Ecole Nationale Supérieure des Télécommunications, Paris

Thèse

en vue de l'obtention du diplôme de docteur de
l'École Nationale Supérieure des Télécommunications

Spécialité: Informatique-Réseaux

Correspondant à l'ENST : Philippe PICOUET

# MULTI-FACET ANALYSIS OF E-NEGOTIATIONS

Willy PICARD
<picard@kti.ae.poznan.pl>

| | |
|---|---|
| **Directeur de thèse :** | Wojciech CELLARY |
| | *Professeur à l'Université des Sciences Économiques de Poznań* |
| **Rapporteurs :** | Geneviève JOMIER |
| | *Professeur à l'Université Paris-Dauphine* |
| | Winfried LAMERSDORF |
| | *Professeur à l'Université de Hambourg* |
| **Examinateurs :** | Michel SCHOLL |
| | *Professeur au Conservatoire National des Arts et Métiers* |
| | Philippe PICOUET |
| | *Maître de Conférences à l'École Nationale Supérieure des Télécommunications de Bretagne* |
| | Jean-Marc SAGLIO |
| | *Directeur d'Études à l'École Nationale Supérieure des Télécommunications* |

Poznań, le 12 octobre 2002

# Contents

# Résumé

## Introduction

Négocier est un acte fondamental dans le monde du commerce. Toute transaction commerciale est fondée sur un contrat négocié au préalable. Dans le contexte de la délocalisation et de la mondialisation de l'économie, les entreprises doivent négocier à l'échelle planétaire. Les négociations réparties sont nécessaires non seulement pour les entreprises multinationales établies dans de nombreux pays, mais également pour les petites et moyennes entreprises (PME) qui évoluent de plus en plus dans un environnement international. Cependant, les méthodes de négociation classiques ne sont pas adaptées aux négociations réparties. Habituellement, les personnes prenant part à la négociation se rencontrent directement afin d'échanger des informations et confronter leurs positions de négociation et leurs buts. Ces rencontres sont, cependant, coûteuses en temps et argent. Elles sont, de plus, difficiles à organiser, en particulier lorsque les négociateurs travaillent dans différents pays. Dans le cadre des négociations classiques, seul un nombre limité de participants est possible.

Avec l'avénement de l'Internet, la localisation géographique des négociateurs n'est plus importante. L'Internet permet à un nombre illimité de négociateurs du monde entier de négocier un même contrat. Les acteurs économiques étant de plus en plus interconnectés, et la quantité d'informations qu'ils échangent étant de plus en plus volumineuse, l'incapacité à traiter ces données représente une vraie menace pour ces nouveaux modèles économiques. Dans le cadre des négociations, la quantité de données générées par le processus de négociation est trop volumineuse pour pouvoir être comprise par un être humain. Aussi, le problème à résoudre actuellement est l'organisation et la gestion des négociations distantes auxquelles participe un nombre important de négociateurs. De nouveaux outils sont nécessaires afin de permettre aux négociateurs de travailler efficacement dans ce nouvel environnement, et ceci aussi bien pour les entreprises multinationales que pour les PME.

Trois approches ayant pour but la construction de tels outils peuvent être distinguées : les modèles de négociations pour la conception de marchés électroniques, les modèles de négociations basés sur les agents logiciels, et les négociations assistées par ordinateur (NAO).

Les modèles de négociations pour la conception de marchés électroniques ont été construits afin d'étudier les mécanismes régissant les négociations. Dans ces modèles, le résultat de la négociation est prédit sur la base des préférences des négociateurs et des règles du jeu qui sont déterminées par le type de négociation (enchères, vente par inscriptions, etc.).

Dans les modèles de négociations basés sur les agents logiciels, la responsabilité de la négociation, qui incombe normalement au négociateur, est déléguée à un agent logiciel. Dans ce cas, la négociation est dite *automatisée*. La négociation est dite *entièrement automatisée* si la négociation est menée uniquement par des agents logiciels sans aucune intervention humaine durant le processus de négociation.

L'approche des négociations assistées par ordinateur (NAO) a donnée naissance à des systèmes informatiques qui ont pour tâche d'aider les négociateurs dans la prise de décision en fournissant des moyens de communication et d'analyse des informations disponibles.

Les trois approches mentionnées ci-dessus réduisent le coût du processus de négociation en l'automatisant en grande partie. Aussi, le nombre de rencontres directes est réduit. Cependant, ces approches s'intéressent principalement aux ventes aux enchères. Dans ce type de négociations, les contrats ont une sémantique connue au préalable, i.e. la signification des différentes clauses du contrat est déterminée. De plus, généralement seul le prix est objet à négociation.

Dans le domaine des négociations électroniques, un vrai défi est de permettre à un nombre important de négociateurs de travailler sur des contrats complexes, i.e. des contrats contenant aussi bien des contreparties quantitatives (par exemple prix ou date de livraison) que qualitatives (par exemple garantie ou clause de confidentialité). De telles négociations incluent aussi bien les négociations classiques dans lesquelles la sémantique des contrats est connue que celles dans laquelles elle est inconnue. Ces négociations électroniques profitent des caractéristiques des médias électroniques — le nombre de négociateurs est potentiellement illimité et ceux-ci peuvent être arbitairement dispersés géographiquement. Cependant, des systèmes de NOA sont nécessaires afin de faciliter les négociations de masse via le réseau. Sans ces systèmes, les négociations de masse à distance sont difficilement imaginables. Les exigences pour de tels systèmes de NOA sont les suivantes :

- la sémantique des contrats n'est pas nécessairement connue,
- les contrats peuvent contenir aussi bien des clauses quantitatives et qualitatives,
- le nombre de négociateurs n'est pas limité.

Les modèles de négociations pour la conception de marchés électroniques et ceux basés sur les agents logiciels satisfont seulement à la dernière exigence. Toutefois, ces modèles ne satisfont pas aux deux premières exigences présentées ci-dessus. A l'opposé, les systèmes de NOA satisfont à la première exigence, certains satisfont également à la deuxième. Ces systèmes ne sont cependant pas extensibles, i.e. ils ne satisfont pas à la dernière exigence.

Dans le cas de contrats multi-attributs contenant aussi bien des contreparties quantitatives que qualitatives, les négociations entièrement automatisées ne sont pas une solution viable. Les agents logiciels, tels que nous les connaissons actuellement, ne peuvent exploiter les contreparties qualitatives du fait du manque de sémantique. Les négociations de tels contrats doivent être menées par des hommes. Cependant, sans outils adaptés, les humains ne sont capables de mener de telles négociations dans lesquelles un nombre important de négociateurs est engagé. La quantité de données générées durant le processus de négociation est trop volumineuse pour être comprise par un être humain. Aussi, les systèmes de NAO pour les négociations de masse via l'Internet doivent fournir aux négociateurs des vues synthétiques du processus de négociation.

Dans ce mémoire, une approche au problème présenté ci-dessus — appelé *analyse multiaspect des négociations électroniques* — est présentée. La solution proposée se compose d'un modèle de contrat multiversion et d'un mécanisme d'analyse hiérarchique multiaspect basé sur les distances ultra-métriques. Le modèle de contrat multiversion proposé capture l'évolution du contrat et les relations existant entre les diverses propositions et contre-propositions. Le mécanisme d'analyse hiérarchique multiaspect permet la création de diverses vues synthétiques du processus de négociation. Une vue synthétique est une classification hiérarchique qui permet aux négociateurs d'explorer un aspect choisi de la négociation à divers niveaux de détails et de se concentrer sur les parties les plus intéressantes.

La thèse de ce mémoire peut être formulée comme suit:

**L'analyse multiaspect des versions du contrat fournit aux négociateurs des vues synthétiques du processus de négociation électronique.**

## Définitions

Plusieurs définitions d'un « contrat » existent, selon le domaine d'application et/ou la localisation géographique. Il est cependant possible de trouver le dénominateur commun suivant à ces définitions. Pour qu'un contrat soit valide, les éléments suivants doivent exister :

- le consentement des parties qui s'obligent ;
- la capacité des parties à contracter ;
- un objet qui forme la matière de l'engagement ;
- une cause licite dans l'obligation.

Dans ce mémoire, seuls les contrats explicites sont considérés, les contrats implicites ne pouvant être l'objet d'une quelconque négociation. Il est supposé que les contrats existent sous forme électronique. Les éléments du contrat qui doivent être présents sont le consentement des parties et un object qui forme la matière de l'engagement. La capacité et la cause licite ne sont pas prises en compte, étant donné qu'aucun système informatique ne peut vérifier ces informations. Les clauses des contrats ne sont soumises à aucune limitation. Elles peuvent concerner des contreparties quantitatives (par exemple prix ou date de livraison) et qualitatives (par exemple garantie ou clause de confidentialité).

Plusieurs classifications ont été proposées dans la littérature pour préciser la notion de négociation, dont notamment :

- négociations classiques *vs* négociations électroniques ;
- négociations distributives *vs* intégratives ;

- négociations multi-attributs *vs* mono-attribut.

De plus, la distinction entre négociations de masse et négociations à nombre restreint de participants est introduite. Une négociation est dite *électronique* si et seulement si les tâches liées au processus de négociation sont effectuées sous une forme écrite via un médium électronique. Une négociation est *distributive* si le gain d'un négociateur implique une perte pour un autre. Par opposition, une négociation est *intégrative* si le gain est réparti entre les négociateurs. Dans le cas d'une négociation *mono-attribut*, une seule contrepartie est négociée. Dans le cas d'une négociation *multi-attributs*, plusieurs contreparties sont négociées. Si toutes les contreparties sont de types similaires, la négociation est dite *homogène*, alors que, dans le cas contraire, on parle de négociation *hétérogène*. Une négociation *de masse* est telle que le nombre de négociateurs y prenant part est supérieur au nombre maximum de personnes pouvant discuter ensemble dans une pièce. Ce nombre est fixé de manière subjective à 20. Dans ce mémoire, les négociations prises en compte sont aussi bien distributives qu'intégratives, multi-attributs, éventuellement hétérogènes, et sont des négociations de masse.

## Domaines de recherche connexes: états de l'art

Trois domaines de recherche ayant rapport à l'approche presentée dans ce mémoire ont été identifiés : les modèles de négociation électronique, les algorithmes de classification hiérarchique automatique, et les techniques de gestion de versions.

Trois approches à la problématique des négociations électroniques peuvent être distinguées : les modèles de négociations pour la conception de marchés électroniques, les modèles de négociations basés sur les agents logiciels, et les négociations assistées par ordinateur (NAO).

Les modèles de négociations pour la conception de marchés électroniques ont été construits afin d'étudier les mécanismes régissant les négociations. Ces modèles de négociations sont basés sur la théorie des jeux [3, 53, 51]. Le processus de négociation est modélisé comme un processus de prise de décision en situation d'incertitude. Utilisant la théorie de l'utilité [87], le résultat de la négociation est prédit sur la base des préférences des négociateurs et des règles du jeu qui sont déterminées par le type de négociation (enchères, vente par inscriptions, etc.).

Dans les modèles de négociations basés sur les agents logiciels, la responsabilité de la négociation, qui incombe normalement au négociateur, est déléguée à un agent logiciel. Ces modèles de négociations se concentrent sur trois aspects [39] : l'établissement d'ontologies, les protocoles de négociation et les modèles de prise de décision. La signification des divers éléments de la négociation et les relations entre ces éléments sont définies par des ontologies [76, 26, 20, 68]. Les ontologies sont nécessaires afin de limiter les malentendus résultant de définitions imprécises des termes utilisés dans les négociations. Les protocoles de négociation [75, 78, 16, 52, 33] définissent les types de participants, les actions valides, les états possibles de la négociation, ainsi que les événements qui causent une modification de l'état de la négociation. Les modèles de prise de décision [65, 82, 83, 5, 84, 47, 73] se concentrent sur le problème de la dynamique du comportement des agents logiciels dans un marché en perpétuel changement.

L'approche des négociations assistées par ordinateurs (NAO) [90, 63, 42, 43, 44, 41, 64, 69, 71, 70] a donné naissance à des systèmes informatiques qui ont pour tâche d'aider les négociateurs dans leurs prises de décision en fournissant des moyens de communications et d'analyse des informations disponibles. Deux approches peuvent être distinguées : les systèmes de préparation et d'évaluation assistent les négociateurs dans la prise de décision avant et durant la négociation, alors que les systèmes de soutien au processus de négociation fournissent les moyens de communication et, dans certains cas, des mécanismes de médiations.

Deux algorithmes de classification hiérarchique automatique peuvent être distingués : les méthodes hiérarchiques basées sur les distances ultra-métriques et les méthodes basées sur les réseaux neuronaux.

Les méthodes hiérarchiques effectuent plusieurs classifications emboîtées les unes dans les autres, permettant ensuite de choisir celle qui correspond au niveau de détail désiré. Ces méthodes reposent sur l'équivalence entre une hiérarchie indicée sur $E$ (espace des échantillons) et une distance ultra-métrique sur $E$. Une distance ultra-métrique sur $E$ est une application $\delta$ de $E \times E$ dans $\mathbb{R}^+$ telle que :

$$\begin{aligned} \forall (a,b) \in E^2, &\quad \delta(a,b) = 0 \Leftrightarrow a = b, \\ \forall (a,b) \in E^2, &\quad \delta(a,b) = \delta(b,a), \\ \forall (a,b,c) \in E^3, &\quad \delta(a,b) \leq \sup[\delta(a,c), \delta(b,c)] \end{aligned}$$

Il est demontré au chapitre 3.2 que, si $\delta$ est une ultra-métrique, la relation $\mathcal{R}_{\delta_0}$ définit une relation d'équivalence sur $E$, avec $\forall \delta_0 \in \mathbb{R}^+, a\mathcal{R}_{\delta_0}b \Leftrightarrow \delta(a,b) \leq \delta_0$.

Dans le cas où $E$ est fini, on appelle hiérarchie sur $E$ un sous-ensemble $\mathcal{H}$ de $P(E)$, où $P(E)$ désigne l'ensemble des parties de $E$, qui vérifie les propriétés suivantes :

$$\begin{aligned} & E \in \mathcal{H} \\ \forall a \in E, \quad & \{a\} \in \mathcal{H} \\ \forall (h_1,h_2) \in \mathcal{H}^2, \quad & \begin{cases} & h_1 \cap h_2 = \emptyset, \\ ou & h_1 \subset h_2, \\ ou & h_2 \subset h_1. \end{cases} \end{aligned}$$

La partition formée des classes d'équivalences de $\mathcal{R}_{\delta_0}$, notée $P(\delta_0)$, induit une hiérarchie sur $E$ : $\mathcal{H} = \bigcup_{\delta_0 \in \mathbb{R}^+} P(\delta_0)$ est une hiérarchie. Si l'on définit la notion de hiérarchie indicée comme étant un couple $(\mathcal{H}, f)$ tel que $\mathcal{H}$ est une hiérarchie et $f$ une application de $\mathcal{H}$ dans $\mathbb{R}^+$ telle que :

$$\begin{aligned} \forall a \in E, &\quad f(\{a\}) = 0, \\ \forall (h_1,h_2) \in \mathcal{H}^2, &\quad h_1 \subset h_2, h_1 \neq h_2 \Rightarrow f(h_1) < f(h_2), \end{aligned}$$

alors le couple $(\mathcal{H}, f)$ est une hiérarchie indicée, avec $\mathcal{H} = \bigcup_{\delta_0 \in \mathbb{R}^+} P(\delta_0)$ et $f$ définie par :

$$\forall h \in \mathcal{H},\ f(h) = \min\{\delta_0 / h \in P(\delta_0)\}$$

Il est également démontré au chapitre 3.2 que toute distance induit une ultra-métrique, et donc, induit une hiérarchie indicée.

Les méthodes de classification basées sur les réseaux neuronaux divisent un ensemble d'échantillons en classes qui sont intrinsèques à l'ensemble donné en entrée. Kohonen [45, 46] a introduit le concept d'ordre dans les réseaux neuronaux. Dans les réseaux de Kohonen — appelés mappes auto-adaptatives (ang. *« self-organizing maps », SOM*) — les nœuds de sortie sont ordonnés, généralement sur une grille bidimensionnelle. Ces réseaux conservent la topologie : deux échantillons proches seront attribués à des nœuds de sortie proches. SOM a été modifiée par Fritzke [29, 30] afin de supprimer la limitation principale de SOM, i.e. la structure du réseau est fixée à l'avance. La solution proposée — appelée structures de cellules croissantes (ang. « *growing cell structures* ») — est basée sur un réseau SOM dont la structure est variable, de nouveaux nœuds pouvant être ajoutés au réseau. Sur la base des structures de cellules croissantes et de SOM, Dopazo et Carazo [23, 24] ont construit un nouveau type de réseau afin de classifier des alignements de séquences génétiques. Cette nouvelle approche — appelée algorithme de l'arbre auto-adaptatif (ang. « *self-organizing tree algorithm* ») — génère une classification d'un ensemble d'alignements sous forme d'arbre binaire. L'espace de sortie peut croître de même manière que dans l'approche des structures de cellules croissantes. Les nœuds de sorties sont des vecteurs de taille identique aux alignements de séquences. L'algorithme de l'arbre auto-adaptatif est présenté en détail au chapitre 3.2.5.

Trois techniques de gestion de versions peuvent être distinguées : les systèmes de gestion de configurations logicielles, la gestion de versions sur le Web, et l'approche des versions de base de données.

Les systèmes de gestion de configurations logicielles ont pour but de permettre l'enregistrement des modifications effectuées durant le développement d'un logiciel et de permettre de revenir à des versions passées. Trois systèmes peuvent être distingués : RCS (*Revision Control System*), SCCS (*Source Code Control System*), et CVS (*Concurrent Versions System*). RCS [81] utilise une technique de capture des modifications pour représenter les différentes versions d'un document. RCS enregistre la version la plus récente, les versions précédentes pouvant être recréées à partir de scripts de modification inverse. Ces scripts — appelés parfois *deltas* — sont générés par le programme *diff* [36]. SCCS [66] modélise l'évolution du document à l'aide de deltas entrelacés. Un fichier est divisé en plusieurs blocs. Pour chaque bloc, un en-tête indique son appartenance à une version donnée. Les blocs sont triés de telle manière qu'une seule lecture du fichier permet de recréer toute version. CVS [7] est construit sur RCS. Il introduit la notion de gestion d'ensemble de répertoires, chacun contenant des fichiers pouvant exister en plusieurs versions.

Le groupe de travail sur la composition et la gestion de versions sur le Web (ang. « *WWW Distributed Authoring and Versioning working group* », WebDAV) a défini un ensemble de spécifications concernant un protocole pour la composition et la gestion de versions sur le Web dans le document RFC 2291 [74]. Le protocole proposé se compose d'un ensemble d'extensions du protocole standard HTTP. Un sous-groupe de WebDAV, appelé Delta-V, a pour but de définir les extensions concernant la gestion de versions. Le travail de Delta-V [17] n'est pas encore terminé, aussi les extensions proposées peuvent encore changer. Delta-V définit uniquement un protocole de gestion de versions, l'implantation des mécanismes proposés n'étant pas l'objet du travail de Delta-V.

Dans l'approche des versions de base de données [13], une base de données multiversion regroupe plusieurs états de l'univers, contrairement à une base de données classique qui représente un seul état de l'univers modélisé. Chacune des versions de base de données (VBD) représente un état du monde modélisé. Une VBD est identifiée par un identificateur, noté $v$. De même, les objets contenus dans la base de données multiversion, nommés *objets multiversions,* sont identifiés par un identificateur, noté $o$. Une VBD contient une *version logique* de chaque object multiversion. Une version logique d'objet possède un identificateur et une valeur. L'identificateur de la version logique de l'objet $o_i$ dans la VBD $v_j$ est le couple $(o_i, v_j)$. Quand un objet n'existe pas dans une VBD, sa valeur est *null*, signifiant *n'existe pas*.

Une VBD est créée à partir d'une VBD existante, par copie logique. La VBD créée est dite *dérivée* de la VBD existante. Immédiatement après la dérivation, les valeurs de toutes les versions logiques d'objets sont identiques entre les deux VBD. Les liens de dérivation entre les différentes VBD de la base de données multiversion sont enregistrés dans un *arbre de dérivation*.

Pour obtenir de bonnes performances, les versions logiques ayant la même valeur dans plusieurs VBD partagent la même *version physique* contenant la valeur. Le gestionnaire de versions fait le lien entre chaque version logique $(o, v)$ de l'objet et la version physique $p$ associée contenant la valeur *val*. Ce lien est établi pour chaque objet multiversion par une *table d'association*. Cette table est composée de deux colonnes et de plusieurs lignes. Chaque ligne contient une liste d'identificateurs de VBD et un identificateur de version physique, associant chaque version logique d'objet à sa version physique. Toutes les associations ne sont pas stockées, certaines sont implicites. La règle est la suivante : *« si un identificateur* v *de VDB n'apparaît pas dans la table d'association d'un object multiversion* o*, alors la version logique* $(o, v)$ *partage sa version physique avec la version logique de l'objet dans la VBD parente »*. Cette règle est récursive.

## Concepts

Les modèles de négociations électroniques qui existent actuellement ne sont pas adaptés aux négociations électroniques de masse menées via l'Internet. Les modèles de négociations pour la conception de marchés électroniques, bien qu'élégantes et ayant de solides bases mathématiques, n'offrent qu'une pauvre représentation des cas réels de négociations [40, 67]. Les modèles de négociations basés sur les agents logiciels sont très prometteuses. Cependant, dans le cas de contrats pour lesquels la sémantique des diverses clauses est inconnue — ce qui est le cas de la majorité des contrats — les agents logiciels ne peuvent mener efficacement les négociations. Enfin, les négociations assistées par ordinateurs (NAO) permettent de négocier tout contrat, la négociation étant menée par des êtres humains pour lequels le manque de sémantique et les contreparties qualitatives ne constituent aucun obstacle. Cependant, les systèmes de NAO, ne proposant pas de vues synthétiques du processus de négociation, sont peu adaptés dans le cas où le nombre de négociateurs est important.

La nouvelle approche de la problématique des négociations électroniques de masse proposée dans ce mémoire se situe dans le cadre des systèmes de NAO, puisqu'elle a pour but d'aider les négociateurs dans la prise de décision en fournissant des moyens de communication et d'analyse des informations disponibles. La solution proposée se compose d'un modèle de contrat multiversion et d'un mécanisme d'analyse hiérarchique multiaspect basé sur les distances ultra-métriques.

Le modèle de contrat multiversion se base sur l'approche des versions de base de données. Le premier problème des négociations de masse est la représentation de l'histoire du processus de négociation. On peut cependant remarquer que le contrat négocié est modifié plusieurs fois au cours de la négociation et que les versions consécutives du contrat reflètent les propositions et contre-propositions des négociateurs. Un ensemble partiellement ordonné de versions de contrat est donc utilisé afin de modéliser l'histoire du processus de négociation.

Les versions de contrat sont organisées hiérarchiquement. La version initiale du contrat se trouve à la racine de l'arbre des versions. Lorsqu'un négociateur se joint à la négociation, il doit dériver une version d'une version existante du contrat. Chaque version du contrat doit être identifiée de manière unique.

Dans le modèle de contrat multiversion, le contenu d'un contrat se compose d'un ou plusieurs paragraphes. Une version du contrat se compose des versions de plusieurs paragraphes. Tous les paragraphes existent dans toutes les versions du contrat. Si un paragraphe n'existe pas dans une version donnée du contrat, la version de ce paragraphe dans cette version est *null*. Le contrat se compose des mêmes paragraphes dans toutes ses versions. Les différences entre les différentes versions du contenu du contrat se résument aux différences entre les différentes versions des paragraphes.

Une *structure multiversion* est responsable du maintien de la structure du contrat dans ses diverses versions. La structure étant multiversion et séparée du contenu du contrat, les différences de structure entre les différentes versions se résument aux différences entre les versions de la structure du contrat.

Un processus de négociation de masse n'est possible que si les négociateurs peuvent avoir accès à des vues synthétiques du processus de négociation. Un élément fondamental de toute stratégie de négociation est le processus de planification ([62], p. 40-51). Le processus de planification se base principalement sur plusieurs analyses de l'état de la négociation. Dans le cadre des négociations de masse, ce processus de planification peut difficilement être effectué sans outils adaptés. En conséquence, un mécanisme d'analyse multiaspect intégré au système de NAO est proposé. Ce mécanisme permet d'analyser différents aspects du processus de négociation. Un tel mécanisme implique la possibilité de définir aussi bien l'aspect de la négociation à analyser que le critère d'analyse utilisé.

Les aspects de la négociation à analyser sont modélisés par des *domaines d'analyse*. Un domaine d'analyse est un ensemble d'*objets de domaine*. Ces objets sont créés par des fonctions appelées *fonctions de domaine d'analyse* (ang. « *analysis domain functions* », ADF). Différentes fonctions sont utilisées pour analyser divers aspects du processus de négociation.

Le résultat de l'analyse est une classification hiérarchique. Une classification hiérarchique fournit aux négociateurs plusieurs classifications emboîtées les unes dans les autres. Les négociateurs peuvent alors choisir le niveau de détails de la classification. L'algorithme de classification hiérarchique basé sur les ultra-métriques a été choisi pour l'approche proposée dans ce mémoire. L'algorithme de l'arbre auto-adaptatif impose aux nœuds du réseau une structure identique aux échantillons à analyser. De plus, le résultat de la classification est un arbre binaire. Aussi cet algorithme n'est pas adapté au problème de la classification des objets de domaine. La solution proposée se basant sur le concept de distance, les critères d'analyse peuvent être définis d'une manière compréhensible pour les êtres humains, afin que les négociateurs puissent choisir les critères d'analyse qui les intéressent, et éventuellement en définir de nouveaux.

## Modèle de contrat

Un contrat multiversion se compose d'un ensemble de négociateurs, d'un ensemble de versions du contrat, et d'un ensemble de *composantes* multiversions. Une composante peut être par exemple un paragraphe, une image, une signature électronique ou une représentation de la structure du contrat. Plus formellement, on note:

- $neg_a$ l'identificateur de $NEG_a$, le $a^{ième}$ négociateur,
- $vc_b$ l'identificateur de $VC_b$, la $b^{ième}$ version du contrat,
- $cm_c$ l'identificateur de $CM_c$, la $c^{ième}$ composante multiversion, et
- $C$ le contrat multiversion :

$$C = (\{NEG_a\}, \{VC_b\}, \{CM_c\})$$

Les versions du contrat sont organisées sous forme d'un arbre. La version initiale du contrat est créée par un négociateur qui la publie et permet aux autres négociateurs de créer de nouvelles versions. La création de nouvelles versions — opération appelée *dérivation* — est régie par les règles suivantes :

**Règle 1.** *Immédiatement après l'opération de dérivation, la nouvelle version — appelée* version fille *— est identique à la version dont elle est dérivée — appelée* version mère.

**Règle 2.** *Une modification de la version fille n'a aucune influence sur la version mère.*

**Règle 3.** *Une modification de la version mère n'a aucune influence sur la version fille.*

Chaque version appartient à un seul négociateur. Tout négociateur peut être responsable de plusieurs versions. Un négociateur peut dériver une nouvelle version d'une version dont il n'est pas reponsable.

La structure hiérarchique des versions est modélisée par les identificateurs de versions. Un identificateur de version se compose de l'identificateur du négociateur qui a créé cette version et d'un sous-identificateur qui identifie la position de la version dans l'arbre des versions. Formellement, $vc_b = (neg_{responsable}, vcID_b)$, où $vcID_b$ est le sous-identificateur. Les sous-identificateurs sont régis par la règle suivante :

**Règle 4.** *Si une version est la n-ième version dérivée d'une version dont le sous-identificateur est* p*, le sous-identificateur de la version fille est* p.n*. Le sous-identificateur de la version initiale est* 0.

Deux types de versions peuvent être distinguées : les *versions de travail* sont des versions qui sont modifiables et visibles seulement par leur responsables, alors que les *versions historiques* sont des versions visibles par tous et non modifiables. Il n'est possible de dériver une nouvelle version qu'à partir d'une version historique. Toute version qui vient d'être créée est une version de travail. Lorsque le négociateur qui est responsable d'une version de travail est satisfait de celle-ci, il peut la publier en la transformant en version historique. Toute modification de cette version est alors impossible.

Ces deux types de versions simplifient notablement le problème du contrôle d'accès simultanés aux versions : les versions historiques ne sont pas modifiables et, de ce fait, tous les accès aux versions historiques ne se font qu'en lecture. Les versions de travail peuvent être modifées mais seulement par leur responsable. Aussi, un seul négociateur travaille sur une version de travail donnée, ce qui supprime le problème des accès simultanés.

Les composantes multiversions du contrat sont présentes dans toutes les versions du contrat, indépendamment du responsable d'une version donnée. Si une composante existe dans une version, elle existe dans toutes les versions. Une version d'une composante est appelée une *instance* de la composante. Toute instance est associée à une composante. En d'autres termes, aucune instance ne peut exister « en dehors » d'une composante. Par conséquence, la création d'une instance implique soit son association à une composante existante, soit la création d'une nouvelle composante et l'association de l'instance à celle-ci. Une instance peut être présente dans plusieurs versions du contrat. Le partage d'instance modélise les relations entre les versions de contrat au niveau des composantes. Ce partage est modélisé à l'aide de *table d'association*. Une table d'association existe pour toute composante. Elle associe chaque instance à une ou plusieurs versions du contrat. Plus formellement, une composante se compose d'un identificateur unique $cm_c$, d'un ensemble d'instances, et d'une table d'association. Notons $IC_d$ la $d^{ième}$ instance d'une composante multiversion, et $ic_d$ son identificateur. Une composante multiversion est alors définie comme suit :

$$CM_d = (cm_c, \{IC_d\}, TA),$$

où *TA* est la table d'association définie par :

$$TA = \{(ic_d, \{vc_d\})\}.$$

Une table d'association se compose de plusieurs lignes, une ligne par instance. Chaque ligne est une paire *(identificateur d'instance, ensemble des identificateurs des versions dans lesquelles cette instance existe)*.

Afin de réduire la redondance dans les tables d'association, le mécanisme d'*héritage de version* est utilisé. Ce mécanisme est défini par la règle suivante :

**Règle 5.** *Une version du contrat est mentionnée dans la table d'association si et seulement si l'instance associée à cette version n'est pas partagée avec la version mère.*

Quatre opérations concernant les composantes sont définies : lecture, modification, suppression et création.

L'opération de lecture pour une composante multiversion $CM_c$ consiste à obtenir l'instance $IC_d$ de cette composante pour une version de contrat $VC_b$ donnée. L'algorithme suivant, présenté en pseudo-code, est utilisé :

```
1.  définir $instance$;
2.  assigner $vc_{temp}$= $vc_b$;
3.  assigner $existeLigne$ = faux;
4.  faire {
5.      pour chaque $ligne$ dans $AT$ {
6.          si ($ligne$ contient $vc_{temp}$) {
7.              $existeLigne$ = vrai;
8.              $instance$ = $ligne.instance$;
9.          }
10.     }
11.     si (!$existeLigne$) {
12.         $vc_{temp}$ = parentDe($vc_{temp}$);
13.     }
14. } tant que (!$existeLigne$);
15. retourner $instance$;
```

La boucle, des lignes 5 à 10, permet de vérifier si une version donnée est explicitement mentionnée dans la table d'association. Si c'est le cas, l'instance associée est retournée. Sinon, l'instance est partagée avec la version fille et doit être recherchée récursivement parmi les versions précédentes dans l'arbre des versions.

L'opération de modification pour une composante multiversion $CM_c$ consiste à modifier l'instance $IC_d$ de cette composante pour une version de contrat $VC_b$ donnée. Deux cas doivent être distingués selon que l'instance est partagée ou non. Une instance est partagée dans le cas où $VC_b$ n'est pas mentionnée dans la table d'association ou lorsque la ligne de la table d'association contenant $VC_b$ fait référence à d'autres versions. Si l'instance n'est pas partagée, elle peut être modifiée sans changement de la table d'association. Si l'instance est partagée, une nouvelle instance $IC_{d'}$ est créée et on lui assigne comme valeur la nouvelle valeur de l'instance. Une nouvelle ligne associant $IC_{d'}$ à $vc_b$ est ajoutée à la table d'association. La ligne de la table d'association mentionnant $IC_d$ est modifiée pour qu'elle ne contienne que :

- les identificateurs de toutes les versions associées de manière explicite à $IC_d$ sauf $vc_b$, dans le cas où $vc_b$ était associée explicitement à $IC_d$,
- les identificateurs de toutes les versions filles de $vc_b$ qui ne sont pas explicitement associées à d'autres instances.

L'opération de suppression se résume à l'opération de modification à l'aide d'une valeur spéciale, notée *null*, et dont la signification est *n'existe pas dans cette version*.

L'opération de création d'une nouvelle instance dans une version donnée consiste à créer une nouvelle composante multiversion, à l'ajouter au contrat, et à modifier cette composante dans la version concernée.

Une composante multiversion peut être *composite*. Une composante composite fait référence à d'autres composantes. Les références aux autres composantes sont effectuées par les identificateurs des composantes multiversions, et non par les identificateurs des instances. Aussi, les composantes composites peuvent être utilisées pour modéliser la structure du contrat. Il faut remarquer que la structure du contrat est orthogonale au mécanisme de gestion des versions grâce aux composantes composites. De ce fait, il est possible de construire des modèles de contrats complexes basés sur le modèle de contrat multiversion proposé dans ce mémoire.

## Domaines d'analyse

Les domaines d'analyse sont utilisés afin de modéliser divers aspects du processus de négociation. Un domaine d'analyse est un ensemble d'*objets de domaine*, appelés également *métaObjets*.

Formellement, notons $D_{aspect}$ le domaine d'analyse modélisant un aspect du processus de négociation noté *aspect*. Un objet de domaine $OD_i$ est identifé de manière unique par son identificateur, noté $od_i$, dans le domaine d'analyse auquel il appartient, ou plus précisément :

$$\begin{aligned} \forall(OD_i, OD_j) \in D^2_{aspect}, &\quad od_i = od_j \Leftrightarrow OD_i = OD_j \\ \forall(OD_i, OD_j) \in D_{aspect} \times D_{aspect'}, &\quad od_i = od_j \text{ and } OD_i \neq OD_j \Rightarrow D_{aspect} \neq D_{aspect'} \end{aligned}$$

Un objet de domaine $OD_i$ se compose de :

- un identificateur noté od$_i$,
- un ensemble d'attributs,
- un type.

Un attribut est une paire *(nom, valeur)*. Le nom d'un attribut est une chaîne de caractères. Une chaîne de catactères se compose d'un ou plusieurs caractères définis dans le standard Unicode. Dans le cadre de négociations de masse, aucune restriction n'est posée ni sur la localisation géographique des négociateurs ni sur la langue qu'ils utilisent. Aussi, les noms des attributs doivent pouvoir être exprimés dans plusieurs langues, ce qui implique l'emploi de divers jeux de caractères. L'emploi d'Unicode permet de supprimer la contrainte d'un unique jeu de caractères. La valeur d'un attribut est un objet de domaine, de même que l'identificateur $do_i$.

Le type d'un objet est également une chaîne de caractères. Le type permet de donner une signification à un objet : un être humain peut, par exemple, aisément imaginer ce que représente un objet de type *négociateur*. De plus, l'existence des types d'objets permet de définir la structure d'un objet de type donné : il est ainsi possible de définir un objet de type *négociateur* se composant par exemple de trois attributs nommés « nom », « prénom » et « telMaison » de types respectifs *nomDeFamille*, *prénom*, et *numéroDeTélephone*. L'identificateur des objets de type *négociateur* peut par exemple être de type *numéroDeSécuritéSociale*.

Six types d'objets de domaine — *String*, *Integer*, *Long*, *Float*, *Double*, et *Boolean* — sont les types de base permettant de construire des types d'objets plus complexes. Ces types de base, appelés également *primitives*, ne font référence à aucun autre type d'objet et ne possèdent qu'un seul attribut nommé *value*. La valeur de cet attribut dépend du type d'objet.

Les objets de domaine sont créés par une *fonction de domaine d'analyse* (ang. « Analysis Domain Function », *ADF*). Une ADF est une fonction dont l'image est un domaine d'analyse. Formellement,

$$f \text{ est une ADF} \Leftrightarrow \begin{cases} f \text{ est une fonction d'un ou plusieurs domaines d'analyse } DA_{orig} \\ \mathrm{Im}(f) = \{MO_i\}, \text{ où } \forall i,\ MO_i \text{ est un métaObjet} \end{cases}$$

Si une ADF est une fonction de deux ou plusieurs domaines d'analyse, elle est dite *de plusieurs variables*. Un domaine d'analyse particulier, noté $\emptyset$, est défini par $\mathrm{card}(\emptyset) = 0$. Ce domaine d'analyse permet de distinguer deux types d'ADF : une ADF est une *génératrice* si et seulement si son seul domaine d'origine est $\emptyset$. Si une ADF n'est pas une génératrice, elle est une *transformatrice*.

Une loi de composition permet d'« emboîter » des ADF : le résultat d'une ADF peut être utilisé comme domaine d'origine d'une autre ADF.

**Règle 6.** *La loi de composition pour les ADF, notée $\circ$, définit une ADF $f_\circ$ à partir d'autres ADF $f_i$ comme présenté ci-dessous :*

- *pour les fonctions d'une seule variable, $f_\circ = f_1 \circ f_2 = f_1(f_2)$;*
- *pour les fonctions de plusieurs variables, $f_\circ = f_1 \circ (f_2, \ldots, f_n) = f_1(f_2, \ldots, f_n)$,*

*où $f_1$ est une transormatrice, alors que $f_2$ et $f_3$ peuvent être aussi bien des transformatrices que des génératrices.*

Un nouveau langage de programmation, nommé *ADL* (pour *Analysis Domain Language*), a été conçu afin de définir des ADF. Le langage ADL est un dialecte d'XML — eXtensible Markup Language. Il y a plusieurs raisons d'utiliser XML :

- XML est un standard du consortium WWW ;

- XML permet de définir des langages dont la grammaire est automatiquement validée par un analyseur ;
- XML est conçu et optimisé pour l'analyse des documents XML ;
- Des analyseurs XML sont disponibles ;
- XML peut aisément être intégré aux autres standards basés sur XML.

Le langage ADL se base sur quatre éléments : *MetaObjets* (ang. « MetaObjects »), *EnsembleDObjets* (ang. « ObjectSets »), *Marqueurs* (ang. « Tags ») et *Fonctions* (ang. « Functions »). Les *métaObjets* correspondent aux objects de domaine. Les *ensembleDObjets* correspondent aux domaines d'analyse. Les *marqueurs* sont les éléments de base des calculs. Les *fonctions* correspondent aux ADF.

Le langage ADL est intrinsèquement extensible. ADL définit un mécanisme d'organisation des extensions basé sur le concept de modules. Un module regroupe les définitions de métaObjects, de fonctions générant ceux-ci et, éventuellement l'implantation des fonctionnalités souhaitées sous forme de marqueurs. Le standard « XML Namespaces », basé sur XML, est utilisé afin d'éviter les collisions de noms entre différents modules. Les noms des métaObjets, des fonctions et des marqueurs sont universels, leur validité s'étendant au-delà du module auxquel ils appartiennent. Chaque module est associé à un *URI* (ang. « Uniform Resource Identifier »), ce qui assure l'universalité des noms. Un module peut utiliser des métaObjets, des fonctions et des marqueurs définis dans d'autres modules. Une référence à un module se fait à l'aide d'une référence à l'URI auquel il est associé, appelé *espace de noms* (ang. « XML namespace »).

Un module est défini par un document XML. Un module est défini par son nom, un URI définissant l'espace de noms associé, éventuellement une liste de références aux définitions de marqueurs, éventuellement une liste de références aux définitions de fonctions, éventuellement une liste de références aux définitions de métaObjets.

Le langage ADL fournit un module, appelé *core*, qui définit les fonctionnalités de base d'ADL. L'espace de noms de ce module est `http://nessy.pl/adl/core`. Ce module fournit les types de base et les marqueurs de base d'ADL.

En ADL, les expressions sont définies par `${unExpression}`. Une expression peut contenir des références à des variables, des opérateurs et des littéraux. Si la variable `maVariable` a été définie, l'expression `${maVariable}` retourne la variable `maVariable`. Différents opérateurs existent en ADL : arithmétiques, logiques, comparatifs, ainsi que l'opérateur `empty` qui teste le contenu des ensembleDObjets et des métaObjets. L'opérateur « `.` » permet d'accéder aux attributs des métaObjets. L'opérateur « `[]` » permet d'accéder aux métaobjets contenus dans un ensembleDObjet par leurs identificateurs.

Un métaObjet est défini par un document XML. Un métaObjet est défini par le nom de son type, le nom du type de son identificateur, une liste d'attributs et de leurs types si le métaObjet n'est pas un type de base, le type de l'attribut *value* si le métaObjet est un type de base.

Un ensembleDObjet est un ensemble de métaObjets. Tous les métaObjets d'un ensembleDObjet donné ont le même type. Le module *core* définit cinq marqueurs de manipulation des ensembleDObjets. La création d'un ensembleDObjet s'effectue à l'aide du marqueur `declare`, l'addition d'un métaObjet à un ensembleDObjet s'effectue à l'aide du marqueur `add`, la suppression à l'aide du marqueur `remove`, la suppression de tous les métaObjets d'un ensembleDObjet s'effectue à l'aide du marqueur `clear`. Le marqueur `for-each` permet d'itérer parmi les métaObjets d'un ensembleDObjet.

Les marqueurs associent des entités de calcul à des marqueurs XML. Une entité de calcul est un logiciel ou une partie de logiciel, par exemple une bibliothèque de fonctions statistiques ou une couche d'accès à une base de données. Les marqueurs permettent d'élargir les fonctionnalités d'ADL. Les entités de calcul peuvent être programmées dans différents langages. Un marqueur est défini par un document XML. Un marqueur est défini par son nom, le nom du langage de programmation utilisé pour son implantation et, éventuellement les paramètres spécifiques au langage de programmation choisi.

Les fonctions sont au cœur d'ADL. Une fonction en ADL est une ADF . Chaque fonction modélise un aspect du processus de négociation. Une fonction est définie par un document XML. Une fonction est définie par (1) son nom, (2) éventuellement le nom des modules qu'elle utilise, (3) éventuellement le nom des ensembleDObjets passés en entrée et le type des métaObjets qu'ils contiennent, (4) le nom de l'ensembleDObjet en sortie et le type des métaObjets qu'il contient et, (5) les actions à effectuer. Les actions à effectuer peuvent être des appels de fonctions ou de marqueurs. Pour les appels de fonctions, le marqueur `execute` du module *core* est utilisé. Pour les appels de marqueurs, il suffit d'insérer le marqueur XML associé.

Le module *core* fournit les types de base, ainsi que les marqueurs définissant les fonctionnalités de base d'ADL. On peut distinguer cinq catégories parmi les marqueurs du module *core* : déclaration de variables, assignation d'attribut de métaObjet, contrôle d'exécution, appel de fonction et manipulation des ensembleDObjets. Une description détaillée de tous les marqueurs du module *core* est donnée au chapitre 6.3.7.

## Classification des objets de domaine

Le but de la classification des objets de domaine est de générer une vue synthétique d'un aspect donné du processus de négociation. Le choix de l'aspect à analyser correspond au choix d'une ADF. Le résultat de l'exécution d'une ADF est un domaine d'analyse, i.e. un ensemble d'objets de domaine. Les objets de domaine pouvant modéliser de manière complexe certains aspects du processus de négociation, et différents négociateurs pouvant être intéressés par diverses analyses d'un aspect donné, plusieurs analyses différentes peuvent être conduites pour un même aspect. Pour cette raison, le concept d'analyse paramétrique est proposé.

**Définition 1.** *Une analyse est paramétrique si plusieurs critères peuvent être utilisés pour analyser de diverses manières un domaine d'analyse donné.*

Tous les objets d'un domaine d'analyse donné sont du même type. Le *type d'un domaine d'analyse* peut donc être défini comme le type des objets qu'il contient. Le type d'un objet permet de déterminer les attributs que tous les objets de ce type possèdent. Aussi, les attributs des objets d'un domaine d'analyse peuvent être déterminés à l'aide du type du domaine d'analyse.

Un critère d'analyse se base sur la présence de certains attributs. Les valeurs des attributs sont les données à analyser. Les noms des attributs définissent la sémantique des données à analyser. Aussi, tout critère d'analyse est associé à un type d'objet de domaine donné. De ce fait, le type d'un critère d'analyse peut être défini comme suit :

**Définition 2.** *Le type d'un critère d'analyse est le type des objets de domaine auquel il est associé.*

Un domaine d'analyse donné ne peut être associé qu'avec des critères d'analyse dont le type est identique au type du domaine d'analyse. Une relation *plusieurs à plusieurs* existe entre les domaines d'analyse et les critères d'analyse. Cette relation est une relation de compatibilité.

**Définition 3.** *Un domain d'analyse* DA *et un critère d'analyse* CA *sont compatibles si et seulement si le type de* DA *et le type de* CA *sont identiques.*

Un critère d'analyse est une distance sur un domaine d'analyse donné. Formellement,

**Définition 4.** *Une fonction* CA *est un critère d'analyse si et seulement si*

$$\begin{cases} \text{CA est une fonction de } DA^2 \text{ dans } \mathbb{R}^+ \\ \forall (x,y) \in DA^2, \quad x = y \Leftrightarrow CA(x,y) = 0 \\ \forall (x,y) \in DA^2, \quad CA(x,y) = CA(y,x) \\ \forall (x,y,z) \in DA^3, \quad CA(x,y) \leq CA(x,z) + CA(y,z) \end{cases}$$

Les critères d'analyse sont un sous-ensemble des transformatrices, et de ce fait, sont des ADF. Ils peuvent donc être définis en ADL.

Un critère d'analyse est une ADF dont les domaines d'entrée sont deux domaines d'analyse ne contenant qu'un seul objet. Le domaine de sortie contient un seul objet qui modélise la distance entre les deux objets à comparer.

Formellement, on désigne par *CA* un critère d'analyse sur le domaine d'analyse *DA*. Le type de *CA*, identique au type de *DA*, est noté *typeCA*. Appelons $od_1$ et $od_2$ deux objets de domaine de *DA*, $d \in \mathbb{R}^+$ la distance entre $od_1$ et $od_2$ selon *CA*, et $DA_d$ le domaine d'analyse résultant de l'exécution de *CA*.

Les domaines d'entrée de *CA* sont $DA_1 = \{od_1\}$ et $DA_2 = \{od_2\}$. $DA_d$ ne contient qu'un seul objet $do_d$. Le type de l'objet $do_d$, qui est également le type de $DA_d$, est noté *typeCA*_`distance`. Les objets de type *typeCA*_`distance` possèdent trois attributs : l'attribut `first` (resp. `second`) qui contient le premier objet (resp. le second objet) de la comparaison, i.e. $od_1$ (resp. $od_2$). L'attribut `distance`, de type numérique, est utilisé pour stocker la distance entre ces deux objets.

Les actions nécessaires au calcul de la distance entre deux objets de domaine sont définies de la même manière que toute ADF grâce à des appels de fonctions et de marqueurs. L'utilisateur définissant un nouveau critère d'analyse doit vérifier que les actions utilisées pour le calcul de la distance entre deux objets définissent une distance. Le compilateur ADF ne vérifie pas si un critère d'analyse vérifie les conditions présentées dans la définition 4.

Une classification peut être calculée lorqu'un domaine d'analyse et un critère compatibles ont été choisis. Une classification est définie à l'aide des concepts de *classes* et de *distance inter-classe*s. Une classe peut contenir au choix :

- d'autres classes — la distance inter-classes associée est explicitement définie et la classe est dite *complexe* ;
- un ou plusieurs métaObjets — la distance inter-classes associée est toujours égale à 0 et la classe est dite *atomique*.

Formellement, appelons *C* une classification du domaine d'analyse *DA*. La classification *C* est un ensemble de classes $c_i$ définies comme suit :

$$\forall c_i \in C, \begin{cases} c_i = (\{c_j\}, d_i), \ \text{où} \ \forall j, \ c_j \in C, \ d_i \in \mathbb{R}^{+*} \\ \text{ou} \\ c_i = \{MO_j\}, \ \text{où} \ \forall j, \ MO_j \in DA \end{cases}$$

$$\forall (c_i, c_j) \in C^2, c_i \subset c_j \text{ ou } c_i \supset c_j \text{ ou } c_i \cap c_j = \emptyset$$

Une classification est une hiérarchie indicée à la condition que les distances inter-classes vérifient :

$$\forall (c_i, c_j) \in C^2 \text{ tels que } c_i \neq c_j,\ c_i \subset c_j \Rightarrow d_i < d_j,$$

où $d_i$ (resp. $d_j$) est la distance inter-classes associée à $c_i$ (resp. $c_j$).

La création d'une classification s'appuie sur l'algorithme de classification hiérarchique automatique basé sur les ultra-métriques. Un critère d'analyse n'est pas une ultra-métrique, il est seulement une distance. Or, toute distance induit une ultra-métrique. Une technique pour dériver une ultra-métrique d'une distance est l'utilisation de la distance de chaîne qui, à deux points *a* et *b*, associe le plus petit des pas maximaux de tous les chemins entre ces deux points. La distance de chaîne est une ultra-métrique. Cependant, la complexité du calcul de la distance de chaîne est très élevée : lorsque le nombre d'objets de domaine est *n*, le nombre de chemins possibles entre deux objets a une complexité en *O(n!)*.

Une autre solution pour dériver une ultra-métrique d'une distance se base sur une propriété des ultra-métriques :

**Lemme 1.** *Dans un espace muni d'une ultra-métrique, tous les triangles sont isocèles.*

En utilisant cette propriété des ultra-métriques, on définit l'opération d'« isocèlisation ». L'opération d'« isocèlisation » transforme tout triangle en un triangle isocèle dans lequel les trois côtés $c_1$, $c_2$ et $c_3$ vérifient $\forall (i, j, k) \in \{1, 2, 3\}^3$, $c_i \leq \sup[c_j, c_k]$.

**Définition 5.** *L'opération d'« isocèlisation » transforme*

*un triangle (a,b,c) tel que* $\begin{cases} \delta(a,b) = c_1 \\ \delta(a,c) = c_2 \\ \delta(b,c) = c_3 \\ c_1 \leq c_2 \leq c_3 \end{cases}$ *en un triangle* $(a,b,c)$*, dans lequel* $\begin{cases} \delta(a,b) = c'_1 = c_1 \\ \delta(a,c) = c'_2 = c_2 \\ \delta(b,c) = c'_3 = c_2 \end{cases}$.

Si on applique l'opération d'« isocèlisation » sur tous les triangles existant dans un espace *E* muni d'une distance *d*, tous les triangles deviennent isocèles. Une nouvelle distance δ peut alors être définie sur *E* comme suit : la distance entre deux éléments de *E*, désignés par *a* et *b*, est la longueur du segment *[a,b]* dans l'espace obtenu après « isocèlisation ». Il faut noter que la nouvelle distance δ est une ultra-métrique.

L'algorithme suivant peut être utilisé afin d'« isocèliser » l'espace *E :*

```
segments = listeTriéeDeTousLesSegments;
isoSegments = nouvelle Liste();
nonIsoSegments = listeTriéeDeTousLesSegments;
tant que (nonIsoSegments.taille() !=0 ){
  [a,b]=nonIsoSegments.premierElement();
  pour chaque $c \in A, c \neq a$ et $c \neq b$ {
    isocèlisation(a,b,c);
  }
  isoSegments.ajouter([a,b]);
  nonIsoSegments.supprimer([a,b]);
  nonIsoSegments.trier();
}
```

La complexité de cette technique — en $O(n^3)$ — est moindre que le calcul direct de la distance de chaîne.

L'opération de seuillage extrait diverses partitions d'une classification selon un seuil. Deux concepts sont nécessaires à la définition de l'opération de seuillage : le concept de *contenu d'une classe* et le concept de *classe t-max*.

**Définition 6.** *Le* contenu *d'une classe $c_i$ d'une classification C est un ensemble d'objets de domaine, noté* Contenu($c_i$).

- *Pour les classes atomiques, le contenu d'une classe est la classe elle-même.*
- *Pour les classes complexes, le contenu d'une classe est l'union des contenus de toutes les classes contenues dans cette classe.*

**Définition 7.** *Pour $t \in \mathbb{R}^+$, une classe $c_i$ est une classe* t-*max si et seulement si*

$$d_i \le t \text{ et } \neg(\exists c_j \text{ tel que } c_i \subset c_j \text{ et } d_j \le t),$$

*où $d_i$ est la distance inter-classe de la classe $c_i$.*

**Définition 8.** *Pour un seuil* t *donné, l'opération de seuillage $T_t$ crée une partition $P_t$ de la classification* C. *$P_t$ est l'ensemble des contenus de toutes les classes* t-*max de* C. *Formellement,*

$$P_t = \{Contenu(c_i),\ où\ c_i est\ une\ classe\ t\text{-}max\}.$$

Le choix du seuil permet de contrôler le niveau de détail de la partition obtenue. Plus le seuil est élevé, moins le nombre de classes est important. Dans le contexte de l'analyse de négociations, cette caractéristique de l'opération de seuillage est une propriété clef qui permet :

- divers niveaux d'analyse ; lorsque le seuil est faible, la partition créée se compose de nombreuses classes, ce qui représente une analyse détaillée. Lorsque le seuil est élevé, la partition se compose de peu de classes, permettant d'avoir une vue plus globale de l'aspect du processus de négociation analysé ;
- de se concentrer sur les parties les plus intéressantes ; en partant d'un niveau d'analyse élevé, un négociateur peut sélectionner les classes qui l'intéressent particulièrement. Ces classes peuvent alors être analysées plus en détail en utilisant un seuil plus faible.

## Application de l'analyse multiaspect aux négociations électroniques

Dans ce chapitre, un exemple d'application de l'analyse multiaspect aux négociations électroniques est présenté. Dans cet exemple, trois négociateurs travaillent sur un contrat concernant la publication d'une nouvelle sur le site Web d'une entreprise de publication sur Internet. Le contrat négocié est un contrat de publication classique.

L'évolution du contrat, dans le cadre du modèle de contrat multiversion, est présentée : l'arbre des versions ainsi que les tables d'association des différentes composantes du contrat sont décrits et analysés.

Trois exemples d'analyses sont présentés en détail afin d'illustrer l'utilisation du mécanisme de classification pour divers aspects et selon divers critères de classification. La possibilité de choix du niveau de détail de la classification est également présentée.

## Prototype de système de négociations électroniques

Un prototype, nommé *NeSSy* (pour *Negotiation Support System*), a été construit afin de permettre aux négociateurs de négocier des contrats complexes et d'effectuer diverses analyses du processus de négociation. Le prototype *NeSSy* permet de contrôler les négociateurs et les contrats. Il permet également d'effectuer des analyses du processus de négociation pour divers aspects et selon divers critères.

Trois modules peuvent être distingués : un module pour les contrats multiversions, un générateur de domaines d'analyse, et un générateur de classifications.

Le module pour les contrats multiversions se compose d'un gestionnaire et d'un éditeur. Le gestionnaire est responsable de la création et la destruction des négociateurs et des contrats. L'éditeur est responsable de la dérivation de nouvelles versions du contrat, ainsi que de l'édition de leur contenu.

Le générateur de domaines d'analyse est responsable de la compilation et de l'exécution des ADF. Le générateur est également responsable de la compilation des documents XML définissant les métaObjets. Enfin, le générateur fournit une implantation du module *core* du langage ADL.

Le générateur de classifications est responsable de la création des classifications de métaObjets selon un critère d'analyse. Il est également responsable de la délégation de l'exécution des critères d'analyse au générateur de domaines d'analyse.

## Conclusion

L'approche de l'analyse multiaspect des négociations électroniques qui est présentée dans ce mémoire résout le problème des négociations de masse par Internet, permettant à un grand nombre de négociateurs dispersés géographiquement de travailler sur des contrats complexes.

Deux idées sont à la base de cette approche : premièrement, des vues synthétiques du processus de négociation sont nécessaires dans les négociations de masse de contrats complexes car la quantité d'informations est très grande, deuxièmement, les relations entre les propositions et contre-propositions sont des informations qui peuvent être analysées plus facilement que le contenu des clauses du contrat, les attributs du contrat pouvant être quantitatifs et leur sémantique n'étant pas toujours connue.

Dans l'approche de l'analyse multiaspect, les contrats peuvent contenir des attributs aussi bien quantitatifs que qualitatifs dont la sémantique n'est pas toujours connue. De tels contrats sont prédominants dans le monde du commerce. Le modèle de contrat proposé n'impose aucune limitation en ce qui concerne la structure des contrats et permet la construction de nouvelles structures de contrat basées sur le modèle de contrat multiversion proposé.

Ce modèle introduit un mécanisme de gestion de versions au sein du processus de négociation. Ce mécanisme permet de s'affranchir du problème de l'extensibilité concernant le nombre de négociateurs. Le modèle d'édition de contrat est un modèle parallèle qui est très extensible, chaque négociateur travaillant de manière isolée sur ses propres versions de travail uniquement. De ce fait, un nombre arbitraire de négociateurs peut participer au processus de négociation. Le mécanisme de gestion de versions introduit également les tables d'association qui modélisent les relations entre les propositions et les contre-propositions.

L'approche proposée fournit également aux négociateurs des outils d'analyse du processus de négociation. Le mécanisme d'analyse a pour but de réduire la grande quantité d'informations créée durant le processus de négociation. Des vues synthétiques du processus de négociation permettent aux négociateurs de comprendre aisément divers aspects d'une négociation et de se concentrer sur les parties les plus intéressantes. Une analyse se compose de deux parties : tout d'abord, il convient de définir l'aspect de la négociation à analyser, puis de choisir le critère d'analyse. Cette décomposition permet une meilleur factorisation des analyses, un aspect de la négociation pouvant être analysé selon divers critères, et un critère donné pouvant être utilié pour classifier divers aspects.

Trois techniques sont utilisées dans l'approche de l'analyse multiaspect : un modèle de contrat multiversion, une technique de création de domaines d'analyse, et un algorithme de classification de ces domaines. Chacune de ces techniques est intéressante en elle-même, mais la combinaison de ces trois techniques constitue une approche qui est particulièrement flexible et riche en fonctionnalités : les tables d'associations peuvent être analysées, même lorsque la sémantique des clauses est inconnue, puisque les tables d'association modélisent certains aspects du processus de négociation et ne s'appuient pas sur le contenu des clauses.

Les résultats principaux de cette thèse sont les suivants :

- Identification et évaluation des approches existantes à la problématique des négociations électroniques, des algorithmes de classification hiérarchique automatique, et des techniques de gestion de versions ;
- Développement d'un modèle de contrat multiversion qui permet de modéliser l'évolution du contrat au cours du processus de négociation. Ce modèle capture les relations entre les diverses propositions et contre-propositions. La structure du contrat est orthogonale au mécanisme de gestion de versions ;
- Développement d'un nouveau langage de programmation, appelé ADL, qui permet de définir des fonctions qui génèrent des objets de domaines modélisant divers aspects du processus de négociation ;
- Développement d'un mécanisme de classification hiérarchique qui crée des vues synthétiques du processus de négociation. Ce mécanisme utilise ADL pour définir les critères d'analyse. L'aspect hiérarchique des vues synthétiques permet aux négociateurs de choisir la finesse de l'analyse ;
- Un exemple d'application de l'analyse multiaspect aux négociations électroniques permettant une meilleure compréhension du modèle de contrat multiversion et du mécanisme d'analyse ;
- Implantation d'un prototype de NAO se composant d'un module de contrat multiversion, d'un compilateur ADL et d'un génerateur de classification.

Une propriété importante de l'approche multiaspect est son extensibilité. L'extensibilité est une exigence pour le mécanisme de classification. De nouvelles facettes du processus de négociation peuvent facilement être analysées grâce à l'utilisation d'ADL pour extraire et classifier les données modélisant ces nouvelles facettes. Le modèle de contrat multiversion est également extensible grâce au fait que la structure des contrats n'est pas figée. Aussi, des structures complexes de contrat peuvent être construites en utilisant les concepts de composantes multiversions et composantes composites proposés dans le modèle de contrat multiversion

L'approche de l'analyse multiaspect des négociations électroniques ouvre de nouvelles perspectives de recherche. Un exemple intéressant est l'application de l'approche proposée au domaine de l'informatique nomade (ang. « mobile computing »), permettant à des négociateurs mobiles, éventuellement déconnectés, d'analyser le processus de négociation. Le modèle de contrat multiversion proposé capture diverses facettes importantes du processus de négociation dans des structures de faible taille — les tables d'association. Ces structures peuvent être transmises de manière efficace par des réseaux à bande passante limitée et peuvent être stockées dans les appareils limités en mémoire, tels que les téléphones portables et les PDA. Les négociateurs peuvent alors analyser certains aspects du processus de négociation sans être obligés de télécharger complètement le contrat multiversion.

Un autre exemple est l'utilisation d'agents logiciels. Des modèles comportementaux, basés sur le mécanisme d'analyse, peuvent être construits. Des modèles psychologiques et sociaux du comportement des agents de négociation peuvent s'appuyer sur les données résultant de l'analyse de plusieurs aspects du processus de négociation. Le problème des ontologies est miminisé dans ce cas, puisque la signification associée à divers aspects du processus de négociation est inclus dans les ADF.

Une autre direction de recherche intéressante est l'utilisation de l'approche proposée dans tout domaine d'application dans lequel une importante quantité d'informations existe et évolue dans le temps. La gestion de configuration logicielle est un bon exemple d'un tel domaine d'application. L'évolution du code source pourrait être modélisée à l'aide du modèle de contrat multiversion, capturant les relations entre les diverses versions du code source. Le mécanisme d'analyse pourrait représenter un grand pas en avant dans le processus de développement logiciel, permettant aux développeurs d'avoir une meilleure compréhension du projet dans sa totalité, et ce sous différents aspects.

# Abstract

In the era of delocalization and globalization of economy, companies need to negotiate at a global scale. The high costs related to face-to-face meetings can be reduced by the use of the Internet as a communication medium. New tools are, however, needed to allow contractors, both from multinational enterprises and from SMEs, to negotiate efficiently in this highly concurrent environment, in which the number of negotiators working on a given contract is potentially unlimited.

The existing e-negotiation approaches — negotiation models for electronic market design, agent-based negotiation models, and computer support for negotiation — provide methods of reducing the cost of negotiation processes. By the use of these approaches, major part of the negotiation process is automated, so the number of face-to-face meetings is significantly reduced. However, up to date, these approaches have been mainly focused on auctions. In the case of auctions, contracts are semantics-enabled, i.e. the meaning of their various elements and clauses is known. Scalability with regard to the number of negotiators is a requirement that may be easily observed.

A real challenge in the area of e-negotiations consists in building negotiation support systems that allow a high number of negotiators to work on real-life contracts, i.e. containing both aggregable attributes (e.g. price, quantity, etc.) and non-aggregable attributes (e.g., legal clauses, appendices, quality clauses, etc.). Such a negotiation support system requires tools that provide negotiators with synthetic views of the negotiation process. In mass electronic negotiations, due to the high number of negotiators, the amount of data generated during a negotiation process is so large that a negotiator is unable to analyze them without a proper support.

In this dissertation, an approach, called *multi-facet analysis of e-negotiation*, is presented that enables various aspect of a negotiation process to be analyzed. In this approach, three new techniques are proposed: first, a multiversion contract model used to represent the negotiation process and to capture the relationships between offers and counter-offers; second, analysis domains, which consists of domain objects modeling various facets of a negotiation process, and finally, classification of domain objects, which provide negotiators with hierarchical synthetic views of analysis domains.

A multiversion contract consists of various versions of the contracts, reflecting various propositions made by negotiators during the negotiation process. The multiversion contract model is a parallel edition model, so each negotiator works on her/his own contract versions. Therefore, no bottleneck occurs due to concurrency control, and the scalability with regard to the number of negotiators is arbitrarily high. In the multiversion contract model the structure of contract versions is not defined, instead versioning mechanism is defined. As the contract structure is orthogonal to the versioning mechanism, new contract models may be built on the top of the proposed multiversion contract model. The versioning mechanism is based on the concept of *association tables* which are entities capturing the relationships between various offers and counter-offers.

A new language — called *ADL* — has been designed to generate analysis domains. The ADL language allows to define various aspects of a negotiation process as domain objects, and to define functions retrieving domain objects, called *ADF*. The ADL language offers programming concepts known from procedural languages like loops, conditions and variables. The ADL language is inherently extensible, allowing new domain objects and functions retrieving them to be added.

The goal of the classification mechanism is to provide a synthetic view of a facet of the negotiation process, generated by an ADF. As domain objects may model complex views of a negotiation process facet, while the interests of a given negotiator may be different from those of another negotiator, several analyses may be performed on the same domain objects. Classification of domain objects may be processed according to various analysis criteria. Analysis criteria are also defined in the ADL language. A classification is hierarchical, which allows negotiators to have different levels of detail and to focus efficiently on the elements of their highest interest.

Theoretical results presented in this dissertation have been validated by implementation of a prototype negotiation support system.

# Streszczenie

W dobie globalizacji gospodarki, przedsiębiorstwa muszą prowadzić negocjacje w skali globalnej. Wysokie koszty związane z bezpośrednimi spotkaniami osobistymi mogą być znacznie zmniejszone dzięki użyciu Internetu jako medium komunikacyjnego w procesach negocjacji. Potrzebne są jednak nowe narzędzia programowe, aby umożliwić negocjatorom, zarówno z międzynarodowych korporacji, jak i z małych i średnich firm, skuteczne negocjowanie w tych nowych warunkach.

Obecnie głównym obiektem badań w dziedzinie elektronicznych negocjacji są licytacje. Kontrakty negocjowane w formie licytacji są charakterystyczne w tym sensie, że mają określoną semantykę, czyli że znaczenie poszczególnych ich części i klauzul jest znane. W przypadku licytacji skalowalność w rozumieniu liczby negocjatorów nie jest wymaganiem trudnym do spełnienia ze względu na prostą komunikację.

Prawdziwym wyzwaniem stojącym przed technikami elektronicznych negocjacji jest zbudowanie systemów komputerowego wspomagania negocjacji, które umożliwiłyby dużej liczbie negocjatorów negocjowanie przez sieć skomplikowanych i rozbudowanych kontraktów. Kontrakty te mogą zawierać atrybuty mierzalne (np. cenę, ilość itp.), oraz niemierzalne (np. klauzule prawne, załączniki itp.). Zbudowanie takiego systemu wymaga nowych narzędzi teoretycznych, które pozwoliłyby na automatyczne generowanie syntetycznych obrazów procesu negocjacji. Z powodu dużej liczby negocjatorów ilość danych wygenerowana w trakcie procesu elektronicznych negocjacji jest bowiem tak duża, że człowiek bez odpowiedniego wspomagania nie jest w stanie ich objąć, a tym bardziej przeanalizować.

W tej rozprawie przedstawiono podejście, nazywane *wieloaspektową analizą elektronicznych negocjacji*, które umożliwia analizę procesu elektronicznych negocjacji. Na to podejście składają się trzy nowe techniki. Po pierwsze, opracowano model wielowersyjnego kontraktu, który dobrze reprezentuje proces negocjacji i odzwierciedla związki między kolejnymi ofertami. Po drugie, wprowadzono pojęcie dziedziny analizy, będącej zbiorem obiektów modelujących różne aspekty procesu negocjacji. Po trzecie, zaproponowano metodę klasyfikacji opartą na analizie dziedzin. Metoda ta umożliwia tworzenie syntetycznych hierarchicznych obrazów dziedzin analizy na potrzeby negocjatorów.

Wielowersyjny kontrakt składa się z wielu wersji kontraktu, odzwierciedlających oferty negocjatorów w trakcie negocjacji. Zaproponowany model wielowersyjnego kontraktu jest modelem równoległym, w którym każdy negocjator pracuje tylko nad swoimi wersjami kontraktu. Dzięki temu nie ma wąskiego gardła wynikającego z zarządzania współbieżnością, a skalowalność w odniesieniu do liczby negocjatorów jest wysoka. W modelu wielowersyjnego kontraktu nie zdefiniowano stałej struktury wersji kontraktu, tylko mechanizm wersjowania. Struktura kontraktu jest ortogonalna do mechanizmu wersjowania. Można zatem budować dowolne modele kontraktu bez zmiany zaproponowanego modelu kontraktu wielowersyjnego. Podstawą mechanizmu wersjowania jest koncepcja *tablic związków*, które odzwierciedlają powiązania między kolejnymi ofertami negocjatorów.

Dziedziny analizy są generowane za pomocą zaprojektowanego w tym celu języka *ADL*. Język ten umożliwia definiowanie różnych aspektów procesu negocjacji jako obiektów dziedzin i definicję funkcji, nazywanych *ADF*, generujących te obiekty. Język ADL zawiera struktury programistyczne znane z proceduralnych języków programowania, takie jak pętle, wyrażenia warunkowe i zmienne. Język ADL jest z założenia rozszerzalny. Pozwala na dodawanie nowych obiektów i funkcji ADF.

Celem mechanizmu klasyfikacji jest tworzenie syntetycznych obrazów różnych aspektów procesu negocjacji, które są modelowane przez obiekty generowane przez ADF. Ponieważ obiekty mogą modelować skomplikowane aspekty procesu negocjacji, a zainteresowania jednego negocjatora mogą się znacząco różnić od zainteresowań drugiego, jest możliwe wykonywanie wielu analiz jednego aspektu. Klasyfikacja obiektów dziedziny może być dokonywana według różnych kryteriów analizy. Kryteria analizy są również definiowane w języku ADL. Klasyfikacja jest hierarchiczna, co pozwala negocjatorom wybierać różne poziomy szczegółowości. Hierarchiczna klasyfikacja pozwala negocjatorom skupić się nad aspektami interesującymi ich w danym momencie negocjacji.

Wyniki teoretyczne przedstawione w rozprawie zostały potwierdzone praktycznie w zaimplementowanym prototypie systemu wspomagania elektronicznych negocjacji.

CHAPTER 1

# INTRODUCTION

Negotiation is a fundamental act in business. Every business transaction is basing on a contract that has been previously negotiated. In the context of economy globalization, companies need to negotiate at a global scale. Distributed negotiations are needed not only for multinational enterprises spread in many countries, but also for small and medium enterprises, which are working more and more in an international environment. However, classical ways of conducting negotiations are not well adapted to distributed negotiations. People involved in a negotiation process are used to personally meet to exchange information and to confront their interests and goals. Personal meetings are, however, costly in terms of time and money, as well as difficult to organize, in particular if negotiators work in different countries. In classical negotiations only a small number of participants are involved.

With the rise of Internet, geographical location of negotiators becomes unimportant. Internet allows a potentially unlimited number of negotiators from the whole world to negotiate on a given contract. As economical actors are more and more interconnected, and the amount of information exchanged between actors is higher and higher, data overflow threatens these new economical models. In the field of contracting, when the Internet gives the possibility to connect a potentially unlimited number of actors, the amount of data generated by the negotiation process is too high to be understood by humans. Now, the problem is the organization and the management of remote negotiations conducted by a high number of negotiators. New tools are needed to allow contractors, both from multinational enterprises and from SMEs, to negotiate efficiently in this highly concurrent environment.

Three approaches which aim at providing such tools may be distinguished: negotiation models for electronic market design, agent-based negotiation models, and computer support for negotiation.

Negotiation models for electronic market design have been build to understand what are the mechanisms involved in negotiations. In these models, the result of the negotiation is predicted on the basis of each contractors preferences and the rules of games, determined by the negotiation type (English, Dutch, Vickrey, first-price sealed-bid auctions, etc. [35]).

In agent-based negotiation models, the responsibility of the negotiation is delegated from a human negotiator to a computer. In such case we talk about automated negotiations. The negotiation is said to be fully automated if negotiation is conducted by software agents without human intervention in the whole negotiation process.

The approach consisting of computer support for negotiation gave birth to negotiation support systems (NSS). These systems are designed to assist negotiators in reaching mutually satisfactory decisions by providing means of communication and analysis of available information.

The three approaches mentioned above reduce the costs of negotiation processes, because they provide automation of a major part of the negotiation process. As a result, the number of face-to-face meetings of negotiators is reduced. However, these approaches mainly focus on auctions. In auctions, contracts are semantics-enabled, i.e. the meaning of various elements or clauses is well determined and commonly known.

A real challenge in the area of electronic negotiation consists in allowing a high number of negotiators to work on real-life contracts, i.e contracts containing both aggregable attributes (e.g. price, quantity, etc.) and non-aggregable attributes (e.g., legal clauses, appendices, quality clauses, etc.). Such electronic negotiations mirror both semantics-enabled and non-semantics-enabled classic contract negotiations. These electronic negotiations take advantage of the features of electronic media — the number of negotiators may be unlimited and they may be arbitrarily dispersed. However, negotiation support systems are required to facilitate mass negotiation processes conducted via the net. Without such support systems, mass, remote negotiations are difficult to imagine. The requirements for such negotiation support systems are the following:

- contracts do not have to be semantics-enabled,
- contracts may contain both aggregable and non-aggregable attributes,
- the number of negotiators is not limited.

Negotiation models for electronic market design and agent-based negotiation models directly address the last requirement only. These models, however, do not address the two first requirements listed above. On the contrary, negotiation support systems address the first requirement, some NSS address also the second requirement. These systems, however, are non-scalable, i.e. they do not address the last requirement.

In the case of multi-attribute contracts with both aggregable attributes and non-aggregable attributes, automated humanless negotiations are not a viable solution. Software agents, as we know them today, cannot operate on non-aggregable attributes, because of the lack of semantics concerning these attributes. Negotiations on such contracts must be conducted by humans. However, humans without a proper support are unable to deal with negotiations involving high number of negotiators. The amount of data generated during such a negotiation process is too high to be understood by humans. Therefore, negotiation support systems facilitating mass negotiation processes conducted via the net, must provide negotiators with synthetic views of the negotiation processes.

In this dissertation, an approach to the above problem is presented, called *multi-facet analysis of e-negotiations*. The proposed solution consists of a multiversion contract model and a multi-facet hierarchical analysis mechanism basing on ultrametrics. The proposed multiversion contract model captures the evolution of the negotiated contract and the relationships existing among offers and counter-offers. The multi-facet hierarchical analysis mechanism provides multiple synthetic views of the negotiation process. A synthetic view is a hierarchical classification, which allows negotiators to have various levels of detail and to focus efficiently on the elements of the highest interest.

The thesis of this dissertation can be formulated as follows:

**Multi-facet analysis of contract versions provides negotiators with synthetic views of the e-negotiation process.**

This dissertation is composed of ten chapters.

In Chapter 2, definitions of basic notions are presented. Both definitions of a contract and of a negotiation are given.

Chapter 3 contains an overview of the current state of the art in the domains related with this dissertation. First, existing approaches to e-negotiations are described. Second, automatic hierarchical classification algorithms are presented. Finally, a review of existing versioning techniques is given.

In Chapter 4, first, the limitations of the current e-negotiation approaches are analyzed in detail. Then, the concept of the multi-facet analysis of e-negotiations approach is presented.

Chapter 5 is devoted to the contract model. First, the concept of the contract model is given. Then, contract version tree is described in detail. Next, multiversion members are described and the contract version management concepts are discussed. Finally, detailed specification of the four operations on multiversion members — reading, updating, deletion, and creation — is provided, followed by a description of composite multiversion members.

Chapter 6 provides description of analysis domains. First, the concept of domain objects is defined. Then, formalization of analysis domain functions (ADF) is given. Finally, detailed specification of the analysis domain language is provided.

Chapter 7 is devoted to the classification of domain objects. First, the concept of parametric analysis is defined. Then, formalization of analysis criteria as ADFs is given. Finally, classification is described: classification structure is formalized, classification generation is discussed, and the threshold operation which creates partitions from classification is presented.

Chapter 8 provides a complete example of the application of multi-facet analysis to e-negotiation. First, a negotiation process is presented. Then, the contract evolution is given. Finally, three examples of analysis are discussed.

The *NeSSy* prototype that proves feasibility of the multi-facet analysis approach is presented in Chapter 9. First, the prototype is described from a user's point of view. Then, architecture of the *NeSSy* prototype is discussed. Finally, presentation of implementations of the three *NeSSy* modules — contract model module, analysis domain module, and classification model module — is provided.

Chapter 10 concludes the dissertation.

CHAPTER 2

# BASIC NOTIONS

## 2.1 Contract

The notion of contract is complex. Many definitions of a "contract" exist, depending on the application domain and/or localization.

The Merriam-Webster's Collegiate Dictionary gives the following definition of a "contract":

> *a binding agreement between two or more persons or parties; especially: one legally enforceable.*

The above definition is the basis for legal definition of contracts. The agreement as the core of most contracts is a set of mutual promises, *clauses* (in legal terminology – "*considerations*"). The promises made by the parties define their rights and obligations. Contracts may be divided into express or implied. An *express* contract is one whose terms of the agreement are openly uttered and avowed at the time of making. *Implied* contracts are such as reason and justice dictates, as well as those the law presumes every man undertakes to perform.

Parties who enter into contracts can rely on them in structuring their business relationships, because contracts are enforceable. The law of contracts is a society's legal mechanism for protecting the expectations that arise from the making of agreements for the future exchange of various types of performance, such as the conveyance of property (tangible and intangible), the performance of services, or the payment of money. Contract laws are defined separately in each country. In most countries of Europe, the Civil Code defines the frame for contract establishment and legal enforcement. In the USA, the Uniform Commercial Code (UCC) applies to commercial transactions (with a few exceptions, as for instance transactions in real estate). The UCC is commented in the Restatements of Contracts of the American Law Institute. In an international context, the Convention on the International Sale of Goods (CIST), created in 1980 under the auspices of the United Nations Commission on International Trade Law (UNCITRAL), is the reference for the sale of goods in transactions in which consumers are not implied.

Definitions of valid contracts in various contract laws are very similar:

- In France, the elements of a valid contract are (1) mutual assent, (2) cause, (3) capacity of the parties, and (4) a lawful object [28],
- In the U.S., the elements are (1) mutual assent, (2) consideration, (3) capacity of the parties, and (4) a lawful object [2].

As a consideration defines a cause, the common denominator of these definitions is:

- mutual assent,
- consideration,
- capacity of the parties,
- a lawful object.

In this dissertation, only express contracts are taken into account, because implied contracts cannot be an object of negotiations. It is assumed that contracts exist in an electronic format. The elements of a contract that are considered are mutual assent and considerations. Capacity of the parties and lawful object requirements are not considered, because they cannot be checked by an information system.

No restrictions are set on clauses. Clauses may concern both measurable considerations (e.g. a price or a delivery date) and unmeasurable considerations (e.g. a warranty or non-disclosure clauses).

## 2.2 Negotiation

According to the Merriam-Webster's Collegiate Dictionary, negotiation is:

> *the process of reaching an agreement by conferring or discussing.*

Various classifications have been proposed in the literature to precise the notion of negotiation, among others, the following ones being orthogonal to one another:

- classical negotiations vs. electronic negotiations,
- distributive vs. integrative negotiations,
- multi-attribute vs. single-attribute negotiations.

Moreover, in this dissertation, *few-persons* negotiations are distinguished from *mass negotiations*.

A negotiation is *electronic* — often shorten to *e-negotiation* — if negotiation tasks (e.g. offer proposal, counter-offers) are performed in a written form via an electronic medium. If contracts are exchange via an electronic medium (e.g. via email or fax), but negotiation tasks are performed in a classical way (i.e. via phone or during face-to-face meeting), the negotiation is not an electronic negotiation. If offer proposals and counter-offers are performed via phone, the negotiation is not electronic, because the used form is not written.

A negotiation is *distributive* if a gain of one negotiator is a loss of another one. Distributive negotiations are also referred as *win-lose* negotiations. A negotiation is *integrative* if the gain is spread among negotiators. Integrative negotiations are also referred as *win-win* negotiations.

In the case of a *single–attribute* negotiation, only one consideration (e.g. price) is negotiated. A common example of single attribute negotiations are auctions. In the case of a *multi-attribute* negotiation, many considerations are to be negotiated. Multi-attribute negotiations are divided into homogeneous and heterogeneous. In *homogeneous* negotiations, all considerations are of the same type (e.g. the price of two items are negotiated), while in *heterogeneous* negotiations, considerations can be of different type (e.g. the price of an item and the warranty associated).

A negotiation is *mass* if the number of negotiators is high, i.e. if it is greater than the maximum number of persons able to have a mutual discussion in one room. This number is subjectively equal to 20.

In this dissertation, considered negotiations are:

- both distributive or integrative,
- multi-attribute, potentially heterogeneous, and
- mass.

CHAPTER 3

# RELATED WORKS

## 3.1 e-Negotiation Approaches

Many works have been done in the research area of e-negotiations [58]. These works can be divided in three groups:

- electronic market design,
- agent-based negotiation models,
- computer support for negotiation.

### 3.1.1 Negotiation Models for Electronic Market Design

With the growth of electronic commerce, the design of electronic market became an important issue. As a consequence, many works have been done to understand what are the mechanisms involved in negotiations. The result of these works are negotiation models based on the game theory [3].

In game-theory models [53, 51], the negotiation process is modeled as a decision making process under uncertainty. Using the utility theory [87], the result of the negotiation is predicted on the basis of each contractor's preferences and the rules of the games, given by the negotiation type (English, Dutch, Vickrey, first-price sealed-bid auctions, etc.) [91].

The most investigated auction model is the *symmetrical independent private values (SIPV)* model. In the SIPV model it is assumed that all contractors are equal (*symmetrical*) and do not depend one another. Each contractor has her/his *private value* for the good being negotiated. Contractors are considered *risk neutral*, i.e. contractors' negotiation strategy is not influenced by the risk an offer or a counter-offer may represent for their gain in the negotiation process. The more interesting characteristic of the SIPV model is that the payoff is the same if the negotiation type is English, Dutch, Vickrey or first-price sealed-bid auctions.

Another interesting model is the *common value* model in which it is assumed that contractors have their own private values and one or more external information, such as the market price for the good or other contractors'opinions. Under these conditions the winner frequently bids more than the good's true value. This phenomenon is referred to as the *winner's curse*. The common value model demonstrates also that contractors should shade their bids as the winner is always the one who provides the most optimistic estimation of the good's value.

### 3.1.2 Agent-based Negotiation Models

Autonomous interacting software agents have been applied to automated negotiations in the area of electronic commerce. One talks about automated negotiations when a negotiation is performed by computers. A negotiation is said to be fully automated if there is no human intervention in the negotiation process.

Topics involved in automated negotiations are the following [39]:

- establishment of **ontologies** defined as agreements among the negotiators about how the negotiation objects are defined and what is the meaning of these definitions;
- **negotiation protocols** defining types of participants, valid actions, negotiation states, and events that cause negotiation states to change;
- **decision-making models** that are used by software agents to achieve their goals.

The above topics became independent areas of research.

#### Ontologies

An ontology is an agreement among the negotiators (humans or software agents) concerning the semantics associated with terms in a given area of knowledge. An ontology can be formalized by providing definitions of:

- classes in the domain of interest,
- relationships among classes, and
- properties (or attributes) of classes.

Ontologies are needed to remove (or, at least, to limit) misunderstanding originating from vague term definitions. A classical example of term ambiguity is the date formatting. In the United States of America, the date 01/02/03 is the 3$^{rd}$ of February 2001, while in Europe, it is the 1$^{st}$ of February 2003. Another example is the definition of the price for many pieces of a given item. Consider the sentence “3 pairs of shoes, for 20 euros”. Does it means that each pair of shoes costs 20 euros or does it means that for 20 euros you can buy 3 pairs of shoes? An ontology may clearly define the meaning of each term of the previous sentence to remove ambiguity. Moreover, ontologies may define relationships between classes. In a ontology, relationships may for instance define synomyms (e.g. shoe and footwear) or generalization (e.g. shoe is a generalization of moccasin). Having such a set of interrelated definitions, an agent can accept an offer for mocassins, while it was searching for shoes.

Two main problems concerning ontologies are: first, the establishment of ontologies for a given domain, and second, reasoning on relationships between terms. Currently, establishment of ontologies is based on *RDF* (Resource Description Framework), event if XML Schemas [76, 26] and UML [20, 68] have also been proposed as candidates to the design of ontologies. The RDF format is a standard established by the W3C Consortium [48, 34]. RDF defines a format to describe resources. The format is based on *{subject, predicate, object}* triplets, called *statements*. A *subject* is a class in the ontology vocabulary. A *predicate* is a property name, while an *object* is a property value in the ontology vocabulary. Relationships between classes are modeled in RDF as special properties:

a refinement of a class is for instance defined by the *subClassOf* property. The RDF Schema Specification [12] defines a set of resources (including classes and properties), and constraints on their relationships.

An object can be an RDF statement, which allows to build complex sentences, in which statements about statements are defined. As an example, consider the sentence "an offer includes a pair of boots, which are a special kind of shoes". This sentence may be split into two simpler sentences: "an offer includes a pair of moccasins" and "moccasins are a special kind of shoes". The overall sentence may be described in RDF as follow:

```
<rdf:Description ID="Shoe">
  <rdf:type resource="http://www.w3.org/2000/01/rdf-schema#Class"/>
</rdf:Description>

<rdf:Description ID="Moccasin">
  <rdfs:subClassOf rdf:resource="#Shoe"/>
</rdf:Description>

<rdf:Description ID="Offer">
  <offer:includeItem rdf:resource="#Moccasin"/>
</rdf:Description>
```

In this example, *Offer* is a subject, *includeItem* is a predicate, and the *reference* to Moccasin is an object. The relationship between *Moccasin* and *Shoe* is defined in the second statement. *Moccasin* is defined as a subclass of *Shoe*.

However, users have expected more from RDF and RDF Schema, including data types, a consistent expression for enumerations, and other facilities. Logicians, some of whom saw RDF as a possible tool for developing long-promised practical AI systems, have deplored the rather poor set of facilities provided by RDF.

In response, the DARPA Agent Markup Language (DAML) [21] sprang from a U.S. government–sponsored effort in August 2000, including DAML-ONT — a simple language for expressing more sophisticated RDF class definitions than permitted by RDFS. The DAML group soon started cooperation with the Ontology Inference Layer (OIL) [54] — another project that provides more sophisticated classification mechanisms based on constructs from frame-based AI. As result, the DAML+OIL language was elaborated to allow expression of far more sophisticated classifications and properties of resources than RDFS. The most recent release of DAML+OIL is that of March 2001 [22], which also includes facilities for data typing based on the type definitions provided in the W3C XML Schema Definition Language. The DAML+OIL specification and its relationship to RDF and RDFS are also available as a series of W3C notes [86, 19, 27, 18]. Currently, the newly commissioned Web Ontology (WebOnt) Working Group [89] is working on a new ontology language, with DAML+OIL as its basis.

Reasoning on relationships between terms, and more generally on ontologies, is currently an active research field. The main stream takes place under the umbrella of the Web Semantics effort [9], supported by the W3C consortium. The Web Semantics aims at allowing software agents to achieve complex tasks on the Internet using RDF files. To enable software agents to deal with data available on the Internet, not only ontologies are needed, but also equivalence relations between ontologies and a logical layer. Equivalence relations allow vocabulary extension, by merging ontologies and mapping definitions from one ontology to other definitions in another ontology. As an example, an ontology may define concepts “shoe” and “Nike shoe”, while another defines concepts “moccasin” and “sport shoe”. An equivalence relation may merge these two ontologies into one ontology in which “moccasin” is a subclass of “shoe” and “Nike shoe” is a synonym of “sport shoe”.

Logic layer allows reasoning on semantic data [8]. The RDF layer provides only assertion and quotations (i.e. assertions about assertions). The logic layer consists of predicate logic (e.g. *not*, *and* operators) and quantification (e.g. *for all(x), f(x)*). The RDF model does define the form of a reasoning engine. The RDF logic mailing list provides a forum for technical discussion concerning the design of logic-based languages for the use in the Web. However, up to date, there is no agreement on standards in this domain.

### Negotiation Protocols

Negotiation protocols define types of participants, valid actions, negotiation states, and events that cause negotiation states to change.

Two areas of research may be distinguished: first, protocol definition frameworks, second, elaboration of proprietary negotiation protocols, aiming at complete solutions to the negotiation process and its context. Not only the negotiation itself is modeled, but also business processes that take place before and after the negotiation process. *SilkRoad* is an example of a project defining a framework for negotiation protocol definition. *Kasbah* and *Cosmos* are two examples of projects defining their own negotiation protocols.

The *SilkRoad* project [75, 78] is an explorative project at IBM Zurich Research Laboratory. The goal of this project is to facilitate the design and implementation of negotiation support systems for specific application domains. The *SilkRoad* project provides the concept of Organization Design Meta-Model (ODMM). The ODMM [77] allows the specification of the structures (roles) and behavior (protocols) of electronic negotiations with design building blocks. The design building blocks represent the functionality of the underlying service components on a conceptual level. The underlying modeling paradigm is the concept of state charts. Roles are defined as the total of all possible events an actor can raise. An agreement scenario represents all necessary roles and the protocol for the complete agreement phase specification. The protocol constitutes all the rules in one scenario, represented by valid states and transitions, which define how agents can achieve an agreement.

With the ODMM, the *SilkRoad* project allows electronic market organizations or negotiation service providers to generate customized negotiation support mechanisms for a broad range of agreement scenarii, such as electronic auctions or agent systems.

*Kasbah* [16] is a project of the MIT Media Lab focused on a Web-based system allowing users creation of autonomous agents that buy and sell good on their behalf. *Kasbah* is a *marketplace*. Users create *buying* and *selling* agents, depending on the role they want to play in the marketplace. Users specify parameters to guide and constrain an agent’s overall behavior. In the *Kasbah* project, parameters concerning a sell are:

- desired date to sell the item by,
- desired price, and

- the lowest acceptable price.

Parameters concerning a buy are:

- desired date to buy the item by,
- desired price, and
- the highest acceptable price.

The *Kasbah* project is a multi-agent system (MAS). It defines a set of messages for communication between buying agents, selling agents and the marketplace. Messages are used for selling and buying announcement, offers and counter-offers. The marketplace is designed to handle any type of agents that support the defined set of messages.

*Cosmos* [52, 33] is a project within the Fourth Framework Programme of the European Union. *Cosmos* stands for Common Open Service Market fOr SMEs. The project goal aims at an Internet-based electronic contracting service that facilitates business transaction processes. The contracting service enables its users to negotiate, sign, and settle electronic contracts across the Internet without leaving a uniform and flexible system environment.

The contracting process is split in three phases, namely, information, negotiation, and execution phase. During the information phase, a broker assembles type-conforming offers, requests and contract templates, matching sellers' offers to buyers' needs. During the negotiation phase, contract proposals are exchanged between parties. Depending on the semantics of such a contract transfer, it may either be considered as a *proposal* or as an *offer*. When all parties accept, the contract is in an *agreed* state and ready for *signing*. After all parties have signed the contract, the electronic contracting service certifies it. Finally, the contract is *executable*. The execution of the contract is defined in the contract itself.

The *Cosmos* project provides a set of software components to support commercial transactions:

- Online Marketplace Service. A number of offers, profiles and electronic contracts are available. They may be combined online to create of an individual contract. In many cases of B2C applications, no further negotiation is required;
- Online Contracting Service. This component facilitates contract negotiation between business partners. Negotiation support is required for B2B contracts with complex product and service specifications;
- Electronic Notary. This module coordinates the signing procedure of electronic contracts. Public key infrastructure services are used to ensure authenticity and non-repudiation of contracts;
- Workflow Management System. A workflow-based control of task execution during the performance phase of the contracts. In service-oriented sectors, this module helps to coordinate fulfillment of contractually agreed obligations.

### Decision-Making Models

In agent-based contract negotiations, continuous market changing is an issue agents have to deal with. Usually, agent behavior is static, as defined by its programmer. Changing markets require agents to behave dynamically to provide the maximum benefits to contractors.

Learning agents may modify their behavior in response to the experience they gain from the interactions with their environment. In recent years, research on software agents able to behave rationally has received a great attention. A theoretical basis of this research is the Belief-Desire-Intention (*BDI*) model [65]. Agents in this model are assumed to have certain *mental attitudes* of believe, desire, and intention, representing information, motivational, and deliberative states of the agents respectively. Within the frame of the BDI model, various kinds of agents can be distinguished: *blindly-committed* agents which deny any changes to its beliefs or desires that would conflict with their commitments; *single-minded* agents which entertain changes to beliefs and drop its commitments accordingly; *open-minded* agents which allow changes in both their beliefs and desires, and drop their commitments if necessary.

Another solution to the problem of dynamically behaving agents is based on modular agent model [82, 83]. In the modular agent model, different parts of agent behavior are modularized. Agent responsibility is to retrieve a behavior appropriated to a given situation and to act as a mediator between the market and the retrieved behavior. To increase possibilities of dynamic composition of agent behavior at the run-time, plug-in mechanism has been proposed. A plug-in mechanism allows agents to delegate a part of their behavior to logical entities, called *plug-ins*. Agents can adapt their behavior to different environments using various plug-ins appropriated to the situation they face.

Finally, an important issue of automated negotiations is the problem of agent behavior in multi-attribute negotiations, where the search space is usually complex and large. Classical agent-based models are not able to deal with such negotiations. Genetic algorithms [5] may be used to solve complex optimization problems of all contractors' satisfaction for all attributes. In [84], genetic algorithms are used to generate negotiation strategies, while in [47] to generate counter-offers. In [73], genetic algorithms are used to provide an empirical evaluation of a range of negotiation strategies and tactics in a number of different types of environment. The aim of this evaluation is to assess operational benefits and drawbacks of a number of negotiation strategies. In [50], the usefulness of agents employing a mix of tactics is demonstrated.

## 3.1.3 Computer Support for Negotiation

Negotiation Support Systems (NSS) are designed to assist negotiators in reaching mutually satisfactory decisions by providing means of communication and analysis of available information. Two aspects of negotiation support systems may be distinguished:

- preparation and evaluation systems, which provide decision support during the negotiation or its preparation;
- process support systems, which provide communication means and, in some cases, mediation mechanisms.

*Preparation and evaluation systems* provide tools to organize information, develop negotiation strategies, and evaluate negotiation offers. To this end, techniques of the Decision Support Systems (DSS) are used. Most common methodologies are: Multiattribute Utility (MAU) representation and Artificial Intelligence-based (AI) approaches. The GENIE project [90] and the NEGOTEX project [63] are two interesting examples of the preparation and evaluation systems.

*Process support systems* provides communication means for negotiators [15]. Process support systems are used in the place of a negotiation table. The format of exchange between parties, and the dynamics and procedures of the negotiation processes are the basis of process support system design. Some process support systems provide computer-assisted mediation. In mediation systems, communications among parties are filtered by a mediator. The mediator prompts the parties toward jointly optimal agreements.

The INSPIRE system [42, 43, 44] is an interesting research project which provides both preparation and evaluation tools, as well as process support. It is based on negotiation analysis. Its precursors were such negotiation support systems as Nego [41] and Negotiation Assistant [64]. INSPIRE uses hybrid conjoint measurement for utility construction and discrete optimization. Conjoint analysis is simple and does not require any specific knowledge or skills from the users. In INSPIRE, the negotiation process goes through three phases: pre-negotiation, negotiation, and post-settlement. In the pre-negotiation phase, the system analyzes the negotiation scenario and evaluates potential offers. Negotiators specify their preferences, which are then used by the system to build the negotiatior's utility function. During the negotiation phases, the system provides negotiators with negotiation history. Basing on the utility functions, the system also provides evaluation of offers submitted by negotiators. When an agreement is reached, the system checks if there is not a better alternative than the reached agreement. If there is such an alternative, the system presents it to the negotiators who continue negotiations.

Another interesting approach to negotiation support systems has been proposed by Schoop and Quix in [69]. It is a model that systematizes the meta-data about the negotiation process. Meta-data are information exchanged during the negotiation about the negotiation process itself. Meta-data can be a remark about some contract part, a refusal of some clauses, pure informational message, etc. Meta-data are an essential part of negotiation and thus they must be taken into account by document management systems that are used to provide contract persistence. A system, named Doc.COM, has been developed on the basis of the proposed model [71, 70].

The speech act theory is the basis of the model proposed in [69]. The speech act theory was published by John Searle in 1969 [72]. According to this theory, the minimal unit of an utterance is a speech act. A speech act consists of the prepositional contents – such as pay two weeks after delivery – and of the illocutionary force, which describe the way the contents were uttered – a promise in the former case. Searle classifies speech acts in five classes, regarding to the illocutionary force. An utterance that represents a fact of the real world, such as technical characteristics of a car or a financial report, is an *assertive*. An utterance that represent a speaker intention to perform an action, e.g. a promise, is a *commissive*. An utterance that represents a speaker attempt to get a listener to perform an action, e.g. a request, is a *directive*. An utterance that represents a speaker feeling or psychological attitude, e.g. an apology, is an *expressive*. An utterance that change the world, such as a prisoner sentencing, is a *declarative*.

The utterance classification can be used to introduce semantics in meta-data (message) exchange during the negotiation process. In [69], a message consists of contents, type (illocutionary force) and timestamp. A message always has a reference to a contract version. A message can also be an answer to a previous message. In Figure 3.1, the message structure is illustrated. The DOC.COM

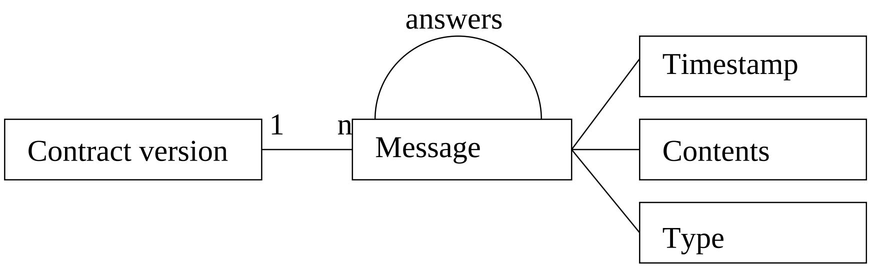


Figure 3.1: Message Structure in the DOC.COM system

model allows monitoring of contractual obligations and traceability of contracts, messages, and their interrelations.

## 3.2 Automatic Hierarchical Classification Algorithms

The goal of any classification is to group items according to their proximity. The concept of proximity can be considered as the similarity between items. The more two items are similar, the closest they are. A well-known classification is the classifications of animals provided by Linné and Lamarck. In these classifications, two animals are in the same class (i.e. mammals) if they share some common characteristics (i.e. they nurse their babies with milk).

### 3.2.1 Partition

More formally, the result of a classification of a set $A$ is a *partition* $P$ of this set. Similarity between items is expressed by a mathematical concept of *equivalence relation*.

**Definition 3.2.1.** *An equivalence relation $\mathcal{R}$ on a set A is a relation that is reflexive ($\forall a \in A, a\mathcal{R}a$), symmetric ($\forall a,b \in A, a\mathcal{R}b \Leftrightarrow b\mathcal{R}a$), and transitive ($\forall a,b,c \in A, a\mathcal{R}b$ and $b\mathcal{R}c \Rightarrow a\mathcal{R}c$).*

Examples of equivalence relations are: “have the same car” or “live in the same country”. An example of an equivalence relation on natural numbers is the “$=$” relation. The keyword of equivalence relations is the word “same”.

Equivalence relations partition the universe into subsets (sometimes called *classes*):

**Definition 3.2.2.** *A* partition *$P$ of a set $A$ is a collection of subsets $\{A_1, \dots A_k\}$ such that any two of them are disjoint (for any $i \neq j$, $A_i \bigcap A_j = \emptyset$) and such that their union is $A$ ($\bigcup_{i\in[1,\dots n]} A_i = A$).*

Every element of $A$ is a member of exactly one subset of the partition $P$. Assume that relation $\mathcal{R}$ is an equivalence relation on the set $A$. Let $[a]$ denote the set $\{b \in A \,|\, a\mathcal{R}b\}$, where $a \in A$.

**Lemma 3.2.1.** *For $a \in A$, the sets $[a]$ constitute a partition of $A$.*

***Proof.*** Let assume that $a,b \in A$ exist, where $[a] \neq [b]$ and $[a]\bigcap[b] \neq \emptyset$. Let $c$ be any element of $[b]-[a]$. Let $d$ be any element of $[a]\bigcap[b]$. First, $a\mathcal{R}d$ because $d \in [a]$. Second, $d\mathcal{R}b$ because $d \in [b]$, and $b\mathcal{R}c$ because $c \in [b]$ and $\mathcal{R}$ is symmetric. By transitivity, $d\mathcal{R}c$. Finally, by transitivity, $a\mathcal{R}c$, which means that $c \in [a]$. The last result is in contradiction with the fact that $c \in [b]-[a]$. Thus, for every $a,b \in A$, either $[a]=[b]$ or $[a]\cap[b]=\emptyset$, which means that, for $a \in A$, the sets $[a]$ constitue a partition of $A$. □

**Lemma 3.2.2.** *Any partition $\{A_1,\ldots,A_k\}$ of $A$ defines an equivalence relation by letting $a\mathcal{R}b$ iff $a$ and $b$ are members of the same $A_i$.*

***Proof.***

Reflexivity. $\forall a \in A, \exists i \in [1,\ldots,k]$ such that $a \in A_i$. Clearly, $a$ and $a$ are members of the same $A_i$, which means that $a\mathcal{R}a$.

Symmetry. If $a\mathcal{R}b$, $\exists i \in [1,\ldots,k]$ such that $a \in A_i$ and $b \in A_i$. Also $b$ and $a$ are members of $A_i$, and therefore $b\mathcal{R}a$.

Transitivity. Assume that $a\mathcal{R}b$ and $b\mathcal{R}c$. Thus, there exist $i,j \in [1,\ldots,k]$ such that $a$ and $b$ are members of the same $A_i$ and $b$ and $c$ are members of the same $A_j$. However, by definition of a partition, $b$ cannot be a member of two different $A_i$ and $A_j$. So $A_i = A_j$ and $a$ and $c$ are members of the same $A_i$, proving $a\mathcal{R}c$. □

By the proof of the two above lemmata, partitions and equivalence relations are exchangeable notions.

### 3.2.2 Indexed Hierarchy

Let assume that $A$ is a finite set. Let $\mathcal{P}(A)$ denote the set of all subsets of $A$.

**Definition 3.2.3.** *A hierarchy $\mathcal{H}$ on $A$ is a subset of $\mathcal{P}(A)$ such that:*

$$A \in \mathcal{H}, \tag{3.1}$$

$$\forall a \in A, \quad \{a\} \in \mathcal{H}, \tag{3.2}$$

$$\forall (h_1,h_2) \in H^2, \quad \begin{cases} & h_1 \cap h_2 = \emptyset, \\ or & h_1 \subset h_2, \\ or & h_2 \subset h_1. \end{cases} \tag{3.3}$$

**Definition 3.2.4.** *An indexed hierarchy on $A$ is a couple $(\mathcal{H}, f)$, where $\mathcal{H}$ is a hierarchy and $f$ is an application from $\mathcal{H}$ to $\mathbb{R}^+$ such that:*

$$\forall a \in A, \quad f(\{a\}) \quad = 0, \tag{3.4}$$

$$\forall (h_1,h_2) \in H^2, h_1 \subset h_2, \quad h_1 \neq h_2, \quad \Rightarrow f(h_1) < f(h_2). \tag{3.5}$$

### 3.2.3 Ultrametrics

**Definition 3.2.5.** *An ultrametric on a set $A$ is an application $\delta$ from $A \times A$ to $\mathbb{R}^+$ such that:*

$$\forall (a,b) \in A^2, \quad \delta(a,b) = 0 \Leftrightarrow a = b, \tag{3.6}$$

$$\forall (a,b) \in A^2, \quad \delta(a,b) = \delta(b,a), \tag{3.7}$$

$$\forall (a,b,c) \in A^3, \quad \delta(a,b) \leq \sup[\delta(a,c), \delta(b,c)] \tag{3.8}$$

**Lemma 3.2.3.** *Consider $\delta$ an ultrametric on a set $A$. The relation $\mathcal{R}_{\delta_0}$ defined by*

$$\forall \delta_0 \in \mathbb{R}^+, a\mathcal{R}_{\delta_0}b \Leftrightarrow \delta(a,b) \leq \delta_0 \tag{3.9}$$

*is an equivalence relation.*

***Proof.***

Reflexivity. $\forall a \in A, \delta(a,a) = 0$. Therefore $\delta(a,a) \leq \delta_0$, which means that $a\mathcal{R}a$.

Symmetry. If $a\mathcal{R}_{\delta_0}b$, then $\delta(a,b) \leq \delta_0$. By symmetry of ultrametrics, $\delta(a,b) = \delta(b,a)$. Therefore $\delta(b,a) \leq \delta_0$, which means that $b\mathcal{R}_{\delta_0}a$.

Transitivity. Assume that $a\mathcal{R}_{\delta_0}b$ and $b\mathcal{R}_{\delta_0}c$. Thus, $\delta(a,b) \leq \delta_0$ and $\delta(b,c) \leq \delta_0$. Because of equation 3.8, $\delta(a,c) \leq \sup[\delta(a,b), \delta(b,c)] \leq \sup[\delta_0, \delta_0] \leq \delta_0$,proving $a\mathcal{R}_{\delta_0}c$. □

As shown in Section 3.2.1, equivalence relations and partitions are exchangeable notions. Therefore, let $\mathcal{P}(\delta_0)$ denote the partition that consists of classes resulting from the equivalent relation $\mathcal{R}_{\delta_0}$.

**Lemma 3.2.4.** *Let A be a finite set. If $\mathcal{H} = \bigcup_{\delta_0 \in \mathbb{R}^+} \mathcal{P}(\delta_0)$, then $\mathcal{H}$ is a hierarchy.*

**Proof.**

Let us show that $A \in \mathcal{H}$. As $A$ is finite, there exists a maximum distance between two of its elements. The partition associated with the maximum distance is the set $A$.

It is obvious that $\forall a \in A, \{a\} \in \mathcal{H}$ because $\{a\} = \mathcal{P}(0)$.

Finally, for the condition 3.3, consider some arbitrary $h_1$ and $h_2 \in \mathcal{H}$. Then, $\exists(\delta_0, \delta_0^{'}) \in \mathbb{R}^+$ such that $h_1 \in \mathcal{P}(\delta_0), h_2 \in \mathcal{P}(\delta_0^{'})$. If $h_1 \cap h_2 = \emptyset$, the condition is observed. Otherwise, consider $a \in h_1 \cap h_2$. $h_1$ can be written as $h_1 = \{b \in A \,|\, \delta(a,b) \leq \delta_0\}$. Respectively, $h_2 = \{b \in A \,|\, \delta(a,b) \leq \delta_0^{'}\}$. If $\delta_0 \leq \delta_0^{'}$, then $h_1 \subset h_2$. Otherwise, $h_2 \subset h_1$, which concludes the proof. □

**Lemma 3.2.5.** *Let $f$ be an application from $\mathcal{H}$ to $\mathbb{R}^+$such that*

$$\forall h \in \mathcal{H}, f(h) = \min\{\delta_0 \,|\, h \in \mathcal{P}(\delta_0)\}.$$

*Then, the couple $(\mathcal{H}, f)$ is an indexed hierarchy.*

**Proof.** $\forall a \in A, \{a\} \in \mathcal{P}(0) \Rightarrow f(\{x\}) = 0$. Moreover, consider some arbitrary $h_1$ and $h_2 \in \mathcal{H}$, such that $h_1 \subset h_2$ and $h_1 \neq h_2$. Consider $a$ being a member of $h_1$. Therefore:

$$\{\delta_0 \,|\, h_1 \in \mathcal{P}(\delta_0)\} \subset \{\delta_0 \,|\, h_2 \in \mathcal{P}(\delta_0)\},$$

and then

$$\min\{\delta_0 \,|\, h_1 \in \mathcal{P}(\delta_0)\} \leq \min\{\delta_0 \,|\, h_2 \in \mathcal{P}(\delta_0)\}.$$

So, $f(h_1) \leq f(h_2)$. Moreover, $h_1 \neq h_2$ and $h_1 \subset h_2$ implies that there exists an element $b$ which is a member of $h_2$ but not a member of $h_1$. Consider $a \in h_1 \cap h_2$. Then,

$$f(h_1) < \delta(a,b) \leq f(h_2),$$

which means that $f(h_1) < f(h_2)$. □

**Lemma 3.2.6.** *Consider an indexed hierarchy $(\mathcal{H}, f)$. An application $\delta$ from $A \times A$ to $\mathbb{R}^+$is an ultrametric if*

$$\forall (a,b) \in A^2, \delta(a,b) = \min_{h \in \mathcal{H}} \left\{ f(h) \,|\, (a,b) \in h^2 \right\}.$$

**Proof.**

Reflexivity. By definition, $\delta(a,a) = 0$ because $f(\{a\}) = 0$. If $\delta(a,b) = 0$, then

$$\exists h \in \mathcal{H} \,\text{such that}\, a \in h, b \in h, f(h) = 0.$$

If $h \neq (\{a\})$ (i.e. $a \neq b$), $f(h) > f(\{a\}) = 0$, which is impossible. Then $a = b$**.**

Symmetry. The demonstration is obvious.

Condition 3.8. Consider $h_1$ such that $\delta(a,b) = f(h_1)$, $h_2$ such that $\delta(a,c) = f(h_2)$, $h_3$ such that $\delta(b,c) = f(h_3)$. As $c \in h_2 \cap h_3$, $h_2 \cap h_3 \neq \emptyset$. Because $\mathcal{H}$ is a hierarchy, let assume that $h_2 \subset h_3$ (if $h_3 \subset h2$, the proof is similar). Then,

$$f(h_2) \leq f(h_3). \tag{3.10}$$

Therefore, $a$, which is a member of $h_2$, is also a member of $h_3$ ($b$ does so). As

$$f(h_1) = \min_{h \in \mathcal{H}} \{ f(h) \,|\, (a,b) \in h^2 \},$$

$h_1 \subset h_3$. Then,

$$f(h_1) \leq f(h_3). \tag{3.11}$$

Because of inequalities 3.10 and 3.11,

$$f(h_1) \leq \sup\left[f(h_2), f(h_3)\right],$$

which proves the condition 3.8. □

From lemmata 3.2.5 and 3.2.6, the notions of an ultrametric and an indexed hierarchy are exchangeable, as illustrated in Figure 3.2. When $\delta_0$ changes, different classes are created and the hierarchical aspect of the classification is visible in the embedment of classes.

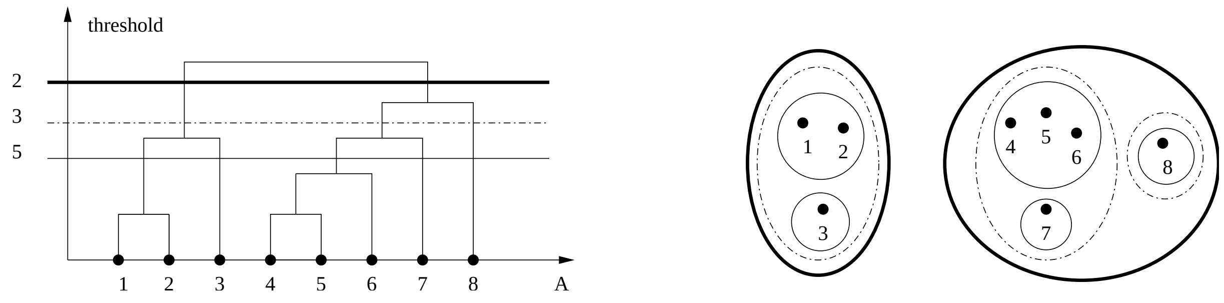


Figure 3.2: Equivalence between indexed hierarchy (on the left side) and ultrametric (on the right side, the classes for various values of $\delta_0$)

## 3.2.4 Path Metrics

**Definition 3.2.6.** *A* path *$p$ between two elements $a$ and $b$ in a set $A$ is a list of elements $p = (a_1, a_2, \ldots a_n)$ such that:*

$$\begin{aligned} \forall i, \quad a_i &\in E, \\ a_1 &= a, \\ a_n &= b. \end{aligned}$$

**Definition 3.2.7.** *Let $d$ be a metric on $A$. A* step $s_d(p)$ *of the path $p$ for the metric $d$ is:*

$$s_d(p) = \sup_{i=1}^{n-1} d(a_i, a_{i+1}).$$

**Definition 3.2.8.** *The* path metric $\delta_d$ *for metric $d$ on a set $A$ is an application from $A \times A$ to $\mathbb{R}^+$ such that:*

$$\delta_d(a,b) = \inf_{p \in \mathbb{P}(a,b)} s_d(p),$$

*where $\mathbb{P}(a,b)$ is the set of all the paths from $a$ to $b$ in $A$.*

**Lemma 3.2.7.** *Each path metric is an ultrametric.*

**Proof.** Conditions 3.6 and 3.7 are obvious. Let prove condition 3.8.

$$\begin{aligned} \delta_d(a,b) &= \inf_{p \in \mathbb{P}(a,b)} s_d(p) \\ &\leq \inf_{p \in \mathbb{P}_c(a,b)} s_d(p), \end{aligned}$$

where $\mathbb{P}_c(a,b)$ is the set of all the paths from $a$ to $b$ in $A$ containing $c$.
By definition of $s_d(p)$,

$$\begin{aligned} \delta_d(a,b) &\leq \inf_{p_1 \in \mathbb{P}(a,c)} \inf_{p_2 \in \mathbb{P}(c,b)} \sup\left[s_d(p_1), s_d(p_2)\right] \\ \delta_d(a,b) &\leq \sup\left[\inf_{p_1 \in \mathbb{P}(a,c)} s_d(p_1), \inf_{p_2 \in \mathbb{P}(c,b)} s_d(p_2)\right] \\ \delta_d(a,b) &\leq \sup\left[\delta_d(a,c), \delta_d(c,b)\right], \end{aligned}$$

which proves condition 3.8. □

A path metric may be derived from every metric. As every path metric is an ultrametric, and every ultrametric is equivalent to an indexed hierarchy,

**Every metric defines an indexed hierarchy.**

### 3.2.5 Neural Networks

#### Basic concepts

Neural networks are computing devices inspired by the way neurons function. Neurons are granular cells performing specialized processes that are the fundamental functional unit of a brain. Neural networks are composed of interconnected computing units, named *nodes*. Each node performs a few simple mathematical operations and communicates the results to the nodes it is connected to.

Nodes of a neural networks are organized into *layers*. Three kinds of nodes are usually distinguished in neural networks: input, output and hidden nodes. Input nodes receive data from the outside of the network. Output nodes send data out of the network. Input and output signals of

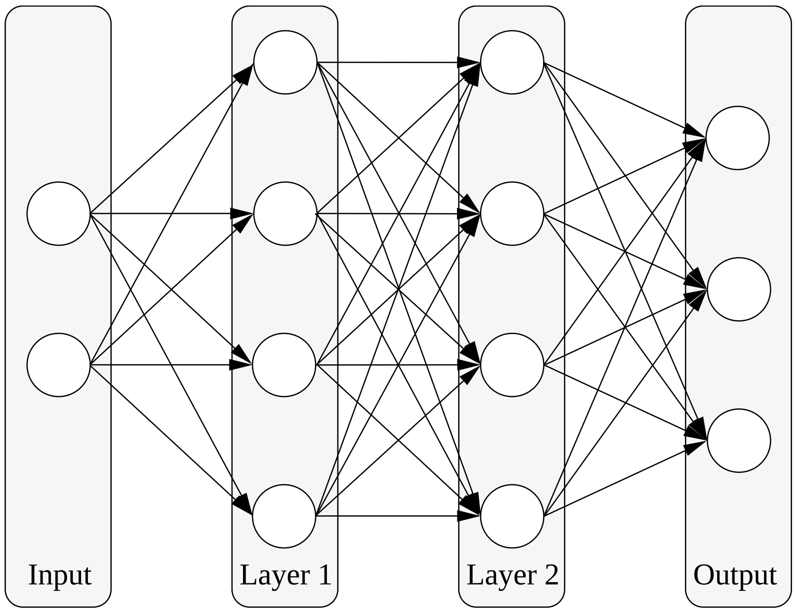


Figure 3.3: Neural network topology

hidden nodes remain inside the network. Each node is connected with every node of the next layer. Figure 3.3 illustrates a neural network topology.

Each node $k$ has an *activation value*, denoted $y_k$, which is equivalent to its output value. Each connection is associated with a *weight* $w_{k'k}$ which determines the effect of the previous node ($k'$) value on the current node value. The effective input $s_k$ of a given node from its external inputs is the weighted sum of the separate outputs from each connected nodes plus an offset term $\theta_k$:

$$s_k(t) = \sum_{k'} w_{k'k}(t).y_{k'}(t) + \theta_k(t)$$

Finally, the output of a given node is the value of a function $F$, called activation function, which takes the previous activation value of the node and the effective input as arguments:

$$y_k(t+1) = F(s_k(t), y_k(t))$$

Usually, threshold functions are used (cf. Figure 3.4): a hard limiting threshold function (a *sgn* function), or a linear or semi-linear function, or a smoothly limiting threshold (such as $F(s,y) = \frac{1}{1+e^{-s}}$).

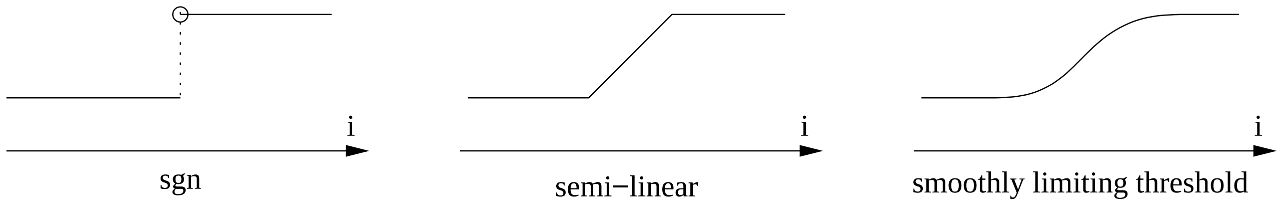


Figure 3.4: Various activation functions

Various methods to set the weights of the connections exist. One way is to set the weights explicitly, using *a priori* knowledge. Another way is to “train” a neural network by feeding it teaching patterns and letting it change its weights according to some learning rules. The learning strategies can be classified in two categories:

- *Supervised learning* or *associative learning* in which a network is trained by providing it with input and matching output patterns. These input-output pairs can be provided by an external teacher, or by the system which contains the network (*self-supervised*);
- *Unsupervised learning* or *self-organization* in which an (output) node is trained to respond to clusters of pattern within the input. The system is supposed to discover statistical features of notable significance of the input population. Unlike the supervised learning, there is no *a priori* set of categories into which the patterns are to be classified; the system must develop its own representation of the input stimuli.

Among various neural network architectures, *competitive learning* is a learning procedure that divides a set of input patterns into clusters that are inherent to the input data. The architecture of competitive learning networks consists of all inputs nodes connected directly to all output nodes (cf. Figure 3.5).

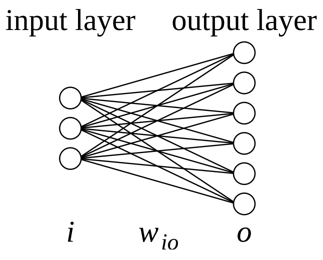


Figure 3.5: Competitive learning network

When an input data $x$ is presented to the input layer, only one output node (the *winner*) is activated. The winner is the the node whose weight vector is the closest to the input data. The weight vector $w_k$ can be defined as follow:

$$\sum_i w_{ik} x_i = w_k x,$$

where $x_i$ are the components of the input data vector $x$. The Euclidian distance measure may be used to determine which node is the closest to the input data. When the winner has been selected, its weigth is updated in order to implement a shift toward the input:

$$w_k(t+1) = w_k(t) + \gamma(x(t) - w_k(t)), \text{ where} \gamma \in \mathbb{R}^+. \tag{3.12}$$

Consequently, weight vectors are rotated towards those areas where many inputs appear: the clusters in the input.

Kohonen [45, 46] provides the ordering concept to competitive learning networks. In the Kohonen networks, named *Self-Organizing Maps (SOM)*, the output nodes are *ordered*, often in a two-dimensional grid or array. The ordering determines which output nodes are neighbors. Input data which are near to each other in the input spaces are mapped on output nodes which are also near to each other. The mapping is said to be *topology preserving.*

At time *t*, an input data *x* is presented to the network. The winner *k* is determined. Next, the weights to this winning node as well as its neighbours are adapted using the learning rule:

$$\forall o \in S, w_o(t+1) = w_o(t) + \gamma.g(o,k)\,(x(t) - w_o(t))$$

where *S* is the output layer and $g(o,k)$ is a decreasing function of the grid-distance between nodes *o* and *k*, such that $g(k,k) = 1$. Due to this collective learning scheme, input signals which are near to each other are mapped on neighbouring neurons.

SOM have been extended by Fritzke [29, 30] to deal with the main disadvantage of SOM, i.e. the fact that the network structure has to be specified in advance. Fritzke proposed a new neural network architecture: the *Growing Cell Structures*. The growing cell structures are a self-organizing network which is able to automatically find a problem specific network structure through a growth process. Basic building blocks are *k*- dimensional *hypertetrahedrons*: lines for $k = 1$, triangles for $k = 2$, tetrahedrons for $k = 3$, etc. Vertices of the hypertetrahedrons are the neurons. Edges of the hypertetrahedrons denot neighborhood relations. In this architecture, only the winner and its *direct* neighbours are updated.

To determine the positions where new neurons should be inserted, the concept of *resource* is introduced. Every neuron has a local resource variable and new neurons are always inserted near the neuron with the highest resource value. New neurons get part of the resources of their neighbors so that the resource is evenly distributed amoung all neurons. Every input data causes an increase of the resource variable of the winner. The value added to the winner resource value may be the number of input data received by the winner or the summed quantization error caused by the winner.

After a constant number of adaptation steps, a new neuron is added. The node with the highest resource value is determined and the edge connecting it to the neighbour with the most different weight vector is split by inserting the new neuron. Additional edges are added to rebuild the *k*-hypertetrahedron based structure.

### The Self-Organising Tree Algorithm

SOM and the growing cell structures have been integrated by Dopazo and Carazo [23] to build a self-organizing neural network that clusters sets of align sequences in genetics. An output of the Self-Organizing Tree Algorithm (SOTA) [24] has a binary tree topology. A series of nodes, arranged in a binary tree, are adapted to the intrinsic characteristics of the input data set. The output space can grow to fit as much as possible the variability of the input space, as in the growing cell structures. In SOTA, the growing of the output nodes can be stopped at the desired taxonomic level, or alternatively, they can grow until a complete classification of every input data is reached.

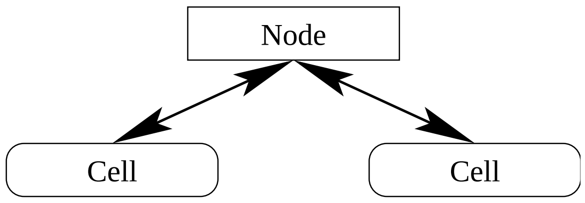


Figure 3.6: SOTA initial network topology

The initial network topology consists of two external elements, denoted *cells*, connected by an internal element, denoted *node* (cf. Figure 3.6). Each cell and each node is a vector with the same size as the input data. The initial value of these vectors is not important and can be set arbitrarily. Contrary to growing cell structures, only cells, but not nodes, are compared to the input data. Thus, the network is trained only through its terminal neurons. Tree generation is performed by cell generation, a given cell generating two descendant cells. The series of operations performed until cell generation is called a *cycle*. During a cycle, cells and nodes are adapted by the input data.

The *adaptation process* consists of a series of *epoch*. During each epoch, the complete input data set is presented to the network. Each input data presentation causes:

1. finding the best matching cell, and
2. updating of this cell and its neighborhood.

Cells are updated according to equation 3.12, known from SOM. The definition of the neighborhood is different than in SOM. If the sister cell of the winning cell has no descendants (cf. Figure 3.7, on the right side, cells *B* and *C*), the neighborhood includes the winning cell, the ancestor node and the sister cell. Three values of $\gamma$ (cf. equation 3.12), denoted $\gamma_w$, $\gamma_a$, and $\gamma_s$, are used for the winner cell, the ancestor node and the sister cell, respectively. The strength of the updating decreases as the neuron is further in the neighborhood, $\gamma_w > \gamma_a > \gamma_s$. If the sister cell of the winning cell has some descendants (cf. Figure 3.7, on the left side, cell *A*), the neighborhood includes only the winning cell.

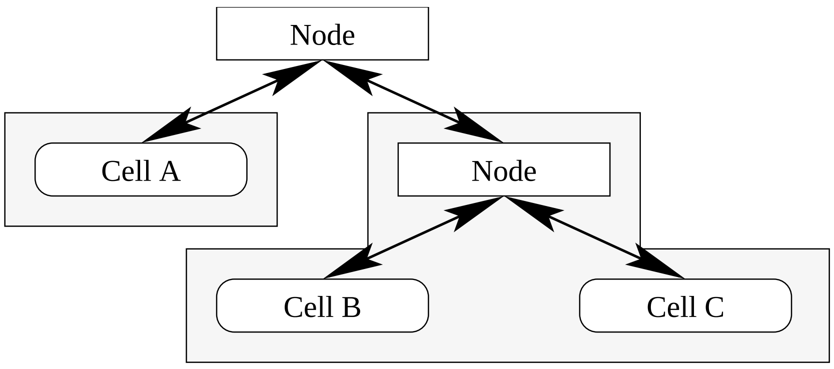


Figure 3.7: Neighborhood in SOTA

When all input data have been presented, the heterogeneity under each cell is computed by its resource *R*. The resource is defined as the mean value of the distances among a cell and the input data associated with it. The resource is used to calculate the total error $\varepsilon$ of the network. The error is defined as the sum of the resource value of all the cells. It measures how close the input data are to their corresponding winning cell after an epoch. The error is used to determine whether a cycle is finished and cells have to be generated. If the relative increase of the error falls below a given threshold $E$, i.e.

$$\left| \frac{\varepsilon_t - \varepsilon_{t-1}}{\varepsilon_{t-1}} \right| < E,$$

then the cycle ends and the cell with the highest resource gives rise to two new descendant cells and becomes a node. The values of the two new cells are identical to the node that generated them.

This process of successive cycles of generation of descendant cells can last until each cell has one single input data assigned, producing a complete classification of the input data set. The expansion can also be stopped at the desired level of heterogeneity in the cells, producing a classification at a higher hierarchical level.

## 3.3 Versioning Techniques

### 3.3.1 Software Configuration Management Systems

The Revision Control System (RCS), the Source Code Control System (SCCS), and the Concurrent Versions System (CVS) have been designed to meet the requirements of designers, developers, and programmers of software working in geographically dispersed teams. When it comes to work on the design of documents, test cases, specifications, and source code that comprise a software project, individual team members need to work on pieces in isolation, then integrate the modified pieces with the modifications of their co-workers, without clobbering anyone else's changes. Changes need to be tracked so that errors and exploratory design changes can be undone. Tracking constitutes a group memory of how files have changed over time – valuable for later reconstruction of detailed design rationales. Released and stable configurations of a software project are tracked so they can be regenerated quickly, and so that bug fixes can be made to the appropriate release.

Two version management schemes used in software configuration management are the Revision Control System (RCS) and the Source Code Control System (SCCS). RCS [81] uses an *edit-based* technique for representing multiple versions of an evolving document. RCS stores the most current version verbatim, while previous revisions are represented via *reverse editing scripts*. Reverse editing scripts describe how to go back in the object's development history. These scripts, also called *deltas* in the RCS terminology, are computed by the *diff* program [36]. Deltas reduce the space consumed, by the cost of an extra processing needed to access to old versions, necessary to run reverse editing script.

SCCS [66], a precursor of RCS, also treats a document as a sequence of lines of text. SCCS models the document history by interleaved deltas. A file containing interleaved deltas is partitioned into blocks of lines. Each block has a header that specifies to which revision(s) the block belongs. The blocks are sorted out in such a way that a single pass over the file can pick up all the lines belonging to a given revision. Thus, the regeneration time for all revisions is the same: all headers must be inspected, and the associated blocks are either copied or skipped.

The Concurrent Versions System (CVS) is a front end to the RCS revision control system. CVS [7] extends the notion of revision control from a collection of files in a single directory to a hierarchical collection of directories, each containing revision controlled files. In the CVS system, directories and files can be combined together in many ways to form a software release. CVS provides the functions necessary to manage software releases and to control the concurrent editing of source files by multiple software developers. To deal with to the inconstancy issue, the *tag* concept is proposed in CVS. A tag is a symbolic name given to a set of files. Tags are used to distinguish various releases of a given program.

### 3.3.2 Web Versioning

The WWW Distributed Authoring and Versioning working group (WebDAV) has defined in the Request For Comments 2291 [74] requirements for a distributed authoring and versioning protocol for the World Wide Web (WWW). The proposed protocol is a set of extensions to the HTTP standard.

It allows for remote loading, editing and publishing of various media types on the WWW. WebDAV not only addresses new methods, headers, and MIME types, but also any required changes to the existing HTTP methods and headers.

One of the goals of the WebDAV working group is to include versioning mechanisms in the proposed protocol. The rationale for versioning mechanism is that "it provides infrastructure for efficient and controlled management of large evolving web sites. (...) It allows parallel development and update of single resources. (...) It provides a framework for coordinating changes to resources. (..) It allows browsing through past and alternative versions of a resource. (...) It allows explicit semantic representation of single resources with multiple states."

RFC 2291 comprises the following definitions:

**Resource** A network data object or service that can be identified by a URI.

**Collection** A resource that contains other resources, either directly or by reference.

**Reservation** A declaration that one intends to edit a resource.

**Lock** A mechanism for preventing anyone other than the owner of the lock from accessing a resource.

**Version Graph** A directed acyclic graph with resources as its nodes, where each node is derived from its predecessor(s).

In the proposed versioning model, a "derived from" relationship exists among each version and its predecessor(s). It is possible to derive several different versions from a single version (*branching*), and to derive a single version from several versions (*merging*). Consequently, a collection of related versions forms a directed acyclic graph, named a *version graph*. Each node of this graph is a *version*. The arcs of the graph capture the "derived from" relationships.

The WebDAV working group identified the following ordered list of logical operations involved in version creation in version management systems:

- Reserve existing version;
- Lock existing version;
- Retrieve existing version;
- Request or suggest identifier for a new version;
- Write new version;
- Release lock;
- Release reservation.

The RFC 2291 states that “the WebDAV extensions must find some balance between allowing versioning servers to adopt whatever policies they wish with regard to these operations and enforcing enough uniformity to keep client implementation simple. (...) Version policies include decisions on the shape of version histories (linear or branched), the granularity of change tracking, locking requirements made by a server, etc. The protocol should clearly identify the policies that it dictates and the policies that are left up to versioning system implementors or administrators.”

The Request for Comment 2518 [32] specifies a set of methods, headers, and content-types ancillary to HTTP/1.1 for the management of resource properties, creation and management of resource collections, namespace manipulation, and resource locking (collision avoidance). While WebDAV’s remote-authoring features are useful for performing remote collaborative authoring, they highlight the need for versioning support to preserve the history of work. The work on Delta-V is intended to fill this role, adding versioning support to WebDAV.

Work on Delta-V is ongoing [17], so details of the protocol may change as the standardization work continues, but there is increasing convergence on its features and benefits. To the base provided by HTTP and WebDAV, DeltaV adds eleven additional methods. Versioning capability is provided by the methods `VERSION-CONTROL`, `CHECKIN`, `CHECKOUT`, `UNCHECKOUT`, `REPORT`, `LABEL`, `UPDATE`, `MERGE`, `MKACTIVITY`, `MKWORKSPACE`, and `BASELINE-CONTROL`. An unversioned resource is put under version control with `VERSION-CONTROL`. While under version control, a typical editing process begins with `CHECKOUT`, involves one or more writes (`PUT`s) to the resource, and ends with a `CHECKIN`. An editing session can be aborted using `UNCHECKOUT`. The version history of a resource can be retrieved using `REPORT`. Unique human-readable names can be associated with specific versions using `LABEL`. The default visible revision can be set using `UPDATE`. Two separate branches in a version history can be brought together using `MERGE`. The `MKACTIVITY` method creates new activities that represent sets of changes and/or branches.

The Delta-V protocol has several unique features. Delta-V assumes that most editing will take place directly on Web resources, which differs from CVS in that there is no replica on the local disk of the client (however, Delta-V provides also this feature). Isolation from the changes of other team members is ensured by “workspaces”, which provide each collaborator with his or her own view on the resources being edited. Workspaces are created using the `MKWORKSPACE` method. Unlike the replicas on the local disk of the client, that provide isolation in CVS, workspaces isolate collaborators when they work on a remote Web server. Overwrite conflicts are avoided because a resource can be checked out by multiple people simultaneously, and each check out creates a separate working resource. Each collaborator actively working on a resource has a separate virtual working area, identified by his or her workspace, and modifications are made first in a workspace, then merged with the changes of other collaborators.

Delta-V provides versioning of collections. When a collection is versioned, collections and their contents follow the check-out/edit/check-in model. When a collection is checked in, its membership is frozen, and cannot be changed until the collection is checked out again. Making a new file or deleting an existing file requires the parent collection to be checked out. When all collections in a project are versioned, it is possible to permanently store the membership of each collection for each time moment, thus making configuration management support possible. Once both collections and their contents are versioned, it is possible to explicitly pick a single revision of each collection and file (often the most recent revision), creating a snapshot of the entire project.

A consistent snapshot of a set of resources is a baseline. It is useful for recording the state of a software system before major ship dates. The `BASELINE-CONTROL` method is used to place both versioned and unversioned collections under baseline control, thus creating a version-controlled configuration. Checking-out then checking-in a version-controlled configuration creates a new baseline (i.e. a new version of the version-controlled configuration).

Delta-V also divides its functionality into two layers: a simple versioning layer, and a more complex software configuration management (SCM) layer. Since authoring clients (word processors, text editors, spreadsheets, etc.) typically work on a single file at a time, they are only expected to use the basic versioning layer to support a check out/edit/check in style of work. The typical authoring client is not expected to provide a user interface for operations like creating and reverting configurations, since a configuration spans an entire project, far greater than their single-file editing scope. A separate SCM control panel application makes use of the features in the SCM layer. This control panel operates at a collection and project level, providing the capability to create a project configuration or revert to a previous configuration. It complements the single-file focus of the authoring tools with project-wide capabilities. A full-featured programming environment is a third class of Delta-V application, one that uses both the versioning and configuration capabilities of Delta-V, providing support for editing individual source-code files, as well as project-level SCM support.

### 3.3.3 The Multiversion Database Approach

The multiversion database approach [13] provides a versioning model to object-oriented databases.

**Definition 3.3.1.** *An entity is a persistent information, i.e. the lifetime of an entity is longer than the process that has created it.*

A multiversion database is a set of logically independent database versions. A database version consists of a database version identifier and a set of entity versions, one version per multiversion entity existing in the multiversion database. Every entity version must exist inside a database version.

The creation of a new database version is done by *derivation*. The derivation operation creates a logical copy, named the *child database version*, of an existing database version, named the *parent database version*. Once a child database version has been created, both the parent and the child database versions may evolve autonomously. The set of database versions is organized as a tree, called the *derivation tree*. Database versions identifiers are constructed in such a way that all database ancestors of a given database version may be retrieved from the database version identifier. If a database version is the $n^{th}$ child of database version identified by $p$, the identifier of the child database version is $p.n$. The identifier of the root version is 0. Database version identifiers are called *database version stamps*.

In Figure 3.8, database version stamps and the derivation tree are given for an arbitrary multiversion database. The root version, labeled 0, is the database parent version of three database versions: 0.1, 0.2 and 0.3. Database version 0.1 has one child version 0.1.1, which has itself one database child version 0.1.1.1. Database version 0.2 has no child version. Database version 0.3 is the parent of two versions 0.3.1 and 0.3.2. Database version 0.3.2 has one child version 0.3.2.1.

Each multiversion entity consists of an identifier $e$ and a set of entity versions. The relationship between entity versions and database versions is captured by an *association table*. An association table consists of pairs $(entity\,version, \{database\,version\})$. If a given entity version $v_1$ is shared between two database versions $db_1$ and $db_2$, the association table contains a pair $(v_1, [db_1, db_2])$. Implicit entity version sharing is provided by a rule that states that:

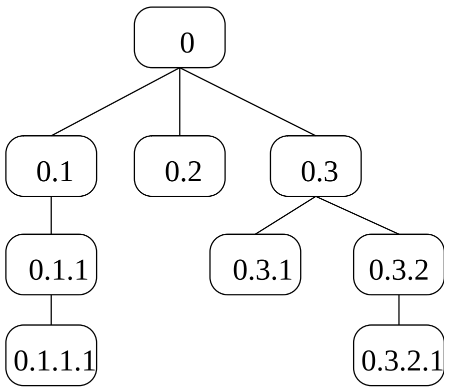


Figure 3.8: Derivation tree

**Implicit Sharing** For a multiversion entity, if no entity version is explicitly associated with database version $db_i$, then the database version $db_i$ shares the entity version with its parent database version.

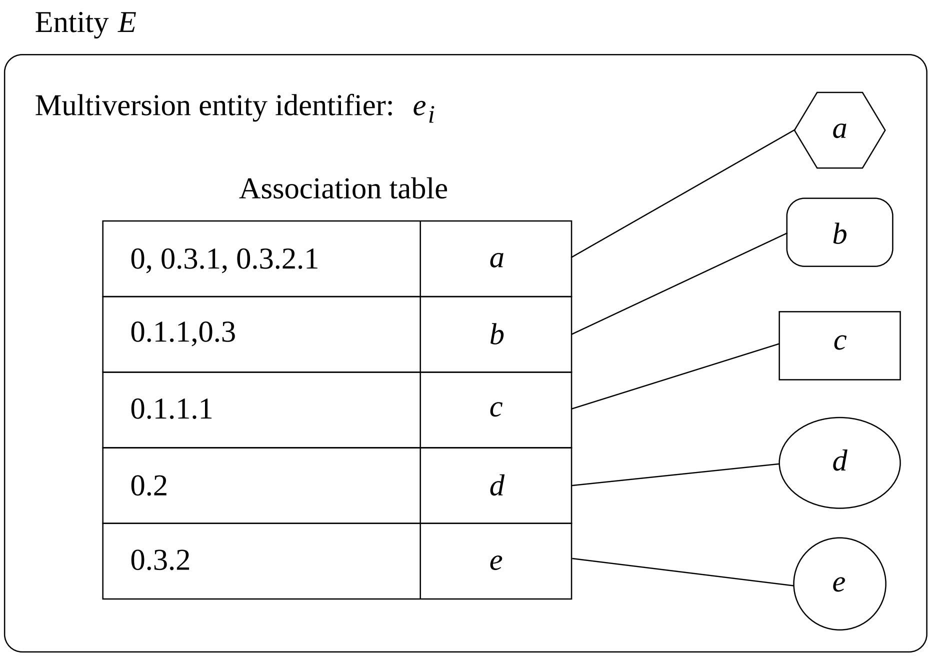


Figure 3.9: A multiversion entity

In Figure 3.9, the internal structure of a multiversion entity $E$ is illustrated. The derivation tree is assumed to be the one presented in Figure 3.8. The multiversion entity $E$ is identified by $e_i$. Five entity versions — $a$, $b$, $c$, $d$, and $e$ — of $E$ exist. In the database version 0, the value of $E$ is $a$. The value of $E$ in the database version 0.1 is not implicitly defined in the association table. According to the implicit sharing rule, it is shared with its parent version, i.e. the value of $E$ in the database version 0.1 equals to $a$. The value of $E$ in the database version 0.1.1 (respectively 0.1.1.1) is equal to $b$ (respectively $c$). The value of $E$ in the database version 0.2 is explicitly defined and equals to $d$. The value of $E$ in the database version 0.3 is $b$. The value of $E$ in database versions 0.3.1 and 0.3.2.1 is equal to $a$. The value of $E$ in the database version 0.3.2 is equal to $e$.

Two kinds of entity version can be distinguished with regard to sharing. *Unshared entity versions* belong to only one database version. *Shared entity versions* belong to several database versions. The above example illustrates two sharing modes. Implicit sharing is used for database version 0.1. Explicit sharing is used for entity versions *a* and *b*.

A modification of an entity version *v* of multiversion entity *E* in database version *db* does not modify other entity versions of *E*, because various database versions are independent. Two cases may happen, depending on the fact that *v* is shared or not. Version *v* is not shared if only one database version stamp is explicitly associated with it, and if all the children of this database version are explicitly associated with other entity versions.

If *v* is not shared, it can be updated without modification of the association table of *E*. If *v* is shared, a new entity version $v_{new}$ is created that is associated with *db* in the association table. The row of the association table concerning *v* is modified so that it consists of:

- all database version stamps, except *db*, if *db* was explicitly associated with *v*,
- the database version stamps of all the children of *db*, which are not explicitly associated with other entity versions.

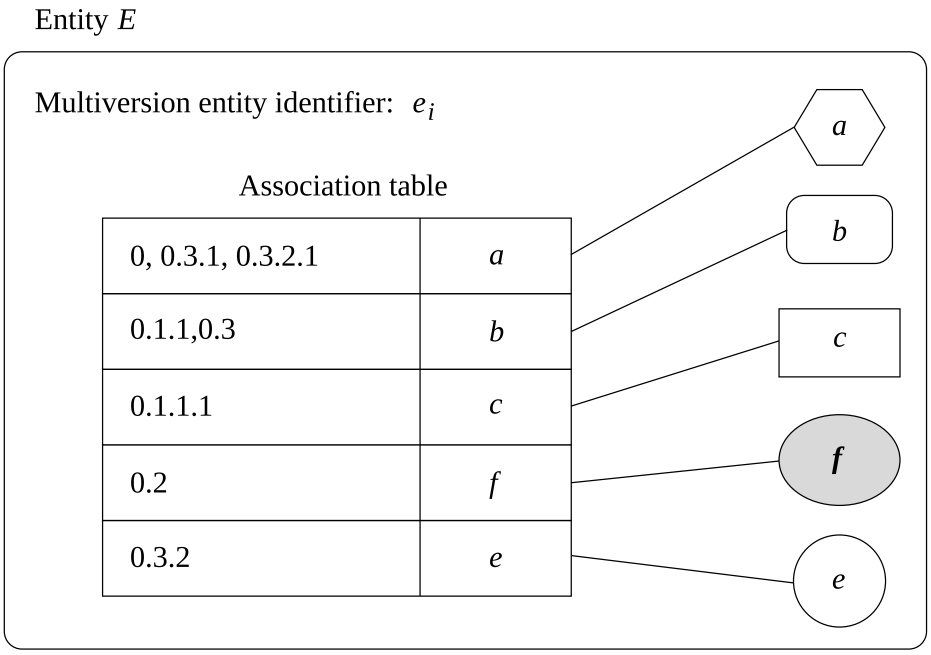
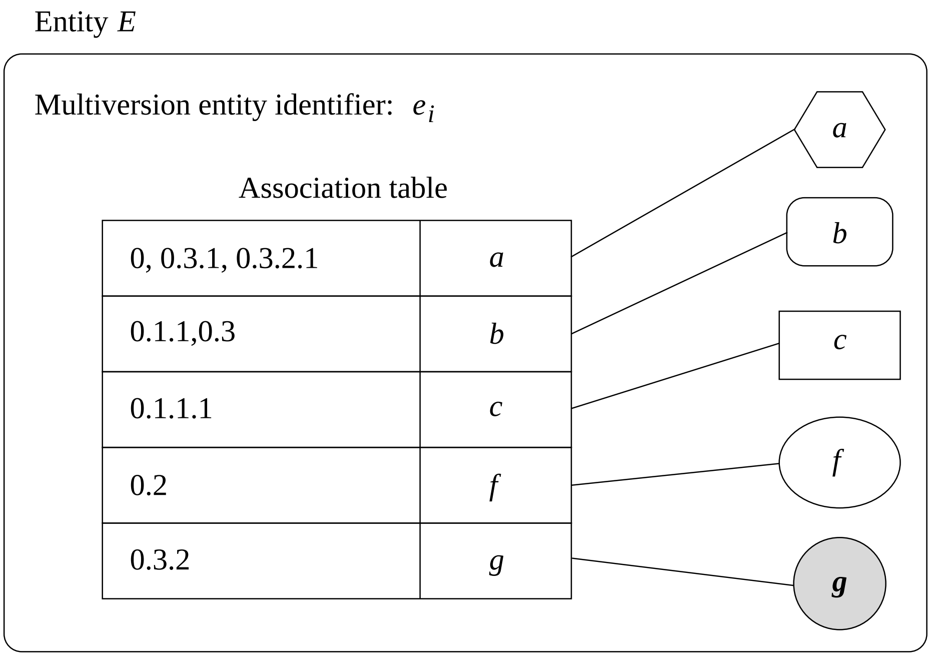


Figure 3.10: Update of an unshared entity version in a database version without children

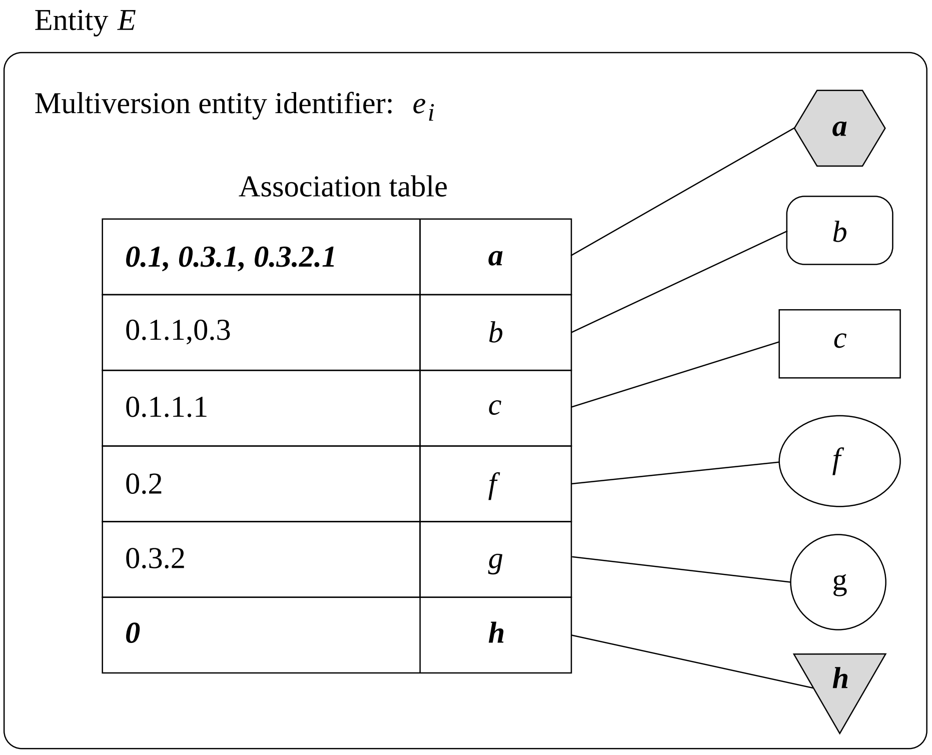
Figure 3.10 illustrates the modification of an unshared entity version. Consider the modification of *E* in database version 0.2. The new entity version is *f*. In database version 0.2, the entity version *a* is not shared because only database version 0.2 is associated with it and no children of database version 0.2 exists.

Figure 3.11 illustrates the modification of another unshared entity version. Consider the modification of *E* in database version 0.3.2. The new entity version is *g*. In database version 0.3.2, the entity version *e* is not shared because only database version 0.3.2 is associated with it and the only database version child 0.3.2.1 is explicitly associated with another entity version *a*.



| | |
|---|---|
| 0, 0.3.1, 0.3.2.1 | *a* |
| 0.1.1,0.3 | *b* |
| 0.1.1.1 | *c* |
| 0.2 | *f* |
| 0.3.2 | *g* |

Figure 3.11: Update of an unshared entity version in a database version with children



| | |
|---|---|
| ***0.1, 0.3.1, 0.3.2.1*** | ***a*** |
| 0.1.1,0.3 | *b* |
| 0.1.1.1 | *c* |
| 0.2 | *f* |
| 0.3.2 | *g* |
| ***0*** | ***h*** |

Figure 3.12: Update of a shared entity version

Figure 3.12 illustrates the modification of a shared entity version. Consider the modification of $E$ in database version 0. The new entity version is $h$. In database version 0, the entity version $a$ is shared because database versions 0, 0.3.1 and 0.3.2.1 are associated with it. Moreover the database version child 0.1 is implicitly associated with entity version $a$. The new entity version $h$ is associated with 0. The old entity version *a* is now associated with database versions 0.3.1 and 0.3.2.1 as before update. Value *a* is now also associated explicitly with database version 0.1.

A special entity version value, named *null*, refers to a non-existent entity version. Thus, if entity version of multiversion *E* is set to *null* in database version *db,* entity *E* does not occur in *db*. The removal of an entity version in a given database version equals to setting its value to *null.*

The addition of a new entity version *v* in database version *db* means the addition of a new multiversion entity *E* in the database. The *null* value is associated with the database root version 0 and *v* is associated with *db.*

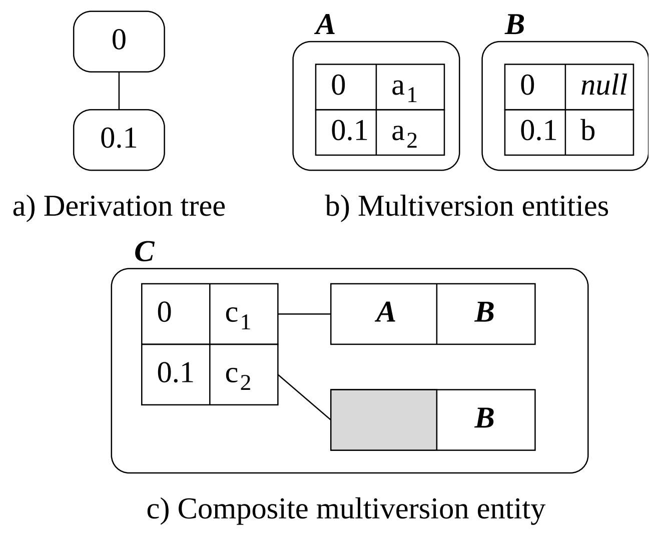


Figure 3.13: Multiversion composite entity

Multiversion composite entities refers to multiversion entities. In Figure 3.13, a case of a multiversion database that consists of three multiversion entities *A*, *B*, and *C*, is illustrated. Multiversion entity *A* exists in two versions: $a_1$ in database version 0 and $a_2$ in database version 0.1. Multiversion entity *B* exists in one version: *b* in database version 0.1. Multiversion entity *C* exists in two versions: $c_1$ in database version 0 and $c_2$ in database version 0.1. Entity version $c_1$ refers to $a_1$. Entity version $c_2$ refers to *b*. The references are indirect, i.e. entity versions of *C* refer to multiversion entities, and not to their entity versions. Access to referenced multiversion entities is needed to reconstruct the composite entity version.

The multiversion database approach has been further extended to provide classical features of database management systems: indexes and large object support [49], query language [1], manipulation language [31], views [6], integrity constraints [25], and concurrency control [14].

CHAPTER 4

# CONCEPTS

## 4.1 Evaluation of the Existing Methods

As mentioned in Section 2, mass negotiations characteristics are:

- A high number of negotiators (a few hundreds);
- A complex contract with both measurable considerations (e.g. price or delivery date) and unmeasurable considerations (e.g. warranty or non-disclosure clauses). It is assumed that a contract is an unstructured document.

Electronic negotiation systems dealing with mass negotiations have to address the two following issues:

- A high amount of data resulting from the negotiation process;
- The lack of semantics due to an unstructured form of the contract.

The existing methods for electronic negotiations do not address at least one of the above issues [56].

### 4.1.1 Negotiation Models for Electronic Market Design

Although these models are quite elegant and based on strong mathematical foundations, they have many limitations. First, most of these models are dealing with single attribute auctions, and cannot be applied to multi-attribute contracts. The object of the negotiation is, for instance, the price of a stock option. Thus, these models are not adapted to the complex contract case.

Another important limitation is that, according to various experiments [40, 67], the existing models provide a poor representation of real negotiation cases. According to several empirical studies conducted by Balakrishnan [4], it seems that fundamental concepts in game theory are not valid for real-world environments.

The negotiation models for electronic market design do not address the high amount of data resulting from the negotiation process. However, these models are not explicitly in opposition with the requirements of high scalability and highly concurrent environment.

### 4.1.2 Agent-based Negotiation Models

The promises of agent-based negotiation models are very attractive: a negotiator only has to specify her/his goals and the negotiation process is performed by the agent. Negotiation costs are highly reduced and the result is often Pareto-optimal.

In the case of mass negotiations, agent-based negotiation models are well-adapted to the high amount of data available. As agent processing capacities are higher than human ones, agent-based models seem to be a good solution to mass negotiations.

The lack of semantics due to the unstructured form of the contract is the main obstacle to the application of agent-based negotiation models in mass negotiations. Agents are currently not able to extract the meaning of a given clause of a contract. Moreover, the lack of ontologies is still a major issue to contract negotiations when the contract is not explicitly associated with some semantics.

### 4.1.3 Computer Support for Negotiation

Systems of computer support for negotiation were used mostly in laboratories. This can be explained by two factors. First, they were mostly elaborated in the 1980s when computer networks were not widely used. Only universities, armies, and research centers could afford the investment in expensive computers. Second, the potential users of theses systems were not used to use computer systems. As a consequence, theses systems are currently mainly used for:

- training purposes and
- theoretical research on negotiations.

The main advantage of the negotiation support systems (NSS) is the fact that negotiators have a direct control on negotiations. NSS are only tools that aim at helping negotiators. On the contrary to agent-based negotiation models, where intelligence is within agents, NSS let humans be the "thinking unit" of the negotiation process. As a consequence, the lack of semantics is not a problem as humans may easily interpret unstructured documents and provide them with some meaning.

The main drawback of NSS is the fact that no support for a high amount of data is provided. NSS are not designed to present synthetic views of the negotiation process. Process support systems (cf. section 3.1.3) do not provide tools to analyze the negotiation process. Preparation and evaluation systems are focusing on data: information organization, negotiation strategies development, and offers evaluation. However, during mass negotiations, in all these fields, data must first be synthesized to be usable. The Doc.COM model is focusing on meta-data. In mass negotiations, the amount of meta-data is even higher than the amount of data. Moreover, the meta-data, at least those proposed by Schoop and Quix, do not capture facts concerning the negotiations, but intentions. When the amount of information is high, facts must be analyzed first. Meta-data are only additive information and cannot replace data.

We proposed a new approach to mass e-negotiation which deals with the issue of analysis of data generated during negotiation processes [60]. The proposed approach is in the NSS research area, because it provides negotiators with tools for negotiation strategies development and negotiation process analysis.

## 4.2 Multiversion Contract Model

The proposed multiversion contract model is based on the multiversion database approach presented in Section 3.3.3.

### 4.2.1 Modeling Negotiation History

The first problem arising in mass negotiations is the need to represent negotiation history. Storing conversation among negotiators is not a right solution, because of natural ambiguities and difficulties in intention interpretations. However, one may notice that a contract, which is under negotiation, is usually modified many times until the final agreement. The various versions of the contract reflect various propositions made by negotiators. Thus, a partially ordered set of contract versions represents the multi-thread history of negotiations. We propose to use the partially ordered set of contract versions as a basis of a negotiation support system.

In the proposed contract model, it is assumed that the contract to be negotiated is a multiversion contract. An initial contract to be negotiated is created before the negotiation process starts. The creator of the initial contract should be one of the negotiators. The initial contract is the starting point in the negotiation. Each negotiator can create and then modify her/his own copy of the initial contract. At this point, many versions of the contract exist. Further, each negotiator may modify her/his version of the contract, creating her/his own set of contract versions.

### 4.2.2 Version Tree

A contract consists of many versions. The various versions of a contract are hierarchically organized.

The tree root is the initial contract version. When a new negotiator joins the negotiation, she/he must derive a version of an existing contract version. Each contract version must be identified by a unique identifier.

The following example shows how the version tree is organized and illustrates the derivation mechanism. Consider a negotiation with three negotiators, denoted $n_i$, where $i \in [1,2,3]$. The initial contract version is denoted $C_{initial}$. When the negotiation process is about to start the version tree is the one shown in Figure 4.1. The only existing version is the initial contract version. In the version tree, only the node labeled 0 exists, corresponding to the initial contract version .

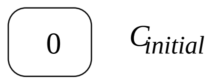


Figure 4.1: Initial version tree

When the negotiation process starts, negotiators $n_1$ and $n_2$ create their own contract versions. In Figure 4.2, the tree root has two children, labeled $0.i$, where $i \in [1,2]$. The node labeled $0.i$ is associated with the contract version created by the negotiator $n_i$.

Assume that negotiator $n_1$ derives two versions and that negotiator $n_2$ derives one version, then derives a new version from the freshly derived version. Negotiator $n_3$ enters the negotiation process. Her/his first contract version bases on the first contract version of $n_2$. The structure of the version tree after these modifications is shown in Figure 4.3. The two versions derived by $n_1$ are labeled 0.1.1 and 0.1.2, while versions derived by $n_2$ are labeled 0.2.1 and 0.2.1.1. The version derived by $n_3$ is labeled 0.2.2.

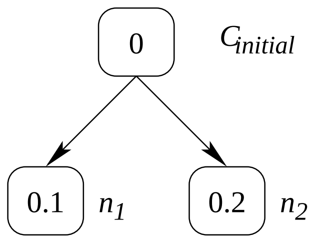


Figure 4.2: Version tree at the beginning of the negotiation process

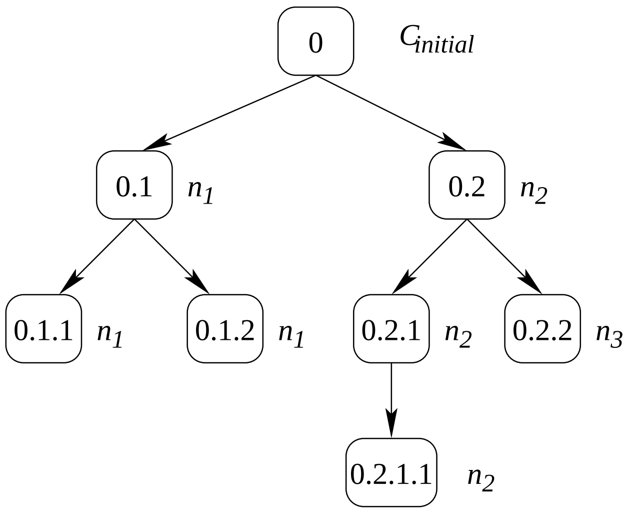


Figure 4.3: Version tree after modifications

### 4.2.3 Multiversion Paragraphs

In the contract model, it is assumed that a contract content consists of multiversion paragraphs, while a given contract version consists of given versions of these paragraphs. In other words, all paragraphs exist in all contract versions. However, if a paragraph is missing in a given contract version, the version of this paragraph in this version is *null*.

Adding a new paragraph $p$ to a given contract version $v$ causes the addition of a multiversion paragraph $P$ to the contract. The newly added multiversion paragraph has the value $p$ for contract version $v$, while it is *null* for all other contract versions.

Deleting an existing paragraph $p$ in a given contract version $v$ causes the modification of the multiversion paragraph $P$, where $p$ is a version of $P$. The multiversion paragraph $P$ has value *null* for version $v$ after deletion.

Modifying an existing paragraph $p$ in a given contract version $v$ to set a new value $val_{new}$ causes the modification of the multiversion paragraph $P$, where $p$ is a version of $P$. The multiversion paragraph $P$ has value $val_{new}$ for version $v$ after modification.

The proposed contract model is characterized by the following: the contract, in all its versions, has the same set of paragraphs. Differences existing between contract content versions can be reduced to differences between paragraph versions.

### 4.2.4 Multiversion Contract Structure

A contract content is a set of multiversion paragraphs. A *multiversion structure* is responsible for maintaining the structure of the contract content in its various versions.

The structure may also be used for contents formatting. Due to the separation of contents and structure, the contract model can easily be extended to support new formats, with new contract structures. As the focus in this dissertation is not on contract contents formatting, the contract structure used in the proposed model maintains a list of paragraphs, with no formatting elements.

As the contract structure is multiversion and separated from the contents, differences between contract version structures can be reduced to differences between contract structure versions.

## 4.3 Multi-facet Classification Mechanism

### 4.3.1 Analysis of the Negotiation Process

In mass negotiations in which the number of negotiators is high, and the number of contract versions is very high, a negotiation process is possible only if negotiators have an access to synthetic views of the negotiation process. A fundamental element of every negotiation strategy is the planning process ([62], pp. 40-51). The planning process is mainly basing on various analyses of the current status of negotiations. In highly concurrent environments, negotiators cannot conduct these analyses manually, because the amount of data to be analyzed is too high. Therefore, a multi-facet analysis mechanism is proposed to be integrated in the negotiation support system.

A negotiation support system has to provide negotiators with a possibility of various analyses of contract versions authored by different negotiators to well understand different aspects of a conducted negotiation process. For instance, a negotiator may want to analyze the involvement of different negotiators in the negotiation process, or analyze the correlation between a contract part defining a delivery date and other contract parts. To analyze the multi-thread history and the current status of a negotiation, both the abstract objects to be analyzed and the analysis criteria must be defined [59].

### 4.3.2 Mapping Functions

Objects to be analyzed are generated by a mapping function $f$ from space $C$ to a set denoted $D_f$. $C$ is the multiversion contract space. Different mapping functions are used to analyze different aspects of the negotiation process.

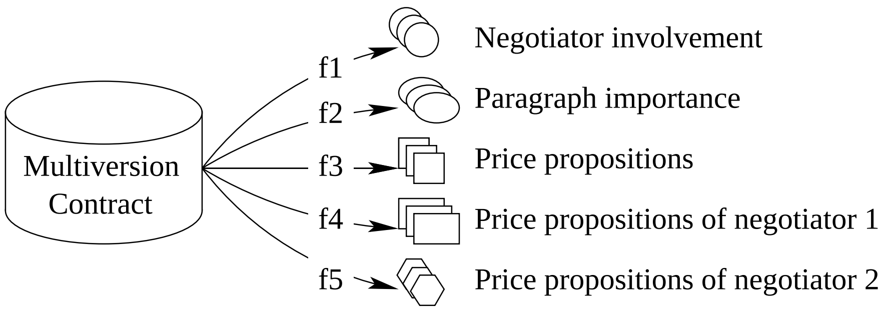


Figure 4.4: Mapping functions

Figure 4.4 illustrates the use of various mapping functions to analyze different aspects of the negotiation process. Mapping function $f_1$ generates a set of objects modeling negotiator involvement. Mapping function $f_2$ generates a set of objects modeling paragraph importance. Mapping functions $f_3$, $f_4$, and $f_5$ generate sets of objects modeling price propositions for various subsets of the negotiation process.

### 4.3.3 Hierarchical Classification

In a highly concurrent environment, the result of an analysis should be a hierarchical classification [57]. Given a set of objects, a classification splits it into subsets of similar objects, denoted classes. Hierarchical classification provides negotiators with a set of embedded classes. Negotiators can then choose a granularity level (the number of classes) of the classification. For instance, the same set of negotiators will be split into few classes if a general involvement characteristic is required, or into many classes if detailed characteristics of negotiators involvement is required.

The hierarchical classification technique chosen is based on ultrametrics. As mentioned in Section 3.2, an alternative to this technique is the SOTA approach. However, the SOTA approach requires that the structure of the neural nodes is identical to the input data structure. Each facet of the negotiation process requires potentially a new mapping function, and thus would require a new neural network, which is not feasible in practice. The SOTA approach is not adapted to the various objects generated by mapping functions.

As the proposed solution is based on metrics, and ultrametrics, the criteria used to analyze set $D_f$ are defined in a human understandable way, so that negotiators may choose and eventually define the criteria they want. The concept of similarity, which is the basis of classification, is closely related with the concept of distance. Two objects are similar if they are close to each other. The concept of distance is intuitive and easily understandable. Negotiators can thus easily define their own analysis criteria.

CHAPTER 5

# CONTRACT MODEL

## 5.1 Multiversion Contract

A multiversion contract consists of:

- a set of negotiators, referenced by their unique identifier,
- a set of contract versions,
- a set of multiversion *members*.

A member is a part of the contract. It may be for instance a paragraph, some multimedia data, e.g. a picture or a digital signature, or a representation of the structure of the contract.

More formally, let us denote:

- $neg_a$ the identifier of $NEG_a$, the $a^{th}$ negotiator,
- $cv_b$ the identifier of $CV_b$, the $b^{th}$ contract version,
- $mm_c$ the identifier of $MM_c$, the $c^{th}$ multiversion member, and
- $C$ the multiversion contract:

$$C = (\{NEG_a\}, \{CV_b\}, \{MM_c\}).$$

## 5.2 Contract Version Tree

### 5.2.1 Derivation

The first contract version is called the *root* version. The root version is created by a given negotiator, who publishes it and allows other negotiators to access it. Other contract versions are created by *derivation*. A derivation operation respects the following rules:

**Rule 5.2.1.** *Just after derivation, the derived version, the child version, is identical to the version it has been derived from, the parent version.*

**Rule 5.2.2.** *The modification of the child version has no influence on the parent version.*

**Rule 5.2.3.** *The modification of the parent version has no influence on the child version.*

A given contract version is owned by only one negotiator. However, derivation of a given contract version does not impose any restrictions on the ownership of the parent and/or a child version. Negotiator $NEG_{a_i}$ can derive contract version from all existing contract versions, even from those owned by other negotiators $NEG_{a_j}$, where $a_i \neq a_j$.

### 5.2.2 Contract Version Identifier Structure

Contract versions are organized as a tree. A good choice of contract version identifier can capture the tree structure and the "is-owner" relationship with related negotiators. As those two aspects are orthogonal, the contract version identifier consists of two independent parts:

- the identifier of the owner;
- a subidentifer which identifies the position of the contract version in the version tree.

Formally,

$$cv_b = (neg_{owner}, cvID_b),$$

where $cvID_b$ is the subidentifier.

Subidentifiers are responsible for capturing the contract version tree. Having a given contract version, we have to be able to find its parent and children versions. The strategy used for subidentifiers is the one used for database version stamps in the multiversion database model (cf. Section 3.3.3):

**Rule 5.2.4.** *If a contract version is the $n^{th}$ child of contract version whose subidentifier is $p$, the subidentifier of the child contract version is $p.n$. The root contract version subidentifier is* 0*.*

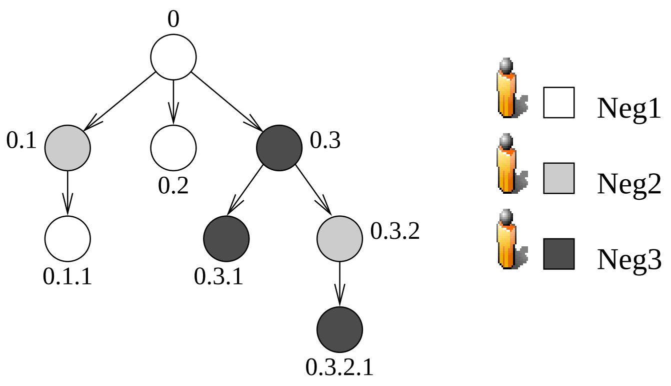


Figure 5.1: An example of a contract version tree

A simple contract version tree is presented in Figure 5.1. Three negotiators are involved in the negotiation process. Negotiator $NEG_1$ starts the negotiation process with the publication of the root version whose subidentifier is 0. Subidentifiers of other versions are built according to Rule 5.2.4. For example, the version derived by negotiator $NEG_2$ from the contract version whose subidentifier is 0 is the first derived version. According to Rule 5.2.4, its subidentifier equals to 0.1.

## 5.3 Multiversion Members

A multiversion contract consists of multiversion members. Each multiversion member appears in all contract versions, independently of the contract version's owner. If negotiators $NEG_1$ and $NEG_2$ are negotiating a multiversion contract $C$, and $C$ consists of a single multiversion member $MM_1$, then $MM_1$ is part of all contract versions, no matter if the owner is $NEG_1$ or $NEG_2$.

The concept of multiversion member is illustrated in Figure 5.2. Two multiversion members exist in the multiversion contract. In all contract versions, both members appear.

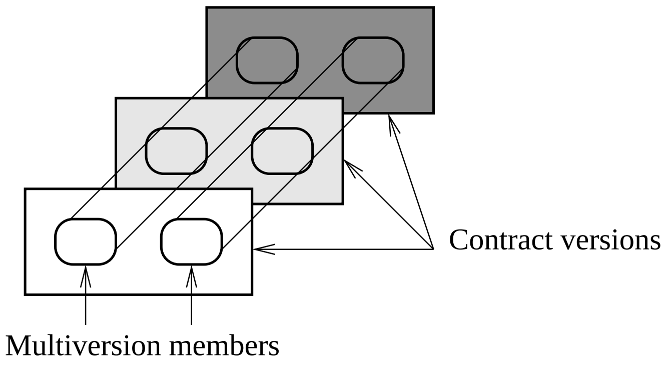


Figure 5.2: Multiversion members in a multiversion contract

### 5.3.1 Member Instances

A state of a multiversion member is called a *member instance*. Each member instance is associated with a multiversion member. In other words, no member instance can exist "outside" a multiversion member. As a consequence, the creation of a member instance causes its association to an existing multiversion member, or the creation of a new multiversion member and association of the new member instance with this new multiversion member.

A given member instance is associated with one or more contract versions. A member instance cannot exist "outside" a contract version. It may be shared by many contract versions. The instance sharing captures the relationship between contract versions at the multiversion member level.

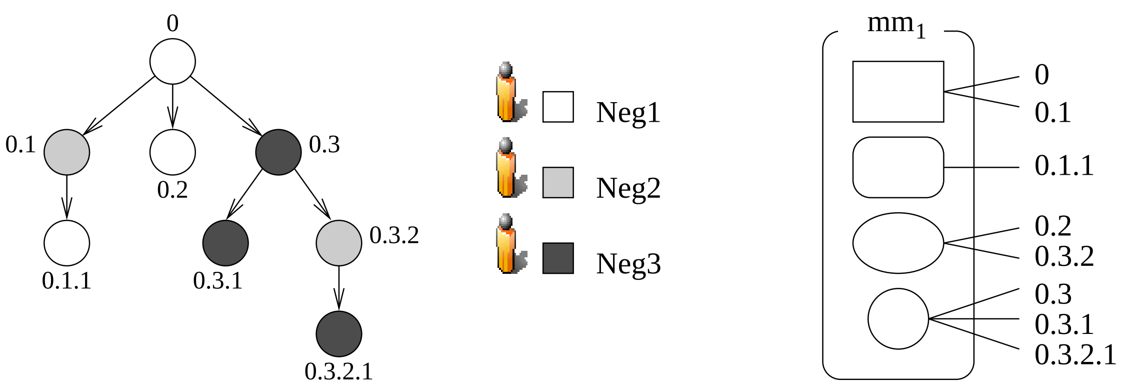


Figure 5.3: Member instances

An example of multiversion member $MM_1$ with a few member instances is given in Figure 5.3. Four member instances are associated with $MM_1$. They are graphically represented by a rectangle, a rounded rectangle, an ellipse and a circle.

The contract version tree is assumed to be the one presented in Figure 5.1. The rounded rectangle member instance is not shared: it is only associated with one contract version, namely 0.1.1. Other member instances are shared. The circle member instance is associated with contract versions 0.3, 0.3.1, and 0.3.2.1. Taking the contract version tree into account one may deduce that the value of $MM_1$ was first a rectangle (in contract versions 0 and 0.1). Then negotiator $NEG_1$ derived a new contract version from 0.1 in which the value of $MM_1$ is a rounded rectangle. Negotiator $NEG_3$ derived a new contract version 0.3 in which the value of $MM_1$ is a circle. Analyzing the version tree and member instances in deep, one may note that negotiator $NEG_3$ keeps a rigid position in the negotiation process as she/he only proposes the circle value for $MM_1$ in all contract versions she/he is the owner.

### 5.3.2 Association Table

Formally, a multiversion member consists of:

- a unique identifier, denoted $mm_c$,
- a set of member instances,
- an association table $AT$ that captures one-to-many relationships between member instances and contract versions.

Let us denote:

- $MI_d$ the $d^{th}$ member instance of a given multiversion member,
- $mi_d$ the identifier of $MI_d$.

Then a multiversion member is defined as:

$$MM_c = (mm_c, \{MI_d\}, AT)\,.$$

An association table associates each member instance with at least one contract version. An association table consists of rows, one row per member instance. Each row is a pair ($mi_d$, set of contract versions associated with $mi_d$) .

Formally,

$$AT = \{(mi_d, \{cv_d\})\}\,.$$

### 5.3.3 Version Inheritance

To limit redundancy in association tables, the rows should be as small as possible. To do so, the concept of *version inheritance* for member instances is introduced.

**Rule 5.3.1.** *A contract version is mentioned in the association table iff the member instance associated with this contract version is not shared with the parent contract version.*

Version inheritance concerns a case when a member instance $MI_d$ is shared by two contract versions, one being a child of the other. In this case, there is no need to mention the child contract version in the association table row concerning $MI_d$, if we assume that a contract version not mentioned in the association table shares the member instance with its parent contract version.

The structure of a contract version subidentifier, defined above, allows the parent contract version subidentifier to be easily retrieved. This characteristics of contract version subidentifiers is crucial for version inheritance. In case of version inheritance, the member instance retrieval implies easy retrieval of parent contract versions.

## 5.4 Contract Version Management

### 5.4.1 Historical and Draft Versions

During the negotiation process, one may distinguish two kinds of contract versions,

- historical versions,
- draft versions.

A *draft version* is a contract version that is viewable and modifiable by its owner. The possible actions on a draft version are:

- retrieval,
- modification,
- publication.

When the negotiator has finished her/his work on a draft version, she/he can make it visible to other negotiators. The draft version then becomes a *historical version*. A historical version is frozen. No one can modify it, even its owner. The only two possible actions on a historical versions are:

- retrieval,
- contract version derivation.

Every freshly derived version is a draft version. As a consequence, derived versions become visible to other negotiators when their owners publish them. Note that a negotiator has an access to all historical versions and only to her/his own draft versions.

### 5.4.2 Contract Version Management Operations

As illustrated in Figure 5.4, the contract version management provides only four operations: contract version retrieval, derivation, modification, and publication.

A retrieval operation consists of retrieval of instances of all multiversion members that belong to a given contract version. As only historical versions may be the subject of retrieval operations, and historical versions are frozen, concurrent accesses may occur in the read-only mode. Therefore, there is no problem concerning concurrency control.

A derivation operation causes the insertion of a new node in the contract version tree. The new node must have a flag indicating that the newly created version is a draft one. No change is needed to multiversion members. The version inheritance mechanism assures that the value of each multiversion member in the derived contract version $CV_{new}$ is identical to the value in the contract version $CV_{new}$ was derived from. The problem of concurrency control is limited to a safe creation of contract version subidentifier. The creation of contract version subidentifiers must be atomic so that the derivation of a new version by two negotiators from the same parent contract version results in two different subidentifiers.

The publication operation does not need concurrency control. As only draft versions may be the subject of publication, and only the owner of a draft version may publish it, the publication operation is by nature an isolated operation. It implies only the removal of the "draft version" flag, set during derivation.

The modification operation is presented in more details in Section 5.5.2.

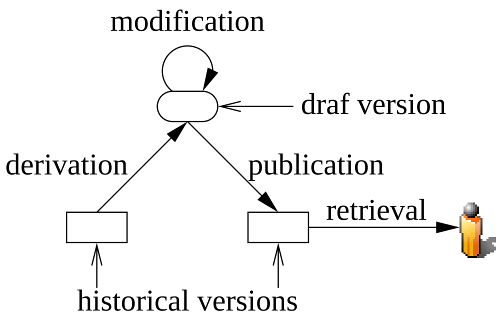


Figure 5.4: Contract version management operations

## 5.5 Operations on Multiversion Members

Operations on multiversion members concern the modification and reading of a given contract version. Four operations on multiversion members are defined: reading, updating, deletion, and creation. The reading operation deals with both historical and draft versions. The updating, deletion and creation operations can be performed only on a draft version. As draft versions are accessible only to their owners, there is no concurrency control needed.

### 5.5.1 Reading

Having a given contract version $CV_b$ and a given multiversion member $MM_c$, the reading of the value of the multiversion member corresponds to the retrieval of the member instance $MI_d$ associated with $CV_b$. The algorithm used, given in a pseudo-code, is the following one:

```
1.  define resultMemberInstance;
2.  set $cv_{temp}$= $cv_b$;
3.  set existsRow = false;
4.  do {
5.      foreach row in AT {
6.          if (row contains $cv_{temp}$) {
```

```
7.              existsRow = true;
8.              resultMemberInstance = row.memberInstance;
9.          }
10.     }
11.     if (!existsRow) {
12.         $cv_{temp}$ = parentOf($cv_{temp}$);
13.     }
14. } while (!existsRow);
15. return resultMemberInstance;
```

The loop, from line 5 to line 10, checks if the given contract version is explicitly mentioned in the association table. If so, the member instance mentioned in the association table row concerning the contract version is retrieved.

If the member instance is an object of version inheritance, the contract version is not explicitly mentioned in the association table. The parent contract versions (lines 11 to 13) must be sought in the association table, potentially recursively.

### 5.5.2 Updating

For the updating operation of a given multiversion member $MM_c$ in a given contract version $CV_b$, two cases must be distinguished, depending on the fact whether $MM_c$ in $CV_b$ is shared or not. $MM_c$ in $CV_b$ is shared iff:

- $CV_b$ is not mentioned in the association table of $MM_c$ (version inheritance);
- the association table row mentioning $CV_b$ mentions also other contract versions.

#### Non-shared version

When $MM_c$ is not shared in $CV_b$, the member instance $MI_d$ associated to $CV_b$ must be retrieved according to the algorithm given in Section 5.5.1, and then updated. No modification of the association table is needed.

#### Shared version

When $MM_c$ is shared in $CV_b$, the member instance $MI_d$ associated with $CV_b$ must be retrieved according to the algorithm presented in Section 5.5.1. Then a new member instance $MI_{d'}$ with the new value must be created. A new row associating $MI_{d'}$ with $cv_d$ must be created in the association table. The association table row previously associated with $MI_d$ must be modify so that it contains only:

- all contract version identifiers explicitly associated with $MI_d$ except $cv_d$, if $cv_d$ was explicitly associated with $MI_d$;
- all contract version identifiers of children version of $cv_d$ which are not explicitly associated to other member instances.

### 5.5.3 Deletion

A special member instance, denoted *null*, is used for the deletion operation. The meaning of *null* is “does not exist”. The use of the *null* value is required because multiversion members MUST exist in all contract versions.

The deletion operation of a given member in a given contract version is reduced to an updating operation to the *null* value. The deletion operation sets the value of the multiversion member to “does not exist”, which means that the given multiversion member is not present is the given contract version.

### 5.5.4 Creation

As assumed in Section 5.3.1, a member instance cannot exist “outside” a multiversion member. As a consequence, the creation of a member instance causes, first, the creation of a new multiversion member and, second, association of the new member instance with the newly created multiversion member.

More precisely, first, a new multiversion member $MM_{new}$ is created and added to the multiversion contract. The *null* member instance is associated with the root contract version in the newly created multiversion member $MM_{new}$. The new member instance is then set using an updating operation.

### 5.5.5 Composite Multiversion Members

A composite multiversion member is a one that points to other members. Other members are referenced by the multiversion member identifier.

Consider as an example a multiversion member $MM_{offer}$ that captures an item offer of the contract. In the multiversion contract whose contract version tree is assumed to be the one presented in Figure 5.1, three other multiversion members exist: $MM_{price}$ represents the price of the object of the negotiation, $MM_{delivery}$ represents the delivery date of this object, and $MM_{warranty}$ represents the warranty concerning the object. Association tables of $MM_{price}$, $MM_{delivery}$, and $MM_{warranty}$ are presented in Tables 5.1, 5.2, and 5.3, respectively.

The association table of $MM_{offer}$ is presented in Table 5.4. The multiversion member consists of:

- a comment,
- the first paragraph,
- the second paragraph.

Paragraphs are references to multiversion member identifiers. In contract version 0, $MM_{offer}$ is referencing $MM_{price}$ and $MM_{delivery}$. The corresponding member instances are, according to Tables 5.1 and 5.2, $p_1$ and $d_1$, respectively. Using version inheritance, $MM_{offer}$ in contract version 0.3 is referencing the same multiversion objects but the corresponding member instances are different ($p_2$ and $d_2$). In contract version 0.3.1, $MM_{offer}$ is referencing $MM_{price}$ and $MM_{warranty}$. The corresponding member instances are $p_2$ and $w_2$, respectively. In contract versions 0 and 0.1, the multiversion member $MM_{offer}$ differs only by the comment associated. The references to the multiversion members modeling paragraphs are identical.

| member instance | contract version(s) |
|---|---|
| $p_1$ | 0 |
| $p_2$ | 0.3 |

Table 5.1: $MM_{price}$ association table

| member instance | contract version(s) |
|---|---|
| $d_1$ | 0 |
| $d_2$ | 0.1.1,0.3 |

Table 5.2: $MM_{delivery}$ association table

| member instance | contract version(s) |
|---|---|
| $w_1$ | 0 |
| $w_2$ | 0.2,0.3 |

Table 5.3: $MM_{warranty}$ association table

| member instance | | | contract version(s) |
|---|---|---|---|
| comment | first paragraph | second paragraph | |
| The first version with price and delivery | $mm_{price}$ | $mm_{delivery}$ | 0 |
| The second version with price and delivery | $mm_{price}$ | $mm_{delivery}$ | 0.1,0.2 |
| The version with price and warranty | $mm_{price}$ | $mm_{warranty}$ | 0.3.1 |

Table 5.4: $MM_{offer}$ association table

Composite multiversion members model the structure of the multiversion contract. In the former example, the composite multiversion member $MM_{offer}$ models the fact that the item offer consists of price and delivery are referenced in all contract versions except 0.3.1, in which the price and warranty are referenced. The value of the price in a given version is not taken into account directly in $MM_{offer}$. Only the structure of the offer is captured.

The proposed contract model does not assume a fixed structure of contracts. It provides a basis for multiversion contract design. Multiversion contract design consists of designing multiversion members — some of them being composite multiversion members — to model the structure and the semantics of a given contract. This allows to model various kind of contracts and do not restrict the contract model to one structure.

A simple multiversion contract may for instance consist of many multiversion members, each modeling a contract paragraph, and a composite multiversion member modeling the structure of the contract as a paragraph list. Another contract with additional semantics may consist of multiversion members modeling various semantically different contract parts, a multiversion member modeling the price, another multiversion member modeling the warranty, etc. A more complex contract may consist of composite multiversion members to capture a tree structure of contract parts. The paragraph concerning the price may then be part of a section concerning offers, which is part of the contract. At a higher level of abstraction, the structure of the contract may be complex, modeling semantics of various parts of a contract, e.g. addenda.

CHAPTER 6

# ANALYSIS DOMAINS

## 6.1 Domain Objects

Domain objects are used to model various facets of the negotiation processes. Domain objects may for instance represent the activity of negotiators, the importance of paragraphs, etc. As a consequence, domain objects must be flexible enough to represent various data types.

Each domain object is an element of an *analysis domain*. An analysis domain is a set of domain objects modeling a facet of a negotiation process. Formally, let $D_{facet}$ denote the analysis domain modeling a facet of a negotiation process, denoted *facet*.

A domain object $DO_i$ is uniquely identified in a given set of domain objects by its identifier $do_i$. Formally,

$$\forall (DO_i, DO_j) \in D^2_{facet},\ do_i = do_j \Leftrightarrow DO_i = DO_j$$

$$\forall (DO_i, DO_j) \in D_{facet} \times D_{facet'},\ do_i = do_j \text{ and } DO_i \neq DO_j \Rightarrow D_{facet} \neq D_{facet'}$$

A domain object $DO_i$ consists of:

- a unique identifier, denoted $do_i$,
- a set of attributes, and
- a type.

An attribute is a pair $(name, value)$. Each attribute models a property of the domain object. To illustrate the use of attributes, let us assume that a negotiator is modeled by a domain object denoted $DO_{neg}$. The attributes of $DO_{neg}$ are pairs $('firstName', 'John')$, $('lastName', 'Smith')$, and $('represents', 'ACME\ Corp.')$.

An attribute name is a character string. A character string consists of one or many characters defined in the Unicode standard [85]. In case of mass negotiations, no limitations are assumed on the location of the negotiators and their language. As a consequence, attribute names should not be limited to one or a few sets of languages and their characters. The use of Unicode allows for an international audience.

An attribute value is a domain object, because domain objects are able to model complex data types. A domain object may, for instance, model a negotiation party, with attributes *name* (the name of the enterprise) and *representant* (the negotiator involved in the negotiation process). The value of the *representant* attribute may be the domain object defined in the former example.

Also identifiers are domain objects in order to associate semantics with the identifiers. For instance, the identifier $do_{neg}$ may be a domain object modeling a person identity. Identifier $do_{neg}$ may have the following attributes: *SSN* (Social Security Number), and *email*. The domain object $DO_{neg}$ is then identified by a domain object defining the social security number and the email of the negotiator.

Object domain types are Unicode character strings. Object domain types are used for two purposes. First, an object domain type associates some semantics with an object domain. In the former example, the type of $DO_{neg}$ may be *negotiator* to indicate the meaning of the data $DO_{neg}$ models. Second, domain object types allows domain object structure to be defined. The structure of domain objects representing negotiators may be defined as follow:

- identifier: type *personIdentity*,
- attributes:
  - firstName: type *String*,
  - lastName: type *String*,
  - represents: type *LegalPerson*.

Types of identifier and attribute values are corresponding to domain object types.

The following six primitive domain objects may be considered as the atomic data elements for domain object building:

- String,
- Integer,
- Long,
- Float.
- Double,
- Boolean.

Primitive domain objects do not reference any other domain object type. Primitive objects have only one attribute denoted *value*. The value of this attribute depends on the domain object type. Primitive domain object values are presented in Table 6.1.

## 6.2 Analysis Domain Definition

Domain objects are generated according to an *Analysis Domain Function (ADF)*. An ADF is a function whose image is an analysis domain. Formally,

$$f \text{ is an ADF } \Leftrightarrow \begin{cases} f \text{ is a function on one or more analysis domains } D_{orig_i} \\ Im(f) = \{DO\}, \text{ where } DO \text{ are domain objects} \end{cases}$$

When two or more analysis domains exist for an ADF, the function is said to be *multivariable*. A special analysis domain, denoted $\emptyset$, is defined by $card(\emptyset) = 0$. The existence of the analysis domain $\emptyset$ allows to distinguish *transformer* functions from *generator* functions.

**Definition 6.2.1.** *An ADF $f$ is a* generator *function iff only one origin domain of $f$ exists that is $\emptyset$.*

A generator function creates an analysis domain without the need of pre-existing data in the form of an analysis domain. A generator function may for instance generate the number $\pi$, or retrieve association tables from the multiversion contract model.

**Definition 6.2.2.** *An ADF $f$ is a* transformer *function iff at least one origin domain of $f$ is different from $\emptyset$.*

A transformer function transforms an existing analysis domain into another analysis domain. A transformer function may for instance transform an analysis domain modeling association tables into another analysis domain representing the number of version of paragraphs.

ADF functions may be embedded according to the *composition law*. The composition law allows "pipelines" of functions to be defined, in which an analysis domain being the results of an ADF is an origin domain of another ADF.

**Definition 6.2.3.** *The ADF composition law, denoted $\circ$, defines an ADF $f_\circ$ from ADF $f_i$ as follow:*

- *for single-variable functions, $f_\circ = f_1 \circ f_2 = f_1(f_2)$;*
- *for multivariable functions, $f_\circ = f_1 \circ (f_2, \ldots, f_n) = f_1(f_2, \ldots, f_n)$,*

*where $f_1$ is a transformator function, while $f_2$ and $f_3$ are either generator or transformator functions.*

| Type | Description | Size/Format |
|---|---|---|
| String | Character string | from 0 to $2^{31} - 1$ Unicode characters |
| Integer | Integer | 32-bit two's complement |
| Long | Long integer | 64-bit two's complement |
| Float | Single-precision floating point | 32-bit IEEE 754 (defined in [37]) |
| Double | Double-precision floating point | 64-bit IEEE 754 (defined in [37]) |
| Boolean | A boolean value (true or false) | true or false |

Table 6.1: Primitive domain objects

## 6.3 Analysis Domain Language

The *Analysis Domain Language (ADL)* is used to define ADFs. ADL is a dialect of XML — the eXtensible Markup Language. The eXtensible Markup Language [11] describes a class of data objects, called *XML documents*, and partially describes the behavior of computer programs that process them. XML is based on SGML – the Standard Generalized Markup Language [38]. XML documents are conforming to SGML documents by construction.

There are multiple reasons for using XML for ADL:

- it is an emerging standard developed by the WWW consortium;
- it is a general-purpose and extensible language;
- it allows defining language grammar that can be automatically validated by a parser;
- it is designed and optimized for parsing structured documents;
- the parsing software for XML is available;
- it can be easily integrated with other XML-based web standards;
- it allows defining human-readable data in a standardized way.

ADL is basing on four elements: *Metaobjects*, *ObjectSets*, *Tags*, and *Functions*. Metaobject correspond to domain objects. ObjectSets correspond to analysis domains. Tags are basic elements of processing. Functions correspond to ADFs. These four elements are presented in more idetails in Sections 6.3.3, 6.3.4, 6.3.5, and 6.3.6, respectively.

### 6.3.1 Modules

ADL is structured in *modules*. A module groups metaobject definitions, definitions of functions generating these metaobjects and potentially implementation of needed features – as tags. A module may for instance define metaobjects modeling multiversion members and their association tables, functions for association table retrieval and some new tags needed to access a database.

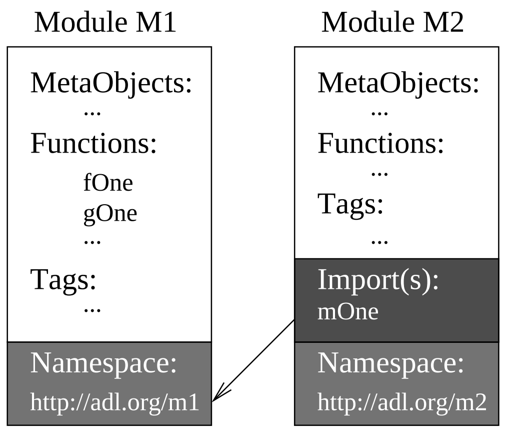


Figure 6.1: Modules in ADL.

XML Namespaces [10] is used to avoid name collisions. Metaobject, function and tag names are universal, their scope extends beyond the module that contain them. Each module is responsible for associating itself with a URI. A module may use metaobjects, functions and tags defined in another module. A namespace referring the URI of the used module must be defined and associated with a prefix. An XML namespace must be associated to every used module.

An example of the use of XML Namespaces for modularization of ADL is presented in Figure 6.1. Metaobjects, functions and tags defined in module $M_1$ may be used in module $M_2$ as qualified names. $M_2$ may, for instance, associate the prefix *mOne* to module $M_1$. Function *fOne* defined in $M_1$ may be then used in $M_2$ as *mOne:fOne*.

A module is defined in an XML document. A module definition document contains:

- the name of the module,
- a URI defining the associated namespace,
- potentially a list of tag definition references,
- potentially a list of function definition references, and
- potentially a list of metaobject definition references.

An example of module definition document is given below:

```
<module name="testModule" uri="http://nessy.pl/adl/testing">
  <tags>
    <tag-decl name="if" definition="if.tdl"/>
    ...
  </tags>
  <functions>
    <function-decl name="myFunction" definition="function1.fdl"/>
    ...
  </functions>
  <objects>
    <object-decl name="Negotiator" definition="Negotiator.odl"/>
    ...
  </objects>
</module>
```

In the module definition document given above, a module named `testModule` is defined. It is associated with the URI `http://nessy.pl/adl/testing`. First, a tag named `if` is defined. Its definition can be found in the `if.tdl` tag definition document. Second, a function named `myFunction` is defined. Its definition can be found in the `function1.fdl` function definition document. Finally, a metaObject named `Negotiator` is defined. Its definition can be found in the `Negotiator.odl` metaobject definition document.

The XML Schema defining the structure of module definition documents is given in Appendix A.1.

ADL provides a module which groups basic functionalities of ADL. The namespace for this module — denoted the *core* module — is *http://nessy.pl/adl/core*. The *core* module provides primitive metaObjects and fundamental tags for ADL.

### 6.3.2 Expressions

In ADL, expressions are defined as ${*someExpression*}. An expression can contain:

- variable references,
- operators, and
- literals.

#### Variable reference

Variable references are done by names. If a variable *myVariable* has been defined, the expression ${*myVariable*} returns the variable *myVariable*. To test if *myVariable* is set, the expression ${*isEmptyObject␣myVariable*} may be used.

#### Operators

The following operators are defined:

- relational operators: ==, !=, <, >, <=, >=
- arithmetical operators: *, +, -, /, div, mod
- logical operators: ||, &&, !
- operator `empty` that checks if a metaobject is unset or if an objectSet contains some metaobject;
- operator "`.`" that retrieves metaobject attribute. For example, ${*myMetaObject.myAttribute*} retrieves the attribute *myAttribute* from metaobject *myMetaObject*;
- operator "`[]`" that retrieves metaobject from objectSet according to their IDs. For example, ${*myObjectSet*[*myMetaObjectID*]} retrieves the metaobject identified by *myMetaObjectID* from the objectSet *myObjectSet*.

#### Literals

The following literals are defined:

- logical: `true` or `false`,
- integers,
- floats,

- character strings surrounded by single or double quotes. The backslash character “\” is used to escape single and double quote characters, i.e. “”’ is obtained by “\”’. The backslash character must be entered as “\\”.
- Unset value: `null`.

### 6.3.3 MetaObjects

MetaObjects are defined in an XML document. A metaObject definition document contains

- the name of the metaObject type,
- the name of the type of the metaObject identifier, a list of the attributes and their type if the metaObject is not a primitive,
- the type of the value if the metaObject is a primitive.

An example of metaObject definition document is given below:

```
<object-def type=”Negotiator”>
  <id type=”SSN”/>
  <attributes>
    <attribute name=”firstName” type=”String”/>
    <attribute name=”lastName” type=”String”/>
    <attribute name=”represents” type=”Enterprise”/>
  </attributes>
</object-def>
```

In the metaObject definition document presented above, a metaObject named `Negotiator` is defined. It is identified by a metaObject whose type is `SSN`. Three `Negotiator` attributes are defined: `firstName`, `lastName` (both of type `String`) and `represents` (of type `Enterprise`). As attributes are defined, the metaObject `Negotiator` is not a primitive metaObject.

An example of a primitive metaObject definition document is given below:

```
<object-def type=”String”>
  <value type=”java.lang.String”>
</object-def>
```

In the metaObject definition document presented above, the `String` primitive metaObject is associated with the `java.lang.String` class. No ID is defined as the `java.lang.String` is responsible for unique self-identification.

The XML Schema defining the structure of metaObject definition documents is presented in Appendix A.2.

### 6.3.4 ObjectSets

An objectSet is a set of metaObjects. All metaObjects of a given objectSet have the same type. An objectSet may be empty, i.e. no metaObject is a member of the objectSet. The emptiness of an objectSet may be checked with the `isEmptySet` operator. If objectSet *OS* is empty, *isEmptySet␣OS* is true. The emptiness of an objectSet may also be checked with the size operator. The size operator returns the size of an objectSet. The size of an objectSet is the number of metaObjects it contains. If the objectSet *OS* is empty, *OS.size* equals to 0.

The *core* module provides tags for basic operations on objectSets. Four operations are defined: objectSet creation (`declare` tag), metaObject addition to an objectSet (`add` tag), metaObject deletion from an objectSet (`remove` tag), and deletion of all metaObjects from an objectSet (`clear` tag). The `for-each` tag is an iterator on objectSets. These tags are described in more details in Section 6.3.7.

All ADF origin domains are objectSets. The image (in the mathematical sense) of all ADF is an objectSet. Figure 6.2 illustrates the relationship between ADF (functions) and objectSets.

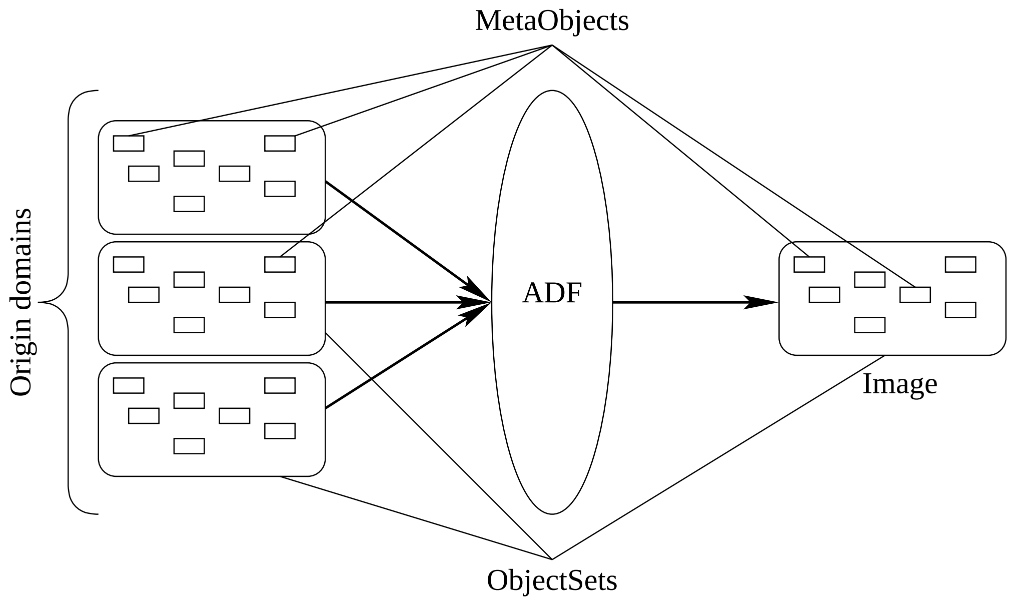


Figure 6.2: ObjectSets and ADF

### 6.3.5 Tags

Tags associate *processing entities* with XML tags. A processing entity is an independent software or a part of a software, such as a database access layer or a statistical library. The XML tags associated with processing entities can be used in function definitions. Tags are the only mechanism to extend ADL. When new features are needed, a new XML tag may be associated with a processing entity that implements the needed feature.

Two kinds of tags may be defined:

- empty tags, and

- non-empty tags.

An empty tag corresponds to an XML empty tag. An empty tag does not have any content. In a function declaration, an empty tag is called by the insertion of the associated empty XML tag.

A non-empty tag corresponds to a non-empty XML tag. A non-empty tag has content, with one or many children tags. In a function declaration, a non-empty tag is called by the associated non-empty XML tag.

Processing entities may be written in various programming languages. For the Java$^{TM}$ language, which has been chosen for the implementation of the ADL compiler, two interfaces have been defined: one for empty tags, and the other for non-empty tags.

Tags are defined in an XML document. A tag definition document contains:

- the name of a tag,
- the name of programming language a tag is implemented in,
- optionally – tag parameters specific to the chosen programming language.

An example of tag definition document is given below:

```
<tag-def name="SQLQuery" lang="java">
  <javaClass name="pl.nessy.db.SQLQuery">
    <params>
      <param name="connection" type="String" required="true"/>
      <param name="query"      type="String" required="true"/>
    </params>
  </javaClass>
</tag-def>
```

In the tag definition document presented above, a tag named `SQLQuery` is defined. It is implemented in the Java$^{TM}$ programming language. All children (`connection` and `query`) elements are specific to tag implementation in Java$^{TM}$.

The XML Schema defining the structure of tag definition documents is presented in Appendix A.3.

### 6.3.6 Functions

Functions are the core of ADL. An ADF is expressed in ADL as a function. Each function models a potential facet of a negotiation process. A function processes zero, one or many objectSets. The result of the processing of a function is an objectSet. Generator ADFs, as defined in Section 6.3.7, are functions that do not process any objectSet. Transformers ADFs are functions that process at least one objectSet.

Functions are defined in an XML document. A function definition document contains:

- the name of a function,

- optionally – the names of modules the function uses,
- optionally – the objectSets to be processed and the type of metaObjects they contain,
- the name of the resulting objectSet and the type of metaObjects it contains,
- the processing actions to be performed.

The processing actions may be calls to tags and functions. Calls to tags are done by inserting the associated XML tags. To call functions, a special tag defined in the *core* module is applied.

An example of function definition document is given below:

```
<function-def name=”getNegotiators”
  xmlns:core=”http://nessy.pl/adl/core”
  xmlns:col =”http://nessy.pl/adl/collections”>

  <param  name=”contracts”    type=”Contract”/>
  <param  name=”firmDB”      type=”Firm”/>
  <result name=”negotiators” type=”Negotiator”/>

  <processing>
    <declare name=”tempNegotiators” type=”Negotiator” isASet=”true”/>

    <core:for-each var=”contractVersion” items=”contract”>
      <core:declare name=”localNegotiator” type=”Negotiator”/>
      <core:set
        obj=”localNegotiator”
        attribute=”Name”
        value=”${contractVersion.negotiator}”/>

     <col:ifContains
        col=”firmDB”
        item=”${contractVersion.negotiator.firm}”>
        <core:set
          obj=”localNegotiator”
          attribute=”represents”
          value=”${contractVersion.negotiator.firm}”/>
      </col:ifContains>

      <add to-set=”tempNegotiators” name=”localNegotiator”/>
    </core:for-each>

    </core:execute name=”col:deleteDoublons” into=”negotiators”>
       <core:param value=”tempNegotiators”/>
    </core:execute>
  </processing>
</function-def>
```

In the function definition document presented above, a function named `getNegotiators` is defined. It uses the `core` module (line 2) and a possible `col` module responsible for extended collection manipulation (3.). The function processes two objectSets `contracts` and `firmDB` containing metaObjects of type `Contract` and `Firm` (lines 5 and 6), respectively. The resulting objectSet `negotiators` contains metaObjects of type `Negotiator` (line 7).

The XML Schema defining the structure of function definition documents is presented in Appendix A.4.

## 6.3.7 The *core* Module

The *core* module defines a set of primitive metaObjects and tags. Primitive objects are defined in Section 6.1. As presented in Table 6.2, tags in the *core* module are classified in five categories:

- variable declaration,
- metaObject attribute setting,
- control flow statements,
- function call, and
- objectSet manipulation operations.

| Tag category | Tag(s) name |
|---|---|
| Variable declaration | `declare` |
| MetaObject attribute setting | `set` |
| Control flow statement | `choose`, `if` |
| Function call | `execute` |
| ObjectSet manipulation | `add`, `for-each`, `remove`, `clear` |

Table 6.2: Tags defined in the *core* module

### Variable Declaration

The `declare` tag allows a variable to be declared. A variable may be a metaObject or an objectSet. Variable declaration associates a name with a variable.

When a metaObject is declared, the type of the newly created metaObject is set. If an attribute value is a primitive object, it is initialized with the default value of the given primitive metaObject. If an attribute value is not a primitive metaObject, it is set to `null`.

When an objectSet is declared, the newly created objectSet does not contain any metaObject. The type of metaObjects that can be added in future is set at the declaration time.

An example of objectSet declaration is the following one:

```
<declare name=”allNegotiators” type=”Negotiator” isASet=”true”/>
```

In the objectSet declaration given above, a variable named `allNegotiators` is defined. The type of metaObjects that can be added to the `allNegotiators` objectSet is `Negotiator`. The `isASet` attribute is set to `true` as the variable `allNegotiators` is an objectSet.

An example of metaObject declaration is the following one:

```
<declare name="aNegotiator" type="Negotiator"/>
```

In the metaObject declaration presented above, a variable named `aNegotiator` is defined. The type of the `aNegotiator` variable is `Negotiator`. The `isASet` attribute is not set as the value of this attribute is `false` by default.

### MetaObject Attribute Setting

The `set` tag allows metaObject attribute to be set. The new value may be a literal or an expression.

An example of metaObject attribute setting with the help of a literal is the following one:

```
<set obj="aNegotiator" attribute="firstName" value="'John"'/>
```

In the metaObject attribute setting presented above, the object of the attribute setting is the metaObject `aNegotiator`. The attribute `firstName` of `aNegotiator` is set to `'John'`.

An example of metaObject attribute setting using an expression is presented below:

```
<set
  obj="aNegotiator"
  attribute="firstName"
  value="${anotherNegotiator.firstName}"/>
```

In the metaObject attribute setting presented above, the object of the attribute setting is the metaObject `aNegotiator`. The attribute `firstName` of `aNegotiator` is set to the value of the attribute `firstName` of metaObject `anotherNegotiator`.

### Control Flow Statements

Two tags — `choose` and `if` — are used to control flow.

The `choose` tag selects one from a number of possible options. It consists of a sequence of `when` elements followed by an optional `otherwise` element. Each `when` element has a single attribute, `test`, which specifies an expression. The content of the `when` and `otherwise` elements is a sequence of processing actions. When a `choose` element is processed, each of the `when` elements is tested in turn, by evaluating the expression and converting it to a boolean. The content of the first, and only the first, `when` element whose test is `true` is processed. If no `when` is `true`, the content of the `otherwise` element is processed. If no `when` element is true, and the `otherwise` element is absent, nothing is done.

An example of `choose` tag application is the following one:

```
<choose>
  <when test="allNegotiators.size==1"/>
  <when test="allNegotiators.size>1">
    <clear set="allNegotiators"/>
    <add to-set="allNegotiators" name="aNegotiator"/>
  </when>
  <otherwise>
    <add to-set="allNegotiators" name="aNegotiator"/>
  </otherwise>
</choose>
```

In the `choose` tag example given above, if the size of the `allNegotiators` objectSet is 1, nothing is done. If the size is greater than 1, the objectSet is cleared and the metaObject `aNegotiator` is added. In all other cases (i.e when size equals 0), the metaObject `aNegotiator` is added.

The `if` element has a `test` attribute which specifies an expression. The content of a `test` attribute is an expression. The expression is evaluated and converted to a boolean. If the result is `true`, then the content of the `if` element is processed; otherwise, nothing is done.

An example of `if` tag application is the following one:

```
<if test="allNegotiators.size == 0"/>
  <add to-set="allNegotiators" name="aNegotiator"/>
</if>
```

In the `if` tag example presented above, if the size of the `allNegotiators` objectSet equals 0, the metaObject `aNegotiator` is added to the `allNegotiators` objectSet. Otherwise, nothing is done.

#### Function Call

The `execute` tag allows functions to be called.

The `execute` tag has a `name` attribute, which specifies the function to be called. When the function to be called is contained in another module, the full qualified name of the function must be used. The full qualified name of a function consists of `moduleNamespace:functionName`. The `into` attribute is used to define the variable the result of the function is bound to. When a function is a tranformer function, the objectSet(s) to be transformed can be specified as child(ren) `param` elements. Variables associated with `param` elements are assigned to function arguments sequentially, from top to bottom, i.e. the variable defined in the first `param` element is bound to the first objectSet defined in the function definition.

An example of function call is the following one:

```
<execute name="neg:getNegotiators" into="negotiators"/>
  <param value="contracts"/>
  <param value="firmDB"/>
</execute>
```

In the `execute` tag example presented above, the `getNegotiators` defined in module `neg` is called. Two objectSets are given as parameters: `contracts` and `firmDB`. The result of the function execution is stored in the objectSet named `negotiators`.

### ObjectSet Manipulation Operations

Four tags: `add`, `for-each`, `remove`, and `clear` are used to manipulate objectSets.

The `add` tag allows to add a metaObject to an objectSet. The name of the objectSet is specified by the `to-set` attribute. The metaObject to be added can be referred by the name of a variable it is bound to, whose name is provided by the `name` attribute, or by its identifier, whose value is provided by the `id` attribute. Only one of those two attributes — `name` or `id` — can exist in a given `add` tag.

An example of objectSet addition by name is the following one:

```
<add to-set="allNegotiators" name="localNegotiator"/>
```

In the `add` tag example presented above, a metaObject bound to a variable named `localNegotiator` is added to the objectSet named `allNegotiators`.

An example of objectSet addition by identifier is the following one:

```
<add to-set="allNegotiators" id="localNegotiatorID"/>
```

In the `add` tag example presented above, a metaObject identified by a metaObject bound to a variable named `localNegotiatorID` is added to the objectSet named `allNegotiators`.

The `for-each` tag contains processing actions, which are executed for any metaObject contained in a given objectSet. The name of the objectSet to be used is specified in the `items` attribute. If the objectSet is empty, nothing is done. The name of the variable the metaObject is bound to in a given loop is specified in the `var` attribute.

An example of `for-each` tag use is the following one:

```
<for-each var="contractVersion" items="contractVersions">
  <declare name="localNegotiator" type="Negotiator"/>
  <set
    obj=      "localNegotiator"
    attribute="Name"
    value=    "${contractVersion.negotiator.Name}"/>
  <add to-set="allNegotiators" name="localNegotiator"/>
</for-each>
```

In the `for-each` tag example given above, the `contractVersions` objectSet is used. For each metaObject of this objectSet bounded to variable `contractVersion`, the processing actions embedded are executed with a new value of `contractVersion`.

The `remove` tag allows to remove a metaObject from an objectSet. The name of the objectSet the metaObject is to be removed from is specified by the `from-set` attribute. The metaObject to be removed can be referred by the name of a variable it is bound to, whose name is provided by the `name` attribute, or by its identifier, whose value is provided by the `id` attribute. Only one of those two attributes – `name` or `id` – can exist in a given `remove` tag.

An example of objectSet removal by name is the following one:

```
<remove from-set="allNegotiators" name="localNegotiator"/>
```

In the `remove` tag example presented above, a metaObject bound to variable named `localNegotiator` is removed from the objectSet named `allNegotiators`.

An example of objectSet removal by identifier is the following one:

```
<remove from-set="allNegotiators" id="localNegotiatorID"/>
```

In the `remove` tag example presented above, a metaObject identified by a metaObject bound to a variable named `localNegotiatorID` is removed from the objectSet named `allNegotiators`.

The `clear` tag removes all the metaObjects from an objectSet. The name of the objectSet the metaObjects are to be removed from is specified by the `set` attribute.

An example of `clear` tag application is the following one:

```
<clear set="allNegotiators"/>
```

In the `clear` tag example presented above, all the metaObjects contained in the objectSet named `allNegotiators` are removed from this objectSet.

CHAPTER 7

# CLASSIFICATION OF DOMAIN OBJECTS

## 7.1 Parametric Analysis

The goal of classification is to provide a synthetic view of an aspect of the negotiation process. The choice of a facet of a negotiation process corresponds to the choice of an ADF. The result of the execution of an ADF is an analysis domain, i.e. a set of domain objects. The ADF defines the facet of the negotiation process to be analyzed, generating domain objects modeling a given facet.

As domain objects may model complex views of a negotiation process, and the interests of a given negotiator may be different from other negotiators' interests, many analyses may be performed on the same domain objects. Having an analysis domain modeling association tables of all multiversion members, a negotiator may be interested in influence of a given multiversion member on others, while another negotiator may be interested in the number of versions of each multiversion member. For this reason, the concept of parametric analysis is proposed.

**Definition 7.1.1.** *An analysis is* parametric *if various criteria may be used to perform various analyses of a given analysis domain.*

A given analysis domain consists of a set of domain objects. All domain objects of a given analysis domain are of the same type. As a consequence, the type of an analysis domain may be defined as follows:

**Definition 7.1.2.** *The* type *of an analysis domain* AD *is the type of domain objects of the analysis domain* AD*.*

The domain object type – and thus the analysis domain type – defines the attributes of all domain objects of this type. So, given an analysis domain, the attributes of domain objects of the analysis domain are known.

Analysis criteria need the presence of some attributes. Attribute values are data to be analyzed. Attribute names are semantics associated with data to be analyzed. Therefore, every analysis criterion is associated with a given domain object type. As a consequence, the type of an analysis criterion may be defined as follows:

**Definition 7.1.3.** *The* type *of an analysis criterion* AC *is the type of domain objects* the AC *is associated with.*

A given analysis domain may be associated only with the analysis criteria corresponding with the analysis domain type. A relationship many-to-many between analysis domains and analysis criteria exists. This relationship is a compatibility relationship.

**Definition 7.1.4.** *An analysis domain* AD *and an analysis criterion* AC *are* compatible *iff the type of* AD *and the type of* AC *are the same*

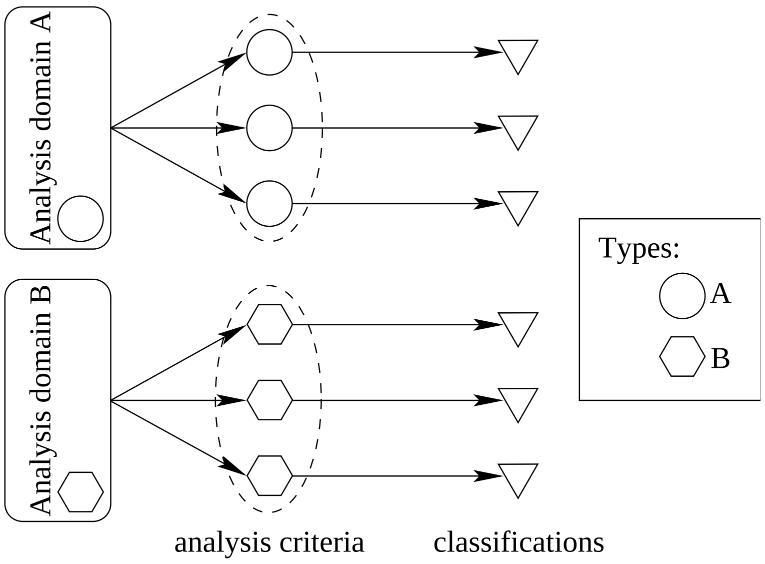


Figure 7.1: Parameterizable analysis

The concept of parametric analysis is illustrated in Figure 7.1. Two different analysis domains may be analyzed according to various criteria. Each criterion produces a classification. Each criterion is associated with an analysis domain type. In Figure 7.1, it is assumed that the domain analysis domain types of A and B are different. As a consequence, no criterion may be used for both analysis domain. The analysis domain A is *compatible* with the analysis criteria represented by a circle. The analysis domain B is *compatible* with the analysis criteria represented by a hexagon. Analysis of A and B are parametric: various criteria may be used to perform various analyses of both A and B.

## 7.2 Analysis Criteria Definition

As presented in Section 3.2, a classification groups domain objects according to their similarity. The concept of promixity can be considered as the similarity between items. The more two items are similar, the closest their are.

An analysis criterion is a metric on a given analysis domain. Formally:

**Definition 7.2.1.** *A function AC is an* analysis criterion *iff*

$$\left\{\begin{array}{lll} \multicolumn{3}{l}{AC \text{ is a function from } AD^2 \text{ to } \mathbb{R}^+} \\ \forall (x,y) \in AD^2, & x = y & \Leftrightarrow AC(x,y) = 0 \\ \forall (x,y) \in AD^2, & AC(x,y) & = AC(y,x) \\ \forall (x,y,z) \in AD^3, & AC(x,y) & \leq AC(x,z) + AC(y,z) \end{array}\right.$$

Analysis criteria are a subset of transformer functions defined in Section 6.2, so they are ADFs. They can, therefore, be defined in ADL.

An analysis criterion is an ADF with two analysis domain, each of them containing only one domain object. The resulting domain contains only one domain object modeling the concept of distance between two domain objects.

Formally, let *AC* denote an analysis criterion on an analysis domain *AD*. The type of *AC*, as well as *AD*, is denoted *typeAC*. Let $do_1$ and $do_2$ denote two domain objects of *AD*. Let $r \in \mathbb{R}^{+*}$ denote the distance between $do_1$ and $do_2$ according to *AC*. Let $AD_r$ denote the resulting analysis domain.

For *AC*, two origin analysis domains $AD_1$ and $AD_2$ are defined by $AD_1 = \{do_1\}$ and $AD_2 = \{do_2\}$. The analysis criterion *AC* may be defined as follows, according to the XML Schema defining the structure of function definition documents (see Appendix A.2):

```
<function-def name="AC"
  xmlns:core="http://nessy.pl/adl/core">

  <param  name="first"  type="typeAC"/>
  <param  name="second" type="typeAC"/>
  <result name="result" type="typeAC_distance"/>

  <processing>
  ...
  </processing>

</function>
```

$AD_r$ contains only one domain object $do_r$. The type of $do_r$ — which is also the type of $AD_r$ — is *typeAC*_`distance`. Domain objects of type *typeAC*_`distance` may be defined as follows, according to the XML Schema defining the structure of metaObject definition documents (see Appendix A.4):

```
<object-def type="typeAC_distance">
  <id type="core:Integer"/>
  <attributes>
    <attribute name="first"    type="typeAC"/>
    <attribute name="second"   type="typeAC"/>
    <attribute name="distance" type="Float"/>
  </attributes>
</object-def>
```

Processing actions to be performed to calculate the distance between $do_1$ and $do_2$ are defined by calls to functions or tags, like for every ADF. The user defining new analysis criterion has to check whether the set of processing actions she/he uses to define the processing of the analysis criterion defines a metric. The ADF compiler does not perform any checking of the conditions given in Definition 7.2.1.

## 7.3 Classifications

Given an analysis domain and an analysis criterion that are compatible, a classification can be processed. A classification is the result of the analysis process. The structure of the classification is based on the inter-class distance. Classifications allow analysis domains to be partitioned at various level of granularity by the threshold operation.

### 7.3.1 Classification Structure

A classification is defined in terms of *classes* and *inter-classes distances*. A class may contain either other classes and an inter-class distance, or one or more metaObjects (with an inter-class distance always equal to 0). More formally, let $C$ denote a classification on domain analysis $DA$. $C$ is a set of classes denoted $c_i$. Classes are formally defined as follows:

$$\forall c_i \in C, \begin{cases} c_i = (\{c_j\}, d_i), \text{ where } \forall j, c_j \in C, d_i \in \mathbb{R}^{+*} \\ \text{or} \\ c_i = \{MO_j\}, \text{ where } \forall j, MO_j \in DA \end{cases}$$

$$\forall (c_i, c_j) \in C^2, c_i \subset c_j \text{ or } c_i \supset c_j \text{ or } c_i \bigcap c_j = \emptyset$$

Two kinds of classes exist. Some classes – denoted *atomic classes* – contains only metaObjects. Others – denoted *complex classes* – contains only classes and an inter-class distance.

The classification is an indexed hierarchy under the condition that an additional constraint is set on inter-class distances:

$$\forall (c_i, c_j) \in C^2 \text{ such as } c_i \neq c_j,\ c_i \subset c_j \Rightarrow d_i < d_j\,,$$

where $d_i$ (respectively $d_j$) is the inter-class distance associated with $c_i$ (respectively $c_j$) and the inter-class distance of atomic classes equals 0.

Let a distance $D$ on $C$ be defined as follows:

- $c_i \subset c_j \Rightarrow D(c_i, c_j) = d_j$ ,
- if $c_i \bigcap c_j = \emptyset$, let $c_k \in C$ be such that $c_i \subset c_k$, $c_j \subset c_k$, and $\forall c_l \in C, c_l \subset c_k \Rightarrow \neg(c_i \subset c_l \text{ and } c_j \subset c_l)$. Then $D(c_i, c_j) = d_k$.

**Lemma 7.3.1.** *In an indexed hierarchical classification, the inter-class distance $d_i$ associated with class $c_i$ observes $D(c_j, c_{j'}) \leq d_i$, where $c_j \subset c_i$ and $c_{j'} \subset c_i$.*

**Proof.**

If $c_j \subset c_{j'}$, $D(c_j, c_{j'}) = d_{j'}$. As $c_j \subset c_i$, $d_j \leq d_j$. Therefore $D(c_j, c_{j'}) \leq d_i$.

If $c_{j'} \subset c_j$, a similar reasoning may be used.

If $c_j \bigcap c_{j'} = \emptyset$, there exists $c_k \in C$ such that $c_j \subset c_k$, $c_{j'} \subset c_k$, and $\forall c_l \in C, c_l \subset c_k \Rightarrow \neg(c_j \subset c_l \text{ and } c_{j'} \subset c_l)$. By definition, $D(c_j, c_{j'}) = d_k$. Or, as $c_i \in C$, $c_j \subset c_i$, and $c_{j'} \subset c_i$, by definition of $c_k$, $c_k \subset c_i$ . Then $d_k \leq d_i$. As a conclusion, $D(c_j, c_{j'}) = d_k \leq d_i$. □

### 7.3.2 Classification Generation

Classification generation is an operation that, given an analysis domain and a compatible analysis criterion, generates a classification. Classification generation is based on ultrametric automatic hierarchical classification algorithm (cf. Section 3.2.3).

An analysis criterion is not an ultrametric. Analysis criteria only have to be metrics. However, a path metric – which is an ultrametric – can be derived from each metric (cf. Section 3.2.4). The path metric derivation is an operation whose computational complexity in terms of processing is very high. When the number of domain objects of an analysis domain is $n$, the number of possible paths between two domain objects has a complexity $O(n!)$.

**Proof.**

Consider an analysis domain $AD$ such that $\mathrm{card}(AD) = n+2$. Let $do_1\}$ and $do_2$ denote two domain objects in $AD$. Let $P_{do_1,do_2}$ denote the set of paths from $do_1\}$ to $do_2$. Then,

$$\mathrm{card}(P_{do_1,do_2}) = n! \sum_{p=0}^{n-1} \frac{1}{(n-p)!}$$

As $\lim_{n\to\infty} \sum_{p=0}^{n-1} = e-1$,

$$\lim_{n\to\infty} \mathrm{card}(P_{do_1,do_2}) = \lim_{n\to\infty} n! \sum_{p=0}^{n-1} \frac{1}{(n-p)!} \simeq n!(e-1)$$

Therefore, the number of possible paths between two domain objects has a complexity $O(n!)$. □

We conclude that the path metric derivation operation cannot be performed directly because of the high complexity of the algorithm.

Another solution to derive ultrametric from metric is based on characteristics of ultrametrics:

**Lemma 7.3.2.** *In an ultrametric space, all triangles are isosceles.*

**Proof.**

Let $\delta$ denote an ultrametric on a space $A$. Let assume that three elements of $A$ exist, $a$, $b$, and $c$, such that the triangle $(a,b,c)$ is not isosceles. Assume that $\delta(a,b) < \delta(b,c) < \delta(a,c)$ — other cases are equivalent to this one by permutation of $a$, $b$, and $c$. Therefore, the condition $\delta(a,c) \leq \sup[\delta(a,b),\delta(b,c)]$ is not observed. As a conclusion, the hypothesis that a non isosceles triangle may exist is false. □

Using this characteristics of ultrametrics, we define the “isoscelization” operation. The “isoscelization” operation transforms every triangle in an isosceles triangle in which the three sides $s_1$, $s_2$, and $s_3$ observe the following clause: $\forall(i,j,k) \in [1,2,3]^3$, $s_i \leq \sup[s_j,s_k]$.

**Definition 7.3.1.** *The “isoscelization” operation transforms*

*a triangle* $(a,b,c)$ *such that* $\begin{cases} \delta(a,b) = s_1 \\ \delta(a,c) = s_2 \\ \delta(b,c) = s_3 \\ s_1 \leq s_2 \leq s_3 \end{cases}$ *into a triangle* $(a,b,c)$*, where* $\begin{cases} \delta(a,b) = s'_1 = s_1 \\ \delta(a,c) = s'_2 = s_2 \\ \delta(b,c) = s'_3 = s_2 \end{cases}$.

The "isoscelization" operation is illustrated in Figure 7.2. The original triangle is the one on the left side. The transformed triangle is the one on the right side. The original triangle is a scalene triangle. After modification, the longest side – $s_3$ in the figure – is altered so that its length equals to the length of side $s_2$. Side $s_2$ is the side whose length is between $s_1$ – the smallest side – and $s_3$ – the longest side.

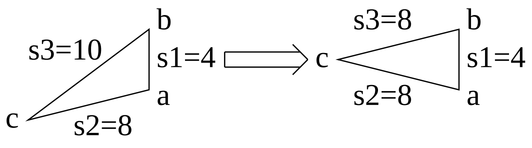


Figure 7.2: The "isoscelization" operation.

If the "isoscelization" operation is performed on all the triangles existing in a space $A$ measurable with distance $d$, all the triangles will be isosceles. A new distance $\delta$ may be then defined on $A$ as follows: the distance between two elements of $A$ — denoted $a_1$ and $a_2$ — is the length of the segment $a_1a_2$ (in the space in which all triangles are isosceles).

It is worth to emphasize that the new distance $\delta$ obtained as explained above is an ultrametric. The proof is obvious as the constraints defining an ultrametrics, defined in Section 3.2.3, are observed. Constraints 3.6 and 3.7 come directly from properties of the metric $d$, and constraint 3.8 is ensured by the "isoscelization" operation.

The complexity of the ultrametric processing based on the "isoscelization" operation is smaller than the processing based on path metrics. The complexity of the proposed algorithm is $O(n^3)$.

**Lemma 7.3.3.** *Given a space A such that $card(A) = n$, the number of triangles existing in A equals* $\frac{n(n-1)(n-2)}{6}$.

**Proof.**

A triangle consists of three points: $a$, $b$, and $c$. There are $n$ possibilities for the choice of $a$. There are then $n-1$ possibilities for the choice of $b$. Finally, there are $n-2$ possibilities for the choice of $c$. So, there are $n(n-1)(n-2)$ triplets in $A$. Each triangle is counted six times with permutations: $(a,b,c)$ is the same triangle as $(a,c,b)$, etc. Therefore, the number of triangle in $A$ equals $\frac{n(n-1)(n-2)}{6}$. □

The following algorithm may be used to perform the "isoscelization" of space $A$:

```
segments = orderedListOfAllSegments();
isoSegments = new List();
nonIsoSegments = orderedListOfAllSegments;
while (nonIsoSegments.size() !=0 ){
  [a,b]=nonIsoSegments.firstElement();
  for-each $c \in A, c \neq a$ and $c \neq b$ {
    isoscelization(a,b,c);
  }
  isoSegments.add([a,b]);
  nonIsoSegments.remove([a,b]);
  nonIsoSegments.sort();
}
```

The complexity of the presented algorithm is $O(n^3)$. The number of segments in a space whose cardinal is $n$ equals $\frac{n(n-1)}{2}$. For each segment, the loop defined between lines 6 and 8 is executed $n-2$ times. Therefore, the algorithm complexity is $\frac{n(n-1)}{2} \times (n-2) \simeq O(n^3)$. □

### 7.3.3 Threshold Operation

Lemma 7.3.1 allows to define a *threshold* operation on classifications. The threshold operation provides various partitions of the analyzed analysis domain according to a threshold.

Two additional concepts are required to define the threshold operation: the concept of *class contents* and the concept of *t-max class.*

**Definition 7.3.2.** *The contents of a given class $c_i$ of a classification $C$ is a set of domain objects, denoted $Contents(c_i)$.*

- *For atomic classes, the class contents is the class itself, i.e. $Contents(c_i) = c_i = \{MO_j,$ where $\forall j, MO_j \in DA\}$.*
- *For complex classes, the class contents is the union of contents of all embedded classes, i.e. $Contents(c_i) = \bigcup_j Contents(c_j)$ with the notations used in Section 7.3.1.*

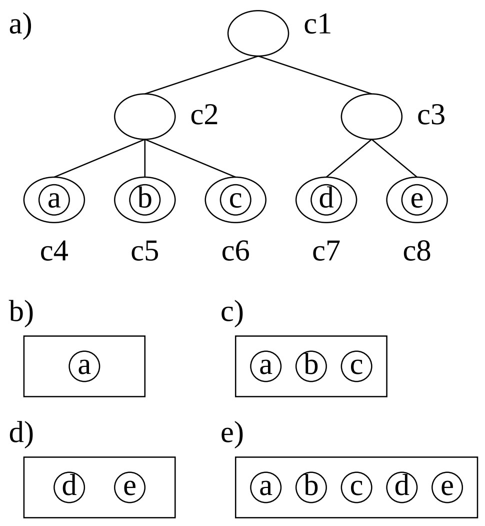


Figure 7.3: The *class contents* operation. a) the full classification $C$; b) $Contents(c_4)$; c) $Contents(c_2)$; d) $Contents(c_3)$; e) $Contents(c_1)$

The class contents operation is illustrated in Figure 7.3. The full classification $C$ is presented in a). Domain objects are represented by circles. Five domain objects exist in $C$, denoted $a$, $b$, $c$, $d$, and $e$. Classes are represented by ellipses. Eight classes exist, denoted $c_i$, where $i \in [1,\dots,8]$. The "is-embedded-in" relationship is represented by lines between classes. Classes $c_2$ and $c_3$ are, for instance, embedded in $c_1$. For atomic classes ($c_i$, where $i \in [4,\dots,8]$), $Contents(c_i)$ is the set of domain objects contained in $c_i$. $Contents(c_4)$ is therefore $\{a\}$. When a class is a complex class, the class contents is a set of object domains which is calculated recursively with the $Contents()$ function. Therefore, $Contents(c_2) = Contents(c_4) \cup Contents(c_5) \cup Contents(c_6) = \{a,b,c\}$.
$Contents(c_3) = Contents(c_7) \cup Contents(c_8) = \{d,e\}$. Finally,
$Contents(c_1) = Contents(c_2) \cup Contents(c_3) = \{a,b,c,d,e\}$.

**Definition 7.3.3.** *Given $t \in \mathbb{R}^{+*}$, a class $c_i$ is a t-max class iff*

$$d_i \leq t \text{ and } \neg(\exists c_j \text{ such that } c_i \subset c_j \text{ and } d_j \leq t)$$

In the classification presented in Figure 7.4, classes are represented by ellipses, distances associated with classes are given inside ellipses, "is-embedded-in" relationship is represented by lines between ellipses. In this classification example, $c_2$ is a 3-max class: the distance $d_2$ associated with $c_2$ observes $d_2 = 3 \leq t = 3$, and there is no class $c_j$ such that $c_2 \subset c_j$ and $d_j \leq 3$ (the only class that contains $c_2$ is $c_1$, but $d_1 = 7$). Class $c_3$ is also a 3-max class: the distance $d_3$ associated with $c_3$ observes $d_3 = 2 \leq t = 3$, and there is no class $c_j$ such that $c_3 \subset c_j$ and $d_j \leq 3$ (the only class that contains $c_3$ is $c_1$, but $d_1 = 7$).

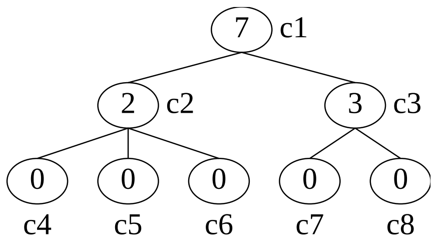


Figure 7.4: Classification example

**Definition 7.3.4.** *Given a threshold $t$, the threshold operation $T_t$ creates a partition $P_t$ of a classification $C$. $P_t$ is the set of contents of all t-max classes of $C$. Formally,*

$$P_t = \{Contents(c_i),\ where\ c_i\ is\ a\ t - max\ class\}$$

Various threshold operation results are illustrated in Figure 7.5. When $t = 0$, the partition obtained by the threshold operation is a set of atomic classes. With the classification given in Figure 7.5 a), $P_0 = P_1$ because all 0-max classes are all 1-max classes. Similarly, $P_5 = P_7$ because all 5-max classes are 7-max classes.

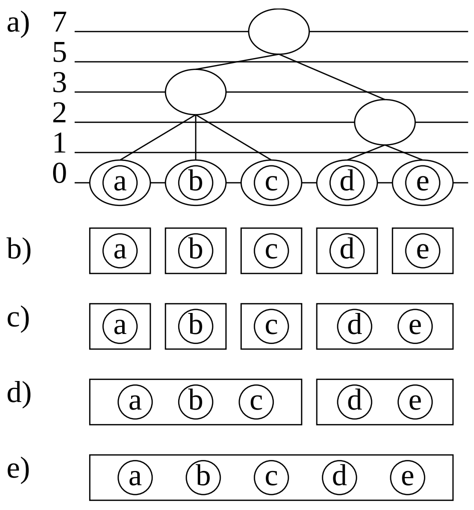


Figure 7.5: The threshold operation. a) Classification; b) $P_1$; c) $P_2$; d) $P_3$; e) $P_7$

The threshold provides the granularity of the obtained partition. The higher the threshold is, the lower the number of classes in the obtained partition is. In the context of negotiation analysis, this characteristics of the threshold operation is a key feature as it allows:

- various analysis levels; when the threshold is low, the generated partition consists of many classes, representing a fine-grained analysis. When the threshold is high, the generated partition consists of a few classes, representing a high-level analysis, giving an overview of the analyzed negotiation facet;
- fast focusing on details; starting from a high-level analysis, a negotiator can select a few classes in the partition which are of special interest. These classes can further be analyzed in details by the application of a threshold operation with a lower threshold. The repetition of this technique allows to focus quickly on interesting details.

CHAPTER 8

# APPLICATION OF DYNAMIC ANALYSIS TO e-NEGOTIATIONS

## 8.1 Negotiation Process

In this section, an example of the application of dynamic analysis to e-negotiations is presented. Though the proposed dynamic analysis deals with mass negotiations, the number of negotiators participating in the negotiation process presented in this section is reduced to 3 to improve readability.

In the following example, three contractors are negotiating a contract concerning the publication of a story on the web site of an Internet-based publishing company. The contract to be negotiated is a typical publishing agreement. Its original version, proposed by the publisher, is the following:

Begining of contract

1 This contract is made between Scott Tiger, whose address is 8i Ora Road, Redwood Shores, CA, USA hereinafter referred to as the PUBLISHER, and Wile E. Coyote and Chuck Jones whose address is 10 Acme Bvd, Hollywood, CA, USA, hereinafter referred to as the AUTHORS.

2

3 The parties agree as follows:

4

5 Authors' Grant.

6 1. The Authors grant permission to include their story entitled "The life of the Roadrunner," a work of approximately 10500 words, hereinafter referred to as the Work, on the ACME Publishing Corporation web site found at www.acme-publishing.com.

7

8 Rights Purchased.

9 1(a). This use of the Work by the Publisher entails the assignment of First World Wide Web Rights, for publication in the English language, for a period limited to one year from the first publication. For the first three months of this period, the publication of the Work shall be exclusive to the Publisher. It is also understood and agreed that the Publisher may use this Work only within the above-mentioned sites and that all rights not expressly granted herewithin reside exclusively with the Authors.

10

11 Options on Further Rights

12 2. If the Publisher desires to exercise any further rights, separate agreements must be entered to and signed for each use of further rights.

13 2(a). The Author grants to the Publisher the right of firstoffer on world anthology rights. The Author also grants to the Publisher the right to make an offer on non-exclusive reprint rights.

14

15 Payments and Royalties.

16 3. For the rights granted to the Publisher above in 1(a) the Authors will receive a payment in the sum of $2500, which will be paid within thirty (30) days of signing this agreement.

17

18 Changes in Text or Title.

19 4. The Publisher will make no major alterations to the Work's text or title without the Author's written approval in e-mail or hardcopy. The Publisher reserves the right to make minor copy-editing changes to conform the style of the text to its customary form and usage. To ensure that no such changes are made without the Authors' approval, the Publisher will furnish the Authors with electronic text proofs or page proofs of the Work in advance of publication. Authors agree to return such proofs with corrections in not more than thirty (30) days from receipt thereof.

20

21 Reversion of Rights and Withdrawal of Offer to Publish.

22 5(a). In the event that the Work is not published within 18 months of signing of this agreement, all rights revert to the Author, and the Author has the right to sell or arrange for publication of the above-named Work in any manner. The Author shall keep any payments made by the Publisher to him.

23 5(b). In the event that a copy of the counter-signed agreement is not returned to the Author within thirty (30) days of signing by the Author, or that payment in 3(a) is not made as specified, the Publisher's offer to purchase the Work shall be considered withdrawn.

24

25 Copyright.

26 6. The Publisher agrees to list a proper copyright notice for the Work in the name of the Authors on the first page of the web-published story, and to take all necessary steps to protect the Authors' copyright in the United States, and in the International Copyright Union.

27

28 Authors' Credit.

29 7. The Authors will be credited on the table of contents page and at the beginning of the story as Wile E. Coyote and Chuck Jones.

30

31 Venue.

32 8. This agreement shall be deemed executed under the laws of the state of California. California state law shall be the applicable law of this agreement.

33

34 The parties acknowledge that each party has read and understood this contract before execution.

End of contract

Negotiators are denoted by their initials, i.e. *ST* for Scott Tiger, *WC* for Wile E. Coyote, and *CJ* for Chuck Jones, respectively. Contract paragraphs are denoted by $p_i$, where $i$ is the number in the black box on the left side of the paragraph, e.g. the paragraph whose text is "The parties agrees as follows:" is refered as $p_3$.

The negotiation process consists of two phases. First, negotiators ST and WC negotiate with no involvement of negotiator CJ. Second, all three negotiators negotiate together.

During the negotiation process, five conflicts occur. The first conflict arises around the clause concerning payments and royalties, i.e. $p_{16}$. The payment delay is the most important topics for negotiator *WC*, who wants to be paid soon. As a consequence, he always proposes shorter payment delay. The payment amount is not so important for him, but he does not aggree with less than $2500. The payment amount is the most important topics for negotiator *CJ*. Negotiator *CJ* wants to get as much money as possible for the story (his lower limit is $2800). Therefore he proposes higher royalties, offering in exchange longer payment delays. Negotiator *ST* understands the two opposite strategies of both authors. As payment fees and delay are not very important for him, he proposes longer delays or lower fees and lets *CJ* and *WC* react to his propositions. The final proposition is made by *WC*: $2900 with a payment delay of twenty-five days.

The second conflict concerns the clause about withdrawal of offer to publish, i.e. $p_{23}$. The conflict arises around the delay for counter-signed agreement deliverable. This delay is closely related to the payment delay. Therefore, changes concerning the delay payment causes modification of the clause $p_{23}$.

Another conflict concerns the credits to authors, i.e. $p_{29}$. In the initial version of the contract, clause $p_{23}$ states that "The Authors will be credited[...] as Wile E. Coyote and Chuck Jones". Negotiator *CJ* wants his name to be first as he is the main autor of the story. Moreover, *CJ* and *WC* are members of an autors' group named "the WB funny group". Therefore *CJ* proposes in version *0.3* "[...] as Chuck Jones and Wile E. Coyote, from the WB funny group". Negotiator *WC* agrees with the credits to the WB funny group but wants his name to be the first, as he brought his experience to the story. Therefore, he proposes in version *0.3.2* "[...] as Wile E. Coyote and Chuck Jones, from the WB funny group". Negotiator *CJ* stops this conflict when he proposes to credit only the WB funny group in version *0.3.2.1*.

Another conflict concerns the number of words in the story, i.e. $p_6$. The initial proposition is "10500 words". Negotiator *WC* proposes a higher number of words ("15000") in version *0.1*. The proposed payment delay — $p_{16}$ — is shorten from thirty days to fifteen. Negotiator *ST* proposes in version *0.1.1* a lower number of words ("10000") with a longer payment delay (twenty-five days). Negotiator *ST* proposes 11500 words for $28000 in version *0*.2. Negotiator *CJ* proposes "12000" words for $3500 in version *0.3* . Negotiator ST disagrees with this proposition and his counter-offer is 12500 words for $2200 (version 0.3.1). This proposition is not accepted. Finally, the proposition made by *CJ* in version *0.3* is accepted by all negotiators.

Another conflict concerns the modification the Publisher may make to the story, i.e $p_{19}$. Negotiator *WC* disagrees with the original proposition of *ST*, which states that the "Publisher will make no major modification [...]". The original proposition is not clear as "minor modifications" are concerned. Therefore, in version *0.1, WC* proposes that the "Publisher will not make any modification [...]", which removes all ambiguity concerning "minor modifications". In version *0.1.1*, negotiator *ST* accepts this modification under the condition that the delay concerning text proofs is shortened from thirty days to fifteen days. In version *0.3*, negotiator CJ proposes that the "Publisher will not make any modification [...]" and the delay concerning text proofs is shortened to twenty days. This proposition is accepted by all negotiators.

The detailed course of the negotiation process is given in Appendix C.

## 8.2 Contract Evolution

A representation of the contract evolution consists of:

- the contract version tree,
- the association table of structure, and
- the association tables of contract paragraphs.

The contract version tree is presented in Figure 8.1. The two phases described in Section 8.1 are visible in the version tree as two independant branches: the branch starting with version *0.1* concerns the discussion between Scott Tiger and Wile E. Scott, while the branch starting with version *0.3* concerns the discussion in which all three negotiators are involved.

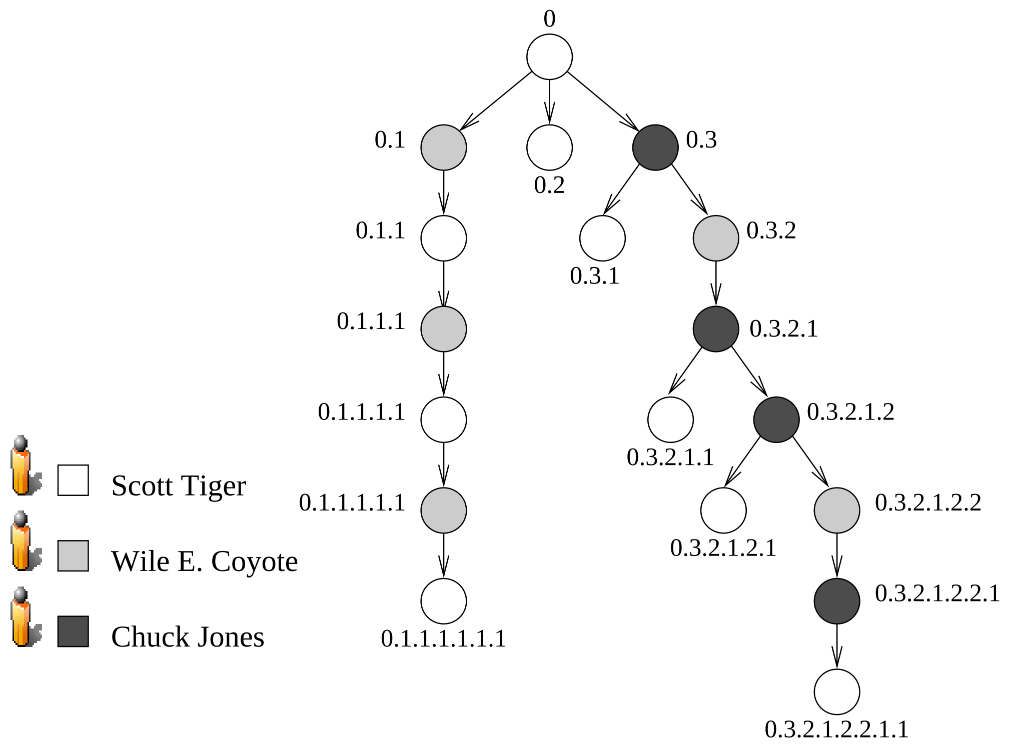


Figure 8.1: Contract version tree

The association table of contract structure is simple as no paragraph has been added or deleted during the negotiation process. Therefore, the multiversion structure consists of one instance, denoted $s_0$. The association table is presented in Table 8.2.

The association tables of contract paragraphs which were not modified are simple, consisting of a single row (*paragraph_instance_ID, version 0)*. For paragraphs that were modified during the negotiation process, their association tables are presented in Tables 8.3, 8.4, 8.5, 8.6, and 8.7.

| Contract structure instance | Contract version |
|---|---|
| $s_0$ | *0* |

Table 8.2: Association table of contract structure

| Paragraph instance identifier | Contract version | Contents overview |
|---|---|---|
| 6 | *0* | 10500 words |
| 35 | *0.1* | 15000 words |
| 39 | *0.1.1* | 10000 words |
| 50 | 0.2 | 11500 words |
| 53 | 0.3 | 12000 words |
| 58 | 0.3.1 | 12500 words |

Table 8.3: Association table of paragraph $p_6$

| Paragraph instance identifier | Contract version | Contents overview |
|---|---|---|
| 16 | *0* | $2500, 30 days |
| 36 | *0.1* | $2500, 15 days |
| 40 | *0.1.1* | $2800, 25 days |
| 43 | *0.1.1.1* | $2700, 20 days |
| 45 | *0.1.1.1.1* | $2500, 20 days |
| 47 | *0.1.1.1.1.1* | $2500, 14 days |
| 49 | *0.1.1.1.1.1.1* | $2500, 18 days |
| 51 | *0.2* | $2800, 30 days |
| 54 | *0.3* | $3500, 15 days |
| 59 | *0.3.1* | $2200, 45 days |
| 61 | *0.3.2* | $3000, 10 days |
| 64 | *0.3.2.1* | $3200, 20 days |
| 66 | *0.3.2.1.1* | $2600, 30 days |
| 67 | *0.3.2.1.2* | $3500, 30 days |
| 68 | *0.3.2.1.2.1* | $2000, 15 days |
| 69 | *0.3.2.1.2.2* | $2900, 25 days |

Table 8.4: Association table of paragraph $p_{16}$

| Paragraph instance identifier | Contract version | Contents overview |
|---|---|---|
| 19 | *0* | no major alterations; 30 days |
| 37 | *0.1* | any alterations; 30 days |
| 41 | *0.1.1* | any alterations; 15 days |
| 55 | *0.3* | any alterations; 20 days |

Table 8.5: Association table of paragraph $p_{19}$

| Paragraph instance identifier | Contract version | Contents overview |
|---|---|---|
| 23 | *0* | 30 days |
| 38 | *0.1* | 10 days |
| 42 | *0.1.1* | 25 days |
| 44 | *0.1.1.1* | 18 days |
| 46 | *0.1.1.1.1* | 20 days |
| 48 | *0.1.1.1.1.1* | 7 days |
| 52 | *0.2* | 21 days |
| 56 | *0.3* | 15 days |
| 60 | *0.3.1* | 28 days |
| 62 | *0.3.2* | 8 days |

Table 8.6: Association table of paragraph $p_{23}$

| Paragraph instance identifier | Contract version | Contents overview |
|---|---|---|
| 29 | *0* | Wile E. Coyote and Chuck Jones |
| 57 | *0.3* | Chuck Jones and Wile E. Coyote, from the WB funny group |
| 63 | *0.3.2* | Wile E. Coyote and Chuck Jones, from the WB funny group |
| 65 | *0.3.2.1* | the WB funny group |

Table 8.7: Association table of paragraph $p_{29}$

## 8.3 Multi-facet Analysis

In this section, three examples of multi-facet analysis are presented. First, the number of paragraph versions are classified to identify conflicts. Then, the number of paragraph versions created by negotiator *ST* are classified to identify the clauses *ST* is interested in. Finally, versions of clause $p_{16}$ will be classified to identify the key proposals concerning payment amount and delay.

A classification of the multiversion paragraphs according to their versions may be used to identify conflicts that occured during the negotiation process. MetaObjects modeling the number of versions of paragraphs, denoted *AssociationTableSummaryMetaObjects*, consist of:

| Attribute name | Type | Description |
|---|---|---|
| `ID` | `core:Integer` | The identifier of the multiversion paragraph |
| `numberOfVersions` | `core:Integer` | The number of versions of the paragraph |
| `contents` | `core:String` | The contents of the last paragraph instance |

AssociationTableSummaryMetaObjects are generated by an ADF denoted *createAssociationTable-Summary*. This ADF retrieves the list of multiversion paragraphs from the database storing the multiversion contract. Then, the association table of each multiversion paragraph is retrieved and the number of version is processed. The ADF assembles the multiversion identifier, the processed number of versions, and the contents of the last paragraph instance into an AssociationTableSummaryMetaObject.

AssociationTableSummaryMetaObjects are classified according to the criterion *associationTable-*

*SummaryDistance.* This criterion, denoted $d_{summary}$ is defined as follows:

- $\forall(as_i, as_j),\ as_i = as_j \Leftrightarrow d_{summary}(as_i, as_j) = 0$
- $\forall(as_i, as_j),\ as_i \neq as_j \Leftrightarrow d_{summary}(as_i, as_j) = 1 + |nv_i - nv_j|,$

where $nv_i$ is the value of the numberOfVersions attribute of the associationTableSummaryMetaObject $as_i$.

The classification obtained by the *createAssociationTableSummary* ADF and the *associationTableSummaryDistance* is illustrated in Figure 8.2.

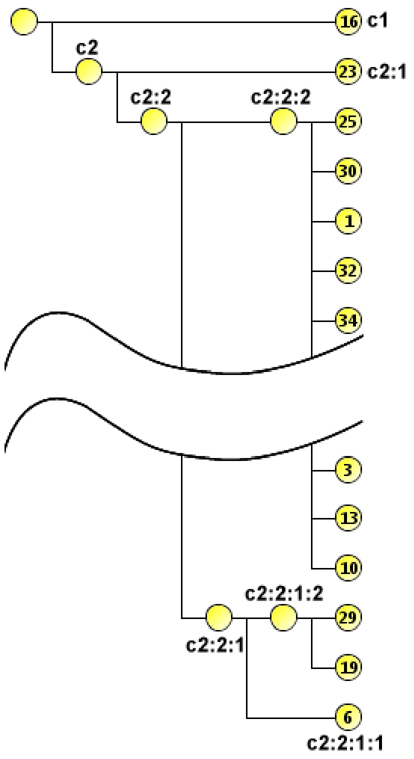


Figure 8.2: Classification of paragraphs according to their number of versions

Each circle in Figure 8.2 represents a class in the classification. Classes on the right side are atomic and therefore contains only one associationTableSummaryMetaObject. They are denoted by the identifiers of the multiversion paragraphs. At a high level of granularity, two classes exists: class $c_1$ contains only the paragraph $p_{16}$, while class $c_2$ contains all the other paragraphs. Sixteen versions of paragraph $p_{16}$ exist; therefore $p_{16}$ is the main cause of conflict. In class $c_2$, two classes are distinguished: class $c_{2:1}$ contains only paragraph $p_{23}$, while class $c_{2:2}$ contains the remaining paragraphs. Ten versions of paragraph $p_{23}$ exists. Paragraph $p_{23}$ is therefore the second cause of conflict. Class $c_{2:2}$ may be further splitted in two classes: class $c_{2:2:1}$ contains paragraphs $p_{29}$, $p_{19}$, and $p_6$, and class $c_{2:2:2}$ contains all paragraphs which have never been modified. Class $c_{2:2:1}$, can be splitted into next two classes: class $c_{2:2:1:1}$ that contains paragraph $p_6$, and class $c_{2:2:1:2}$ that contains paragraphs $p_{29}$ and $p_{19}$. Six versions of paragraph $p_6$ exist, while four version of paragraphs $p_{29}$ and $p_{19}$ exist.

The classification obtained with the *createAssociationTableSummary* ADF and the *associationTableSummaryDistance* allows negotiators to identify conflicts. At the higher level of granularity, one may distinguish the paragraph which is the main source of conflict from the other paragraphs, i.e. $p_{16}$. The paragraph concerning the payment amount and delay is indeed the most negotiated paragraph. The second source of conflict, i.e. $p_{23}$, may be identified at a lower level of granularity. The paragraph concerning the reversion of rights and withdrawal of offer to publish is the second most negotiated paragraph. All conflicts can be identified at various levels of granularity: the conflict concerning the number of words, i.e. $p_6$, is visible in class $c_{2:2:2:1}$, while conflicts concerning the credits to authors, i.e. $p_{29}$, and changes in the text or title, i.e. $p_{19}$, may be identified in class $c_{2:2:1:2}$.

A classification of the multiversion paragraphs according to their versions created by negotiator *ST* may be used to identify the clauses *ST* is interested in. MetaObjects modeling the number of versions of paragraphs are the same as the one used in the previous classification.

AssociationTableSummaryMetaObjects are generated by an ADF denoted *createAssotiationTableSymmaryForST*. This ADF retrieves the list of multiversion paragraphs that *ST* has been created from the database storing the multiversion contract. Then, the association table of each multiversion paragraph is retrieved and the number of version is processed. The ADF assembles the multiversion identifier, the processed number of version, and the contents of the last paragraph instance into an AssociationTableSummaryMetaObject.

AssociationTableSummaryMetaObjects are classified according to the same *associationTableSummaryDistance* criterion used in the previous classification.

The classification obtained with the *createAssociationTableSummaryForST* ADF and the *associationTableSummaryDistance* is illustrated in Figure 8.3.

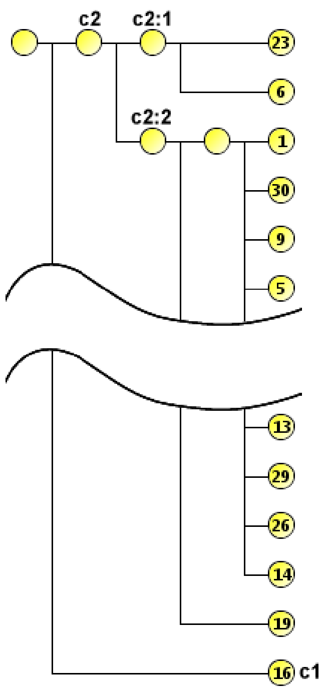


Figure 8.3: Classification of paragraphs according to the number of versions that *ST* has been created.

At a high level of granularity, two classes exists: class $c_1$ contains only the paragraph $p_{16}$, while class $c_2$ contains all the other paragraphs. Eight versions of paragraph $p_{16}$ have been created by

*ST*; therefore $p_{16}$ is of the main interest to *ST*. In class $c_2$, two classes are distinguished: class $c_{2:1}$ contains paragraphs $p_{23}$ and $p_6$ , while class $c_{2:2}$ contains all the remaining paragraphs. Five versions of paragraph $p_{23}$ have been created by *ST*. Four versions of paragraph $p_6$ have been created by *ST*. Paragraphs $p_{23}$ and $p_6$ are therefore the secondary center of interest of *ST*. Finally, paragraph $p_{19}$ is distinguished from other paragraphs in class $c_{2:2}$. Negotiator *ST* has created two versions of paragraph $p_{19}$ which is therefore the last center of interest of *ST.* Briefly, *ST* is interested in:

1. the payment and amount delay,
2. the reversion of rights and withdrawal of offer to publish,
3. the number of words of the story, and
4. changes in text or title.

One may notice that *ST* is not involved in the conflict concerning credits to authors.

A classification of clause $p_{16}$ may be used to identify the key proposals concerning payment amount and delay. MetaObjects modeling the paragraph instances concerning payment, denoted *PaymentMetaObjects*, consist of:

| Attribute name | Type | Description |
|---|---|---|
| `ID` | `core:Integer` | The identifier of the paragraph instance |
| `amount` | `core:Integer` | The payment amount |
| `delay` | `core:Integer` | The payment delay |

PaymentMetaObjects are generated by an ADF denoted *createPayments*. This ADF retrieves the list of paragraph instances concerning the payment from the database storing the multiversion contract. Then, the amount and the delay proposed in each retrieved paragraph instance are processed. The ADF assembles the instance identifier, the processed amount, and the processed delay into a PaymentMetaObject.

PaymentMetaObjects are classified according to the criterion *paymentDistance.* This criterion, denoted $d^{\alpha}_{payment}$ is defined as follows:

- $\forall(p_i,p_j),\ p_i = p_j \Leftrightarrow d^{\alpha}_{payment}(p_i,p_j) = 0$
- $\forall(p_i,p_j),\ p_i \neq p_j \Leftrightarrow d^{\alpha}_{payment}(p_i,p_j) = 1+\alpha\left|a_i - a_j\right| + 200\,(1-\alpha)\left|d_i - d_j\right|,$

where $a_i$ , repectively $d_i$ , is the value of the amount attribute, respectively the delay attribute, of the paymentMetaObject $p_i$, and $\alpha \in [0,1]$. When the ratio $\alpha$ equals to 0, only the delay is taken into account. When the ratio equals to 1, only the amount is taken into account. For values in between 0 and 1, both the delay and the amount are taken into account. The multiplication by 200 is a scaling factor betweem payment amounts and delays.

The classifications obtained with the *createPayments* ADF and the *paymentDistance* for various values of the ratio are illustrated in Figures 8.4, 8.5, and 8.6. Each circle in Figures 8.4, 8.5 and 8.6 represents a class in the classification. Classes at the bottom are atomic and therefore contains only one paymentMetaObject. These classes are denoted by the identifiers of paragraph instances.

As far as the payment amount is concerned, three classes exists at a high level of granularity: class $c_1$ contains paragraph instances *68* and *59.* Class $c_2$ contains paragraph instances *54* and *67.* Class

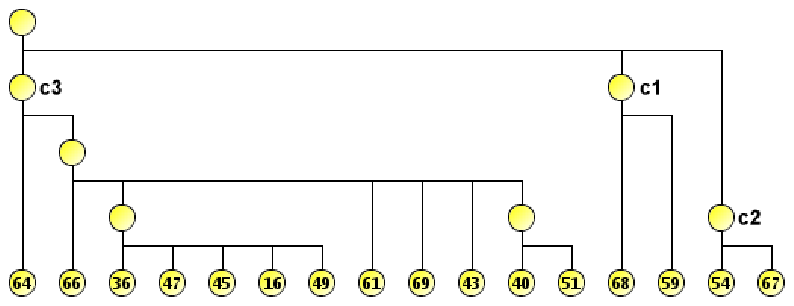


Figure 8.4: Classification of paragraph $p_{16}$ according to the payment amount ($\alpha = 1$)

$c_3$ contains all the other paragraph instances. Low payment amounts are proposed in paragraph instances *68* and *59* — \$2000 and \$2200. The highest payment amount is proposed in paragraph instances *54* and *67* — \$3500. Payment amount proposals in paragraph instances contained in class $c_3$ are in between \$3200 and \$2500. In class $c_3$, paragraph instance *64*, in which payment amount proposal is \$3200, may be distinguished at a lower level of granularity. At the lowest level of analysis, one may notice that the payment amount \$2500 is common for paragraph instances *36*, *47*, *45*, *16*, and *49*. It it the most often proposed payment amount, when the payment delay is not taken into account.

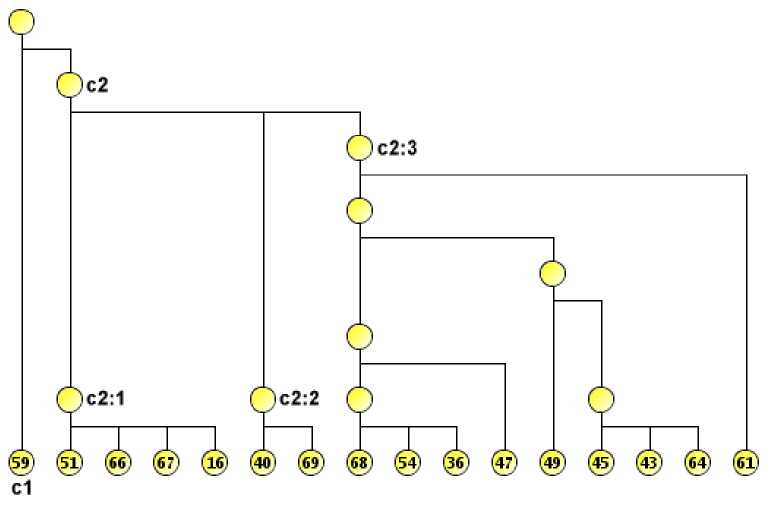


Figure 8.5: Classification of paragraph $p_{16}$ according to the payment delay ($\alpha = 0$)

As far as the payment delay is concerned, two classes exists at a high level of granularity: class $c_1$ contains paragraph instance *59*, while class $c_2$ contains all the other paragraph instances. The payment delay proposed in paragraph instance *59* is the longest in the whole negotiation process — 45 days. In class $c_2$, three classes are distinguished: class $c_{2:1}$ contains paragraph instances *51*, *66*, *67*, and *16*, in which the proposed payment delay is 30 days. Class $c_{2:2}$ contains paragraph instances *40* and *69*, in which the proposed payment delay is 25 days. Class $c_{2:3}$ contains all the remaining paragraph instances, in which the proposed payment delays are in between 10 and 20 days.

As far as the payment delay and amount are concerned, three proposals are distinguished at a high level: proposals in paragraph instances *59* (\$2200, 45 days), *61* (\$3000, 10 days), and *54* (\$3500, 15 days) are excessive. An analysis of the remaining proposals allows to distinguished "low amount - short delay" proposals (paragraph instances *68*, *64*, *43*, *45*, *49*, *47*, and *36*) from "high amount - long delay" proposals (paragraph instances *67*, *66*, *16*, *51*, *40*, and *69*). One may notice that paragraph instance *43* (\$2700, 20 days) is considered as a "low amount - short delay" proposal, while paragraph instance *16* (\$2500, 30 days) is considered as a "high amount - long delay". This may be explained by the fact that the chosen ratio $\alpha = 0.45$ favors payment delay.

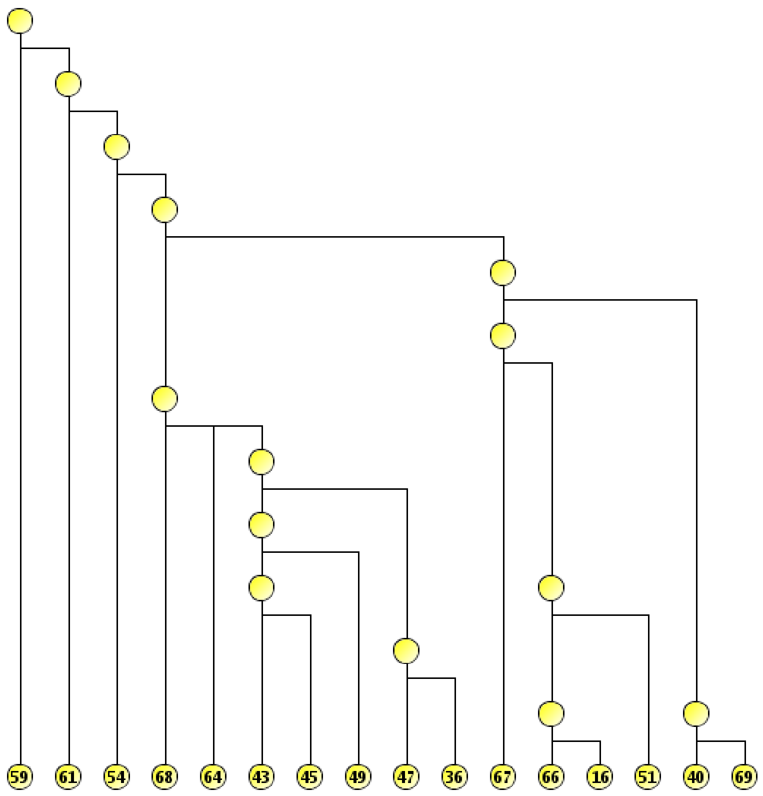


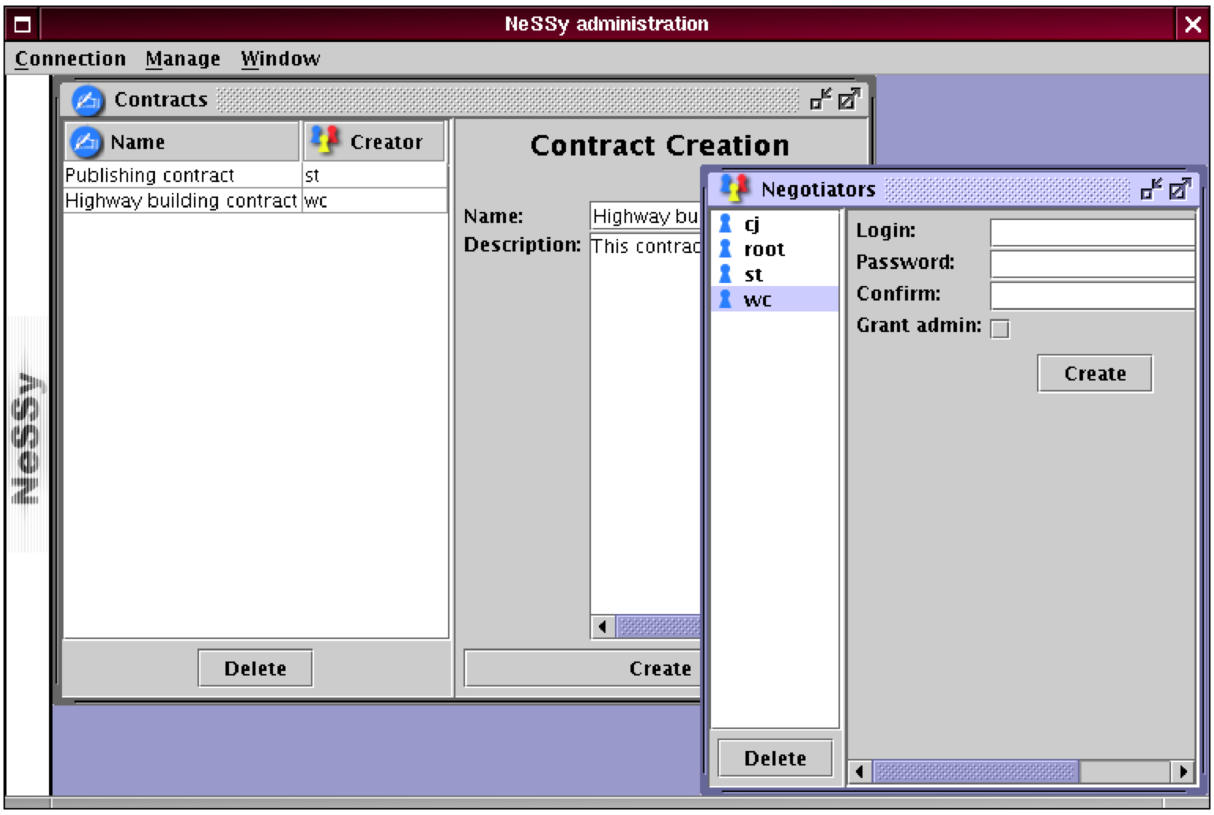
Figure 8.6: Classification of paragraph $p_{16}$ according to the payment delay and amount ($\alpha = 0.45$)

CHAPTER 9

# PROTOTYPE OF E-NEGOTIATIONS SYSTEM

## 9.1 Prototype Description

A prototype named *NeSSy* (for *Negotiation Support System*) has been built to allow negotiators to negotiate and perform multi-facet analysis of e-negotiations. The *NeSSy* prototype allows to manage negotiators and contracts. The *NeSSy* prototype allows also to perform various analysis of the negotiation process.



| Name | Creator |
|---|---|
| Publishing contract | st |
| Highway building contract | wc |

Name: Highway bu
Description: This contrac

cj
root
st
wc

Login:
Password:
Confirm:
Grant admin: ☐

Figure 9.1: Negotiators and contracts management windows in the *NeSSy* prototype

Negotiators management window, presented on the right side of Figure 9.1, allows for negotiator creation and removal. A negotiator is defined by her/his login, and password. A negotiator may be an administrator. An administrator can manage negotiators. The removal of a negotiator causes the removal of all multiversion contracts she/he created. If a negotiator is the owner of a contract version of a contract he/she did not create, the removal of this negotiator is not possible.

Contracts management window, presented on the left side of Figure 9.1, allows for multiversion contract creation and removal. A multiversion contract is defined by its name and its description. A multiversion contract may be deleted only by the negotiator that created it.

When a negotiator creates a multiversion contract, she/he can manage negotiation participants and add the initial contract version (as presented in Figure 9.2).

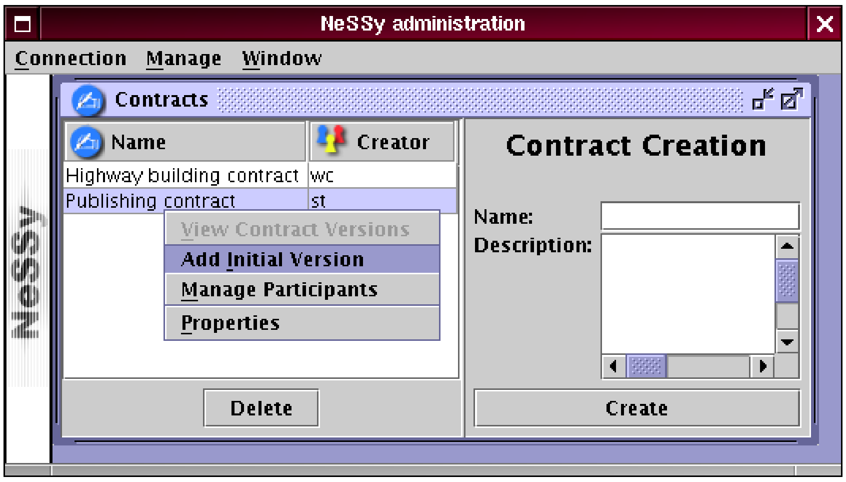


Figure 9.2: The popup menu for newly created multiversion contract

Only the creator of a multiversion contract can manage participants. The “Contract Participants” window, presented in Figure 9.3, allows to add and remove participants from the contract negotiation process. A participant can be removed iff she/he is not the owner of a contract version. A participant may be added at any time during the negotiation process.

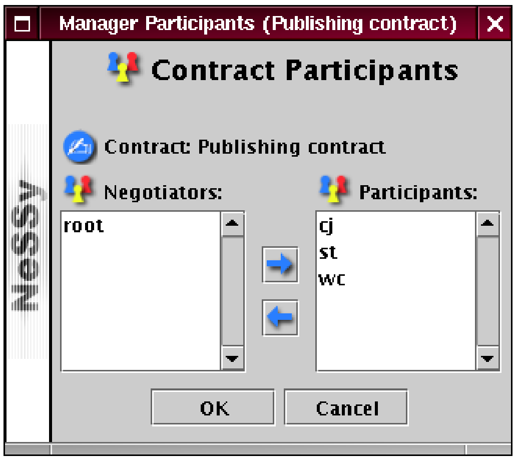
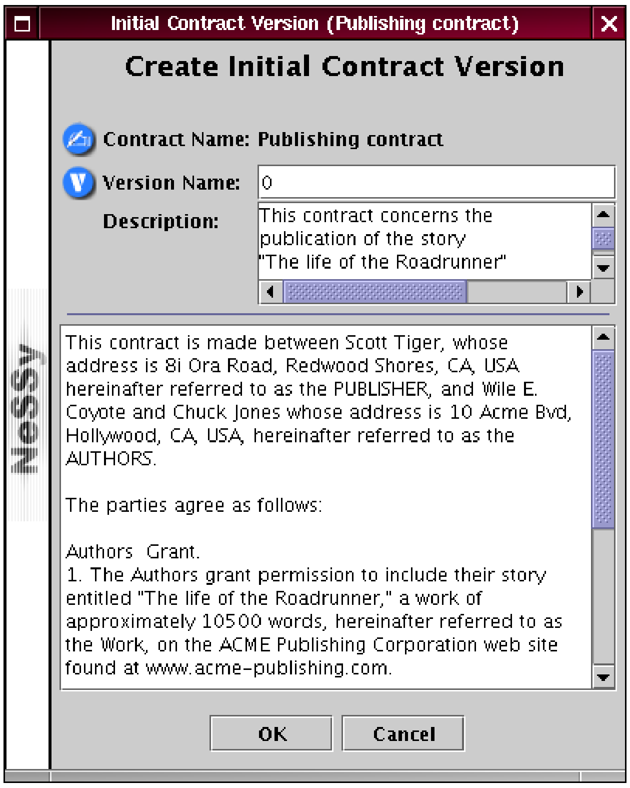

Manager Participants (Publishing contract)
Contract Participants
Contract: Publishing contract
Negotiators:
root
Participants:
cj
st
wc
OK
Cancel
NeSSy

Figure 9.3: Contract participants management window

Only the creator of a multiversion contract can add the initial contract version. The initial contract version can be added with the “Create Initial Contract Version” window, presented in Figure 9.4. The contract initial version is defined by its version name, its version description, and its contents. The version name does not have to follow the rule 5.2.4 which states that “the root contract version subidentifier is 0”. The version name is a label that allows negotiators to easily identify a given contract version.



Create Initial Contract Version

Contract Name: Publishing contract

Version Name: 0

Description: This contract concerns the publication of the story "The life of the Roadrunner"

This contract is made between Scott Tiger, whose address is 8i Ora Road, Redwood Shores, CA, USA hereinafter referred to as the PUBLISHER, and Wile E. Coyote and Chuck Jones whose address is 10 Acme Bvd, Hollywood, CA, USA, hereinafter referred to as the AUTHORS.

The parties agree as follows:

Authors Grant.
1. The Authors grant permission to include their story entitled "The life of the Roadrunner," a work of approximately 10500 words, hereinafter referred to as the Work, on the ACME Publishing Corporation web site found at www.acme-publishing.com.

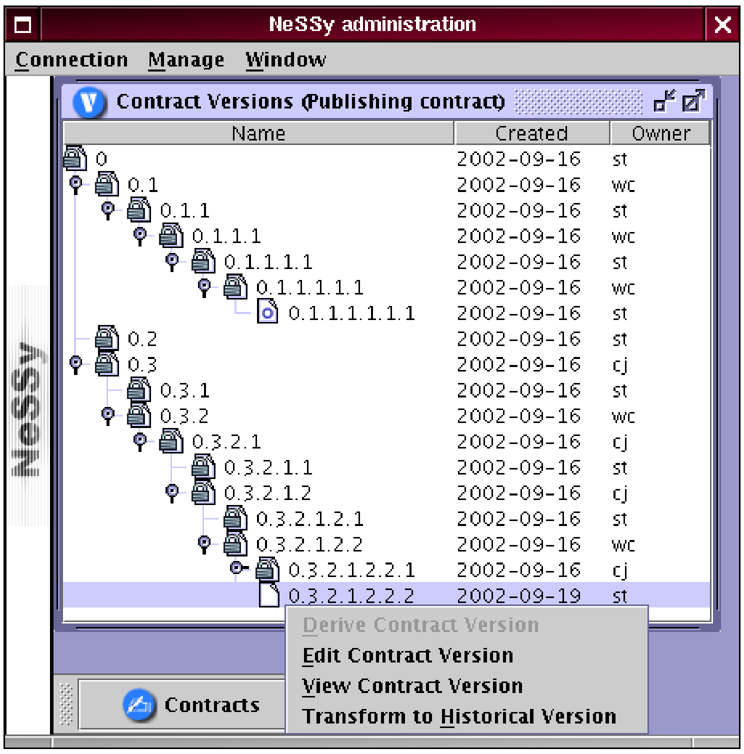
Figure 9.4: The “Create Initial Contract Version” window

When the initial contract version has been created, negotiators that are participants can negotiate, creating new contract versions. The contract version tree can be browsed with the “Contract Version Tree” window, presented in Figure 9.5. This window presents all the existing contract versions. Various contract versions are represented differently in the contract version tree, depending on their type:

- a draft version, i.e. version *0.3.2.1.2.2.2*, is represented by the icon 🗋 ;
- a historical version, i.e. version *0*, is represented by the icon 🗐 ;
- the last proposed version of each negotiator, i.e. version *0.1.1.1.1.1.1*, is represented by the icon 🗎 .

A new contract version can be derived from all historical and last proposed contract version. Draft versions can be edited by the negotiator that creates it. Any modifications can be made inserted to draft version. When a negotiator finishes her/his contract version modifications, she/he can transform the draft version into the last proposed version. The contract version that was previously the last created historical contract version becomes then a historical version. Only one last proposed version exists for each negotiator.

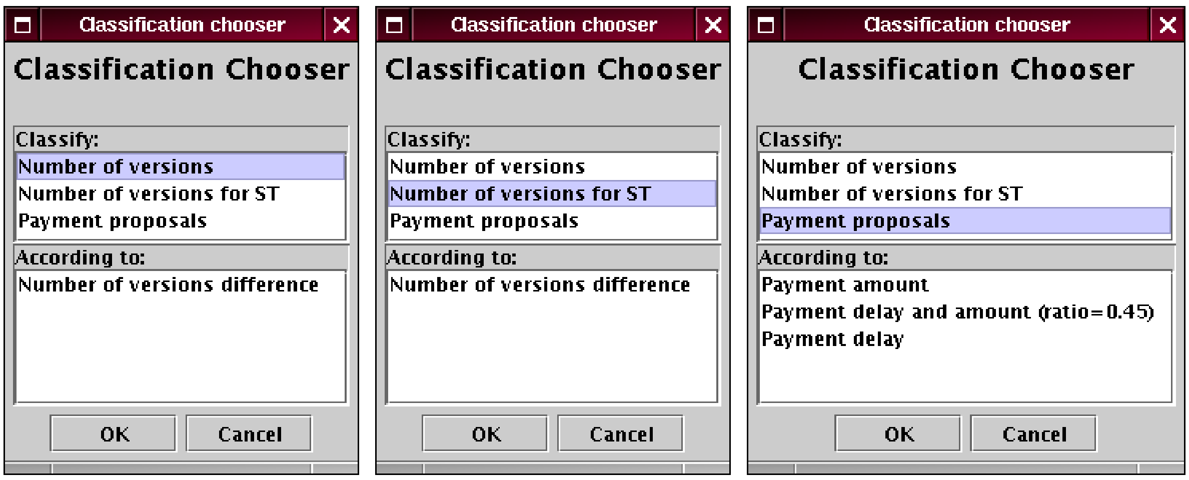
Historical and last proposed contract versions can be read by all participants. No modification can be made on historical and the last proposed contract versions.



| Name | Created | Owner |
|---|---|---|
| 0 | 2002-09-16 | st |
| 0.1 | 2002-09-16 | wc |
| 0.1.1 | 2002-09-16 | st |
| 0.1.1.1 | 2002-09-16 | wc |
| 0.1.1.1.1 | 2002-09-16 | st |
| 0.1.1.1.1.1 | 2002-09-16 | wc |
| 0.1.1.1.1.1.1 | 2002-09-16 | st |
| 0.2 | 2002-09-16 | st |
| 0.3 | 2002-09-16 | cj |
| 0.3.1 | 2002-09-16 | st |
| 0.3.2 | 2002-09-16 | wc |
| 0.3.2.1 | 2002-09-16 | cj |
| 0.3.2.1.1 | 2002-09-16 | st |
| 0.3.2.1.2 | 2002-09-16 | cj |
| 0.3.2.1.2.1 | 2002-09-16 | st |
| 0.3.2.1.2.2 | 2002-09-16 | wc |
| 0.3.2.1.2.2.1 | 2002-09-16 | cj |
| 0.3.2.1.2.2.2 | 2002-09-19 | st |

Figure 9.5: Contract version tree window

Every participant of a given negotiation may analyze the negotiation process. The “Classification Chooser” window, presented in Figure 9.7, allows negotiators to choose an analysis from among all possible ones. Negotiators can choose the object of the analysis, e.g. number of paragraph version or proposed payment clauses. Having a given object of the analysis, negotiators can potentially choose various analysis criteria, e.g. relative importance of payment amount or payment delay. Various cases are presented in Figure 9.7. The two windows on the left side present two analyses of two different facets with a common criterion. The window on the right side presents a negotiation facet that can be analyzed according to various criteria.



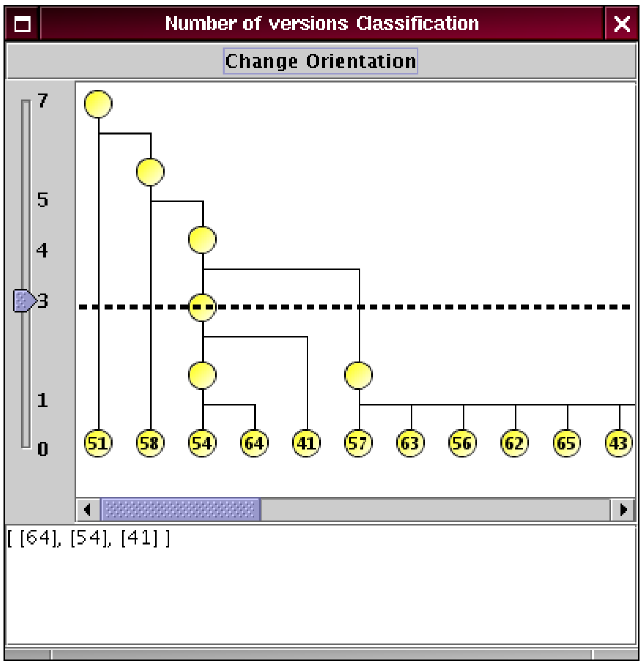
Classify:
Number of versions
Number of versions for ST
Payment proposals
According to:
Number of versions difference

Classify:
Number of versions
Number of versions for ST
Payment proposals
According to:
Number of versions difference

Classify:
Number of versions
Number of versions for ST
Payment proposals
According to:
Payment amount
Payment delay and amount (ratio=0.45)
Payment delay

Figure 9.6: Contract version tree windows for various negotiation facets.

The result of an analysis is presented in Figure 9.7. Three parts may be distinguished in the "Analysis Results" window. In the center, a tree represents the hierarchical classification. The horizontal dashed line represents the chosen threshold. On the left side, a slider allows negotiator to set the threshold at the given value. When a negotiator clicks on a class in the tree, represented as a circle, the value of the class is displayed on the text area at the bottom of the window. If a negotiator clicks on an atomic class, the metaObjects contained in this class are displayed. Otherwise, the identifiers of the metaObjects contained in the class are displayed, as presented in Figure 9.7.



[ [64], [54], [41] ]

Figure 9.7: The "Classification Results" window

## 9.2 Architecture

Three modules may be distinguished in the *NeSSy* prototype: a multiversion contract module, an analysis domain generator, and a classification generator.

The multiversion contract module consists of a multiversion contract manager and an editor. The multiversion contract manager is responsible for negotiators and multiversion contract creation and removal. The multiversion contract editor is responsible for contract version derivation, and contract version contents edition. Data persistence is provided by a database.

The analysis domain generator is responsible for ADF compilation and execution. The analysis domain generator is also responsible for compilation of metaObject definition documents. Finally, the analysis domain generator provides an implementation of the *core* module, described in Section 6.3.7.

The classification generator is responsible for metaObjects classification generation according to an analysis criterion. It is also responsible for the delegation of criteria execution to the analysis domain generator.

The interactions between *NeSSy* modules are presented in Figure 9.8. A solid line arrow represents a mandatory relationship. A dashed line arrow represents a potential relationship. The multiversion contract module interacts with a database that is used to perform contract persistence. The analysis domain generator may interact with the database to retrieve data needed to generate metaObjects. The analysis domain generator may also interact with the multiversion contract module to retrieve data concerning a contract of negotiators. Finally, the classification generator interacts with the analysis domain generator to retrieve metaObjects to be classified. Moreover, as defined in Section 7.2, analysis criteria are defined as ADFs. Therefore, the classification generator also interacts with the analysis domain generator to execute analysis criteria.

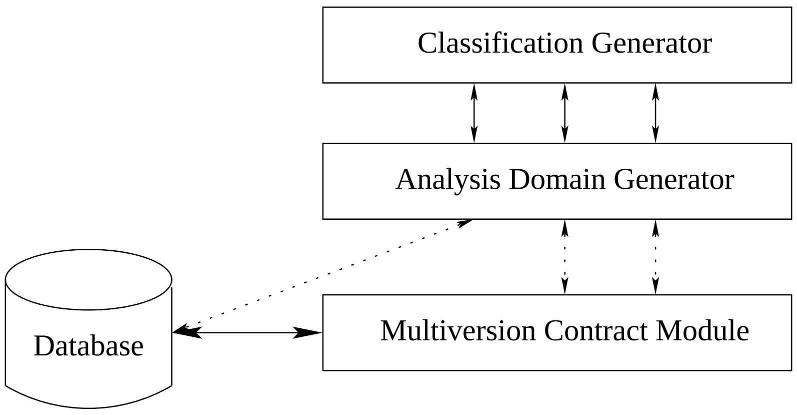


Figure 9.8: Interactions between *NeSSy* modules

The *NeSSy* system is distributed. It is implemented as a client/server system: it is organized according to the user interface, business logic, and data storage tiers. The user interface tier is implemented by the *NeSSy* client. The business logic tier is implemented by the *Nessy* server. The data storage tier is implemented by the Oracle 8i database. For the *NeSSy* prototype, the Oracle 8i database has been chosen for its wide acceptance. The distribution of the *NeSSY* prototype is presented in Figure 9.9.

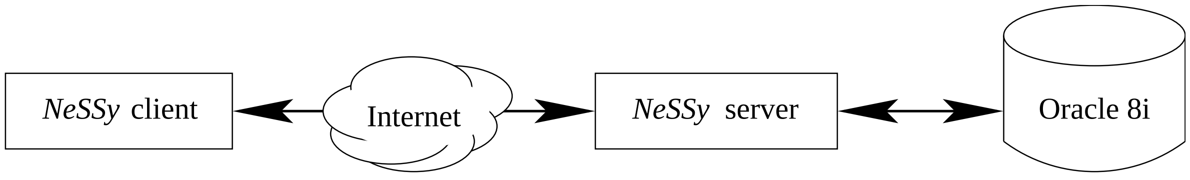


Figure 9.9: Distribution of the *NeSSy* prototype

The *NeSSy* client is a Java$^{TM}$application. It is implemented with the Swing framework [80]. The *NeSSy* client provides a front-end to the multiversion contract module and the classification generator. It allows negotiators to manage contracts, edit contract versions and perform analyses of the negotiation process.

The *NeSSy* server is a Java$^{TM}$application. It implements the logic for multiversion contract management, analysis domain generation, and classification generation. It is also responsible for the access to the data storage tier.

The communication between the *NeSSy* client and the *NeSSy* server uses the Java$^{TM}$Remote Method Invocation (RMI). Other mechanisms, such as CORBA or SOAP, may be used: the *NeSSy* prototype is designed to be independent of any specific middleware.

## 9.3 Contract Model Implementation

The contract model presented in Section 4.2 is implemented by the multiversion contract module. The structure of the multiversion contract module architecture is presented in Figure 9.10.

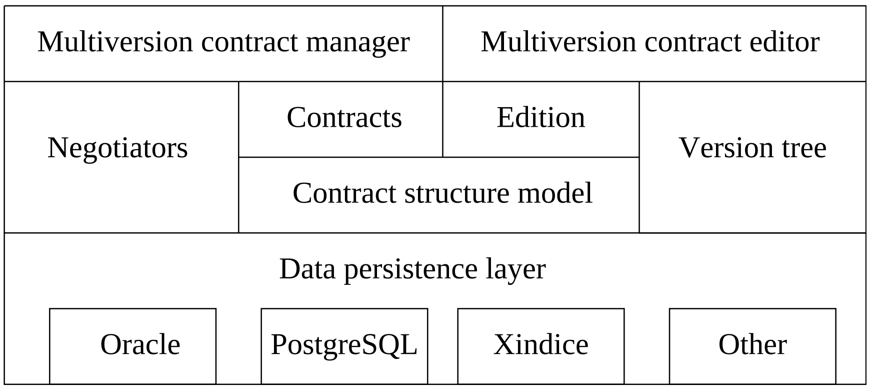


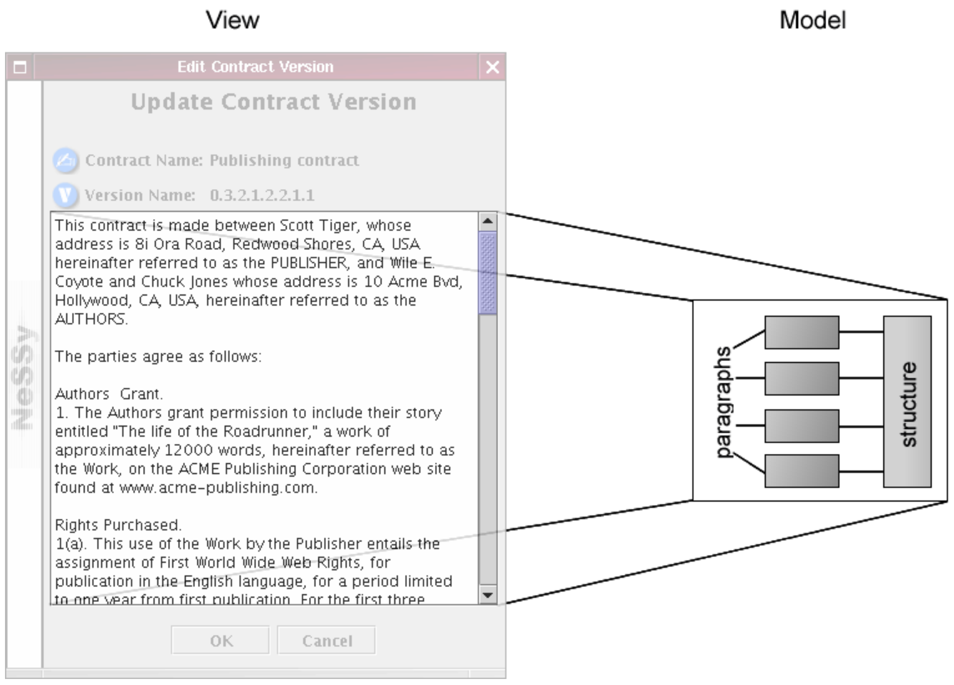
Figure 9.10: Contract module architecture

Two main parts may be distinguished: a data persistence layer and a business logic module. The data persistence layer is responsible for separating the data storage from the business logic module. The data persistence layer provides an API (Application Programming Interface) and may use various implementations to access specific data storage system via Data Storage Implementations (DSI). The data persistence API allows the business logic module to access transparently a variety of storage data systems. A plugin mechanism allows to extend the functionality of the data persistence layer to access new data storage systems by addition of new DSI. In Figure 9.10, DSIs for Oracle, PostgreSQL, and Xindice are presented. Currently, DSI for Oracle and PostgreSQL have been implemented.

Three parts may be distinguished in the business logic module: negotiator manager, version tree manager, and contract manager and editor. The negotiator manager is responsible for negotiator creation and removal. It accesses directly the data persistence layer. The version tree manager is responsible for contract version tree retrieval, derivation and contract version status modification (from draft to historical). It accesses directly the data persistence layer. Finally, the contract manager, respectively contract editor, is responsible for multiversion contract additions and removals, respectively for contract version modifications. Both these parts are based on a contract structure model module, which is responsible for mapping specific contract structure model to the contract model presented in Section 5. It isolates contract manager and contract editor from the data persistence layer. Therefore, various contracts, structured differently, can be stored with the existing data persistent layer API — only the contract structure model module has to be adapted.

The contract structure model implemented in the *NeSSy* system is the one presented in Section 4.2. In this contract structure model, a contract consists of a set of multiversion paragraphs and a multiversion structure. A paragraph version consists of a character string. The multiversion structure is responsible for maintaining the order of paragraphs in each contract version. Therefore, each contract version is a list of plain text paragraphs.

On the client side, a contract editor basing on the proposed contract model allows negotiators to easily read existing contract versions and to create new ones. The contract editor is basing on the Swing text framework [80, 61]. Swing's text components, like all Swing components, are based on an architecture that features model-view separation. The model-view separation, as presented in Figure 9.11, allows building custom document structures and specific views associated with these structures. In the *NeSSy* system, the contract structure model is mapped to a Swing document and is plugged to the Swing text framework.



Contract Name: Publishing contract
Version Name: 0.3.2.1.2.2.1.1

This contract is made between Scott Tiger, whose address is 8i Ora Road, Redwood Shores, CA, USA hereinafter referred to as the PUBLISHER, and Wile E. Coyote and Chuck Jones whose address is 10 Acme Bvd, Hollywood, CA, USA, hereinafter referred to as the AUTHORS.

The parties agree as follows:

Authors Grant.
1. The Authors grant permission to include their story entitled "The life of the Roadrunner," a work of approximately 12000 words, hereinafter referred to as the Work, on the ACME Publishing Corporation web site found at www.acme-publishing.com.

Rights Purchased.
1(a). This use of the Work by the Publisher entails the assignment of First World Wide Web Rights, for publication in the English language, for a period limited to one year from first publication. For the first three

Figure 9.11: Model-view separation in the contract editor of the *NeSSy* client

## 9.4 Analysis Domain Implementation

The analysis domain language presented in Section 6.3 is supported by the analysis domain generator. The analysis domain generator is an ADL compiler. Given a set of metaObjects, tag, and function definitions, the analysis domain generator creates a set of Java$^{TM}$classes and compiles them. Function executions – to generate analysis domains – are then Java$^{TM}$method executions.

The analysis domain generator is a framework, containing:

- ADL basic Java$^{TM}$classes and interfaces,
- a compiler,
- an implementation of the *core* module.

ADL basic Java$^{TM}$classes and interfaces provides basic ADL mechanisms, such as:

- support for metaObjects,
- support objectSets,
- support for tag implementations,
- support for ADF.

All ADL basic Java$^{TM}$classes and interfaces are in the `org.nessy.xf` package.

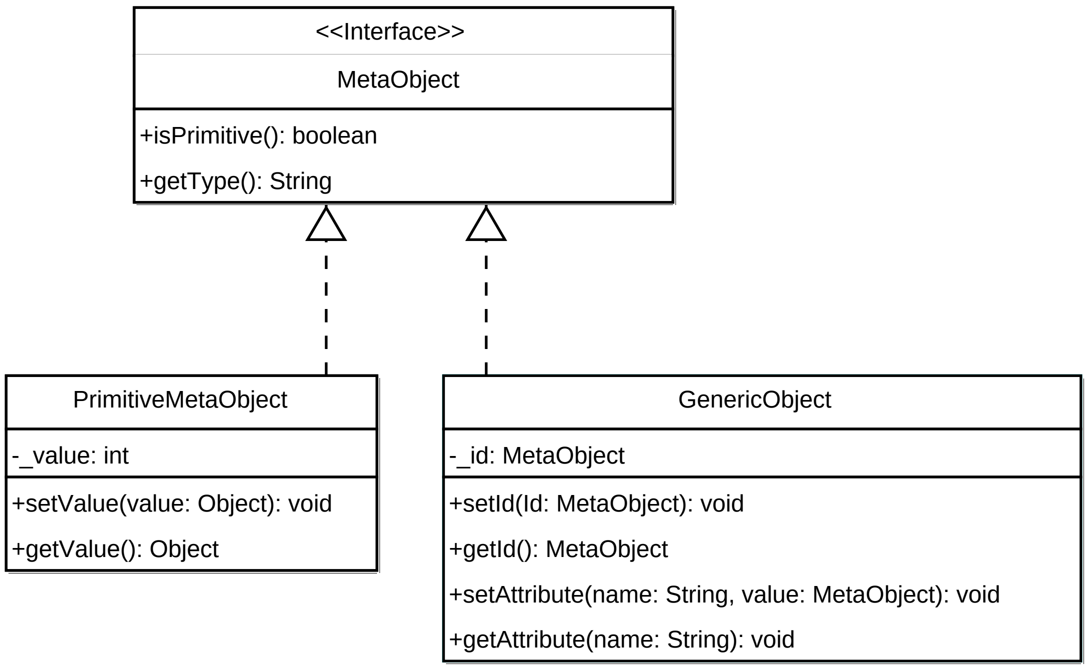


Figure 9.12: Java$^{TM}$classes and interface for metaObjects support.

The metaObjects support in the analysis domain generator consists of a Java$^{TM}$interface and two classes, as presented in Figure 9.12. All metaObjects *implement* the `MetaObject` interface. The `MetaObject` interface defines the minimal behavior of all metaObjects: a metaObject must have a type (returned by the `getType()` method) and is or is not a primitive metaObject (depending on the result of the `isPrimitive()` method. The `PrimitiveMetaObject` class inherits from the `MetaObject` interface and is the parent class of all primitive metaObjects. The value of the primitive object is accessible by the `setValue()` and `getValue()` methods, which uses Java$^{TM}$`Object`s. The `GenericObject` class inherits from the `MetaObject` interface and is the parent class of all non-primitive metaObjects. The identifier of a given `GenericObject` is accessible by the `getId()` and `setId()` methods, which uses `MetaObject`s. Attributes are accessible by the `setAttribute()` and `getAttribute()` methods. Attribute type-checking is encapsulated in the `GenericObject` class, in the `setAttribute()` method. Type-checking is also performed for identifiers in the `setId()` method.

ObjectSets are modeled by the `ObjectSet` class. The `ObjectSet` class methods are:

- `isEmpty()`: returns `true` if the objectSet does not contain any metaObject,

- `size()`: returns the number of metaObjects contained in the given objectSet,
- `iterator()`: returns a `java.util.Iterator` to access metaObjects contained in the given objectSet,
- `clear()`: deletes all metaObjects from the given objectSet,
- `add(MetaObject)`: adds a metaObject to the given objectSet.

The tag support in the analysis domain generator consists of a Java$^{TM}$interface and four classes, as presented in Figure 9.13. All tags implemented in Java$^{TM}$must implement the `Tag` interface. The `Tag` interface defines the minimal behavior of all tags:

- `findAttribute()`, `getAttribute()`, `setAttribute()`, and `removeAttribute()` methods are responsible for tag context management. A tag context is a set of attributes, i.e. a pair (name, value), where name is a `java.lang.String` and value is a `java.lang.Object`;
- `getParent()` and `setParent()` methods are responsible for parent tag management;
- `getFunctionContext()` and `setFunctionContext()` methods are responsible for function context management. Function context is defined in the `FunctionContext` class. A function context is a set of attributes, i.e. a pair (name, value), where name is a `java.lang.String` and value is a `java.lang.Object`;
- `init()` method is called before tag execution and may be used to set resources needed by a tag, such as for instance connection to a database;
- `release()` method is called after tag execution and may be used to release resources needed by a tag, such as for instance database connection release.

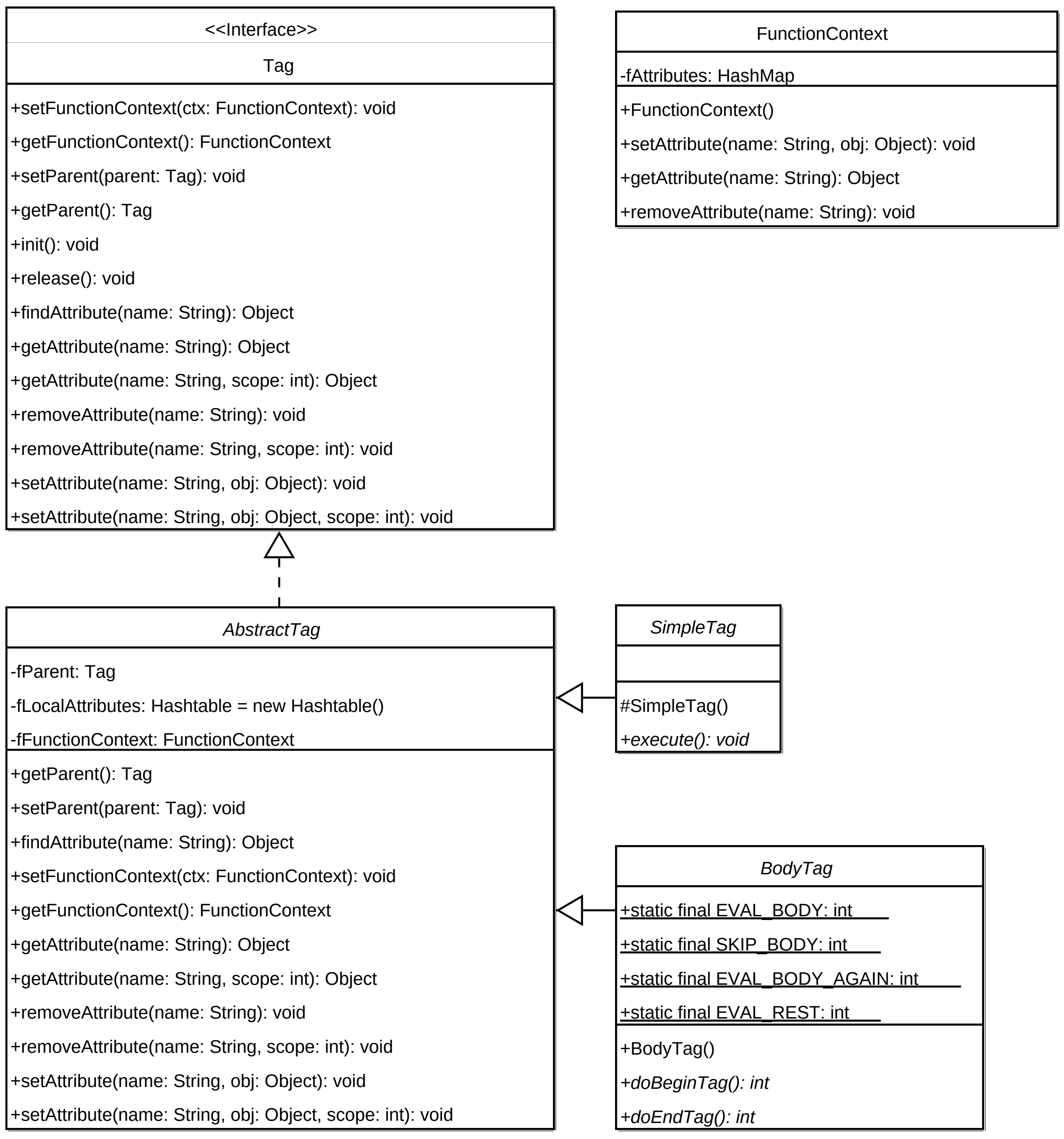
The abstract `AbstractTag` class implements the previous functions. However, as an abstract class, it cannot be instantiated. Two children abstract classes are defined: `SimpleTag` and `BodyTag`. The proposed solution is similar to tag extension mechanism of the Java Server Pages$^{TM}$(JSP$^{TM}$) [79, 55].

Empty tags must be children of the `SimpleTag` class. The `SimpleTag` defines only an abstract method, the `execute()` method, which must be defined in children class and in which the processing actions to be done must be defined.

Non-empty tags must be children of the `BodyTag` class. The BodyTag defines two abstract methods, the `doBeginTag()` and `doEndTag()` methods. The `doBeginTag()` method is executed at the beginning of the tag execution and may return constants `EVAL_BODY` and `SKIP_BODY`. If the `doBeginTag()` method returns `EVAL_BODY`, the tag body is evaluated. Otherwise, the body tag is not evaluated. The `doEndBody()` is executed after `doBeginTag()` execution and potentially tag body evaluation. The `doEndBody()` may return constants `EVAL_BODY_AGAIN` and `EVAL_REST`. If the `doEndTag()` method returns `EVAL_BODY_AGAIN`, the `doBeginTag()` method is executed again, and potentially the tag body is evaluated. If the `doEndTag()` method returns `EVAL_REST`, the execution of the tag ends.

Support for ADL consists of the abstract `FunctionBody` class. The `FunctionBody` class methods are:

- `initializeContext()` and `getContext()` methods are responsible for function context initialization and retrieval, respectively;



<<Interface>>
Tag
+setFunctionContext(ctx: FunctionContext): void
+getFunctionContext(): FunctionContext
+setParent(parent: Tag): void
+getParent(): Tag
+init(): void
+release(): void
+findAttribute(name: String): Object
+getAttribute(name: String): Object
+getAttribute(name: String, scope: int): Object
+removeAttribute(name: String): void
+removeAttribute(name: String, scope: int): void
+setAttribute(name: String, obj: Object): void
+setAttribute(name: String, obj: Object, scope: int): void
FunctionContext
-fAttributes: HashMap
+FunctionContext()
+setAttribute(name: String, obj: Object): void
+getAttribute(name: String): Object
+removeAttribute(name: String): void
AbstractTag
-fParent: Tag
-fLocalAttributes: Hashtable = new Hashtable()
-fFunctionContext: FunctionContext
+getParent(): Tag
+setParent(parent: Tag): void
+findAttribute(name: String): Object
+setFunctionContext(ctx: FunctionContext): void
+getFunctionContext(): FunctionContext
+getAttribute(name: String): Object
+getAttribute(name: String, scope: int): Object
+removeAttribute(name: String): void
+removeAttribute(name: String, scope: int): void
+setAttribute(name: String, obj: Object): void
+setAttribute(name: String, obj: Object, scope: int): void
SimpleTag
#SimpleTag()
+execute(): void
BodyTag
+static final EVAL_BODY: int
+static final SKIP_BODY: int
+static final EVAL_BODY_AGAIN: int
+static final EVAL_REST: int
+BodyTag()
+doBeginTag(): int
+doEndTag(): int

Figure 9.13: Java$^{TM}$classes and interface for tag support.

- `addParameter()`: sets attribute to the associated function context;
- abstract `process()`: processing actions must be defined in this method.

The contents of the process() methods in ADLs are generated by the parser.

The compiler is responsible for parsing the metaobject, tag, and function definition documents. It generates and compiles Java$^{TM}$classes for metaObjects and ADLs. Generated classes are in sub-packages of the `org.nessy.work` package. If an ADL or a metaObject is defined in a module myModule, the associated generated classes are in the `org.nessy.work.myModule` package. The compiler generates a Java$^{TM}$class for each metaObject definition document. The compiler generates a Java$^{TM}$class for each ADL definition document.

A special module of the compiler is responsible for ADL expressions compilation. This module consists of a set of Java$^{TM}$classes generated by the Java Compiler Compiler$^{TM}$(JavaCC) [88]. JavaCC compiles a grammar into a set of Java$^{TM}$classes. The grammar for ADL expressions is given in Appendix B.

The implementation of the *core* module consists of a set of metaObject and tag definition document, and a set of Java$^{TM}$classes implementing the *core* module tags.

## 9.5 Classification Implementation

The domain objects classification mechanism presented in Section 7 is implemented by the classification generator. The classification generator is responsible for retrieving the metaObjects generated by the analysis domain generator and for classifying them according to analysis criterion. The classification generator is also responsible for ADFs and analysis criteria management.

Generation of a classification consists of four phases:

1. The classification generator calls the domain analysis generator to create a set of metaObjects to be analyzed.
2. The classification generator processes the distances between metaObjects according to a given analysis criterion:
   a) The classification generator crates an empty space *D*.
   b) For each pair of metaObjects ($metaObject_1$, $metaObject_2$),
      i. the classification generator calls the domain analysis generator to invoke the analysis criterion as an ADF to process *distance*, and
      ii. the classification generator adds a triplet ($metaObject_1$, $metaObject_2$, *distance*) to the space *D*.
3. The classification generator “isoscelizes” the space *D*.
4. The classification generator creates a classification from space *D*.

The classification generator is a repository for ADF and analysis criteria. New ADFs and analysis criteria may be added to the classification generator. Negotiators may retrieve the list of registered ADFs. Given an ADF, the list of compatible analysis criteria may be retrieved from the classification generator.

The classification generator is implemented in the Java$^{TM}$language. Two main parts may be distinguished: one part is responsible for modeling space *D*, another part is responsible for modeling classifications, partitions and classes.

Classes modeling space *D* are presented in Figure 9.14. Class `Segment` models a triplet (*metaObject*$_1$, *metaObject*$_2$, *distance*). Class `SegmentList` models space *D* as a set of Segment instances. Class `SegmentList` is responsible for "isoscelization" (implemented by the `isoscelizeSpace()` method). It is also responsible for classification generation (implemented by the `createClassification()` method).

Classes modeling classifications, partitions and classes are presented in Figure 9.15. Class `Classification` models the result of the analysis process. An instance of the `Classification` class consists of a set of `HierarchicalClass` instances. Class `HierarchicalClass` is a specialization of `Class`. Instances of `Class` consist of a set of `MetaObject`s. An instance of the `HierarchicalClass` may contain `HierarchicalClass` instances. Instances of the class `Classification` are generated by the `createClassification()` method of `SegmentList` instances. An `AnalysisCriterion` class models analysis criteria used during classification generation. Method `threshold()` of the class `Classification` generates `Partition` instances. A `Partition` is a set of instances of class `Class`.

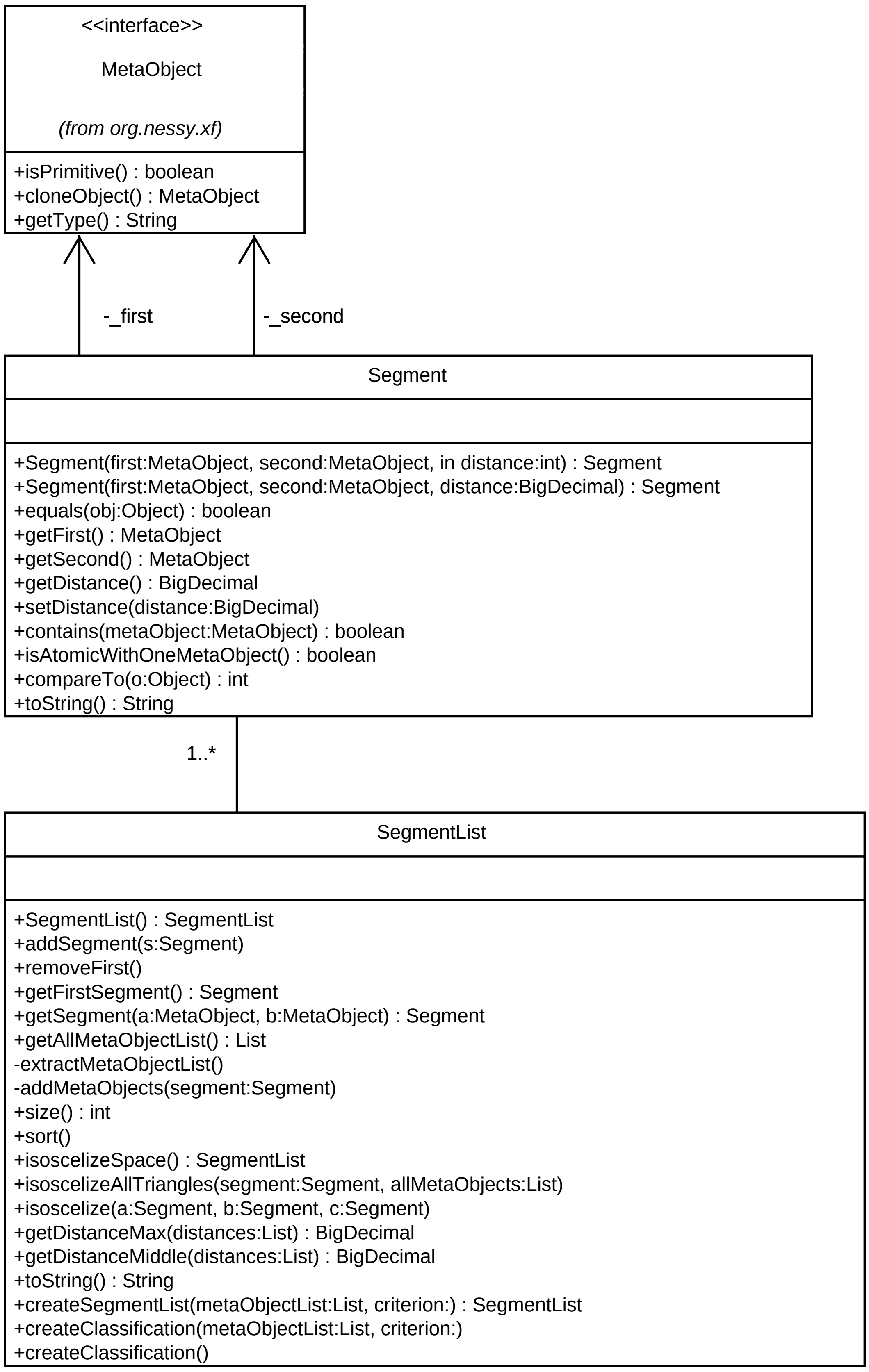

<<interface>>
MetaObject
(from org.nessy.xf)
+isPrimitive() : boolean
+cloneObject() : MetaObject
+getType() : String
-_first
-_second
Segment
+Segment(first:MetaObject, second:MetaObject, in distance:int) : Segment
+Segment(first:MetaObject, second:MetaObject, distance:BigDecimal) : Segment
+equals(obj:Object) : boolean
+getFirst() : MetaObject
+getSecond() : MetaObject
+getDistance() : BigDecimal
+setDistance(distance:BigDecimal)
+contains(metaObject:MetaObject) : boolean
+isAtomicWithOneMetaObject() : boolean
+compareTo(o:Object) : int
+toString() : String
1..*
SegmentList
+SegmentList() : SegmentList
+addSegment(s:Segment)
+removeFirst()
+getFirstSegment() : Segment
+getSegment(a:MetaObject, b:MetaObject) : Segment
+getAllMetaObjectList() : List
-extractMetaObjectList()
-addMetaObjects(segment:Segment)
+size() : int
+sort()
+isoscelizeSpace() : SegmentList
+isoscelizeAllTriangles(segment:Segment, allMetaObjects:List)
+isoscelize(a:Segment, b:Segment, c:Segment)
+getDistanceMax(distances:List) : BigDecimal
+getDistanceMiddle(distances:List) : BigDecimal
+toString() : String
+createSegmentList(metaObjectList:List, criterion:) : SegmentList
+createClassification(metaObjectList:List, criterion:)
+createClassification()

Figure 9.14: Classes modeling space *D*

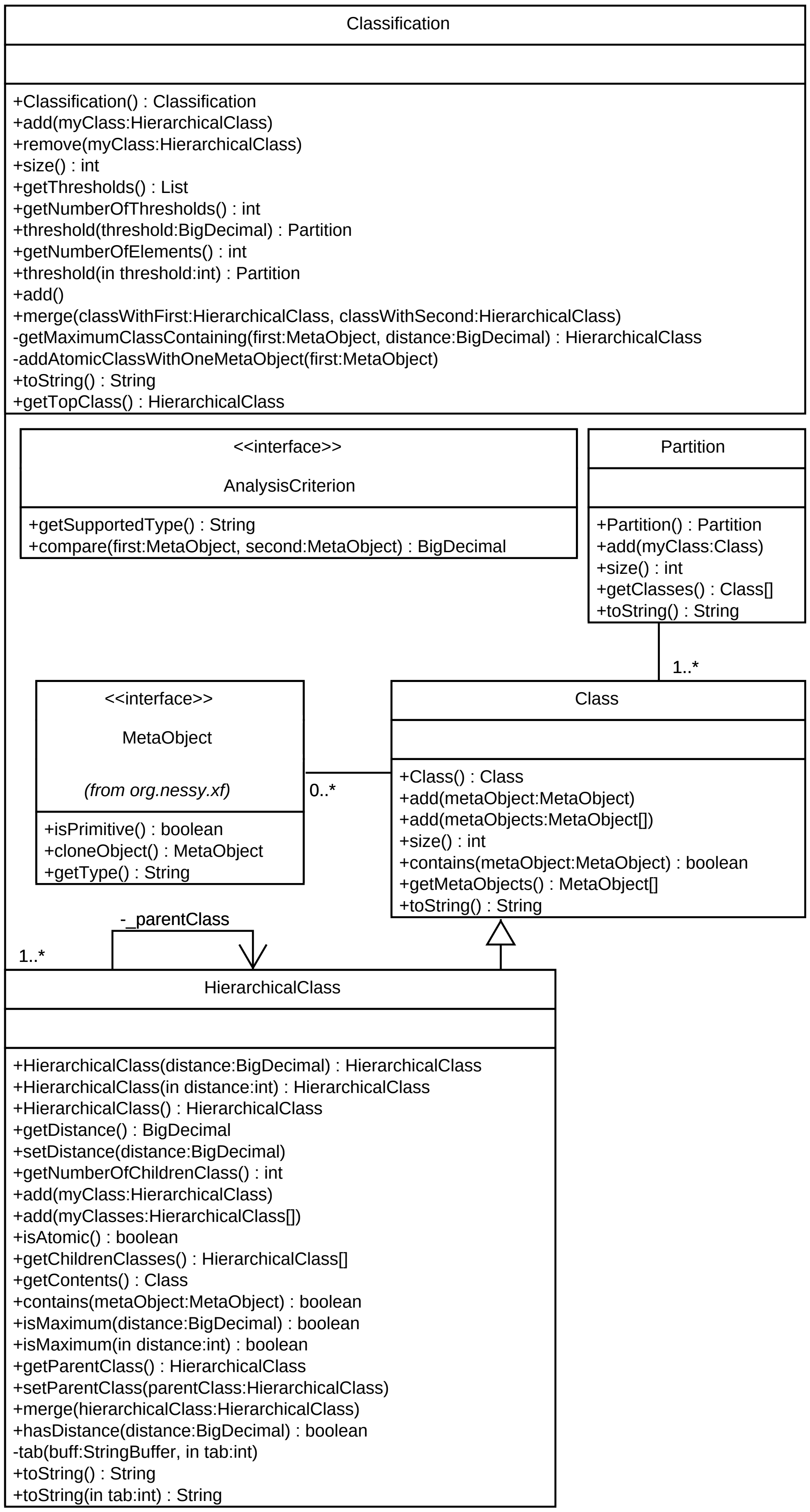

Classification
+Classification() : Classification
+add(myClass:HierarchicalClass)
+remove(myClass:HierarchicalClass)
+size() : int
+getThresholds() : List
+getNumberOfThresholds() : int
+threshold(threshold:BigDecimal) : Partition
+getNumberOfElements() : int
+threshold(in threshold:int) : Partition
+add()
+merge(classWithFirst:HierarchicalClass, classWithSecond:HierarchicalClass)
-getMaximumClassContaining(first:MetaObject, distance:BigDecimal) : HierarchicalClass
-addAtomicClassWithOneMetaObject(first:MetaObject)
+toString() : String
+getTopClass() : HierarchicalClass
<<interface>>
AnalysisCriterion
+getSupportedType() : String
+compare(first:MetaObject, second:MetaObject) : BigDecimal
Partition
+Partition() : Partition
+add(myClass:Class)
+size() : int
+getClasses() : Class[]
+toString() : String
1..*
<<interface>>
MetaObject
(from org.nessy.xf)
+isPrimitive() : boolean
+cloneObject() : MetaObject
+getType() : String
0..*
Class
+Class() : Class
+add(metaObject:MetaObject)
+add(metaObjects:MetaObject[])
+size() : int
+contains(metaObject:MetaObject) : boolean
+getMetaObjects() : MetaObject[]
+toString() : String
-_parentClass
1..*
HierarchicalClass
+HierarchicalClass(distance:BigDecimal) : HierarchicalClass
+HierarchicalClass(in distance:int) : HierarchicalClass
+HierarchicalClass() : HierarchicalClass
+getDistance() : BigDecimal
+setDistance(distance:BigDecimal)
+getNumberOfChildrenClass() : int
+add(myClass:HierarchicalClass)
+add(myClasses:HierarchicalClass[])
+isAtomic() : boolean
+getChildrenClasses() : HierarchicalClass[]
+getContents() : Class
+contains(metaObject:MetaObject) : boolean
+isMaximum(distance:BigDecimal) : boolean
+isMaximum(in distance:int) : boolean
+getParentClass() : HierarchicalClass
+setParentClass(parentClass:HierarchicalClass)
+merge(hierarchicalClass:HierarchicalClass)
+hasDistance(distance:BigDecimal) : boolean
-tab(buff:StringBuffer, in tab:int)
+toString() : String
+toString(in tab:int) : String

Figure 9.15: Classes modeling classifications, partitions and classes

CHAPTER 10

# CONCLUSIONS

The multi-facet analysis approach to e-negotiation that has been presented in this dissertation provides a solution to the problem of mass distributed negotiations via Internet, allowing a high number of geographically dispersed negotiators to work on real-life contracts.

Two ideas that are the basis of the multi-facet analysis approach to e-negotiation are: first, synthetic views of the negotiation process are needed in mass negotiation of real-life contracts, because of the high amount of data, second, relationships between offers and counter-offers contains the information that can be analyzed more easily than clauses contents, because attributes can be non-aggregable and their semantics is not always known.

In multi-facet analysis approach, contracts do not have to be semantics-enabled and may contain both aggregable and non-aggregable attributes. Such contracts are predominant in the business world. The proposed contract model does not limit contracts with regard to their structure and allows new contract structures to be built on the top of the proposed multiversion contract model.

The proposed contract model provides versioning mechanism to the negotiation process. The versioning mechanism provides scalability with respect to the number of negotiators. Proposed contract edition model is a parallel model which scales well, as each negotiator works on her/his draft versions in isolation. Therefore, an arbitrary number of negotiators may be involved in a negotiation process. The versioning mechanism also provides association tables which model relationships between offers and counter-offers.

The proposed approach provides negotiators with tools to analyze the negotiation process. The analysis mechanism aims at reducing the high amount of data resulting from the negotiation process. Synthetic views of the negotiation process allow negotiators to easily understand various aspects of an on-going negotiation and to focus efficiently on the elements of their highest interest. An analysis is decomposed into two parts: first, the negotiation aspect to be analyzed is defined, second, an appropriate analysis criterion is chosen. This decomposition allows for a better code reuse, as an aspect may be analyzed according to various criteria, and a given criterion may be used to classify various aspects.

Three techniques are used in the multi-facet analysis approach to e-negotiation: multiversion contract model, domain object generation, and analysis domain classification. Each of these techniques offers a value by itself, but combined together constitute an approach that provides particularly rich functionality and flexibility: association tables may be analyzed even when the semantics of clauses are unknown, because association tables capture some aspects of the negotiation process (e.g. importance of clauses as the number of their versions), and do not rely on clauses contents.

The main achievements of this dissertation are the following:

- Identification and evaluation of existing e-negotiation approaches, automatic hierarchical classification algorithms, and versioning techniques.
- Development of a multiversion contract model that enable modeling of contract evolution during the negotiation process. The model captures the relationships between various offers and counter-offers. The structure of contracts is orthogonal to the versioning mechanism.
- Development of a new language — called ADL — that enables building of functions that extract domain objects modeling various facets of the negotiation process.
- Development of a hierarchical classification mechanism that provides synthetic views of the negotiation process. This mechanism uses ADL as an analysis criteria definition language. The hierarchical aspect of synthetic views allows negotiators to choose the granularity of the analysis.
- Example application of the multi-facet analysis to e-negotiations offering better insight into the multiversion contract model and the analysis mechanism.
- Implementation of a prototype e-negotiation support system *NeSSy* composed of a multiversion contract module, an ADL compiler, and a classification generator.

An important feature of the multi-facet analysis approach to e-negotiation is its extensibility. Extensibility is an inherent requirement for the classification mechanism. New facets can easily be analyzed because of the use of ADL to extract and classify data. The multiversion contract model is also extensible because the structure of contracts is not fixed in the model. Therefore, advanced contract structures (e.g. tree structured contracts) may be built using the concepts of multiversion members and multiversion composite members proposed in the multiversion contract model.

The multi-facet analysis approach to e-negotiation opens new directions of research. An interesting example is application of the proposed approach to mobile computing, allowing mobile negotiators, which are potentially off-line, to analyze the negotiation process. The proposed multiversion contract model captures various important facets of the negotiation process (such as contract member sharing) in small size structures — association tables. These structures can be send efficiently over limited-bandwidth network and can be stored in memory-limited devices like mobile phones or PDAs. Negotiators could therefore analyze some aspects of the negotiation process without having to download the whole multiversion contract.

Another example is the use of software agents. Using the analysis mechanism, advanced behavior models can be build. Psychological and social models for negotiating agents may base on data retrieved from the analysis of various facets of the negotiation process. An agent may for example have a “collaborative” behavior, i.e. may look for negotiators having similar proposals to build a group of negotiators in order to increase its weigh in the negotiation process. The problem of ontologies would be minimized in this case, as the analyses “encapsulate” the meaning of various facets of the negotiation process.

Another interesting research direction is application of the proposed approach to any application areas characterized by great amount of data that evolves. Source code management in software development is a good example of such area. The evolution of source code could be modeled by the multiversion contract model, capturing the relationships between various versions. The analysis mechanism would be an important step-ahead in the process of software development, allowing developers to have a better understanding of the whole project from various points of view.

# References

[1] ABDESSALEM, T., AND JOMIER, G. VQL, a Query Language for Multiversion Databases. In *International Workshop on Databases Programming Languages, DBPL-97* (Colorado, USA, 1997).

[2] AMERICAN LAW INSTITUTE. *Restatement of the Law*, Second: Contracts, Art. 3,8,12.

[3] AUMANN, R. J., AND HART, S. *Handbook of Game Theory*, vol. I. North-Holland, 1992.

[4] BALAKRISHNAN, P., SUNDAR, V., AND ELIASHBERG, J. An Analytical Process Model of Two-Party Negotiations. *Management Science 41* (1995), 226–243.

[5] BANZHAF, W., NORDIN, P., KELLER, R. E., AND FRANCONE, F. D. *Genetic Programming — an Introduction: On the Automatic Evolution of Computer Programs and its Applications*. Morgan Kaufmann Publishers, 1998.

[6] BAUZER MEDIEROS, C., BELLOSTA, M. J., AND JOMIER, G. Multiversion Views: Constructing Views in a Multiversion Database. *Data and Knowledge Engineering 33* (2000).

[7] BERLINER, B. CVS II: Parallelizing Software Development. In *Proceedings of the USENIX Winter 1990 Technical Conference* (Berkeley, USA, 1990), USENIX Association, pp. 341–352.

[8] BERNERS-LEE, T. A Roadmap to the Semantic Web, Sept. 1998.
http://www.w3.org/DesignIssues/Semantic.html.

[9] BERNERS-LEE, T., HENDLER, J., AND LASSILA, O. The Semantic Web. *Scientific American* (May 2001).

[10] BRAY, T., HOLLANDER, D., AND LAYMAN, A. Namespaces in XML, W3C Recommendation, 14 Jan. 1999.
http://www.w3.org/TR/1999/REC-xml-names-19990114/.

[11] BRAY, T., PAOLI, J., SPERBERG-MCQUEEN, C. M., AND MALER, E. Extensible Markup Language (XML) 1.0 (Second Edition), W3C Recommendation, 6 Oct. 2000.
http://www.w3.org/TR/2000/REC-xml-20001006.

[12] BRICKLEY, D., AND GUHA, R. V. Resource Description Framework (RDF) Schema Specification 1.0, W3C Candidate Recommendation, 27 Mar. 2000.
http://www.w3.org/TR/REC-rdf-schema.

[13] CELLARY, W., AND JOMIER, G. Consistency of Versions in Object-Oriented Databases. In *Proceedings of the 16th VLDB Conference* (Brisbane, Australia, Aug. 1990), pp. 432–441.

[14] CELLARY, W., AND JOMIER, G. Apparent Versioning and Concurrency Control in Multi-version Databases. In *6th International Conference on Computing and Information* (Ontario, Canada, 1994).

[15] CELLARY, W., **PICARD, W.**, AND WIECZERZYCKI, W. Web-based business-to-business negotiation support. In *Int. Conference on Electronic Commerce EC-98* (Hamburg, Germany, 1998).

[16] CHAVEZ, A., AND MAES, P. Kasbah: An Agent Marketplace for Buying and Selling Goods. In *Proceedings of the 1st International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology* (London, UK, Apr. 1996).

[17] CLEMM, G., AMSDEN, J., ELLISON, T., KALER, C., AND WHITEHEAD, J. Versioning Extensions to WebDAV (Web Distributed Authoring and Versioning), RFC 3253, Mar. 2002. http://www.ietf.org/rfc/rfc3253.txt.

[18] CONNOLY, D., VAN HARMELEN, F., HORROCKS, I., MCGUINNESS, D., PATEL-SCHEIDER, P. F., AND STEIN, L. A. Annotated DAML+OIL Ontology Markup, W3C Note, 18 Dec. 2001. http://www.w3.org/TR/daml+oil-walkthru.

[19] CONNOLY, D., VAN HARMELEN, F., HORROCKS, I., MCGUINNESS, D. L., PATEL-SCHEIDER, P. F., AND STEIN, L. A. DAML+OIL (March 2001) Reference Description, W3C Note, 18 Dec. 2001.
http://www.w3.org/TR/daml+oil-reference.

[20] CRANEFIELD, S., AND PURVIS, M. UML as an Ontology Modeling Language. In *Proc. of the IJCAI-99 Workshop on Intelligent Information Integration* (Stockholm, Sweden, 1999).

[21] DAML. The DARPA Agent Markup Language (DAML) Program home page.
http://www.daml.org/.

[22] DAML+OIL. DAML+OIL (March 2001).
http://www.daml.org/2001/03/daml+oil-index.html.

[23] DOPAZO, J., AND CARAZO, J. M. Phylogenetic Reconstruction Using a Growing Neural Network that Adopts the Topology of a Phylogenetic Tree. *Journal of Molecular Evolution 44* (1997), 226–233.

[24] DOPAZO, J., WANG, H.-C., AND CARAZO, J. M. A New Type of Unsupervised Growing Neural Network for Biological Sequence Classification that Adopts the Topology of a Phylogenetic Tree. In *Lecture Notes in Computer Science* (1997), Springer, Ed., vol. 1240, pp. 932–941.

[25] DOUCET, A., GANÇARSKI, S., JOMIER, G., AND MONTIES, S. Integrity Constraints in Multiversion Databases. In *14th British National Conference on Database Proceedings* (UK, 1996).

[26] FALLSIDE, D. C. XML Schema Part 0: Primer, W3C Recommendation, 2 May 2001.
http://www.w3.org/TR/xmlschema-0.

[27] Fikes, R., and McGuinness, D. An Axiomatic Semantics for RDF, RDF-S, and DAML+OIL (March 2001), W3C Note, 18 Dec. 2001.
http://www.w3.org/TR/daml+oil-axioms.

[28] French Civil Code. Art. 1108, 1101-1369.

[29] Fritzke, B. Growing Cell Structures—a Self-Organizing Network in *k* Dimensions. In *Artificial Neural Networks, 2* (Amsterdam, Netherlands, 1992), I. Aleksander and J. Taylor, Eds., vol. II, North-Holland, pp. 1051–1056.

[30] Fritzke, B. Growing Cell Structures – a Self-Organizing Network for Unsupervised and Supervised Learning. *Neural Networks 7*, 9 (1994), 1441–1460.

[31] Gançarski, S., and Jomier, G. A Framework for Programming Multiversion Databases. *Data and Knowledge Engineering 36*, 1 (2001), 29–53.

[32] Goland, Y., Whitehead, E., Faizi, A., Carter, S., and Jensen, D. HTTP Extensions for Distributed Authoring – WEBDAV, RFC 2518, Feb. 1999.
http://www.ietf.org/rfc/rfc2518.txt.

[33] Griffel, F., Boger, M., Weinreich, H., Lamersdorf, W., and Merz, M. Electronic Contracting with COSMOS – How to Established and Execute Electronic Contracts on the Internet. In *Proceedings of IEEE/OMG Second Enterprise Distributed Object Computing Workshop (EDOC'98)* (La Jolla, CA, Nov. 1998).

[34] Hayes, P. RDF Model Theory, W3C Working Draft, 14 Feb. 2002.
http://www.w3.org/TR/rdf-mt.

[35] Houba, H., and W., B. *Credible Threats in Negotiations. A Game-theoretic Approach*, vol. 32 of *Theory and Decision Library C: Game Theory, Mathematical Programming and Operations Research*. Kluwer Academic Publishers, Boston, USA, Aug. 2002.

[36] Hunt, J. W., and McIlroy, M. D. An Algorithm for Differential File Comparison. *Computing Science Technical Report*, 41 (June 1976). Bell Labs, N.J.

[37] IEEE. IEEE Standard for Binary Floating-Point Arithmetic. ANSI/IEEE Std. 754-1985. *ACM SIGPLAN Notices 22*, 2 (Feb. 1987).

[38] ISO (International Organization for Standardization). Information Processing – Text and Office Systems – Standard Generalized Markup Language (SGML). Tech. Rep. 8879:1986(E), ISO, Geneva, Swissland, 15 Oct. 1986.

[39] Jennings, N. R., Faratin, P., Lomuscio, A. R., Parsons, S., Sierra, C., and Woodridge, M. Automated Negotiations: Prospects, Methods and Challenges. *International Journal of Group Decision and Negotiation 10*, 2 (2001), 199–215.

[40] Kagel, J. H., and Roth, A. E. *The Handbook of Experimental Economics*. Princeton University Press, Princeton, NJ, USA, 1995.

[41] Kersten, G. E. NEGO – Group Decision Support System. *Information and Management 8*, 5 (1985), 237–246.

[42] KERSTEN, G. E., KÖSHEGI, S., AND VETSCHERA, R. The Effects of Culture in Anonymous Negotiations: a Four Countries Experiment. Interim Report IR-99-023, IIASA, Laxenburg, Austria, July 1999.

[43] KERSTEN, G. E., AND NORONHA, S. J. Negotiation via the World Wide Web: A Cross-cultural Study of Decision Making. *Group Decision and Negotiations 8* (1999), 251–279.

[44] KERSTEN, G. E., AND NORONHA, S. J. WWW-based Negotiation Support: Design, Implementation and Use. *Decision Support Systems 25* (1999), 135–154.

[45] KOHONEN, T. The Self-Organizing Map. In *Proceedings of the IEEE* (Sept. 1990), vol. 78, pp. 1464–1480.

[46] KOHONEN, T. *Self-Organizing Maps*, 3rd ed. Information Sciences. Springer-Verlag, New York, Jan. 2001.

[47] KROVI, R., AND GRAESSER, A. C. Agent Behavior in Virtual Negotiation Environments. *IEEE Transaction on Systems, Man, and Cybernectics 29*, 1 (Feb. 1999).

[48] LASSILA, O., AND SWICK, R. R. Resource Description Framework (RDF) Model and Syntax, W3C Recommendation, 22 Feb. 1999.
http://www.w3.org/TR/REC-rdf-syntax.

[49] MANOUVRIER, M. *Objects similaires de grande taille dans les bases de données*. PhD thesis, University Paris-Dauphine, 2000.

[50] MATOS, N., SIERRA, C., AND JENNINGS, N. R. Determining Successful Negotiation Strategies: An Evolutionary Approach. In *Proc. of 3rd International Conference on Multi-Agent Systems, ICMAS-98* (Paris, France, 1998).

[51] MCAFEE, R., AND MCMILLAN, P. Auctions and Bidding. *Journal of Economic Literature 25* (1987), 699–738.

[52] MERZ, M., GRIFFEL, F., TU, M. T., MULLER-WILKEN, S., WEINREICH, H., BOGER, M., AND LAMERSDORF, W. Supporting Electronic Commerce Transactions with Contracting Services. *International Journal of Cooperative Information Systems 7*, 4 (1998), 249–274.

[53] NASH, J. Two-Person Cooperative Game. *Econometrica 18* (1953), 128–140.

[54] OIL. The Ontology Inference Layer (OIL) home page.
http://www.ontoknowledge.org/oil/.

[55] PELEGRÌ-LLOPART, E., AND CABLE, L. JavaServer Pages™Specification, version 1.1.
http://java.sun.com/products/jsp/download.html.

[56] **PICARD, W.** E-Negotiations of Contracts: Strategies and Issues. In *Economy in Transition – problems, ideas, solutions. Proceedings of Lubniewice 2000* (Lubniewice, Poland, May 2000), A. Janc, Ed., The Poznań University of Economics, Wydawnictwo Akademii Ekonomiczej w Poznaniu, pp. 242–247.

[57] **PICARD, W.** Collaborative Document Edition in a Highly Concurrent Environment. In *First International Workshop on Web-Based Collaboration, at the 12th International Workshop on Database and Expert Systems Applications, DEXA 2001* (Munich, Germany, Sept. 2001), A. M. Tjoa and R. R. Wagner, Eds., IEEE Computer Society, pp. 514–518.

[58] **PICARD, W.** Survey of e-contract negotiations issues. In *Research of Contemporary Economic, Issues by Young Economonists, Proceedings of Lubniewice 2001* (Lubniewice, Poland, May 2001), R. I. Zalewski, Ed., The Poznań University of Economics, Wydawnictwo Akademii Ekonomicznej w Poznaniu, pp. 378–385.

[59] **PICARD, W.** AND CELLARY, W. Generic Hierarchical Classification Using the Single-Link Clustering. In *Knowledge-Based Information Retrieval and Filtering*, W. Abramowicz, Ed., Kluwer Academic Publishers, Boston, USA. to appear.

[60] **PICARD, W.** AND CELLARY, W. Electronic negotiations in a highly concurrent environment. In *Towards the Knowledge Society. eCommerce, eBusiness and eGovernment* (2002), J. Monteiro, P. M. C. Swatman, and L. Valaderes Tavares, Eds., The Second IFIP Conference on E-Commerce, E-Business, E-Government (I3E 2002), Oct. 2002, Lisbon, Portugal, Kluwer Academic Publishers, Boston, USA, pp. 525–536.

[61] PRIZING, T. Using the Swing Text Package. http://java.sun.com/products/jfc/tsc/articles/text/overview/.

[62] RAIFFA, H. *The Art and Science of Negotiation*. Harvard University Press, 1982.

[63] RANGASWAMY, A., ELIASHBERG, J., BURKE, R., AND WIND, J. Developing Marketing Expert Systems: an Application to International Negotiations. *Journal of Marketing 53*, 4 (Oct. 1989), 24–49.

[64] RANGASWAMY, A., AND SHELL, G. R. Using Computers to Realize Joint Gains in Negotiations: Towards an "Electronic Bargaining Table". *Management Science 43*, 8 (Aug. 1997), 1147–1163.

[65] RAO, A. S., AND GEORGEFF, M. P. BDI Agents: From Theory to Practice. In *Proc. of 1st International Conference on Multi-Agent Systems, ICMAS-95* (San Francisco, USA, 1995).

[66] ROCHKIND, M. J. The Source Code Control System. *IEEE Transactions on Software Engineering SE-1*, 4 (Dec. 1975), 364–370.

[67] ROTHKOPF, M. H., AND HARSTAD, R. M. Modeling Competitive Bidding: A Critical Essay. *Management Science 40* (1994), 364–384.

[68] RUMBAUGH, J., JACOBSON, I., AND BOOCH, G. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.

[69] SCHOOP, M., AND QUIX, C. Towards Effective Negotiation Support in Electronic Marketplaces. In *Proc. of 10th Annual Workshop on Information Technologies & Systems, WITS 2000* (Brisbane, Australia, Dec. 2000), pp. 1–6.

[70] SCHOOP, M., AND QUIX, C. DOC.COM: a Framework for Effective Negotiation Support in Electronic Marketplaces. *Computer Networks 37*, 2 (2001), pp 153–170. Elsevier.

[71] SCHOOP, M., AND QUIX, C. DOC.COM: Combining Document and Communication Management for Negotiation Support in Business-to-Business Commerce. In *Proc. of the 34th Hawaii International Conference on System Sciences (HICSS-34)* (Maui, Hawaii, 2001).

[72] SEARLE, J. R. *Speech Acts – An Essay in the Philosophy of Language*. Cambridge University Press, 1969.

[73] SIERRA, C., FARATIN, P., AND JENNINGS, N. R. A Service-Oriented Negotiation Model between Autonomous Agents. In *Proc. of 8th European Workshop on Modeling Autonomous Agents in a Multi-Agent World, MAAMAW-99* (Ronneby, Sweden, 1997), pp. 17–35.

[74] SLEIN, J., VITALI, F., WHITEHEAD, E., AND DURAND, D. Requirements for a Distributed Authoring and Versioning Protocol for the World Wide Web, RFC 2291, Feb. 1998. http://www.ietf.org/rfc/rfc2291.txt.

[75] STRÖBEL, M. A Framework for Electronic Negotiations Based on Adjusted Winner Mediation. In *Proceedings of the 1st Workshop on e-Negotiations, DEXA2000* (London, UK, 2000).

[76] STRÖBEL, M. Communication Design for Electronic Negotiations on the Basis of XML Schema. In *Proceedings of the 10th World Wide Web Conference* (Hong Kong, China, 2001), ACM Press, New York, pp. 9–20.

[77] STRÖBEL, M. Design of Roles and Protocols for Electronic Negotiations. *Electronic Commerce Research, Special Issue on Market Design 1*, 3 (2001), 335–353.

[78] STRÖBEL, M., AND STOLZE, M. A Matchmaking Component for the Discovery of Agreement and Negotiation Spaces in Electronic Markets. In *Proceedings of the Group Decision and Negotiation Conference* (La Rochelle, France, 2001), pp. 61–75.

[79] SUN MICROSYSTEMS. JavaServer Pages$^{TM}$(JSP$^{TM}$). http://java.sun.com/products/jsp/.

[80] SUN MICROSYSTEMS. The Java$^{TM}$Foundation Classes (JFC). http://java.sun.com/products/jfc/.

[81] TICHY, W. F. RCS – A System for Version Control. *Software - Practice and Experience 15*, 7 (1985), 637–654.

[82] TU, M. T., GRIFFEL, F., MERZ, M., AND LAMERSDORF, W. A Plug-in Architecture Providing Dynamic Negotiation Capabilities for Mobile Agents. In *Proc. 2. Intl. Workshop on Mobile Agents, MA'98* (Stuttgart, Germany, Sept. 1998), Springer LNCS.

[83] TU, M. T., SEEBODE, C., AND LAMERSDORF, W. DynamiCS: An Actor-based Framework for Negotiating Mobile Agents. *Electronic Commerce Research Journal 1*, 1/2 (2001), 101–117.

[84] TU, M. T., WOLFF, E., AND LAMERSDORF, W. Genetic Algorithms for Automated Negotiations: A FSM-Based Application Approach. In *Proc. of 11th International Conference on Database and Expert Systems, DEXA 2000* (Greenwich, United Kingdom, 2000), IEEE.

[85] UNICODE CONSORTIUM, THE. *The Unicode Standard, Version 3.0.* Addison-Wesley, 11 Feb. 2000.

[86] VAN HARMELEN, F., HORROCKS, I., AND PATEL-SCHEIDER, P. F. A Model-Theoretic Semantics for DAML+OIL (March 2001), W3C Note, 18 Dec. 2001.
http://www.w3.org/TR/daml+oil-model.

[87] VON NEUMANN, J., AND MORGENSTERN, O. *Theory of Games and Economics Behavior*. Princeton University Press, Princeton NJ, USA, 1944.

[88] WEBGAIN, INC., AND SUN MICROSYSTEMS. The Java Compiler Compiler$^{TM}$(JavaCC) - The Java Parser Generator.
http://www.webgain.com/products/java_cc/.

[89] WEBONT. The W3C Web-Ontology (WebOnt) Working Group home page.
http://www.w3.org/2001/sw/WebOnt/.

[90] WILKENFELD, J., KRAUS, S., HOLLEY, K. M., AND HARRIS, M. A. GENIE: a Decision Support System for Crisis Negotiations. *Decision Support Systems 14* (Aug. 1995), 369–391.

[91] WOLFSTETTER, E. Auctions: An Introduction. *Journal of Economics Surveys 10* (1996), 367–420.

Appendix A

# XML Schemas for ADL

## A.1 Module Definition Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:attribute name="nameAttribute" type="xs:string"/>
  <xs:attribute name="pathAttribute" type="xs:string"/>
  <xs:attribute name="definitionAttribute" type="xs:string"/>

  <xs:complexType name="elementType">
    <xs:attribute ref="nameAttribute" use="required"/>
    <xs:attribute ref="definitionAttribute"/>
  </xs:complexType>

  <xs:complexType name="tagsType">
    <xs:sequence>
      <xs:element name="tag-decl"
                  type="elementType"
                  maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="functionsType">
    <xs:sequence>
      <xs:element name="function-decl"
                  type="elementType"
                  maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="objectsType">
    <xs:sequence>
      <xs:element name="object-decl"
```

```
                  type="elementType"
                  maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>

  <xs:element name="module">
    <xs:complexType>
      <xs:all>
        <xs:element name="tags" type="tagsType" minOccurs="0"/>
        <xs:element name="functions" type="functionsType" minOccurs="0"/>
        <xs:element name="objects" type="objectsType" minOccurs="0"/>
      </xs:all>
      <xs:attribute ref="nameAttribute" use="required"/>
      <xs:attribute ref="pathAttribute"/>
    </xs:complexType>
  </xs:element>

</xs:schema>
```

## A.2 MetaObject Definition Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:attribute name="type" type="xs:string"/>

  <xs:attribute name="name" type="xs:string"/>

  <xs:complexType name="attributeType">
    <xs:attribute ref="type" use="required"/>
  </xs:complexType>

  <xs:complexType name="attributesNameType">
    <xs:attribute ref="name" use="required"/>
    <xs:attribute ref="type" use="required"/>
  </xs:complexType>

  <xs:complexType name="attributesType">
    <xs:sequence>
      <xs:element name="attribute" type="attributesNameType"
                  minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>

  <xs:element name="object-def">
```

```
    <xs:complexType>
      <xs:sequence>
        <xs:element name="id" type="attributeType"/>
        <xs:choice>
          <xs:element name="value" type="attributeType"/>
          <xs:element name="attributes" type="attributesType"/>
        </xs:choice>
      </xs:sequence>
      <xs:attribute ref="type" use="required"/>
    </xs:complexType>
  </xs:element>

</xs:schema>
```

## A.3 Tag Definition Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:attribute name="name" type="xs:string"/>
  <xs:attribute name="lang" type="xs:string" default="Java"/>
  <xs:attribute name="type" type="xs:string"/>
  <xs:attribute name="required" type="xs:boolean" default="false"/>

  <xs:complexType name="paramType">
    <xs:attribute ref="name" use="required"/>
    <xs:attribute ref="type"/>
    <xs:attribute ref="required"/>
  </xs:complexType>

  <xs:complexType name="paramsType">
    <xs:sequence>
      <xs:element name="param" type="paramType"
                  minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="javaClassType">
    <xs:sequence>
      <xs:element name="params" type="paramsType"/>
    </xs:sequence>
    <xs:attribute ref="name" use="required"/>
    <xs:attribute ref="method"/>
    <xs:attribute ref="hasBody"/>
  </xs:complexType>
  <xs:element name="tag-def">
    <xs:complexType mixed="true">
      <xs:sequence>
        <xs:element name="javaClass" type="javaClassType" minOccurs="0"/>
      </xs:sequence>
      <xs:attribute ref="name" use="required"/>
      <xs:attribute ref="lang"/>
    </xs:complexType>
  </xs:element>

</xs:schema>
```

## A.4 Function Definition Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:attribute name="name" type="xs:string"/>
  <xs:attribute name="type" type="xs:string"/>

  <xs:complexType name="paramType">
    <xs:attribute ref="name" use="required"/>
    <xs:attribute ref="type" use="required"/>
  </xs:complexType>

  <xs:complexType name="resultType">
    <xs:attribute ref="name" use="required"/>
    <xs:attribute ref="type" use="required"/>
  </xs:complexType>

  <xs:complexType name="processingType">
    <xs:sequence>
      <xs:element name="anything" type="xs:anyType"
                  minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>

  <xs:element name="function-def">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="param" type="paramType"
                    minOccurs="0" maxOccurs="unbounded"/>
        <xs:element name="result" type="resultType" minOccurs="0"/>
        <xs:element name="processing" type="processingType"/>
      </xs:sequence>
      <xs:attribute ref="name" use="required"/>
    </xs:complexType>
  </xs:element>

</xs:schema>
```

APPENDIX B

# ADL EXPRESSIONS GRAMMAR

The ADL expressions grammar is specified as a JavaCC grammar[88].

```
<DEFAULT> TOKEN:
{
  <NOT_EXPRESSION : ((~["$"] | ("$" ~["{"]))+) | "$" >
  |
  <START_EXPRESSION: "${"> : IN_EXPRESSION
}

<IN_EXPRESSION> SKIP:
{
   " "
 | "\t"
 | "\n"
 | "\r"
}

<IN_EXPRESSION> TOKEN:
{
   <INTEGER_LITERAL : (["0"-"9"])+>
 | <#EXPONENT: ["e","E"] (["+","-"])? (["0"-"9"])+>
 | <FLOATING_POINT_LITERAL:
    (["0"-"9"])+ <EXPONENT>
    | "." (["0"-"9"])+ (<EXPONENT>)?
    | (["0"-"9"])+ "." (["0"-"9"])* (<EXPONENT>)?
   >
 | <STRING_LITERAL:
     ("\"" ((~["\"","\\"]) | ("\\" ( ["\\","\""] )))* "\"")
     | ("\'" ((~["\'","\\"]) | ("\\" ( ["\\","\'"] )))* "\'")
   >

 | <END_EXPRESSION: "}"> : DEFAULT
 | <TRUE: "true">
 | <FALSE: "false">
 | <NULL: "null" >
 | <DOT: ".">
 | <GT1: ">">
 | <GT2: "gt">
 | <LT1: "<">
 | <LT2: "lt">
 | <EQ: "==">
 | <LE1: "<=">
 | <LE2: "le">
 | <GE1: ">=">
 | <GE2: "ge">
 | <NE: "!=">
 | <LPAREN: "(">
 | <RPAREN: ")">
 | <LBRACKET: "[">
 | <RBRACKET: "]">
 | <PLUS: "+">
 | <MINUS: "-">
 | <MULTIPLY: "*">
 | <DIVIDE1: "/">
 | <DIVIDE2: "div">
 | <MODULUS: "mod">
 | <NOT: "not">
 | <AND: "and">
 | <OR: "or">
 | <EMPTY: "empty">

 | <IDENTIFIER: ["a"-"z","A"-"Z","_"] (["a"-"z","A"-"Z","_","0"-"9"])*>
}
```

APPENDIX C

# DETAILED COURSE OF NEGOTIATION EXAMPLE

Negotiators are denoted by their initials, i.e. *ST* for Scott Tiger, *WC* for Wile E. Coyote, and *CJ* for Chuck Jones, respectively. The original version of the contract is given in Section 8.1. For each contract version, the modifications provided in regards to its parent version are presented below.

**Version:0.1**

Owner:wc

| Previous version | New version |
|---|---|
| 10500 words | 15000 words |
| $2500, which will be paid within thirty (30) | $2500, which will be paid within fifteen (15) |
| Author within thirty (30) days | Author within ten (10) days |
| will make no major alterations | will not make any alterations |

**Version:0.1.1**

Owner:st

| Previous version | New version |
|---|---|
| 15000 words | 10000 words |
| $2500, which will be paid within fifteen (15) | $2800, which will be paid within twenty-five (25) |
| Author within ten (10) days | Author within twenty-five (25) days |
| thirty (30) days from receipt thereof | fifteen (15) days from receipt thereof |

**Version:0.1.1.1**

Owner:wc

| Previous version | New version |
|---|---|
| $2800, which will be paid within twenty-five (25) | $2700, which will be paid within twenty (20) |
| Author within twenty-five (25) days | Author within eighteen (18) days |

**Version:0.1.1.1.1**

Owner:st

| Previous version | New version |
|---|---|
| $2700, which will be paid within twenty (20) | $2500, which will be paid within twenty (20) |
| Author within eighteen (18) days | Author within twenty (20) days |

**Version:0.1.1.1.1.1**

Owner:wc

| Previous version | New version |
|---|---|
| $2500, which will be paid within twenty (20) | $2500, which will be paid within fourteen (14) |
| Author within twenty (20) days | Author within seven (7) days |

**Version:0.1.1.1.1.1.1**

Owner:st

| Previous version | New version |
|---|---|
| $2500, which will be paid within fourteen (14) | $2500, which will be paid within eighteen (18) |

**Version:0.2**

Owner:st

| Previous version | New version |
|---|---|
| 10500 words | 11500 words |
| $2500 | $2800 |
| Author within thirty (30) days | Author within twenty-one (21) days |

**Version:0.3**

Owner:cj

| Previous version | New version |
|---|---|
| 10500 words | 12000 words |
| $2500, which will be paid within thirty (30) | $3500, which will be paid within fifteen (15) |
| Author within thirty (30) days | Author within fifteen (15) days |
| will make no major alterations | will not make any alterations |
| thirty (30) days from receipt thereof | twenty (20) days from receipt thereof |
| the story as Wile E. Coyote and Chuck Jones | the story as Chuck Jones and Wile E. Coyote, from the WB funny group |

**Version:0.3.1**

Owner:st

| Previous version | New version |
|---|---|
| 12000 words | 12500 words |
| $3500, which will be paid within fifteen (15) | $2200, which will be paid within forty-five (45) |
| Author within fifteen (15) days | Author within twenty-eight (28) days |

**Version:0.3.2**

Owner:wc

| Previous version | New version |
|---|---|
| $3500, which will be paid within fifteen (15) | $3000, which will be paid within ten (10) |
| Author within fifteen (15) days | Author within eight (8) days |
| the story as Chuck Jones and Wile E. Coyote, from the WB funny group | the story as Wile E. Coyote and Chuck Jones, from the WB funny group |

**Version:0.3.2.1**

Owner:cj

| Previous version | New version |
|---|---|
| $3000, which will be paid within ten (10) | $3200, which will be paid within twenty (20) |
| the story as Wile E. Coyote and Chuck Jones, from the WB funny group | the story as the WB funny group |

**Version:0.3.2.1.1**

Owner:st

| Previous version | New version |
|---|---|
| $3200, which will be paid within twenty (20) | $2600, which will be paid within thirty (30) |

**Version:0.3.2.1.2**

Owner:cj

| Previous version | New version |
|---|---|
| $3200, which will be paid within twenty (20) | $3500, which will be paid within thirty (30) |

**Version:0.3.2.1.2.1**

Owner:st

| Previous version | New version |
|---|---|
| $3500, which will be paid within thirty (30) | $2000, which will be paid within fifteen (15) |

**Version:0.3.2.1.2.2**

Owner:wc

| Previous version | New version |
|---|---|
| $3500, which will be paid within thirty (30) | $2900, which will be paid within twenty-five (25) |

**Version:0.3.2.1.2.2.1**

Owner:cj

No modification

**Version:0.3.2.1.2.2.1.1**

Owner:st

No modification