UFR Informatique **École doctorale IIS Lyon**

Modéliser l'expérience pour en assister la réutilisation

De la Conception Assistée par Ordinateur au Web Sémantique

THÈSE

présentée et soutenue publiquement le 2 décembre 2002

pour l'obtention du

Doctorat de l'université Claude Bernard – Lyon 1

(spécialité informatique)

par

Pierre-Antoine Champin

Composition du jury

Président :	Jean-Marie Pinon
Rapporteurs :	Rose Dieng-Kuntz Amedeo Napoli
Examinateurs :	Laurent Di Cesare Jérôme Euzenat Alain Mille
Invitée :	Françoise Détienne

Laboratoire l'Ingénierie des Systèmes d'Information (LISI)
Laboratoire d' Informatique en Images et Systèmes d'information (LIRIS)

Mis en page avec la classe thloria.

Remerciements

Je tiens avant tout à remercier Alain Mille, mon directeur de thèse, dont j'ai grandement apprécié les qualités humaines et scientifiques pendant ces trois années de travail. Grâce à lui j'ai beaucoup appris, et apprécié, ce qui fait le métier de chercheur.

Je remercie Jean-Marie Pinon d'avoir accepté de présider mon jury. Sa présence pour conclure ce travail à d'autant plus de sens que c'est, entre autre, grâce à lui que j'ai pu le commencer.

Merci à Rose Dieng-Kuntz et Amedeo Napoli d'avoir accepté de rapporter cette thèse. L'intérêt qu'ils ont manifesté à l'égard de mon travail, par des remarques constructives et par les délais courts qu'ils ont acceptés et tenus, est un grand encouragement pour continuer.

Merci aussi à Françoise Détienne d'avoir accepté l'invitation à participer à ce jury, pour le point de vue «sciences humaines» qu'elle y a apporté.

Je souhaite enfin remercier Jérôme Euzenat pour ses conseils précieux, nos nombreuses discussions, et pour ses remarques parfois déroutantes mais toujours stimulantes.

Je souhaite remercier l'équipe Knowledgeware de Dassault Systèmes pour son accueil amical, et tout particulièrement Laurent Di Cesare et Pierre Grignon, qui m'ont encadré et soutenu par leur aide et leurs conseils.

Merci aussi à tous les habitants du bâtiment Nautibus pour l'ambiance chaleureuse qu'ils y font régner et l'émulation qu'elle permet. En particulier, ce travail doit beaucoup à Yannick Prié, Saurabh Sharma,Christine Solnon et Sébastien Sorlin.

Merci enfin à ma famille et ma belle famille, pour m'avoir épaulé, notamment pendant la rédaction de ce mémoire. Merci surtout à Marie-Pierre pour son soutien continu et sa confiance.

À Marie-Pierre, à Camil

Table des matières

Introduction

L'activité de conception tient une place très particulière dans les activités de résolution de problème. La complexité, au sens systémique, des problèmes de conception est ce qui les caractérise en premier lieu. Cette complexité se retrouve dans les connaissances mobilisées par les concepteurs, que ce soit dans le domaine de la conception mécanique, architecturale, logicielle ou autre. Nulle surprise, donc, à ce que ces connaissances se manifestent en grande partie sous la forme d'*expérience* : les concepteurs sont enclins à réutiliser les résultats ou les processus de conceptions antérieures.

Bien que les activités de conception soient le plus souvent médiées par une application informatique, les logiciels de Conception Assistée par Ordinateur (CAO) fournissent peu ou pas d'assistance pour la réutilisation. L'objectif de ce travail est donc de poser les bases à la construction d'un tel assistant, permettant la capitalisation de l'expérience des concepteurs, et sa réutilisation dans un contexte approprié.

Le projet ARDECO

Ce travail de thèse s'est inscrit dans le cadre du projet ARDECO (Aide à la Réutilisation D'Épisodes de COnception), lui même projet membre du programme PROSPER[1]. PROSPER est un programme du CNRS visant l'amélioration des connaissances sur les systèmes de production afin d'accroître leur efficacité. Il met l'accent sur le caractère interdisciplinaire de ces recherches afin de placer l'homme et son environnement au centre de la réflexion.

Le projet ARDECO[2] réunit, autour de la problématique de la réutilisation d'expérience, les partenaires suivants :

- Équipe Cognition et Expérience, Laboratoire l'Ingénierie des Systèmes d'Information (LISI), Université Claude Bernard Lyon 1 (http://experience.univ-lyon1.fr/)
- Équipe EIFFEL, INRIA Rocquencourt (http://www.inria.fr/recherche/equipes/eiffel)
- Équipe EXMO, INRIA Rhône-Alpes (http://www.inrialpes.fr/exmo/)
- Équipe KNOWLEDGEWARE, Dassault Systèmes (http://www.3ds.com/)

Le but de ce projet est la mise en place d'un assistant à la réutilisation d'expérience dans le cadre du logiciel de CAO développé par Dassault Systèmes : CATIA. Depuis la version 5 de CATIA, Dassault Systèmes met en effet l'accent sur la possibilité d'expliciter des connaissances, par opposition aux seules données géométriques manipulées en général dans les logiciels de CAO.

Pour modéliser cette expérience réutilisable, la notion d'*épisode de conception* est apparue comme une charnière entre les domaines de l'ergonomie cognitive (équipe EIFFEL) et de l'informatique (équipes EXMO et Cognition et Expérience). C'est donc cette notion qui a servi de base à la collaboration interdisciplinaire dans ce projet.

[1] http://www.univ-valenciennes.fr/PROSPER/
[2] http://www710.univ-lyon1.fr/~champin/ardeco

Assister la réutilisation d'expérience

Lorsqu'un concepteur sollicitera l'aide de l'assistant ARDECO, celui-ci lui proposera de réappliquer une partie d'une activité de conception antérieure, dont il aura jugé qu'elle est pertinente dans la situation courante. C'est cette partie réutilisable de l'activité de conception qui constitue un épisode de conception. Pour pouvoir décider de la pertinence à réutiliser un épisode, l'assistant doit donc garder la trace du *contexte* dans lequel cet épisode a eu lieu initialement. Le raisonnement à partir de cas, un champ de l'intelligence artificielle, s'intéresse justement à l'exploitation de connaissances contextualisées, et sert de base à ce travail.

L'assistant doit donc permettre la capitalisation et la réutilisation, sous forme d'épisodes, de l'expérience de conception accumulée par les concepteurs. Un point important dans ce cahier des charges est le caractère non intrusif que devrait conserver cet assistant : la capitalisation devrait en effet interférer le moins possible avec l'activité des concepteurs. Le but n'est pas de forcer les concepteurs à favoriser la réutilisation de leur travail, mais de permettre cette réutilisation le cas échéant.

Il en découle que toutes les connaissances mises en œuvre par les concepteurs ne seront pas nécessairement explicitées par ces derniers. Si certaines connaissances *vont sans dire*, littéralement, c'est parce que les concepteurs sont généralement capables de les retrouver ou de les reconstruire lorsqu'ils se trouvent dans le contexte approprié. Le *document* de conception, produit de l'activité du concepteur, porte donc ces connaissances aux yeux des concepteurs, bien qu'elles ne soient pas sous une forme exploitable par la machine. Cette problématique de l'exploitation des connaissances « enfouies » dans les documents numériques intéresse depuis quelques années les chercheurs dans le domaine plus général du Web Sémantique. C'est pourquoi ce travail aboutit à une réflexion sur la place de la réutilisation d'expérience dans le Web Sémantique, et propose un modèle général de capitalisation et réutilisation d'épisodes dans ce cadre.

Plan du mémoire

Dans la première partie du mémoire seront exposées les bases scientifiques de ce travail. Elles sont organisées selon trois axes qui en constituent en quelque sorte le référentiel scientifique. Dans le chapitre 1, la notion d'assistant sera présentée du point de vue de l'intelligence artificielle, et plus particulièrement du raisonnement à partir de cas. Les difficultés rencontrées par l'intelligence artificielle y seront également soulignées, et dans quelle mesure le raisonnement à partir de cas permet de les contourner. Le chapitre 2 donnera un aperçu pluridisciplinaire de l'activité de conception, dont les spécificités sont mises en évidence. Ce chapitre présentera également un certain nombre d'assistants informatiques à la conception dans divers domaines. Enfin, le chapitre 3 s'intéressera à la notion de document et aux relations qu'elle entretient avec la notion de connaissance. Cela conduira à présenter les travaux récents autour du Web Sémantique.

La seconde partie présentera le travail relevant directement du projet ARDECO. Dans le chapitre 4, l'application CATIA sera présentée, et la notion d'épisode de conception sera définie précisément. Une modélisation des épisodes de conception sera également proposée. Le chapitre 5 s'intéressera quant à lui à l'exploitation des épisodes ainsi modélisés, pour leur réutilisation. Cette exploitation s'inspire notamment des travaux en raisonnement à partir de cas autour des phases de remémoration et de réutilisation. Enfin, le chapitre 6 décrira le prototype d'assistant implantant les mécanismes décrits dans les deux chapitres précédents.

La troisième et dernière partie proposera une ouverture de ce travail. Le chapitre 7 présentera dans ce but une généralisation des idées développées dans la deuxième partie, afin de permettre

leur application à des domaines différents, notamment à celui du Web Sémantique. Enfin, le chapitre 8 conclura et discutera d'un certain nombre de perspectives à ce travail.

Première partie

Référentiel scientifique

Chapitre 1

De l'expert à l'assistant

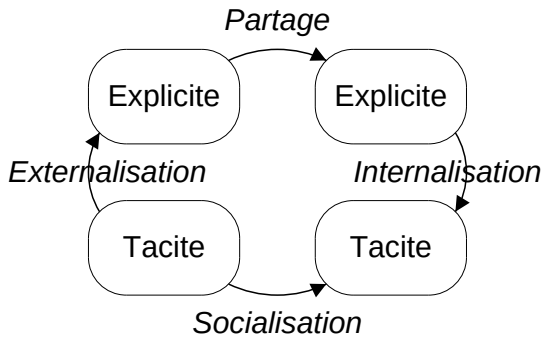
Ce chapitre a pour objectif de présenter les fondements en intelligence artificielle (IA) de ce travail, notamment le paradigme du raisonnement à partir de cas (RàPC). On discutera aussi du changement d'ambition que représente ce nouveau paradigme dans la discipline : le modèle (au sens de ce que l'on cherche à modéliser) n'est plus un expert capable de répondre à une question bien posée, mais un assistant capable d'aider l'utilisateur à répondre à des questions complexes et souvent mal posées. La section 1.1 décrit les écueils rencontrés par le modèle de l'expert. La section 1.2 présente le raisonnement à partir de cas, modèle de raisonnement qui permet, dans une certaine mesure, de contourner ces écueils. Enfin, la section 1.3 présente les spécificités du modèle de l'assistant.

1.1 Premières ambitions de l'intelligence artificielle

Une des premières ambitions de l'IA a été de capturer et d'exploiter automatiquement les connaissances relatives à un domaine particulier. C'est le modèle de l'*expert*, que l'on cherche, sinon à supplanter, au moins à égaler. Ce paradigme des systèmes experts se heurte à plusieurs écueils.

La difficulté à capturer les connaissances Outre le problème social (l'expert est réticent à dévoiler des connaissances qui le rendent nécessaire, parfois indispensable à son entreprise, de peur d'être remplacé par la machine), se pose un véritable problème cognitif : nombre des connaissances d'un expert sont des connaissances *tacites*, qu'elles le soient devenues par la bonne appropriation d'un apprentissage explicite, ou bien qu'elles aient même été *acquises* sous forme tacite, ce que Nonaka et Takeuchi [1995] appellent la «socialisation» (cf. figure 1.1).

La difficulté à représenter les connaissances Les langages issus de la logique utilisés dans les systèmes experts, s'ils sont appropriés pour l'automatisation du raisonnement, ne le sont pas forcément pour représenter les connaissances liées à un domaine d'expertise. Les langages de représentation des connaissances par objets [Euzenat et Rechenmann, 1995; Pachet, 1997], ainsi que des variantes de langages logiques intégrant des éléments de l'approche objet [Kifer *et al.*, 1995; Napoli, 1997] ont permis, dans une certaine mesure, de combler cet écart. D'autres efforts, comme les graphes conceptuels de Sowa [1999] proposent des représentations graphiques sous forme de réseaux sémantiques afin de faciliter leur compréhension par des novices. Enfin des systèmes comme CommonKADS [Schreiber *et al.*, 1994] proposent un cadre plus abstrait de mod-



Les flèches dans le sens vertical représentent les transformations des connaissances individuelles : l'« explicitation » de tacite vers explicite, et l'« internalisation » ou appropriation, d'explicite vers tacite. Les flèches dans le sens horizontal représentent les transferts interindividuels de connaissances, explicites dans le cas du « partage », ou tacites dans le cas de la « socialisation ».

FIG. 1.1 – Transformations et transferts des connaissances d'après Nonaka et Takeuchi [1995]

élisation des connaissances, suivant l'hypothèse de Newell [1982] selon laquelle les connaissances peuvent être modélisées indépendamment de la représentation utilisée pour leur exploitation.

La difficulté à exploiter les connaissances L'enjeu de nouveaux langages pour représenter les connaissances n'est pas seulement dans l'expressivité qu'ils offrent pour modéliser les connaissances de l'expert, ou dans leur facilité à être utilisés par des non-informaticiens (pour renseigner le système, pour interpréter ses résultats ou ses explications, etc.). Les mécanismes de raisonnement sur ces langages doivent posséder de bonnes propriétés de *complétude* et de *correction* : ils doivent fournir uniquement de bonnes réponses (être corrects) et fournir toutes les bonnes réponses (être complets), et ce bien sûr avec une complexité algorithmique qui reste praticable. C'est sans doute dans le domaine des logiques de descriptions que le compromis entre expressivité et efficacité a été le mieux étudié : en choisissant le type d'opérateurs dont on veut disposer, on peut savoir quelle sera la complexité des algorithmes d'inférence (corrects et complets) correspondants [Napoli, 1997].

1.2 Le raisonnement à partir de cas

1.2.1 Fondements cognitifs

Le paradigme du raisonnement à partir de cas apparaît à la fin des années 70 avec la prise de conscience en psychologie cognitive de l'importance relative de la mémoire *épisodique* par rapport à la mémoire *sémantique*, c'est-à-dire des connaissances contextualisées, sous forme de souvenirs ou d'*expériences*, par rapport aux connaissances générales et abstraites, sous forme de concepts ou de règles. Jusqu'alors, on considérait que la mémoire sémantique occupait une place prépondérante dans les mécanismes de raisonnement. Schank et Abelson [1977] suggèrent alors que le déroulement d'une situation est anticipé à l'aide de *scripts* qui permettent de guider le comportement à adopter. Ces scripts peuvent êtres issus d'une situation antérieure particulière ou bien d'une généralisation. Un script *Restaurant* par exemple, décrira le déroulement habituel d'un dîner au restaurant (placement, commande à table, dîner, paiement). Applicable dans la plupart des situations, il devra être révisé dans certains cas particuliers comme un fast-food (commande à la caisse, paiement, placement, dîner).

Certains vont même jusqu'à proposer des modèles de système unique de mémoire [Whittlesea, 1987; Rousset, 2000] dans lesquels la mémoire sémantique ne serait qu'une émergence de la mémoire épisodique[1].

[1] Par exemple, dans MINERVA II [Hintzman, 1986], la mémoire n'est constituée que d'un ensemble de traces

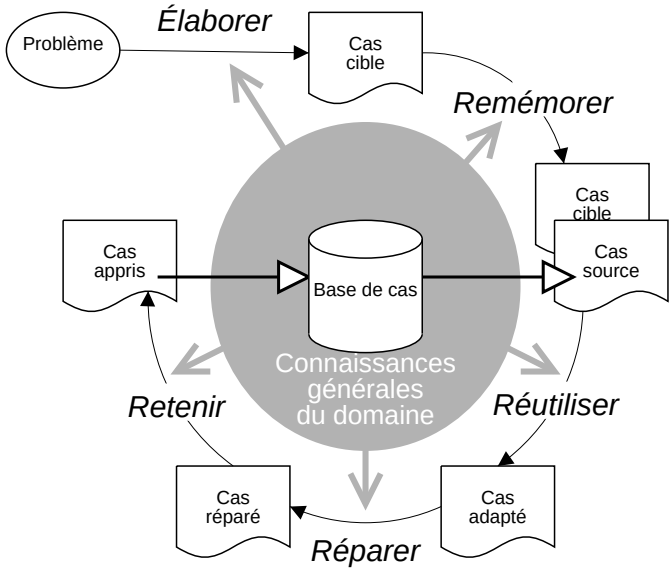


FIG. 1.2 – Le cycle du raisonnement à partir de cas, d'après Aamodt et Plaza [1994] et Mille [1998]

Nombre des modèles proposés font l'objet d'implantations informatiques à des fins de simulation et de validation. C'est notamment le cas du modèle de Schank [1982] dont l'implantation, MOPS (*Memory Organisation PacketS*), permet l'indexation dans la même structure, d'expériences concrètes et de connaissances abstraites. Ce modèle a donné lieu aux premiers systèmes de RàPC [Lebowitz, 1983; Kolodner, 1983].

1.2.2 Le cycle du raisonnement à partir de cas

Puisque l'être humain trouve souvent plus facile de résoudre des problèmes en réutilisant des solutions connues à des problèmes similaires, pourquoi ne pas appliquer ce principe à l'IA ? Aamodt et Plaza [1994] ont fait une revue des travaux fondés sur cette idée, mais ont également proposé une formalisation générale des mécanismes impliqués dans ce type de raisonnement : le cycle du RàPC (figure 1.2). Ce cycle, organisé autour d'une base de connaissances, se compose de 5 phases détaillées par la suite.

Base de connaissances

Comme il a déjà été dit, la particularité première du raisonnement à partir de cas est de reposer sur des connaissances *concrètes*, c'est-à-dire décrivant des situations réelles. Cette particularité constitue une façon de représenter l'expérience, notion difficile à prendre en compte dans les systèmes à base de règles, et qui nous intéresse au premier chef dans ce travail. Ces situations réelles, généralement présentées comme un problème et sa solution, sont appelées *cas* et sont stockées dans une *base de cas* représentée au centre de la figure 1.2.

correspondant à des souvenir précis. Tout nouveau stimulus (appelé *sonde*) active plus ou moins chacune des traces en fonction de leur similarité à la sonde. La somme de toutes ces activations (l'*écho* de la sonde) ne correspond à aucune des traces stockées, mais est une généralisation des traces les plus activées. On peut en ce sens l'assimiler à un concept.

Cependant, il serait une erreur de considérer que le RàPC ne doit exploiter que ce type de connaissances. Richter [1995] souligne que le RàPC utilise en fait quatre «conteneurs de connaissances» : la base de cas, la mesure de similarité, les mécanismes d'adaptation, et le vocabulaire pour décrire les trois précédents. On peut généraliser cette position en considérant que des connaissances supplémentaires à la base de cas sont nécessaires à toutes les étapes du cycle (ce qui est représenté par les flèches grises sur la figure 1.2). Sans ces connaissances, seul l'outil statistique permettrait l'exploitation des cas. Ce sont elles qui permettent d'expliquer les cas, et donc qui leur confèrent à leur tour un statut de connaissances en soi.

Le premier type de connaissances mobilisées par le cycle et indissociables de la base de cas, réside dans le langage de description des cas ; il correspond à ce que Richter appelle le vocabulaire. Toutes les étapes du cycle s'appuient nécessairement sur la façon dont les cas sont représentés, et notamment sur la capacité de cette représentation à capturer les éléments pertinents liant un problème à sa solution. Les cas peuvent être représentés comme de simples listes d'attributs-valeurs, on parle alors de cas-vecteurs. Ils peuvent aussi avoir des représentations plus structurées [Plaza, 1995], voire même être reliés à d'autres cas par des relations structurelles (décomposition hiérarchique de cas en sous-cas pour Smyth et Keane 1995b). Enfin, en plus du problème et de sa solution, la structure des cas peut également permettre de représenter des explications sur les dépendances entre éléments de problèmes et éléments de solution, ce qui sera utile lors de la phase de réutilisation.

Les cinq étapes

Élaborer Cette phase, absente du cycle original, a été suggérée par Mille [1998]. Elle consiste dans la transformation d'un problème «brut» en un cas, bien entendu incomplet puisqu'il n'a pas encore de solution. Ce cas est appelé le *cas cible* (celui que l'on cherche à résoudre). Cette transformation ne se réduit pas à la simple transcription du problème dans le formalisme de représentation : les connaissances du domaine y sont déjà utilisées pour déduire ou requérir, selon le cas, les éléments nécessaires manquants.

Remémorer Cette phase consiste dans la récupération, dans la base de cas, d'un cas complet (comportant une solution) supposé réutilisable pour résoudre le cas cible. Le cas remémoré est appelé le *cas source*. C'est sans doute cette phase du RàPC qui a été la plus étudiée, de nombreuses mesures de *similarité* ayant été proposées pour divers types de cas et divers domaines d'application. Bouchon-Meunier *et al.* [1996] proposent une classification des mesures de similarité s'inspirant notamment des travaux en psychologie de Tversky [1977].

Un critère d'évaluation d'une mesure de similarité est bien sûr la pertinence des cas remémorés. On distingue généralement dans un cas les traits de structure des traits de surface [Chi *et al.*, 1981], éléments qu'il est pertinent (respectivement non pertinent) de prendre en compte. Sebag et Shoenauer [1993] proposent d'appliquer des techniques d'apprentissage automatique sur la base de cas pour définir une mesure de similarité pertinente. Rifqi *et al.* [2000] définissent quant à elles un critère de «pouvoir de discrimination» de différentes mesures de similarité afin de pouvoir en sélectionner une.

Un autre critère est bien sûr l'efficacité, puisque la quantité de cas à comparer au cas cible peut être assez importante. Une méthode classique est d'appliquer une première phase de filtrage, ne conservant qu'une partie des cas, sur la base d'une mesure de comparaison grossière mais rapide, puis d'une phase de similarité à proprement parler, appliquée seulement aux cas retenus par le filtrage [Gentner et Forbus, 1991].

Enfin, il est important de souligner que la phase de remémoration est en amont de la phase de réutilisation, et que sa vocation est à ce titre de fournir un cas *réutilisable*. Certains auteurs suggèrent donc que les connaissances de similarité sont duales des connaissances d'adaptation [Lieber et Napoli, 1998], et que la remémoration doit être guidée par les connaissances d'adaptation [Smyth et Keane, 1995b; Griffiths *et al.*, 1999; Bergmann *et al.*, 2001] (ce qui remet en cause l'« étanchéité » des conteneurs de connaissances de Richter).

Réutiliser Durant cette phase, la solution du cas source est réutilisée pour résoudre le problème du cas cible. Cette réutilisation fait le plus souvent l'objet d'une *adaptation*. Au contraire de la précédente, cette phase a été longtemps délaissée, sans doute à cause de sa difficulté, les connaissances d'adaptation étant difficiles à acquérir et à formaliser. On a même argumenté qu'il était préférable de laisser le soin à l'utilisateur d'effectuer lui même l'adaptation, processus créatif plus adapté à l'humain qu'à la machine [cité par Hanney, 1996, p.10].

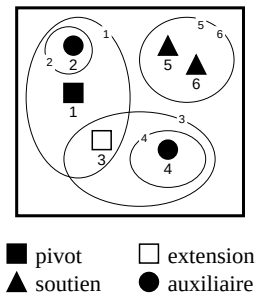
On compte malgré tout un certain nombre d'efforts pour formaliser l'adaptation. D'abord, pour essayer de distinguer les différents types d'adaptation [Voss, 1996; Wilke et Bergmann, 1998]. On peut en distinguer trois :

- L'adaptation nulle consiste simplement à recopier la solution source dans le cas cible. Lieber [2002] étudie les différentes situations pouvant impliquer ce type d'adaptation.
- L'adaptation par transformation consiste à modifier la solution source. Cette modification peut, selon le cas, se limiter à changer des valeurs d'attributs, ou modifier plus profondément la structure de la solution. Fuchs *et al.* [2001] ont proposé un algorithme générique pour ce type d'adaptation.
- L'adaptation générative consiste à « rejouer » la résolution du cas source, en l'appliquant aux données du cas cible. Elle suppose donc en général l'existence d'un algorithme capable en principe de résoudre la solution seule, mais tirant parti des connaissances apportées par le cas source. Elle suppose aussi que les cas contiennent non seulement leur solution, mais aussi une description plus ou moins détaillée de la façon dont cette solution a été obtenue.

D'autres travaux s'intéressent aussi à la possibilité d'extraire de la base de cas les connaissances d'adaptation. Hanney [1996] propose d'appliquer des techniques d'apprentissage pour extraire des règles d'adaptation. Leake *et al.* [1997] proposent quant à eux de stocker de précédentes adaptations comme des *cas d'adaptation*, et de ré-appliquer un mécanisme de RàPC pour la seule phase de réutilisation, donnant lieu en quelque sort à une « adaptation à partir de cas ».

Réparer Les connaissances d'adaptation ne sont pas toujours suffisantes pour garantir que la solution du cas adapté soit valide. Dans certains systèmes, le cas adapté est alors soumis à une vérification (test en situation réelle, simulation ou évaluation par un utilisateur) et à une correction (réparation) de la solution en cas de problème. On peut citer par exemple le système CHEF, capable de réviser un plan qu'il a réutilisé et adapté, sur la base de relations causales expliquant l'échec de ce plan [Hammond, 1990].

Retenir Une conséquence du rôle central des cas dans le RàPC est que les problèmes résolus par le système peuvent venir enrichir la base de cas pour servir à des résolutions ultérieures. C'est ce bouclage qui légitime l'analogie entre le RàPC et la notion d'expérience. Ce mécanisme d'apprentissage suppose bien sûr une *indexation* des cas dans la base de cas à l'aide d'une structure adéquate, comme par exemple celle fournie par le système MOPS [Schank, 1982] évoqué précédemment.



On peut distinguer quatre types de cas, en fonction de leur accessibilité et de leur couverture. Les cas auxiliaires *ont une couverture totalement englobée par d'autres cas, et peuvent donc être supprimés sans perte de compétence. Les cas de* soutien *forment des groupes ayant la même couverture, on peut donc les supprimer à condition de garder un membre du groupe. Les cas d'*extension *sont couverts par d'autres, mais leur couverture ne l'est pas. Les cas* pivots*, quant à eux, ne sont couverts par aucun autre cas. Les cas des deux dernières catégories ne peuvent pas être supprimés sans diminuer les compétences du système.*

FIG. 1.3 – Compétence des cas, d'après Smyth et Keane [1995a]

Comme il n'existe généralement pas de méthode d'indexation parfaite pour les cas (et que son existence supposerait qu'on sache résoudre les problèmes sans avoir recours au RàPC), la taille de la base de cas ne peut pas croître indéfiniment sous peine de rendre la phase de remémoration prohibitivement longue. Cette phase d'apprentissage ne se limite donc pas à l'ajout et à l'indexation de chaque nouveau cas dans la base ; elle suscite aussi un processus de *maintenance* de la base de cas, qui peut être réalisé à chaque nouveau cycle (« Ce cas doit-il être retenu ou non ? ») ou comme un cycle parallèle à celui du raisonnement [Roth-Berghofer et Iglezakis, 2001]. Cette phase de maintenance doit contrôler l'entrée de nouveau cas dans la base, (par exemple, en abstrayant les cas pour augmenter leur couverture, comme le proposent Bergmann *et al.* 1999) mais aussi éventuellement en supprimer certains, et ce en préservant la *compétence* du système [Smyth et Keane, 1995a].

Préserver les compétences du système suppose de tenir compte de la *couverture* de chaque cas, c'est-à-dire de la partie de l'espace des problèmes pour laquelle sa solution peut être réutilisée. Comme la couverture théorique d'un cas ne peut généralement pas être calculée, Smyth et McKenna [2002] proposent de l'évaluer par l'ensemble des cas de la base couverts par ce cas (en faisant l'hypothèse que cette dernière est représentative de l'espace des problèmes[2]). La compétence d'un cas découle alors de son accessibilité (l'ensemble des cas qui le couvrent) et de sa couverture (l'ensemble des cas qu'il est le seul à couvrir, cf. figure 1.3).

1.2.3 Apports et limites du RàPC

Il est intéressant de mettre en regard les spécificités du RàPC et les difficultés rencontrées par l'IA et qui ont été évoquées plus haut. D'abord, concernant la **capture des connaissances** : il est beaucoup plus facile aux experts de faire part de leur expérience à travers des anecdotes et des situations vécues qu'en essayant de les généraliser. La gestion des *exceptions* a donné lieu à plusieurs travaux dans le domaine des systèmes à base de règles, notamment avec les logiques non monotones [Schlechta, 1997] et les logiques de défaut [Reiter, 1999; van der Torre et Tan, 1997]. En RàPC, la notion d'exception n'a même pas lieu d'être, puisque la notion de règle est également absente : un cas sera plus ou moins typique, selon qu'il est souvent réutilisable ou non, c'est-à-dire qu'il correspond à une situation *régulière* ou *exceptionnelle*[3].

[2] Smyth et McKenna justifient cette hypothèse en avançant qu'elle est faite implicitement dès que le RàPC est considéré pour résoudre une classe de problème (sans quoi c'est sa capacité même à répondre à ce cahier des charges qui est remise en cause).

[3] Cette règle souffre cependant des exceptions ! Dans un système comme CHEF, lorsqu'une adaptation échoue, le cas réparé est mémorisé comme une *exception* au cas initialement adapté, et ce afin d'éviter de reproduire la même erreur d'adaptation. Ce mécanisme de structuration de la mémoire permet en fait de palier des régularités

Par ailleurs, et c'est un argument souvent avancé en faveur des systèmes de RàPC, une bonne partie des connaissances nécessaires au système peut être acquise de façon automatique ou semi-automatique : la base de cas peut être constituée en réutilisant des traces d'activité généralement disponibles (sous forme de bases de données le plus souvent). Comme on l'a vu plus haut, on peut aussi envisager d'extraire de cette base de cas initiale des connaissances de similarité [Sebag et Shoenauer, 1993] ou d'adaptation [Hanney, 1996; Leake *et al.*, 1997]. Enfin, même si la base de cas initiale est pauvre, ce n'est pas un problème à long terme puisque le système apprend de nouveaux cas au fur et à mesure des utilisations.

En revanche, si le problème de l'acquisition des connaissances semble simplifié par le RàPC, ce dernier pose sous une nouvelle forme le problème de la **représentation des connaissances**. Ce que Richter [1995] appelle le vocabulaire, l'ensemble des connaissances impliquées dans la première phase du cycle, est très dépendant du domaine, et influence toutes les autres phases.

Qu'en est-il de l'**exploitation des connaissances** du point de vue du RàPC ? Son hypothèse principale est que pour résoudre un problème, il est parfois plus intéressant d'adapter une solution connue à un problème proche, que d'essayer de le résoudre à partir de rien — par «plus intéressant», on entendra «plus rapide», «plus facile», «plus économique» ou tout autre critère jugé d'intérêt pour le problème considéré. Cette hypothèse est séduisante car elle est définitivement vérifiée dans notre expérience quotidienne. Cependant sa validité informatique est moins assurée.

Dans le cas particulier de la planification à partir de cas, Nebel et Köhler [1995] démontrent que la remémoration et l'adaptation d'un plan réutilisable est souvent plus complexe (du point de vue théorique) que la génération d'un plan à partir de zéro. Empiriquement cependant, la réutilisation de plans existants améliore ou dégrade les performances selon le domaine où elle est appliquée : dans un domaine homogène (où tous les objets ont le même type, tel un monde de blocs), la remémoration est pénalisante car le nombre de combinaisons possible pour comparer deux cas est très important (ce qui relève du problème plus général de *symétries*, bien connu en résolution de problèmes combinatoires, cf. Flener *et al.* 2002). À l'inverse, dans un domaine hétérogène, la remémoration est suffisamment efficace pour rendre la réutilisation de plans plus performante que la génération.

Ces résultats mitigés sont bien sûr à relativiser dans la mesure où ils s'appliquent à un domaine pour lequel on sait en principe construire une solution optimale depuis zéro (fût-ce en un temps très grand !). En revanche dans de nombreux domaines complexes, comme celui de la conception qui nous intéresse dans ce travail, de telles méthodes n'existent pas, et la notion même de solution optimale n'a pas forcément de sens. Dans de tels domaines, le critère définissant l'intérêt du RàPC passe au niveau méta : il est plus facile de *mettre en place* ce type de raisonnement que de tenter de développer des méthodes de résolution pour le type de problème considéré, car les connaissances de similarité et d'adaptation sont plus faciles à acquérir et formaliser que des connaissances de résolution classiques.

Enfin, l'apprentissage de nouveaux cas au cours de l'exploitation du système pose des problèmes supplémentaires. D'une part, le raisonnement n'est pas strictement reproductible, puisque chaque itération du cycle est susceptible d'enrichir la base de cas et donc d'influer sur le déroulement des itérations suivantes. Idéalement, en soumettant deux fois de suite le même problème au système, la seconde réponse devrait consister dans la réutilisation de la première. D'autre part, on peut s'inquiéter de la propension d'un tel système à apprendre de «mauvais» cas[4]. On

inappropriées induites par le mécanisme de remémoration (le fait que des situations similaires à l'«exception» risquent de remémorer la «règle» en premier lieu).

[4] À prendre ici du point de vue du domaine d'application plutôt que de celui du RàPC, déjà évoqué à propos

voit bien que cette faculté d'apprentissage, souvent présentée comme un « effet de bord » des mécanismes du RàPC, constitue son originalité et une différence fondamentale par rapport à d'autres paradigmes.

1.3 Le modèle de l'assistant

L'ordinateur constitue aujourd'hui le poste de travail d'innombrables personnes, et est à la fois un outil de travail, un moyen de communication et un accessoire de loisir. Le modèle de l'assistant prend tout son sens dans ce contexte où cet ordinateur n'est plus l'exécutant d'une tâche particulière qui réclame qu'on s'adapte à lui, mais participe à une myriade de tâches diverses pour lesquelles on souhaite qu'il s'adapte à son utilisateur.

1.3.1 Un assistant « intelligent »

L'assistance à l'utilisateur est d'abord du ressort de l'interface homme-machine et de l'ergonomie. C'est avec les exigences croissantes à l'égard des systèmes à large utilisation qu'elle commence à concerner l'IA. Il faut noter que l'assistance telle qu'elle est envisagée dans ce travail ne se limite pas à une documentation de l'application, mais à l'ensemble des mécanismes qui permettent d'améliorer ses conditions d'utilisation. Cette définition, à l'image de certaines définitions de l'IA, est relative à un certain état des connaissances, puisqu'une fois implanté, un mécanisme d'assistance *fait partie* de l'application... et devient lui même sujet à amélioration. Comment caractériser l'« intelligence » d'un système d'assistance ? Suit une liste, sans doute non exhaustive, de qualités qu'un tel système devrait posséder.

Intelligible Pour paraître intelligent, un système doit d'abord être *intelligible* pour son interlocuteur, à savoir l'utilisateur. Au minimum, cette intelligibilité est constituée par l'auto-documentation. Pour les commandes en ligne, c'est la capacité d'un programme à donner un bref mode d'emploi à son sujet, à la demande ou spontanément lorsqu'il est mal utilisé. Quant aux interfaces graphiques, on peut considérer qu'elles sont également une sorte d'auto-documentation : les commandes disponibles sont présentes à l'écran, qui plus est sous une forme métaphorique évoquant leur utilisation (bouton, interrupteur, potentiomètre). Leur regroupement (géométrique, en menus, en pages) permet également d'indiquer la façon dont elles sont reliées les unes aux autres. Enfin, l'homogénéité (y compris inter-applications) des interfaces graphiques, prônée depuis leur apparition, participe de leur intelligibilité.

Notons que cette intelligibilité passe aussi par une certaine transparence, notamment à l'égard des actions entreprises automatiquement par l'assistant : celles-ci doivent conserver une certaine visibilité pour l'utilisateur (être soumises à validation, ou au moins signalées et révocables). En effet, une trop grande autonomie est parfois perçue comme une gêne ou une perte de contrôle par l'utilisateur. On y reviendra ci-après en abordant la robustesse et l'adaptativité.

Contextualisé Un système d'assistance doit être capable de tenir compte de l'état de l'application dans ses interactions avec l'utilisateur. C'est le cas de l'aide contextuelle, affichant la page de l'aide en ligne la plus appropriée en fonction du contexte, ou de certains compilateurs, notamment ADA, accompagnant les messages d'erreurs du numéro de la page correspondante dans le manuel (papier).

---

de la phase « Retenir » ; mais un cas sortant des canons du domaine est-il mauvais pour autant ?

Plus récemment sont apparus les *wizards* [5] qui guident l'utilisateur dans une tâche particulière, en tenant compte à la fois des spécificités de la tâche et des choix qui sont faits par l'utilisateur au cours de leur déroulement. On peut les rapprocher des systèmes de complétion de certains interpréteurs de ligne de commande, capables de compléter automatiquement un mot en fonction de son contexte (nom de commande pour le premier mot, nom de fichier pour les suivants, par exemple). Notons que les *wizards* et les systèmes de complétion suffisamment évolués, s'ils sont au départ destinés à *limiter* le champ des actions possibles pour faciliter la tâche de l'utilisateur et limiter les risques d'erreur, peuvent aussi lui servir de documentation sur la façon dont fonctionne l'application.

Robuste L'intelligence d'un système se caractérise aussi par sa robustesse, c'est-à-dire sa capacité à passer outre une différence entre ce qu'il attend de l'utilisateur, et ce que l'utilisateur lui fournit effectivement. C'est la vocation des outils d'interrogation acceptant les questions en langue naturelle (où la même requête peut être formulée par une grande variété de questions), ou des systèmes de correction automatique des fautes de frappe.

Cette caractéristique est plus délicate à mettre en œuvre dans la mesure où elle peut nuire à l'intelligibilité du système si ses « interprétations » s'avèrent non pertinentes. Un système robuste devrait donc être capable de signaler, voire d'expliquer ses interprétations à l'utilisateur, et si possible de lui permettre de les remettre en cause.

Adaptatif L'adaptativité des interfaces homme-machine est le sujet de recherche d'une communauté croissante de chercheurs [Langley, 1999], notamment dans le domaine des documents numériques [Garlatti et Crampes, 2002] et des hypermédia [Brusilovsky, 2001]. On y distingue les objets *adaptables* des objets *adaptatifs*. Les premiers peuvent être adaptés par une action émanant de l'extérieur — généralement l'utilisateur, mais il peut aussi s'agir, par exemple, d'un enseignant adaptant un document pour ses élèves. Rentrent dans cette catégorie les applications proposant plusieurs niveaux d'utilisateur (novice, confirmé, expert), offrant des interfaces plus ou moins complètes et adaptées à chaque niveau.

Les objets adaptatifs, quant à eux, sont capables de s'adapter automatiquement en fonction du contexte. Pour reprendre l'exemple précédent, une application adaptative changera d'elle-même le niveau de l'interface lorsque la tâche de l'utilisateur nécessitera l'accès à des fonctionnalités jusqu'alors cachées. C'est cette notion, plus que la précédente, qui nous intéresse dans la perspective d'assistants intelligents. On notera que l'adaptativité rajoute une dimension *temporelle* à la qualité de contextualité mentionnée plus haut. Ici encore, l'intelligibilité du système doit rester satisfaisante : l'utilisateur risque fort d'être dérouté par un système dont le comportement change avec le temps, s'il n'a pas une vision claire de la nature et de la cause de ces changements. Certains auteurs, comme de Bra [2002], considèrent même que l'adaptativité est satisfaisante si les utilisateurs ne se *rendent pas compte* des changements. Ce point de vue peut se défendre dans le contexte de documents adaptatifs, il paraît plus discutable dans le contexte d'une interface homme-machine avec une application.

1.3.2 L'IA pour réaliser des assistants intelligents

La plupart des travaux visant au développement d'interfaces intelligentes et adaptatives placent au centre de leur démarche la modélisation de l'*utilisateur* [Langley, 1999]. Cela s'explique

[5] Ce terme est malheureusement traduit par « assistant » en français, qui possède alors une acception très différente de celle adoptée ici. C'est pourquoi on gardera l'appellation anglaise.

peut être par une parenté de ce champ de recherche notamment avec celui des environnements informatiques d'apprentissage humain (EIAH) qui s'attachent à modéliser les apprenants afin de les guider [Jean-Daubias, 2002; Héraud, 2002]. Toutefois Langley [1999] remarque que les objectifs en sont différents, puisque les EIAH visent à modifier le comportement de l'utilisateur, tandis que les interfaces intelligentes doivent au contraire s'y adapter, avec un effort cognitif minimum pour l'utilisateur[6].

Ce souci de réduire l'effort nécessaire à l'utilisateur apparaît également lors de la phase de constitution de ce modèle : alors que certains systèmes s'appuient sur des questionnaires remplis préalablement par l'utilisateur, d'autres tentent de construire le modèle en se basant uniquement sur ses interactions avec le système [Langley et Fehling, 1998]. Un autre argument avancé en faveur de cette approche est que certains éléments nécessaires ne peuvent pas être explicités par introspection.

Ces différents problèmes rencontrés par la modélisation de l'utilisateur font pencher vers une approche centrée sur les *tâches* plutôt que sur les utilisateurs. On peut en effet ajouter à l'argument de Langley que les systèmes d'EIAH ne s'intéressent qu'à une famille assez restreinte de tâches à la fois. En revanche, certaines applications (dont les applications de CAO) permettent une grande variété de tâches, selon lesquelles le même utilisateur aura des profils fort différents. Cette position a été développée dans [Champin et Prié, 2002], et on y reviendra au chapitre 7.

[6] Cette opposition est toutefois discutable : certains EIAH utilisent également le modèle de l'apprenant pour s'adapter, afin de faciliter son apprentissage [Héraud, 2002].

Chapitre 2

L'activité de conception

L'objectif de ce travail étant d'assister les concepteurs dans leur tâche, il convient de comprendre ce qui constitue l'*activité de conception* et ses spécificités par rapport aux autres activités de résolution de problème. Cette interrogation se justifie d'autant plus qu'elle intéresse une large communauté interdisciplinaire depuis quelques dizaines d'années. En section 2.1, après avoir donné un aperçu de la question de la place qu'occupe la conception par rapport aux sciences dites «exactes», on s'attachera plus précisément au point de vue de l'ergonomie cognitive avant de présenter un modèle de l'activité de conception. À la lumière de ce modèle, un certain nombre de systèmes d'assistance aux concepteurs sera ensuite présenté en section 2.2. Tous ces systèmes appliquent les principes du raisonnement à partir de cas, présentés dans le chapitre précédent.

2.1 Comment concevoir la conception ?

2.1.1 Un aperçu épistémologique

Depuis la fin des années 60 se poursuit une réflexion lancée par Simon [1969] sur la nature de ce qu'il appelle les «sciences de l'artificiel». Ces sciences se distinguent des sciences fondamentales (comme la physique, la chimie, etc.), qui s'intéressent à expliquer les phénomènes *naturels*. La méthode analytique et hypothético-déductive prônée par ces «sciences de la nature» est alors considérée comme le seul critère de «scientificité». La démarche de conception s'articule quant à elle autour de modes de raisonnement plus synthétiques, tels que l'abduction ou l'induction [Perrin, 2002]. En France, elles sont encore appelées sciences *pour* l'ingénieur, ce qui perpétue l'image de l'ingénieur/concepteur relégué à l'application plutôt qu'à la réelle démarche scientifique, comme le fait remarquer Lemoigne [2002]. Ce dernier raille même en s'interrogeant sur l'existence de sciences *contre* l'ingénieur !

Pourtant, si la démarche du concepteur vise d'abord la création/invention d'un *artefact*, elle est également une démarche de création/découverte de connaissances. L'opposition entre invention et découverte (selon que leur objet préexiste ou non) n'est d'ailleurs pas aussi claire d'un point de vue linguistique, comme le souligne Gire [2001] : *inventer* un trésor, par exemple, c'est le trouver. Il aura fallu les évolutions récentes de l'épistémologie, reconnaissant l'importance, dans les sciences fondamentales, des *modèles* par rapport aux lois[1] [Schmid, 2001], pour envisager la reconnaissance de ces sciences de la conception. La modélisation est en effet au centre de l'activité de conception, ce qui est élégamment illustré par le terme italien *Disegno* utilisé par Léonard de

[1] Les lois ne sont valides que dans un modèle particulier, qui n'a pas vocation à être une représentation vraie et unique de la réalité, mais de répondre à une *finalité* particulière.

Vinci (quel plus prestigieux concepteur ?) et rapporté par Lemoigne [2002]. Ce dernier propose pour le terme *Disegno* la traduction suivante : « le dessin à dessein », ce qui souligne bien son rôle de représentation et d'outil de raisonnement, ainsi que son inscription dans un *projet* (notion chère à Simon).

Reste la complexité inhérente à l'activité de conception, due notamment à la multiplicité et à la transdisciplinarité des modèles qu'elle implique, ainsi bien sûr qu'à son caractère créatif et non déterministe. Gire [2001], à qui a été emprunté le titre de cette section (2.1) en forme d'anneau de Möbius, cite Köstler [1965] et sa théorie de la « bissociation », expliquant tout acte créatif par la mise en correspondance de deux idées jusqu'alors disjointes. Ce « choc bissociatif » serait, d'après Köstler, à l'origine de la création scientifique et artistique[2]. Sans aller jusque là, on ne peut nier le rôle, sinon d'artiste, au moins d'*artisan*, du concepteur, ce qui situe irréductiblement la conception « entre science et art », pour paraphraser Perrin [2001].

2.1.2 Le point de vue de l'ergonomie cognitive

L'ergonomie cognitive caractérise les problèmes de conception comme des problèmes ne préexistant pas à leur résolution. En effet, les problèmes de conception sont par essence des problèmes *mal posés* : les besoins que doit satisfaire le futur artefact, et les contraintes de son utilisation, ne sont pas bien définis, et certains besoins ou contraintes restent implicites. C'est en construisant la solution que ces ambiguïtés sont levées, et donc que le problème se construit.

Par ailleurs, un problème de conception est le plus souvent à l'intersection de plusieurs disciplines (ne serait-ce que par les implications sociales ou économiques que son inscription dans un projet ajoute aux dimensions scientifiques). La nécessité, évoquée ci-dessus, de lever des ambiguïtés dans la formulation du problème, n'est d'ailleurs pas étrangère à cette pluridisciplinarité. Cela explique notamment l'intérêt pour les approches de conception collaborative ou participatives, qui visent à impliquer, même dans les phases précoces de la conception, tous les acteurs concernés [Toussaint et Zimmermann, 2001; Elsner et Blessing, 2002; Carroll, 2002]. On pense bien sûr aux utilisateurs finaux, mais également aux utilisateurs « involontaires », tels les voisins d'une usine, ou les acteurs impliqués durant son cycle de vie : fabrication, maintenance, recyclage.

Enfin, les problèmes de conceptions se caractérisent par une complexité intrinsèque qui, ajoutée aux points soulevés ci-dessus, rendent souvent impraticable leur décomposition analytique classique en sous-problèmes (notamment parce que l'énoncé du problème évolue en cours de résolution). La résolution est donc le plus souvent *opportuniste* [Visser, 2002] : l'activité du concepteur n'est pas seulement guidée par des objectifs fixés au départ, mais aussi par des objectifs suscités par l'apparition, en cours de conception, de nouveaux éléments.

2.1.3 Un modèle de l'activité de conception

D'après Simon, « est concepteur toute personne qui met au point une procédure visant à changer des situations existantes en des situations préférées ». Ces situations « préférées », à l'origine de toute démarche de conception, s'expriment par les spécifications de l'artefact à concevoir, par les *fonctions* que cet artefact va remplir. Après avoir présenté un modèle de l'activité de conception (le modèle FBS), on reviendra sur cette notion de centrale de fonction.

[2] ...mais également comique.

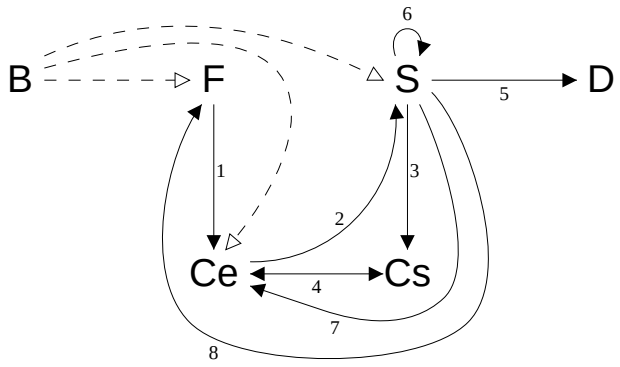


FIG. 2.1 – Le modèle FBS d'après Qian et Gero [1996]

Le modèle FBS

On a adopté pour modéliser l'activité de conception, le modèle FBS (pour *Function-Behaviour-Structure*) proposé par Gero [1990] et raffiné par Qian et Gero [1996] puis Gero et Kannengiesser [2002]. Ce modèle est représenté par la figure 2.1.

Tout artefact dispose d'une *structure* S. Celle-ci exhibe un ensemble Cs de *comportements*, qui peuvent être structurels, c'est-à-dire qu'ils ne dépendent que de la structure (par exemple, peser, pour un presse-papiers) ou exogènes, s'ils dépendent en plus d'évènements extérieurs (s'ouvrir et se fermer pour une porte).

Par ailleurs, cet artefact est conçu pour remplir un certain nombre de *fonctions* notées F. Des besoins B à l'origine du problème de conception, on déduit un certain nombre de fonctions *primaires* qui satisfont ces besoins. Mais le contexte de ces besoins impose un certain nombre de contraintes, pouvant s'exprimer par d'autres fonctions, dites *secondaires*, ou des contraintes sur les comportements ou la structure. Par exemple, la loi « Informatique et liberté » impose à tout système permettant de stocker des données personnelles sur des personnes (fonction primaire) de leur permettre d'accéder et de corriger ces informations (fonction secondaire). La configuration des berges d'une rivière impose des contraintes sur la structure d'un pont qu'on veut y construire. Ces influences sont représentées sur la figure 2.1 par les flèches en pointillés.

Les 8 procédés constituant l'activité de conception dans le modèle FBS sont représentés sur la figure 2.1 par les flèches pleines numérotées :

- La *formalisation* (1) consiste à déterminer un ensemble de comportements escomptés Ce, c'est-à-dire de comportements remplissant les fonctions F, que l'on cherchera ensuite à implémenter par une structure.
- La *synthèse* (2) consiste à proposer une structure S censée exhiber les comportements Ce.
- L'*analyse* (3) dérive les comportements effectifs de la structure, Cs.
- Lors de l'*évaluation* (4), on compare les comportements de la structure aux comportements escomptés, afin de déterminer si la structure répond bien aux besoins exprimés (c'est-à-dire si elle remplit les fonctions de F).
- La *documentation* (5) est la production d'une description D destinée à la réalisation de l'artefact.
- Les trois procédés de *reformulation* (6,7,8) constituent la principale originalité du modèle

FBS et sa caractéristique la plus intéressante, puisqu'ils permettent de rendre compte de l'aspect dynamique et opportuniste de l'activité de conception : au fur et à mesure du développement de la structure, le concepteur découvre de nouveaux éléments qui le poussent à réviser la structure elle-même (6), mais également les comportements escomptés (7) ou même les fonctions (8).

Prenons pour exemple un cas de conception «quotidienne» : un courant d'air menace de faire s'envoler une pile de papier sur mon bureau. Mon besoin s'exprime par la fonction primaire «empêcher la pile de papier de s'envoler», et de deux fonctions secondaires : «garder un accès facile à la pile de papier» et «ne pas avoir trop chaud», ce qui exclut d'emblée les solutions consistant à ranger les papiers dans un tiroir ou fermer la fenêtre. Je décide donc de poser un objet lourd sur la pile (phase de formalisation, définition du comportement escompté : peser sur le papier). Une première idée consiste à y poser ma tasse de café (phase de synthèse, définition de la structure), dont le comportement répond bien au comportement voulu (phases d'analyse et d'évaluation). Cependant je risque ainsi de tacher les papiers (un effet secondaire du comportement effectif de la tasse), ce qui n'est pas souhaitable. Je rajoute donc une fonction secondaire à F :«ne pas abîmer les papiers», et je dois donc réviser ma solution. Une nouvelle solution peut consister à vider préalablement la tasse. On note que la structure dans le modèle FBS comporte des objets mais aussi des *procédés* (ici : vider la tasse, la poser sur la pile de papier).

Sur la notion de fonction

Comme il a été dit plus haut, la notion de fonction est fondamentale dans l'activité de conception puisque c'est pour remplir certaines fonctions qu'un artefact est conçu. L'AFNOR définit, dans la norme X50-150, une fonction comme les «actions d'un produit ou de l'un de ses constituants exprimées exclusivement en termes de finalité». Cette importance de la *finalité* distingue donc la notion de fonction de celle de comportement, qui s'attache à décrire les moyens. Ces deux notions doivent donc bien être distinguées, comme le font remarquer Chandrasekaran et Josephson [1997].

Pour Qian et Gero, les fonctions ne sont représentées que par des étiquettes, comme par exemple «contrôler le flux lumineux» pour décrire la fonction d'un rideau, ou «augmenter la température» pour décrire celle d'un radiateur. Plus précisément, toute fonction peut être vue comme une *action* sur des *flux* (de matière, d'énergie ou d'information). Des taxonomies d'actions et de flux permettent éventuellement de structurer et de comparer différentes fonctions.

2.2 Des assistants pour les concepteurs

La complexité de leur tâche conduit naturellement les concepteurs à puiser dans leur expérience des situations spécifiques pour résoudre de nouveaux problèmes, et ce quel que soit leur domaine. Dans celui de l'architecture, Alexander [1964] propose même une véritable méthodologie destinée à faciliter la compilation et la réutilisation de solutions de conception éprouvées. Cette méthodologie a depuis pris un essor considérable, notamment dans le domaine de la conception logicielle, autour de la notion de *patrons de conception* [Gamma *et al.*, 1995].

Au delà d'une pratique individuelle, la capitalisation et la réutilisation des connaissances apparaît en effet de plus en plus comme une pratique collective déterminante dans la compétitivité d'une entreprise [Dieng-Kuntz *et al.*, 2001]. La réutilisabilité devient une clause implicite des spécifications fonctionnelles, de sorte que l'expérience des concepteurs puisse être partagée.

Dans ce contexte, le raisonnement à partir de cas apparaît adapté [Maher et Gómez, 1996]. Non seulement il s'applique bien à des domaines complexes et mal définis, mais il modélise également de façon satisfaisante la manière de procéder des concepteurs. Cette section présente un certain nombre de système d'assistance aux concepteurs fondés sur le paradigme du RàPC.

PRECEDENTS [Oxman, 1994] permet à des architectes de rechercher des cas dans une base en fonction des problèmes que ces cas ont servi à résoudre. Chaque cas est en effet indexé par un certain nombre d'*histoires*, chaque histoire se composant d'un problème, d'un concept répondant à ce problème, et d'une forme sous laquelle ce concept est instancié dans le cas en question. Un même cas peut donner lieu à de nombreuses histoires. Ce système ne fournit aucun mécanisme d'adaptation. En revanche, il permet aux architectes de *naviguer* entre les cas et les histoires, en explorant les différents cas liés à un problème ou les différentes histoires attachées à un même cas, ou encore les analogies entre problèmes ou concepts. Ces derniers sont en effet organisés dans un réseau sémantique.

Le modèle utilisé dans PRECEDENTS, nommé ICF (pour *Issue-Concept-Form*), résonne fortement avec le modèle FBS présenté plus haut, bien que les notions de problème et d'histoire soient plus contextualisées (et peut-être plus faciles à appréhender pour l'utilisateur) que la notion de fonction. Ce type d'indexation évoque également celui des patrons de conception, tel qu'ils ont été proposés par Alexander [1964]. Si le système PRECEDENTS n'est pas à proprement parler un système de RàPC (il se limite à la remémoration), le modèle ICF a été appliqué au système FABEL [Oxman et Voss, 1996] qui est quant à lui muni de mécanismes d'adaptations.

Le système RODEO se focalise sur la récupération d'artefacts réutilisables dans le domaine de la CAO électronique. Altmeyer *et al.* [1994] proposent une formalisation des caractéristiques qui permettent de décrire à la fois les artefacts et les spécifications sous forme de prédicats : par exemple, *FonctionAdditioneur*, *Arité2*, *Largeur16bits*. Les connaissances du domaine sont représentées par des relations hiérarchiques de généralisation entre prédicats (par exemple *Arité2* $<$ *AritéPaire*), et par des règles de réécriture correspondant à la composition d'artefacts plus simples (par exemple, si j'ai besoin d'un multiplexeur n voies, je peux rechercher deux multiplexeurs $n/2$ voies et un multiplexeur 2 voies). Cette approche suppose donc des connaissances assez importantes sur le domaine, et les cas comportent des descripteurs de haut niveau, y compris fonctionnels.

Le système DÉJÀ VU [Smyth et Keane, 1995b] a pour domaine d'application la conception de logiciels de pilotage de robots pour une usine. L'adaptation est le propos principal de ce système. Partant du principe que les processus d'adaptation sont des processus heuristiques susceptibles de créer des incohérences, Smyth et Keane insistent sur la nécessité de remémorer le cas qui nécessitera le moins d'adaptation possible. Ils introduisent pour cela la *remémoration guidée par l'adaptation* : les connaissances d'adaptations sont représentées sous une forme déclarative qui permet leur exploitation pendant la remémoration.

Une autre particularité notable de DÉJÀ VU tient au fait qu'il utilise des cas décomposés *hiérarchiquement*. En effet, les problèmes considérés sont trop complexes, et par conséquent trop spécifiques : deux d'entre eux ne se ressembleront jamais assez pour permettre une réutilisation (toujours dans l'optique cherchant à minimiser les adaptations nécessaires). Il existe donc deux types de cas : les cas de *décomposition*, qui décomposent un problème en sous-problèmes, et les cas de *conception*, qui implantent une solution pour un problème atomique. On maximise ainsi

les chances qu'à chaque partie du problème en cours corresponde un cas suffisamment proche pour être réutilisé.

Le système FAMING [Faltings et Sun, 1996] est un système d'assistance pour la conception mécanique de paires cinématiques : des systèmes mécaniques bidimensionnels impliquant deux composants. De façon assez atypique, cette application du RàPC laisse complètement de coté la remémoration pour se focaliser sur l'adaptation. Son objectif est en effet de décharger l'utilisateur des parties les moins créatives de la réutilisation. L'utilisateur choisit donc le ou les cas à adapter ou combiner ainsi que certains paramètres de l'adaptation. FAMING est capable de calculer le comportement qualitatif d'une paire cinématique en fonction de sa structure, et donc de guider l'adaptation vers la fonction recherchée. Cette dernière, en revanche, doit être saisie manuellement par l'utilisateur, en terme de contraintes sur le comportement ; de même, chaque cas de la base est indexé par sa ou ses fonctions, qui doivent donc être saisies préalablement par un opérateur humain.

Hua *et al.* [1996] s'intéressent, dans le domaine de la conception architecturale, au raisonnement géométrique. Ils distinguent deux façons d'envisager le RàPC : avec des cas décrits en profondeurs, ou bien superficiellement. Les cas décrits en profondeur (*deep cases*) comportent non seulement le résultat d'une conception, mais également la façon d'obtenir ce résultat, les choix qui y ont conduit, etc. Les cas décrits superficiellement (*shallow cases*) ne comportent généralement que le résultat de la conception. Si les premiers sont plus faciles à exploiter, car plus riches, ce sont souvent les seconds qui sont effectivement disponibles. Le système CADRE manipule des cas ne comportant que des données géométriques, assortis de contraintes inférées par le système grâce aux connaissances générales du domaine.

La réutilisation d'un cas se fait par analyse de dimensionnalité : des degrés de liberté sont ajoutés ou supprimés selon que le problème est sur- ou sous-contraint. Une distinction intéressante est faite entre cas et *prototype* : un prototype est nécessairement abstrait pour être réutilisable. Un cas, en revanche, n'est abstrait qu'au moment de sa réutilisation, et uniquement dans la mesure du nécessaire, en fonction du problème à résoudre. Cette distinction est très importante dans CADRE où le nombre de paramètres potentiels est trop grand pour les considérer tous. L'analyse de dimensionnalité permet de ne prendre en compte que ceux qui violent les contraintes du nouveau problème.

RESYN/CBR [Lieber et Napoli, 1996] propose des plans de synthèse de molécule en chimie organique, en réutilisant des plans connus. La particularité de ce système est d'utiliser une méthode de classification pour la phase de remémoration : chaque plan est indexé par une abstraction de la molécule qu'il sert à synthétiser. Lorsque la classification pure échoue, une méthode de classification *élastique* est utilisée. La chaîne de généralisation, spécialisations et transformation (pour la classification hiérarchique seulement) qui lie le cas source au cas cible est appelée *chemin de similarité*. L'intérêt de ce chemin est qu'il permet de guider l'adaptation : à chaque type d'étape de ce chemin sont en effet associées des connaissances d'adaptations qui permettent de faire le chemin inverse avec la solution source. Ce principe est également discuté en détail et généralisé dans Lieber et Napoli [1998].

Herbeaux et Mille [2001] décrivent le système ACCELERE d'aide à la conception de caoutchoucs élastomères. Dans ce domaine, les connaissances liant la structure (formule et mode de fabrication) et les comportements (propriétés physiques et mécaniques des caoutchoucs) sont très faibles.

Aussi le processus de conception est-il itératif, où chaque *essai* consiste à réaliser effectivement un produit pour le tester. Un enjeu majeur consiste donc à réduire le nombre d'essais en réutilisant l'expérience, d'une part pour proposer rapidement une structure ayant des chances de répondre aux besoins exprimés, d'autre part pour anticiper le comportement d'une structure avant de la réaliser. Il s'est avéré qu'un troisième type d'utilisation du système est apparu spontanément chez les concepteurs : proposer de compléter une structure partiellement spécifiée, les différents produits impliqués dans un produit n'étant en général pas indépendants.

On peut distinguer deux grandes tendances dans les systèmes présentés ici. D'une part, les systèmes *spécialisés* assistent les concepteurs dans une tâche ciblée. C'est le cas par exemple d'ACCELERE ou de RESYN/CBR qui offrent une modélisation précise des cas et effectuent la phase de réutilisation du cycle du RàPC. D'autre part, les systèmes *généralistes* se limitent le plus souvent à la remémoration : c'est le cas de RODEO, PRECEDENTS et de plusieurs autres systèmes présentés par Maher et Gómez [1996]. Ils offrent en général une vue beaucoup plus abstraite des cas (PRECEDENTS) ou requièrent un effort significatif de formalisation de la part des concepteurs (RODEO). L'objectif de ce travail le rapproche plus des systèmes généralistes, tout en mettant l'accent sur la réutilisation, et ce dès la phase de remémoration (similarité guidée par l'adaptation).

Chapitre 3

Connaissances et documents

Les connaissances mobilisées par les concepteurs sont complexes, souvent implicites, et difficiles à modéliser pour une exploitation informatique. Elles sont cependant consignées dans des *documents de conception* qui constituent pour les concepteurs un moyen efficace de stockage et de communication de leurs connaissances. Qui plus est, ces documents sont numériques : leur création et leur consultation est nécessairement médiée par l'outil informatique. Il est donc pertinent de s'intéresser à la notion de document (section 3.1), aux langages documentaires (section 3.2) et plus particulièrement aux travaux autour du Web Sémantique, qui depuis quelques années cherchent aussi à faciliter l'exploitation des connaissances « enfouies » dans les documents numériques (section 3.3).

3.1 Sur la notion de document

Si la notion de document ne paraît pas poser de problème *a priori*, elle soulève quelques ambiguïtés qu'il convient de lever en la définissant précisément. On s'intéressera aussi aux particularités des documents numériques.

3.1.1 Document, document numérique

D'après Bachimont [1999], un document est un contenu exprimé sur un support matériel. Cette notion de support (écran, feuille de papier, haut-parleur, etc.) est le premier point important souvent éludé, particulièrement lorsqu'on considère des documents numériques, qui semblent par nature immatériels. Bachimont insiste donc sur le fait qu'il faut toujours ce support d'inscription pour exprimer le contenu. Contenu qui est lui une forme interprétable (image, lettre, parole, etc.), « sémiotique dans la mesure où elle fait signe pour un lecteur[1] d'un sens qui lui est adressé ». On a coutume de discerner dans les documents la structure logique de la structure physique. La première contraint la forme interprétable du document (choix d'énoncer tel argument avant tel autre) tandis que la seconde contraint sa manifestation physique (choix d'utiliser une police de quatorze points).

La particularité des documents *numériques* tient au fait que, contrairement par exemple aux livres, ils ne sont pas stockés sur un support qui permette leur consultation. Cette dernière requiert la médiation d'un procédé, en l'occurrence calculatoire, pour passer du *support d'enregistrement* (disque magnétique, CD-ROM) au *support d'appropriation*. On remarquera toutefois

[1] On gardera l'appellation « lecteur » pour tout type de document, faute de terme plus générique satisfaisant.

que cette dichotomie n'est pas exclusivement réservée aux documents numériques : les documents audio ou audiovisuels nécessitent eux aussi la médiation d'un procédé, *analogique* celui-là (lecture magnétique, projection). Par contraste, cette analogie n'existe pas pour les documents numériques : la *forme* d'appropriation est tout à fait différente de la *forme* d'enregistrement. On peut alors définir quatre types d'opérations permettant de passer d'une forme à l'autre, qui sont détaillées ci-après.

Projection C'est l'opération qui permet de passer du numérique à l'intelligible. Cela consiste, par exemple, dans l'affichage à l'écran, l'impression, ou l'émission d'un son. De même qu'un document ne peut exister qu'avec une forme physique pour médier son contenu, un document numérique n'existe que lorsqu'il a une forme d'appropriation (et donc, un support d'appropriation). Un document numérique n'est donc pas seulement un contenu, mais contrairement à ce que son appellation peut laisser croire, il n'est pas non plus seulement un fichier informatique (forme d'enregistrement) ni un CD-ROM (support d'enregistrement). Ces derniers sont tout au plus des *ressources* qui constituent un *ou plusieurs* document(s) en puissance ou virtuel(s) [Garlatti et Crampes, 2002], réalisé(s) par une opération de projection.

Cette notion de ressource est particulièrement présente dans le domaine du *World Wide Web*, et dans les recherches récentes autour du thème du Web Sémantique (cf. sections 3.2.2 et 3.2.4). On va en effet voir par la suite que l'accès à un document sur le Web passe par un processus de projection complexe transparent pour l'utilisateur : sélection d'un serveur, sélection d'un fichier, parfois même génération du contenu à la volée.

Abstraction Opération duale de la précédente, elle permet de passer de l'intelligible au numérique. Rentre dans cette catégorie le fait de scanner une image ou d'enregistrer une musique sur un clavier MIDI. Cependant on peut aussi y voir la prise en compte continue des actions de l'utilisateur sur la forme intelligible pour mettre à jour le modèle numérique en mémoire (par exemple, déplacer un élément avec la souris).

Transformation Cette opération va du numérique vers le numérique. Elle permet donc de passer d'une forme d'enregistrement à une autre. Cela concerne bien sûr la conversion de fichiers d'un format dans un autre, mais également de nombreuses autres situations, à commencer par la simple duplication. Tout dépend en fait du niveau d'abstraction auquel on considère les deux précédents types d'opération. L'application d'une feuille de style peut être considérée comme une opération de projection (puisque la feuille de style sert à construire une forme d'appropriation), mais c'est d'abord une transformation d'une forme numérique vers une *autre* forme numérique, plus propice à la projection. Il en va de même avec l'abstraction, la première forme d'un document numérisé, très dépendante du matériel de numérisation, étant rarement conservée pour l'enregistrement.

Navigation Le dernier type d'opération, la navigation, permet de passer d'une forme intelligible à une autre forme intelligible (pas nécessairement du même document). On pourrait au premier abord considérer que ce quatrième type d'opération est une composition des trois précédents, ce qui est vrai au niveau de détail le plus fin : naviguer au sein d'un document numérique, et *a fortiori* d'un document à un autre, requiert au moins une opération de projection pour restituer la nouvelle forme intelligible, éventuellement après un traitement (transformation) ou même une entrée de l'utilisateur (abstraction).

Cependant, à un niveau plus abstrait, la navigation apparaît bien comme une opération de plein droit sur les documents numériques. En effet, même si les notions de renvoi et de référence existent dans les documents papiers, les liens correspondants ne sont pas instrumentés. À l'inverse, les documents numériques permettent l'instrumentation de ces liens (on parle alors d'hyperdocuments) de sorte qu'il est aussi aisé de suivre une référence que de tourner la page dans un livre.

3.1.2 Indexation et annotation

Un *index* est un dispositif permettant d'accéder à l'objet indexé [Prié, 1999]. Pour être utile, l'index doit *documenter* son objet en vue d'une tâche particulière, ou dans le cas de l'indexation de document, le *paraphraser*. En ce sens, pour Bachimont [1999] l'index peut être lui même considéré tantôt comme un document, tantôt comme un moyen d'accès à son objet[2], et tout document peut éventuellement servir d'index. Par exemple, le sommaire d'un magazine de cinéma indexe la critique du film *Le nom de la rose*, qui elle même indexe le film, qui indexe à son tour *La poétique, livre II* d'Aristote ; par ailleurs le film indexe aussi le roman dont il est tiré [Eco, 1980]. Une spécificité des documents numériques est qu'ils sont intrinsèquement indexés, et que toute manipulation de ces documents, à commencer par leur consultation, passe nécessairement par un index (ne serait-ce qu'un nom de fichier, une URL ou une clé dans une base de données).

De la même façon, l'*annotation* peut être vue comme une indexation en sens inverse : c'est la mise en place d'un contenu documentaire dont l'accès *est permis par* la consultation du document annoté. Étant un document, l'annotation peut bien sûr faire office d'index pour des documents tiers. Là encore, le numérique offre des possibilités inédites en terme d'annotation. D'abord, parce qu'aucune contrainte physique ne pèse sur leur mise en place comme c'est le cas pour les documents traditionnels. Ensuite, parce que l'annotation n'a pas besoin d'altérer le document original. Il en découle que ce dernier peut toujours être visualisé sans ses annotations, comme à l'inverse ces dernières peuvent lui être intégrées de façon transparente ou non au moment de la projection.

On voit que l'*unité* d'un document numérique est difficile à définir : la différence s'estompe entre fragment documentaire, document et corpus, et ni les supports physiques (support et forme d'enregistrement), ni les modalités de consultation ne permettent de cerner précisément les frontières d'*un* document, tant les processus correspondant aux quatre types d'opération ci-dessus sont complexes. D'un certain point de vue, c'est le lecteur, devenu coauteur, qui construit un document unique à travers son propre parcours de lecture et ses éventuelles annotations, impliquant un réseau de ressources diverses. En ce sens, toute trace de l'activité (de consultation mais également d'édition) peut à son tour prendre un caractère documentaire.

3.2 Les langages documentaires

3.2.1 SGML

Le *Standard Generalized Markup Language* ou SGML [ISO 8879] est une norme ISO pour la représentation informatique des documents. Un document SGML se présente comme un texte marqué par des *balises* ; ces balises définissent une structure arborescente sur le texte (cf. figure 3.1). Mais SGML est en fait un métalangage, permettant de décrire les balises et la structure

[2] Cela est vrai sur le principe, même si certains index présentent peu d'intérêt en tant que document, tant la documentation qu'ils constituent est faible : une entrée dans une table des matières, un nom de fichier, etc.

```
<body>
    <h1>
        Un exemple de texte balisé
        par <a name="sgml">SGML</a>
    </h1>
    Les balises utilisées sont en fait
    celles de HTML,<br>
    une <em>DTD</em> SGML bien
    connue.
</body>
```

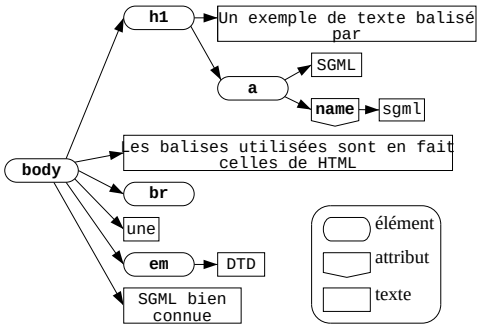


FIG. 3.1 – Un exemple de texte balisé par SGML (DTD HTML)

de tous les documents d'un même type à l'aide d'une DTD (*Document Type Definition*). Chaque DTD définit donc le vocabulaire et la syntaxe d'un langage de description de document.

La volonté derrière la norme SGML est d'encourager la représentation des documents selon leur structure logique plutôt que selon leur structure physique. Cette dernière peut être reconstruite à partir d'une *feuille de style* valable pour tous les documents d'une DTD. L'avantage de cette notion de feuille de style est double : elle assure l'homogénéité de la structure physique des documents conformes à une même DTD, et elle permet d'envisager pour une DTD plusieurs styles de présentation selon le support d'appropriation visé (moniteur d'ordinateur, impression, écran de téléphone portable, etc.).

On pourrait discuter le principe sur lequel repose ce modèle, selon lequel la forme découle du fond, mais pas l'inverse. Il n'en demeure pas moins qu'on peut voir, dans cette préoccupation de favoriser le contenu sur la présentation, un début d'intérêt pour une représentation du contenu structurée, et donc exploitable par la machine.

3.2.2 Le *World Wide Web*

Le *World Wide Web* (WWW, ou simplement Web) est un espace hyperdocumentaire distribué à l'échelle de l'internet. Il est devenu en une dizaine d'années tellement incontournable qu'il est difficile de parler de document numérique sans se référer aux langages et aux technologies du Web, principalement proposées par le W3C (*World Wide Web Consortium*[3]), par le biais de normes nommées *recommandations*. Les trois composants initiaux du Web sont présentés ici.

HTML

Hyper-Text Markup Language (HTML), le langage du *World Wide Web*, est une DTD SGML, sans doute la plus connue. Il a été proposé pour permettre une publication facile d'information sur un réseau informatique. Une des clés de son succès est la robustesse qu'il autorise pour les logiciels l'interprétant : une erreur de structure, une balise ouverte et non fermée, une balise inconnue, n'empêchent généralement pas l'affichage d'une page. Cette robustesse tient bien sûr à l'implantation des navigateurs, mais également à la redondance inhérente de SGML, et sans doute en partie aux choix de conceptions dans la DTD de HTML.

Si la première version de HTML était conforme à l'esprit SGML, proposant essentiellement des balises portant sur le fond, son succès a eu raison de cette bonne volonté. Les versions

[3] http://www.w3.org/

successives, sous la pression des *desiderata* des utilisateurs et des extensions introduites par les éditeurs de logiciels, se sont vues enrichies de nombreuses balises relevant de la présentation. Les efforts récents pour découpler à nouveau les deux aspects, en promouvant l'utilisation de feuilles de style et en retirant de la norme certaines balises, ne sont pas largement suivis.

URL

HTML est destiné à la représentation d'hypertextes, et propose donc un mécanisme de liens vers d'autres documents. Ces liens sont réalisés à l'aide d'URLs (*Uniform Resource Locator*). Une URL est un index vers une ressource (et non un document), qu'on peut voir comme un programme permettant l'accès à cette ressource en vue de sa projection en un document. Par exemple, l'URL http://www.ietf.org/rfc/rfc2396.txt peut s'interpréter comme suit : contacter le serveur www.ietf.org *via* le protocole HTTP, et demander la ressource identifiée par /rfc/rfc2396.txt. Comme son nom l'indique, ce type d'index propose une syntaxe suffisamment générale pour permettre d'unifier les protocoles existants et d'intégrer ceux à venir.

Cette uniformité des URLs participe au succès du Web, dans la mesure où l'espace couvert par ces index n'est pas soumis à une autorité centralisée. Pour la plupart, les personnes ayant un accès au Web « possèdent » une partie de cet espace (généralement sous la forme d'un répertoire personnel sur un serveur accessible par le Web), et peut donc facilement *mettre en ligne* n'importe quel contenu. Réciproquement, il n'est pas nécessaire d'avoir des droits particuliers sur une ressource pour pointer vers elle[4]. N'importe qui publiant un document sur le Web peut donc faire pointer ce document vers n'importe quelle ressource. Cette souplesse a bien sûr des contreparties : l'auteur d'un lien ne peut pas garantir que la destination de ce lien existe aussi longtemps que le lien lui même. En pratique, nombre de liens sur le Web sont « pendants », et résultent dans la classique « erreur 404 » — le code signifiant qu'une ressource n'est pas accessible dans le protocole HTTP (cf. ci-après).

Notons que les URLs sont en fait un cas particulier d'identifiants plus généraux, les URIs (*Uniform Resource Identifier*, Berners-Lee *et al.* 1998). La particularité des URLs par rapport aux URIs est originellement de fournir une méthode opérationnelle pour accéder à la ressource identifiée. D'autres types d'URIs n'ont pas ce type de prérequis, mais visent à fournir des identifiants plus durables (puisque indépendants d'une solution technique particulière). C'est le cas des URNs (*Uniform Resource Name*, Sollins et Masinter 1994). Cependant, les protocoles actuels autorisent un niveau d'abstraction suffisant pour permettre aux URLs de garder une certaine stabilité dans le temps, et donc de répondre raisonnablement aux deux contraintes. Les URLs sont donc l'immense majorité des URIs utilisés.

HTTP

Bien que les URLs permettent d'utiliser n'importe lequel des protocoles existants, le Web dispose d'un protocole de prédilection : *Hyper-Text Transfer Protocol* ou HTTP [Fielding *et al.*, 1999], qui contrairement à ce que son nom laisse supposer, ne se limite pas au transfert d'hypertextes, mais permet bien le transfert de n'importe quel type de données. La particularité de HTTP est de faire explicitement la distinction entre une ressource (l'objet d'une requête par le biais d'une URL) et une *entité* (les données envoyées en réponse à la requête). L'entité n'est pas encore un document puisqu'elle n'est pas une forme intelligible. Cependant, elle se distingue de la ressource à un certain nombre d'égards qui justifient de considérer HTTP comme une première phase de projection de la ressource.

[4] Pas même celui d'y accéder, d'ailleurs, pourvu qu'on connaisse son URL.

Tout d'abord, la ressource n'est pas nécessairement un fichier (bien que ce soit l'usage le plus commun, et que la forme des URLs puisse le laisser croire). Certaines ressources HTTP apparaissent plus comme des *services*[5], c'est-à-dire des programmes produisant une entité à la demande[6]. Une application classique de ressource dynamique est la génération d'une entité à partir d'une requête dans une base de données. Par ailleurs, HTTP permet une *négociation* entre client et serveur, lorsque plusieurs *variantes* existent pour une ressource donnée. Cette négociation peut par exemple porter sur le format des données de l'entité ou sur la langue. HTTP fournit également des mécanismes permettant d'identifier une variante de façon non ambiguë. Enfin, ce protocole prévoit un mécanisme d'extension, notamment pour permettre la prise en compte d'autres critères dans la négociation de contenu.

3.2.3 XML

Le langage XML (*eXtensible Markup Language*, Bray *et al.* 1998) est le successeur de HTML pour le Web, cherchant à conserver ses avantages tout en palliant ses défauts. XML est un métalangage, à l'instar de SGML, permettant lui aussi de définir des DTDs. De HTML, il garde la simplicité et la robustesse : il est beaucoup plus simple que SGML et permet, contrairement à lui, de vérifier qu'un document est *bien formé* sans connaître sa DTD. La figure 3.2 donne un exemple de document XML court.

L'extensibilité d'XML, c'est-à-dire la possibilité qu'il offre de définir ses propres *termes* (nom de balises, nom d'attributs) pour répondre à un besoin spécifique, a souvent été présentée comme un apport *sémantique* par rapport à HTML, ce qui est à la fois vrai et faux. C'est faux, car si les noms des balises `<subject>` ou `<email>` portent du sens pour un humain anglophone, ils n'ont aucune sémantique pour la machine qui exploite ce fichier, alors que la balise `<h1>` de HTML possède une sémantique connue et implantée dans tous les navigateurs (c'est un titre de premier niveau, qui doit par conséquent être mis en évidence par la typographie). On ne peut cependant pas mettre les deux au même niveau puisque XML est un métalangage, contrairement à HTML. Ce qui est vrai, c'est donc que XML permet de créer des langages ayant une sémantique spécifique à un besoin donné, et donc plus appropriée que la sémantique prédéfinie de HTML. Par exemple, pour décrire un courrier électronique, HTML dispose d'une balise `<address>`, mais cette dernière ne permettrait pas de distinguer l'expéditeur du destinataire, comme le font les balises `<from>` et `<to>` de l'exemple. Toujours est-il que cette sémantique spécifique n'est pas contenue dans les documents (tout au plus, elle est suggérée par le nom de la balise si l'auteur de la DTD a jugé bon de le rendre explicite), et elle ne l'est pas non plus dans l'expression de la DTD (même si là encore, les contraintes structurelles et le choix des termes peuvent aider un lecteur humain à reconstruire cette sémantique). De même que la sémantique de HTML est *explicitée* dans les spécifications du langage, la sémantique d'une DTD XML doit être explicitée dans un document, ou au moins implantée dans une feuille de style ou dans une application afin d'être *opérationnelle*. XML prend donc tout son intérêt lorsque des DTDs consensuelles existent. Parmi celles-ci, on peut citer MathML, proposées par Ion et Robert [1998] pour représenter les formules mathématiques, SVG de Ferraiolo [2001] pour les dessins vectoriels, XHTML [Pemberton, 2000] pour les pages Web (cette dernière recommandation est en fait une version compatible XML de

[5] Toutes sont en fait des services, le plus courant consistant à retourner systématiquement un fichier donné, d'où son assimilation au fichier lui même.

[6] À l'aide de protocoles tels que :
le protocole CGI : http://hoohoo.ncsa.uiuc.edu/cgi/
le protocole *servlet* : http://java.sun.com/products/servlet/
le protocole spécifique au serveur Apache : http://modules.apache.org/doc/API.html

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE email SYSTEM "e-mail.dtd">
<email>
    <from href="mailto:champin@bat710.univ-lyon1.fr">
        <name>Pierre-Antoine Champin</name>
    </from>
    <to href="mailto:amille@bat710.univ-lyon1.fr"/>
    <subject>Dernière version</subject>
    <body>
        Alain,<html:br/>
        tu trouveras
        <html:a href="http://www710/~champin/publis/these">
            ici
        </html:a>
        la dernière version de ma thèse.
        <html:p>
            Bonne lecture !
        </html:p>
    </body>
    <signature>
        Pierre-Antoine
    </signature>
</email>
```

FIG. 3.2 – Un fichier XML décrivant un courrier électronique

HTML).

XML-Names

La recommandation XML-Names [Bray *et al.*, 1999] permet de mixer des termes provenant de divers vocabulaires tout en prévenant les conflits de noms. Elle permet donc la réutilisation et prévient la redondance entre DTDs. Par exemple dans la figure 3.2, la DTD servant à décrire des courriers électroniques spécifie l'utilisation du vocabulaire XHTML (différencié ici par le préfixe `html:`) à l'intérieur de l'élément `<body>`. Ce dernier n'ayant pas de préfixe (et donc dans cet exemple, supposé provenir de la DTD e-mail.dtd) ne risque pas d'être confondu avec son homologue `<html:body>`. Toute DTD ou tout schéma (cf. ci-après) conforme à XML-Names doit donc spécifier l'URI identifiant son *espace de noms*. Ces URIs sont ensuite associés à des préfixes (ou au préfixe vide) par des déclarations apparaissant dans n'importe quelle balise XML, avec pour portée le contenu de cette balise. Notons que dans l'exemple, ces déclarations ont été omises pour faciliter la lecture. Le début de l'exemple aurait été :

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE email SYSTEM "e-mail.dtd">
<email xmlns="http://example.com/email"
       xmlns:html="http://www.w3.org/1999/xhtml">
    <from...
```

XSLT

Enfin, XML est également muni d'un mécanisme de feuille de style : *eXtensible Stylesheet Language* (XSL), qui contient notamment la recommandation *XSL Transformations* ou XSLT [Clark, 1999]. Elle propose un langage pour représenter en XML des feuilles de style pour les documents XML. Comme l'indique le T dans l'acronyme, les feuilles de styles XSLT décrivent une *transformation* d'un document XML vers une autre structure. Bien que la notion de feuille de style suppose au départ que cette autre structure soit une forme d'appropriation, cette limite n'est absolument pas imposée par XSLT. Ce dernier est de fait utilisé à de nombreuses fins autres que la mise en forme de documents, impliquant toute sortes de transformations, et joue sans doute un rôle important dans l'utilisation massive de XML pour la représentation de *données* semi-structurées n'ayant pas de vocation documentaire.

3.2.4 Vers le Web Sémantique

Le Web Sémantique [Berners-Lee et Fischetti, 1999] est la vision de ce que le Web actuel pourra devenir. Cette vision part du constat que la grande majorité des informations disponibles sur le Web sont formatées pour être utilisées par des humains. Or un certain nombre d'arguments plaident en faveur d'un traitement automatique ou semi-automatique de ces informations, au centre desquels réside la *quantité* colossale et toujours croissante d'informations disponibles. Sont présentés ci-après quelques scénarios servant de ligne directrice aux travaux dans ce domaine [Euzenat, 2001].

En recherche d'information, un effet symptomatique de la surabondance de données a été de voir le moteur de recherche AltaVista®[7], dont le succès était fondé sur l'exhaustivité, supplanté par Google™[8], qui tire son succès de la pertinence de ses résultats. Faute de pouvoir encore

[7] http://www.altavista.com
[8] http://www.google.com

faire le tri lui même, l'utilisateur attend de la machine qu'elle le décharge partiellement de cette tâche. Cela suppose que les moteurs de recherche aient une meilleure « compréhension » du contenu des pages.

Dans le domaine du commerce électronique, cette recherche d'information est plus précise (recherche du meilleur produit au meilleur prix), mais pose des problèmes d'interopérabilité : les catalogues des fournisseurs adoptent des formats différents, contenant des informations différentes (quantitativement et qualitativement). L'enjeu de cette automatisation est une accélération de certaines transactions dans le commerce B2B (*business to business*), ou dans certaines visions domotiques (réfrigérateur commandant en ligne les produits épuisés).

Certains domaines scientifiques, notamment la biologie génétique, impliquent de grosses quantités de données et de lourds traitements, qui ne peuvent être le fait d'un seul organisme et sont donc nécessairement distribués. La consultation de ces données et l'exécution de ces traitements devraient pouvoir se faire de façon transparente pour les chercheurs concernés, sans qu'ils aient à se soucier du caractère distribué de ces ressources. Là encore, des problèmes d'interopérabilité entre agents, mais également de gestion des droits d'accès à travers le réseau, se posent.

Enfin, la gestion des connaissances au sein d'une entreprise, relève également de l'instrumentation d'un ensemble de documents afin de faciliter l'exploitation des connaissances qu'ils renferment. Pour être efficace, cette instrumentation doit être personnalisée (dépendante de l'utilisateur) et contextualisée (dépendante de sa situation, de sa tâche) afin de lui éviter une surcharge d'information qui serait plus néfaste que bénéfique. L'utilisation d'informatique embarquée est également envisagée, notamment celle des assistants électroniques (*Personal Digital Assistants* ou PDA) connectés au Web et capables de communiquer directement les uns avec les autres. L'enjeu de taille de ce type de scénario, et qui résonne particulièrement avec le travail présenté ici, est que la création de connaissance doit être une *conséquence automatisée et transparente* de toute action de l'utilisateur.

La réalisation du Web Sémantique suppose donc que les ressources soient appropriables par des agents logiciels, et non plus seulement par des lecteurs humains. Les techniques de représentation de connaissances et de raisonnement, développées dans le domaine de l'IA, semblent alors pertinentes, et on peut voir les deux recommandations présentées ci-après comme un premier pas pour intégrer ces techniques dans le contexte du Web.

XML-Schemas

Alors que XML sort du cadre purement documentaire hérité de SGML pour représenter des données semi-structurées et des connaissances, le pouvoir d'expression des DTDs devient insuffisant. La recommandation XML-Schemas [Fallside, 2001] vise à intégrer des mécanismes hérités des bases de données et de la représentation de connaissances. Les types de documents ne sont plus décrits par des DTDs mais par des *schémas*, eux même décrits en XML. Les schémas autorisent la séparation d'un terme (nom de balise ou d'attribut) de son *type*, c'est-à-dire des contraintes sur la forme de son contenu, qu'il soit simple (uniquement du texte, comme `<subject>` dans l'exemple de la figure 3.2) ou complexe (avec des attributs et du texte balisé, comme `<from>` dans le même exemple). XML-Schemas inclut des notions d'héritage (un type peut être défini par rapport à un autre), de polymorphisme (un nom de balise peut être employé pour un autre si leurs types sont compatibles), de type abstrait.

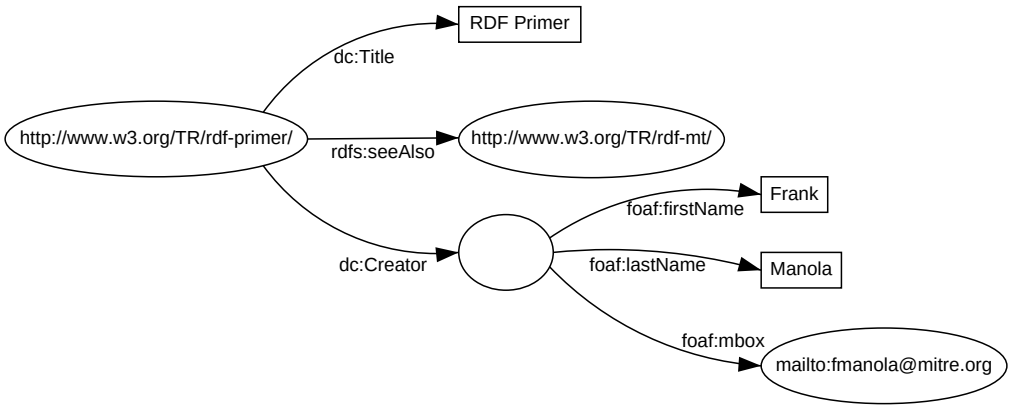


FIG. 3.3 – Un exemple de graphe RDF

RDF

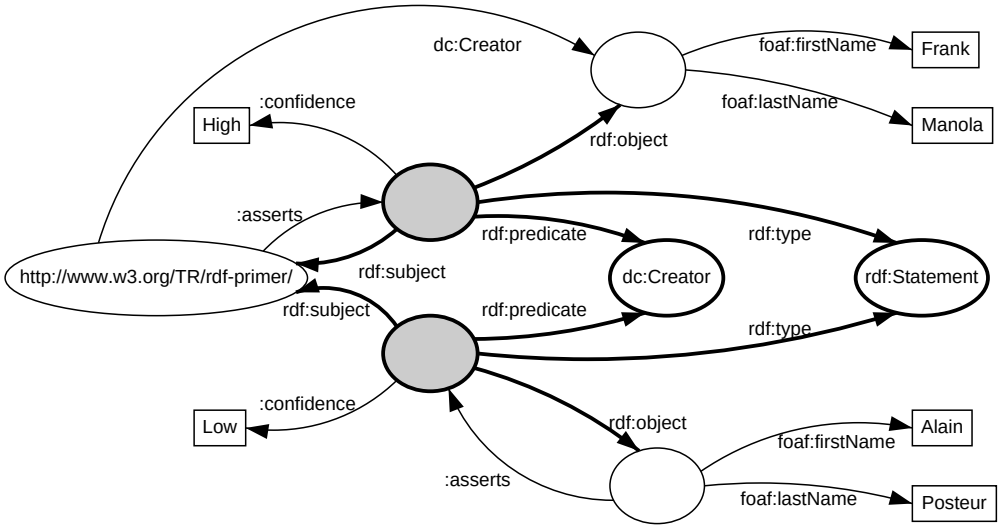
Alors que XML-Schemas ne s'attache qu'à définir une classe de documents à venir, *Resource Description Framework* ou RDF [Manola et Miller, 2002] permet quant à lui la *description* de ressources déjà existantes. Ceci peut s'avérer utile dans différentes situations : lorsque le langage de description de la ressource ne permet pas une représentation suffisamment riche, ou lorsque la ressource ne peut pas être modifiée pour y intégrer plus d'information. RDF est donc souvent présenté comme un langage de représentation de *méta-données*, de nombreuses ressources du Web étant effectivement réductibles à un ensemble de données. Il convient toutefois de garder à l'esprit que dans le cas général, une ressource est plus à considérer comme un *service*, produisant éventuellement[9] des données.

Le principe de RDF peut se résumer par la notion de *propriété* : chaque ressource possède un certain nombre de propriétés ayant une certaine valeur. Les propriétés d'une page Web sont par exemple son auteur, son titre, sa date de dernière modification, etc. La valeur de ces propriétés peut être une valeur littérale ou bien une autre ressource. Chaque *triplet* (ressource, propriété, valeur) est représenté en RDF par une *déclaration*, et un ensemble de déclarations RDF peut être vu comme un graphe orienté étiqueté : chaque sommet représente une ressource (s'il est étiqueté par un URI) ou une valeur littérale (s'il est étiqueté par cette valeur), et chaque arc représente une déclaration (étiqueté par la propriété correspondante, et pointant de la ressource possédant la propriété vers sa valeur). Notons que pour RDF, tout objet identifiable est une ressource ; si cet objet ne possède pas d'URI connue, il sera représenté par un sommet sans étiquette. La figure 3.3 peut donc être lue comme :

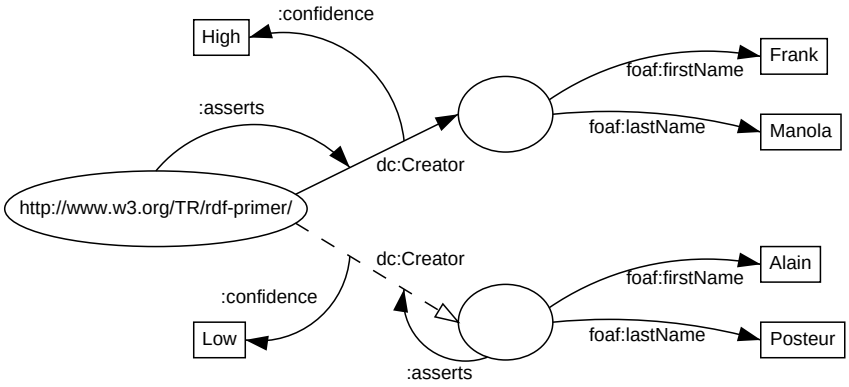
> La ressource http://www.w3.org/TR/rdf-primer a pour titre *RDF Primer*, elle est apparentée à la ressource http://www.w3.org/TR/rdf-mt, et elle a pour auteur une ressource (d'URI inconnue) dont le prénom est Frank, le nom de famille est Manola, et l'adresse électronique est mailto:fmanola@mitre.org.

Un autre aspect notable de RDF est sa réflexivité : il est en effet important de pouvoir décrire toute sorte de ressource, y compris les éléments qui constituent un graphe RDF (propriétés, déclarations). Les propriétés sont donc des ressources, identifiées bien entendu par des URIs. À noter que sur la figure 3.3, les étiquettes d'arcs sont des notations abrégées représentant bien des URIs (notations inspirées de XML-Names et de Notation 3, cf. Berners-Lee 1999). De la même

[9] À titre de contre-exemple, les adresses de courrier électronique, identifiées par des URIs de type mailto:, ne permettent pas directement d'accéder à des données.



(a)



(b)

FIG. 3.4 – Représentations syntaxique (a) et concise (b) de la réification RDF

façon, un procédé de *réification* permet de représenter une déclaration (un arc du graphe) par une ressource, et donc d'affecter des propriétés à cette déclaration, ou d'en faire la valeur des propriétés d'autres ressources. Ce procédé assez lourd syntaxiquement (ajout d'une ressource ayant quatre propriétés pour identifier la déclaration réifiée) peut cependant s'interpréter de façon assez naturelle [Conen *et al.*, 2002], comme le montre la figure 3.4. On peut lire cette figure comme :

> L'auteur de la ressource *RDF Primer* est Frank Manola[10]. Ceci est affirmé par la ressource elle même, et cette affirmation est hautement fiable. Alain Posteur affirme également être auteur de cette ressource, information peu fiable.

Dans la représentation correspondant littéralement à la syntaxe RDF (figure 3.4-a), les deux déclarations dont il est question (le fait que Frank Manola soit auteur de la ressource d'une

[10] Plus exactement « une ressource ayant prénom Frank et nom de famille Manola ». Idem pour Alain Posteur dans la suite.

part, et le fait qu'Alain Posteur soit également auteur de cette ressource) sont réifiées par les ressources représentées en gris. Chacune représente de façon non ambiguë une déclaration grâce aux quatre propriétés représentées en trait épais, qui indiquent respectivement leur qualité de déclaration (rdf:type→ rdf:Statement), la ressource décrite (rdf:subject), la propriété (rdf:predicate) et sa valeur (rdf:object). Tous les éléments représentés en trait épais sont ceux ajoutés pour les besoins de la réification (on y retrouve la ressource représentant la propriété dc:Creator). Dans une représentation graphique plus concise (figure 3.4-b), on peut réifier la déclaration simplement en autorisant l'*arc* la représentant à être à l'extrémité d'autres arcs (les déclarations impliquant la ressource réifiante).

Enfin, on notera que le graphe RDF de cet exemple ne déclare *pas* qu'Alain Posteur est l'auteur de la ressource considérée, bien qu'il *mentionne* cette déclaration pour lui attribuer des propriétés. Ceci se traduit par l'absence, dans la représentation syntaxique, d'un arc étiqueté par dc:Creator entre les deux ressources. Dans la représentation concise, les déclarations réifiées sont représentées par des arcs ; cette déclaration uniquement mentionnée est représentée en pointillés, pour la différencier des arcs effectivement déclarés.

Les rôles de XML et de RDF dans le futur Web Sémantique sont complémentaires. Le premier vise à fournir un cadre dans lequel peuvent être définies des *syntaxes* de document appropriées à un domaine particulier, sans aucune implication sémantique particulière. RDF propose en revanche un cadre *sémantique* minimum, spécifié par Hayes [2002], qui ne repose sur aucune syntaxe particulière : on peut l'exprimer en XML [Beckett, 2002], mais des syntaxes alternatives ont également été proposées [Berners-Lee, 1999]. De même que des applications peuvent être fondées sur la sémantique de RDF sans utiliser sa syntaxe XML, des applications peuvent utiliser la syntaxe XML avec une autre sémantique. Cependant, leur interopérabilité au sein du Web Sémantique sera maximale si elles adoptent les deux.

3.2.5 Au delà de RDF

La sémantique de RDF est spécifiée pour être « ontologiquement neutre », ce qui signifie qu'elle ne repose sur aucune conceptualisation particulière du monde qu'elle décrit. Il en découle que RDF seul ne permet que très peu d'inférences. Pour augmenter ce pouvoir inférentiel, il faut associer à certaines ressources une sémantique particulière. C'est le rôle des *ontologies*. Une ontologie est une description formelle d'un domaine de connaissance, décrivant généralement les catégories d'objets qui constituent ce domaine, et les relations pouvant exister entre les individus appartenant à ces catégories.

Par exemple, dans une ontologie simple de sens commun, des catégories seraient ⟨Personne⟩ et ⟨Adresse-électronique⟩. Une personne peut être en relation ⟨a-pour-adresse⟩ avec aucune, une ou plusieurs adresses électroniques, et en relation ⟨a-pour-mère⟩ avec une personne (différente d'elle même) et une seule. Cette description permet bien de faire des inférences : si une personne est en relation ⟨a-pour-adresse⟩ avec une ressource inconnue, on peut déduire que cette ressource est une adresse électronique ; si une personne est en relation ⟨a-pour-mère⟩ avec plusieurs ressources (c'est-à-dire des ressources ayant des URIs différents), on peut en déduire que ces ressources sont en fait une seule et même personne (donc que leurs URIs sont interchangeables).

Notons que les ontologies ne sont pas apparues avec le Web Sémantique. Elle sont un sujet de recherche actif dans le domaine de l'ingénierie des connaissances [Gil *et al.*, 2001; Bachimont, 2002]. Les langages présentés ci-après s'inspirent d'ailleurs des travaux de cette communauté, mais sont plus directement destinés à être déployés dans le contexte du Web.

RDF-Schema

Les schémas RDF, proposés par Brickley et Guha [2001], offrent quelques primitives pour décrire des ontologies. Ils sont composés de *classes* (catégories) et de *propriétés* (relations). Les classes sont structurées par des relations d'inclusion (sous-classe, super-classe) : par exemple, la classe ⟨Femme⟩ est une sous-classe de (ou est incluse dans) la classe ⟨Personne⟩ (toute femme est également une personne). Les propriétés sont également structurées par une relation d'inclusion (sous-propriété, super-propriété) : la relation ⟨a-pour-parent⟩ par exemple est une super-propriété de ⟨a-pour-mère⟩ (la mère d'une personne est également un de ses parents). Enfin, il est possible de contraindre les classes des individus pouvant être impliqués dans une propriété : par exemple, la propriété ⟨a-pour-mère⟩ s'applique aux personnes, et a pour valeur un individu de la classe ⟨Femme⟩. On parle respectivement du *domaine* et de la *portée* d'une propriété.

Un point notable des schémas RDF est qu'ils ne séparent pas strictement les classes des individus : les unes et les autres sont des ressources, considérées sur le même plan. Cette approche est cohérente avec la volonté de réflexivité de RDF (en RDF les propriétés sont déjà des ressources). Elle peut être utile dans certains domaines d'application : en biologie, par exemple, ⟨Poisson⟩ est une classe, mais c'est également une instance de la classe ⟨Espèce⟩[11]. Cette expressivité a bien sûr une contrepartie computationnelle, augmentant la complexité des algorithmes d'inférence. Cependant comme le fait remarquer Hayes [2002], une application particulière peut ignorer cette possibilité si elle ne l'utilise pas, et appliquer des mécanismes d'inférence plus efficaces. Par ailleurs, considérer classes et propriétés comme des individus permet de décrire un schéma directement en RDF, voire de mélanger dans le même graphe des données de schéma avec des données utilisant le schéma. Ce que Brickley et Guha [2001] proposent est d'ailleurs un vocabulaire RDF pour décrire des schémas (un méta-schéma, donc).

Une dernière conséquence de cette réflexivité est que toute sur-couche de RDF-Schema peut être décrite à l'aide de RDF-Schema lui même. C'est d'ailleurs le cas des langages DAML+OIL et OWL décrits ci-après. Si l'on prend l'exemple de OWL, qui a vocation à être plus expressif que RDF-Schema, sa description en RDF-Schema est nécessairement partielle. Cependant, tout agent «comprenant» RDF-Schema pourra s'appuyer sur cette description pour interpréter un énoncé OWL, ou plus exactement la *projection* de cet énoncé dans la sémantique de RDF-Schema. RDF-Schema fournit même un mécanisme pour exprimer que l'emploi d'un terme est soumis à des contraintes non exprimables.

DAML+OIL, OWL

Les primitives offertes par RDF-Schema sont insuffisantes pour décrire des ontologies complexes. Elles ne permettent pas, par exemple, de stipuler qu'une propriété ne peut prendre qu'une seule valeur (comme ⟨a-pour-mère⟩), que deux classes n'ont aucun individu en commun (comme ⟨Homme⟩ et ⟨Femme⟩), ou encore de définir une classe par un ensemble de contraintes sur ses instances, comme c'est le cas dans les logiques de description (par exemple, la classe ⟨Jeune-cadre⟩ peut être définie comme l'ensemble des personnes dont la propriété ⟨age⟩ a une valeur inférieure à trente, et dont la propriété ⟨profession⟩ a pour valeur «cadre»).

Le langage DAML+OIL est issu des efforts conjoints du projet européen On-To-Knowledge[12] et du groupe de travail américain Darpa Agent Markup Language[13], pour développer un langage de description d'ontologies pour le Web au dessus de RDF-Schema et plus puissant que ce

[11] Notons que ce n'est en aucun cas une *sous-classe* de ⟨Espèce⟩, car les individus de la classe ⟨Poisson⟩ ne sont *pas* des espèces.

[12] http://www.ontoknowledge.org

[13] http://www.daml.org

dernier. Ce langage est notamment décrit par Connolly *et al.* [2001]. Sur la base de ces travaux, le W3C a lancé un groupe de travail pour publier une recommandation du consortium portant sur un langage de description d'ontologies pour le Web. Ce groupe de travail a déjà publié une liste de critères que doit remplir un tel langage [Heflin *et al.*, 2002], ainsi qu'une première version du langage OWL [McGuinness et van Harmelen, 2002].

3.3 Le Web Sémantique et l'assistance

On retrouve dans les thèmes abordés pour le Web Sémantique un certain nombre des critères évoqués dans le chapitre 1 pour le modèle de l'assistant. L'idée centrale du Web Sémantique est bien en effet d'*assister* un utilisateur dans sa tâche, tâche qui implique l'appropriation d'un certain nombre de ressources. L'informatique documentaire s'est donc rapprochée petit à petit de la représentation des connaissances (séparation des structures logique et physique, importance des méta-données, introduction des ontologies) en gardant un lien fort avec l'utilisateur et les contraintes afférentes (notamment de facilité et de souplesse d'utilisation). À l'inverse, ces contraintes sont apparues de manière de plus en plus présentes dans le domaine de l'intelligence artificielle. D'une certaine manière, le Web Sémantique est le point de rencontre des deux disciplines, et un terrain de recherche porteur pour la création d'assistants intelligibles et intelligents.

Deuxième partie

Un assistant pour la conception

Chapitre 4

Représenter l'expérience de conception

L'objectif de ce travail est de capturer, pour en permettre la réutilisation, des connaissances de conception. Plutôt que de chercher à généraliser des connaissances complexes et multiples, on s'attachera à capturer des connaissances contextualisées, correspondant à des situations précises : les épisodes de conception. En ce sens, c'est bien une modélisation de l'*expérience* de conception qui est proposée. L'ambition de cette représentation est de permettre à un assistant de répondre à des situations diverses (à la mesure de son expérience dans ces situations) plutôt qu'un assistant spécialisé dans une tâche particulière. Après avoir présenté le contexte applicatif en section 4.1, on définira la notion d'épisode de conception pour modéliser l'expérience (section 4.2). Enfin, la section 4.3 propose une représentation à l'aide du langage RDF de ces épisodes.

4.1 Le contexte applicatif

L'application visée par ce travail est le logiciel de CFAO CATIA de Dassault Systèmes, une suite applicative offrant de nombreux modules prenant en charge toutes les étapes de la conception d'un produit. Elle ne sera pas décrite intégralement ici, mais on se focalisera plutôt sur certains aspects clés à l'égard de ce travail.

4.1.1 Un outil versatile

L'application CATIA couvre une très large partie de la conception d'un produit. Par exemple, elle permet de définir la géométrie des pièces d'un assemblage, les contraintes mécaniques entre ces pièces, une analyse de résistance des matériaux tenant compte de ces contraintes, puis les gammes d'usinages des pièces.

La version 5 introduit pour cela la notion d'*atelier*. Chaque atelier correspond à un type précis de tâche (dessiner un profil en deux dimensions, définir les contraintes dans un assemblage, etc.), et comporte uniquement les menus et les barres d'outils utiles à cette tâche (cf. figure 4.1). Certains ateliers sont liés à un métier particulier (installation électrique, usinage), d'autres sont plus liés à des tâches transversales ; c'est notamment le cas des différents ateliers de la famille KNOWLEDGEWARE, dédiés comme leur nom l'indique à la gestion des connaissances de conception.

Le premier intérêt de ce principe d'ateliers est l'ergonomie. Ils évitent de surcharger l'interface avec des informations inutiles : seules sont présentes les commandes utiles à la tâche courante, ainsi que celles permettant le passage vers des ateliers connexes. Par ailleurs, les ateliers sont en fait des modules indépendants : chaque poste de travail ne contient que les ateliers nécessaires

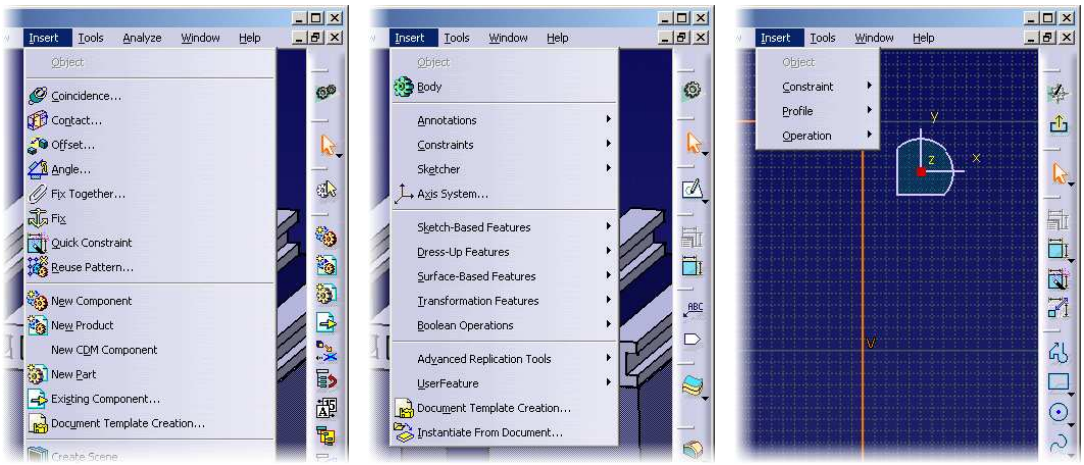


FIG. 4.1 – Extraits de l'interface CATIA dans trois ateliers différents (*Assembly design*, *Part design* et *Sketcher*)

au concepteur qui l'utilise. D'un certain point de vue, on pourrait donc considérer que CATIA est un ensemble d'applications relativement indépendantes. Cependant, ces ateliers travaillent pour la plupart sur une même et unique structure de données, ce qui permet d'éviter les problèmes de conversion soulevés par l'utilisation d'outils différents, et assure une intégration optimale des différentes étapes de la conception.

4.1.2 Des points de vue multiples

Outre les multiples points de vue offerts par les différents ateliers, CATIA présente au concepteur deux vues complémentaires de l'objet qu'il est en train de concevoir : une vue *géométrique* et une vue *hiérarchique* (cf. figure 4.2). La première est une représentation « figurative » de l'objet, et elle est habituelle en CAO. La seconde est quant à elle beaucoup plus riche sémantiquement : elle permet de représenter des éléments n'ayant pas de représentation géométrique immédiate (un profil, une contrainte, un paramètre, etc.), et les relations entre ces éléments. La vue hiérarchique fournit donc une meilleure visibilité que la vue géométrique sur la façon dont l'objet a été construit avec CATIA. Ces raisons font que la vue hiérarchique est utilisée largement autant que la vue géométrique par les concepteurs, dès lors qu'ils ont à interagir avec des éléments non purement géométriques.

Cette familiarité des concepteurs avec une vue plus symbolique de leur travail que la simple représentation géométrique du résultat, encourage à utiliser ce type de représentation pour un système d'assistance. En effet, elle ne nuit pas à son intelligibilité, et les informations qu'elle contient sont plus pertinentes (parce que d'un niveau d'abstraction plus élevé) pour la réutilisation. Si la représentation géométrique comporte des traits de structure, elle comporte également de nombreux traits de surface qu'il est difficile de discriminer des premiers sans connaissances approfondies du domaine d'application dont relève l'artefact (par exemple : la forme de telle pièce a-t-elle un rôle fonctionnel particulier ?). Au contraire, la représentation hiérarchique présente une vue plus abstraite et donc dépouillée de ses traits de surface, et ceux qui demeurent peuvent être déterminés à l'aide de connaissances sur le logiciel, plus faciles à acquérir (par exemple, l'ordre de ces éléments est-il significatif ?).

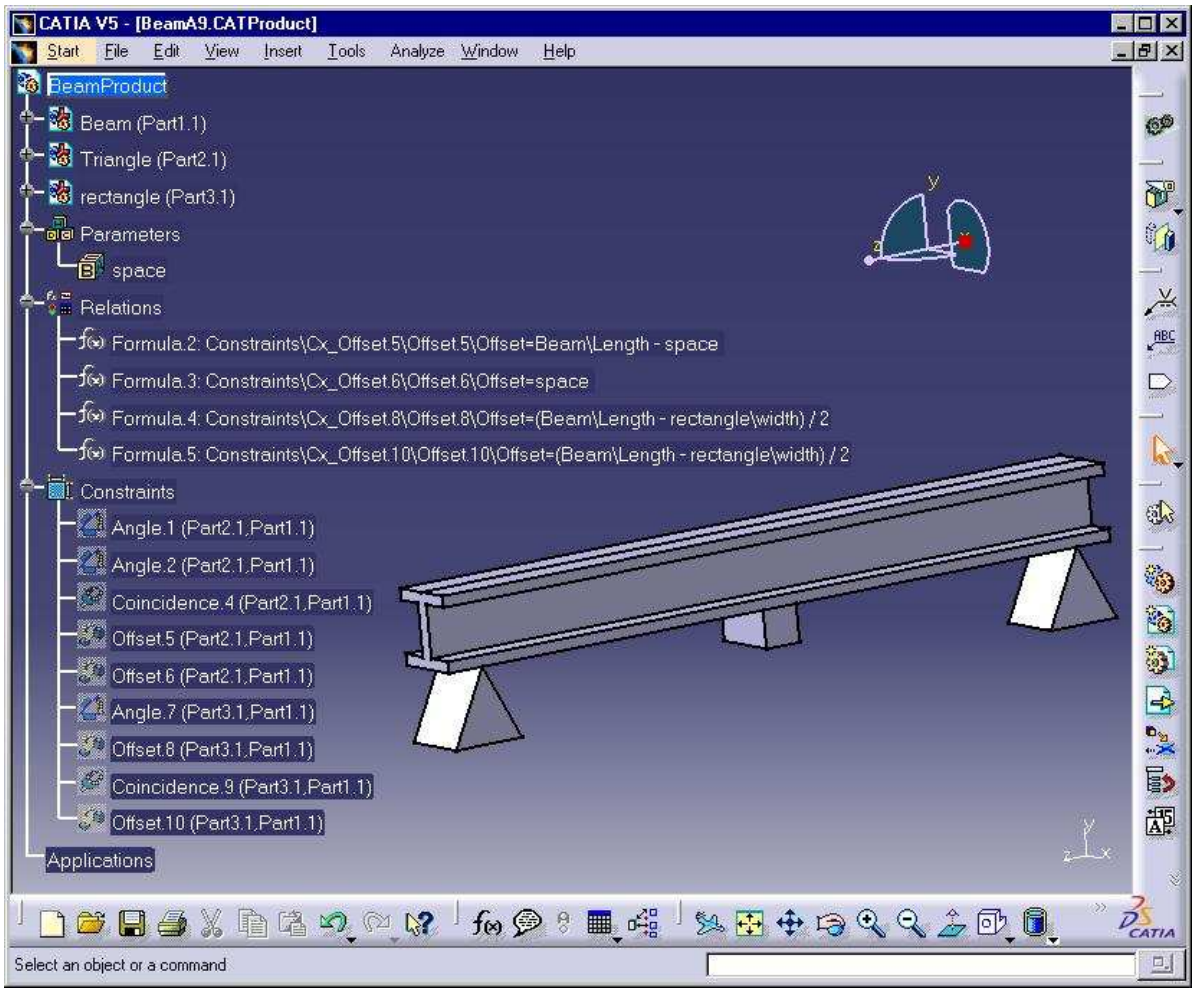


FIG. 4.2 – Les deux vues hiérarchique et géométrique dans CATIA

4.1.3 Un environnement ouvert

Enfin, CATIA offre une interface de programmation : CAA (*Component Application Architecture*). Cette dernière permet d'enrichir l'application par des modules développés pour des besoins spécifiques. Cette interface de programmation reste dans la lignée de ce qui est présenté ci-avant : un objet (au sens de la programmation) n'est jamais manipulé directement, mais toujours à travers une *interface* — similaire aux interfaces du langage JAVA [Flanagan, 2002]. Un objet adhère généralement à de nombreuses interfaces, ce qui donne de multiples points de vue sur lui. Ces points de vue ne sont d'ailleurs pas très éloignés de ceux qu'a l'utilisateur final de l'application, la programmation CAA ayant pour but d'avoir un niveau d'abstraction proche de celui des utilisateurs.

4.2 Épisodes de conception

4.2.1 Qu'est-ce qu'un épisode ?

L'application du raisonnement à partir de cas au domaine de la conception soulève une difficulté : le RàPC s'appuie sur les notions de problème et de solution. Or, en tant que processus de résolution de problèmes, la conception traite par définition de problèmes mal posés, construits en même temps que leur solution, et extrêmement spécifiques et complexes, ce qui rend la réutilisation de leur solution pratiquement impossible. Il faut donc identifier des parties de l'activité de conception qui soient suffisamment simples pour être réutilisables. Cette simplicité peut provenir d'une restriction à un sous-problème, ou d'une abstraction de la solution afin de masquer sa complexité.

La notion d'*épisode de conception*, déjà proposée par Mille *et al.* [1999] pour répondre à ce type de problème, vise à capturer ces parties réutilisables de l'activité de conception. En se fondant sur le point de vue opportuniste de l'activité de conception (cf. section 2.1.2), la définition adoptée au sein du projet ARDECO est la suivante :

> Un **épisode de conception** est une partie de l'activité de conception située entre le moment où un objectif est identifié, et le moment où cet objectif est jugé satisfait ou non pertinent.

Cette définition rend bien compte de l'aspect opportuniste de la conception : les buts ne sont pas déterminés *a priori* suivant un plan préétabli, mais bien découverts au fur et à mesure de la conception. De la même façon, l'épisode prend fin à la satisfaction de l'objectif (ou du moins lorsque le concepteur estime que l'objectif est satisfait), ou le cas échéant lorsque le concepteur révise son jugement et estime que cet objectif n'est plus pertinent. En se limitant à la satisfaction d'un objectif, cette définition satisfait bien la nécessité de simplicité évoquée plus haut : soit cet objectif est suffisamment élémentaire pour être résolu simplement, soit il se décompose lui même en sous-objectifs dont on peut faire abstraction de la résolution.

4.2.2 Comment les délimiter ?

Une fois donnée la définition d'un épisode de conception, reste à être capable de les identifier pendant l'activité du concepteur. Cela a été l'objet d'une partie des travaux des chercheurs en ergonomie cognitive du projet ARDECO, travaux décrits dans [Bougé *et al.*, 2001]. Ces travaux ont consisté dans l'observation de concepteurs utilisant CATIA et verbalisant leur activité, afin de mettre en lumière des invariants comportementaux correspondant aux limites d'un épisode de conception.

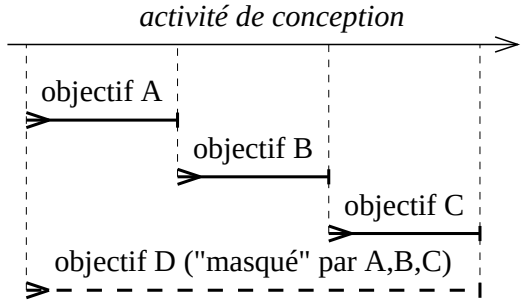


FIG. 4.3 – Délimitation automatique et objectif complexe

Il a été mis en évidence que les épisodes sont encadrés par deux phases caractéristiques. En début d'épisode, une phase dite de «repérage» consiste à explorer l'environnement de travail (l'artefact, mais également les menus de l'application) afin de déterminer le nouvel objectif. Cette phase se manifeste donc par une utilisation intensive des fonctions de changement de point de vue : rotations, translations et changement d'échelle. En fin d'épisode, la phase dite d'«évaluation» consiste à vérifier un certain nombre de points indiquant la satisfaction de l'objectif. Cette phase se manifeste également par de nombreux changements de points de vue, quoique plus rapides et plus grossiers que dans la phase précédente.

Il est donc possible d'instrumenter CATIA de façon à détecter automatiquement les bornes des épisodes de conception : toute manipulation intensive du point de vue peut être interprétée comme la fin d'un épisode et le début du suivant. Notons que cette détection automatique des bornes d'épisode est nécessaire pour initier le fonctionnement du système, mais elle ne constitue pas la seule façon. Une instrumentation permettant une déclaration *explicite* des épisodes par le concepteur est également nécessaire si ce dernier souhaite donner des informations plus précises à l'assistant. Le système ne peut cependant pas s'appuyer uniquement sur une saisie manuelle des épisodes, trop contraignante pour le concepteur.

Notons également que la délimitation automatique des épisodes n'a de visibilité que sur les objectifs simples, c'est-à-dire ne comportant pas de sous-objectif. Dans le cas contraire, seuls les épisodes correspondant aux sous-objectifs seront détectés directement (cf. figure 4.3). Par ailleurs, dans cette figure l'objectif C pourrait n'être considéré que comme la partie de la résolution de A comprise entre la résolution de B et l'identification de D. On prendra le parti de considérer malgré tout ce fragment de l'activité de conception comme un épisode à part entière, donc l'objectif serait «passer de B à D dans le contexte de A».

Enfin, les expérimentations ont montré une grande influence du niveau d'expertise sur les épisodes délimités. Plus précisément, un utilisateur expérimenté de CATIA se fixera des objectifs de haut niveau, alors qu'un débutant réalisera la même tâche avec de plus nombreux épisodes, correspondant à des objectifs plus élémentaires. Ce résultat pose la question de la réutilisabilité des épisodes d'une personne à l'autre : un épisode de conception d'expert sera difficilement compréhensible par un débutant, parce que trop abstrait, tandis qu'un épisode de conception de débutant sera inutile pour un expert, qui le considérera comme une opération élémentaire. Bien sûr, le niveau d'expertise dépend également du type de tâche, et en pratique, un concepteur passera par des épisodes abstraits et d'autres plus détaillés selon sa familiarité avec certains aspects du problème.

4.2.3 Comment les modéliser ?

Deux façons de modéliser les épisodes ont été envisagées : une modélisation *dynamique*, et une modélisation *différentielle*. Dans la première, un épisode est représenté comme la séquence d'actions élémentaires effectuées pendant cet épisode. La seconde consiste a représenter simplement un épisode par deux « photographies » de l'*état* de l'application, correspondants au début et à la fin de l'épisode.

La représentation dynamique peut sembler préférable car elle porte plus d'information : en théorie, on peut reconstruire les états en rejouant les actions élémentaires. On pourrait même penser que cette possibilité de rejouer un épisode soit un avantage pour la phase de réutilisation. Cependant, la représentation dynamique pèche par cette richesse d'information et par son manque d'abstraction. Notamment, l'ordre des actions élémentaires n'est pas toujours significatif, et cela dépend en grande partie de leur contexte d'application. Il est donc difficilement envisageable d'acquérir les connaissances du domaine permettant de juger si l'ordre entre deux actions est significatif ou non, et ce afin de comparer les épisodes entre eux. Qui plus est, pour des épisodes abstraits (tels ceux délimités par un expert), certaines actions relèvent d'un niveau de détail non pertinent, et là encore, une discrimination automatique réclamerait beaucoup trop de connaissances du domaine. Enfin, la réutilisation d'une séquence d'action suppose de pouvoir adapter cette séquence à un nouveau contexte.

La représentation différentielle ne s'attache quant à elle qu'au contexte d'application de l'épisode (son état initial) et à son résultat (son état final). La séquence des actions ayant conduit à ce résultat est ignorée, ce qui permet de s'abstraire de ses spécificités contextuelles. Toutefois, on peut déterminer par différence sur quelles parties des états ces actions ignorées ont porté. Par ailleurs, en utilisant une représentation hiérarchique des états, du type de celle manipulée par les utilisateurs en plus de la vue géométrique, on peut localiser dans cette hiérarchie les modifications apportées pendant l'épisode, et ne considérer que celle au dessus d'un certain niveau de détail.

4.2.4 Modèles d'utilisation

La représentation différentielle suppose de représenter l'état de l'application à deux instants pour modéliser un épisode. Se pose alors le choix de la façon de représenter ces états. On a déjà fait allusion à la représentation hiérarchique présentée par CATIA aux utilisateurs. Cette représentation n'est cependant pas tout à fait satisfaisante telle quelle, car elle est strictement arborescente, et un certain nombre de relations importantes n'y figurent pas (par exemple, la relation entre une contrainte mécanique et les pièces qu'elle contraint, ou la relation entre une formule de calcul de longueur et l'élément dont une longueur est basée sur cette formule). Une représentation plus structurée est donc nécessaire. D'un autre coté, garder toutes les relations effectivement implantées dans chaque objet n'est pas une solution : de nombreux objets de l'implantation n'ont qu'un rôle technique et ne sont pas connus des utilisateurs.

Les ateliers KNOWLEDGEWARE proposent un jeu d'interfaces (au sens de CAA) nommé CATITYPE, spécifié pour donner une vue des objets centrée sur l'utilisateur, et ne se limitant pas à un atelier particulier — rappelons que KNOWLEDGEWARE propose des ateliers *transversaux*, non liés à un métier particulier. Cette vue homogène est particulièrement intéressante pour assurer la généralité d'un assistant l'utilisant pour représenter les cas. Hélas, il s'est avéré que de nombreuses classes adhérant à ces interfaces ne respectent pas strictement les spécifications de KNOWLEDGEWARE. Il en découle que la structure résultante est beaucoup moins porteuse de sémantique que ce que ces spécifications laissaient supposer, ce qui a conduit à l'abandon de

cette piste.

On propose donc la notion de *modèle d'utilisation* (MU) pour répondre au problème de la modélisation des états. Cette notion sera présentée de façon plus détaillée au chapitre 7, mais on peut considérer ici qu'un MU est simplement une manière de décrire les états de l'application. Les épisodes décrits à l'aide d'un MU donné ne peuvent bien sûr pas être comparés avec ceux décrits à l'aide d'un autre MU. Bien que l'idéal eût été d'avoir un MU unique comme celui proposé par KNOWLEDGEWARE, cette architecture présente l'avantage de la modularité : on peut envisager de développer des MUs généralistes couvrant une grande variété de tâches, mais également des MUs très spécialisés ne couvrant que quelques tâches mais offrant en contrepartie une représentation spécifique plus facile à exploiter.

4.3 Représenter les épisodes de conception

4.3.1 Le choix de RDF

Les épisodes de conception ne sont pas des connaissances explicites, mais contiennent de façon implicite les connaissances qui ont présidé à leur réalisation. L'objectif principal de notre assistant n'est donc pas de manipuler intensivement ces connaissances épisodiques, mais de mettre les plus pertinentes à disposition de l'utilisateur. La représentation des épisodes a donc une vocation *documentaire*, en plus de permettre la mise en œuvre de mécanismes de RàPC. C'est pourquoi on s'est intéressé aux technologies du Web Sémantique (cf. chapitre 3), qui sont à la frontière des deux domaines.

RDF est en passe de devenir le standard de fait du futur Web Sémantique et paraît, avec le vocabulaire de représentation de schémas RDF-Schema, un bon candidat pour la représentation des épisodes. Ses principes de base sont suffisamment simples pour permettre l'implantation de modèles d'utilisation par des développeurs non spécialistes en représentation des connaissances, et la représentation graphique est proche de la représentation hiérarchique à laquelle les utilisateurs de CATIA sont accoutumés. Cette simplicité n'entrave cependant pas l'expressivité du langage, puisque comme on l'a vu, des langages plus expressifs comme DAML+OIL et OWL peuvent être développés comme des sur-couches de RDF-Schema. Un modèle d'utilisation particulier pourra donc s'appuyer sur ces langages si nécessaire, et ainsi bénéficier des mécanismes d'inférence associés. Les mécanismes développés pendant la thèse et présentés ici ne s'appuient en revanche que sur RDF-Schema.

4.3.2 États et transitions

Cette section donne une description formelle de la modélisation des épisodes. Un schéma RDF-Schema correspondant à cette modélisation est donné en annexe B.1.1. Les termes RDF correspondant à ce schéma seront donnés en police sans-serif avec le préfixe ep:.

État

Un état de l'application est représenté par un graphe RDF à l'aide d'un modèle d'utilisation. C'est le modèle d'utilisation qui définit les termes employés dans ce graphe RDF. Au minimum, le modèle d'utilisation s'appuie sur RDF-Schema pour décrire ces termes (classes, instances et propriétés), mais il peut également recourir à des langages plus expressifs. Il convient de s'attarder sur deux propriétés particulières, ep:component et rdf:type.

La propriété ep:component est le seul terme préconisé indépendamment de tout modèle d'utilisation pour la description des épisodes. Elle a la sémantique de la relation de composition : elle est transitive, et une ressource ne peut pas être un composant direct de plus d'une autre ressource. Ces méta-propriétés ne peuvent pas s'exprimer en RDF-Schema, mais il semble important de donner à tout modèle d'utilisation un moyen standard de représenter cette relation. Elle a en effet une grande importance dans la plupart des domaines de conception, et des expérimentations menées au sein du projet ARDECO par les chercheurs en ergonomie cognitive ont montré qu'elle jouait pour la remémoration un rôle prépondérant par rapport à d'autres types de relation [Bougé, 2003].

La propriété rdf:type a également un rôle très particulier dans RDF-Schema, puisqu'elle permet d'indiquer l'appartenance d'une ressource à une classe. Plus généralement, on préférera dire qu'elle indique une *caractéristique* de la ressource, un prédicat unaire que cette ressource vérifie. La notion d'appartenance à une classe est en effet trop connotée : dans de nombreux formalismes, elle est considérée comme intrinsèque à l'objet, et non susceptible de changer ; on verra qu'au contraire, certaines caractéristiques peuvent apparaître ou disparaître pour un objet donné (être sélectionné pour un objet d'interface, être violée pour une contrainte mécanique)[1]. Le modèle formel adopté pour les états est une abstraction du modèle RDF dans laquelle les propriétés rdf:type n'apparaissent plus comme des arcs, mais comme des étiquettes sur les sommets du graphe — ces propriétés ont un rôle particulier qui nécessite un traitement différent des autres arcs et sommets. Par ailleurs, on préférera considérer sans perte de généralité que des propriétés multiples liant un sommet à un autre sont un seul arc étiqueté par plusieurs prédicats, plutôt que plusieurs arcs comme on le fait habituellement en RDF. On appellera ce modèle un *graphe orienté multi-étiqueté* qu'on définira comme un sextuplet :

$$\mathcal{G} = \langle V, A, \Phi_V, \Phi_A, \lambda_V, \lambda_A \rangle$$

tel que :

- V est l'ensemble fini des sommets,
- $A \subseteq V \times V$ est l'ensemble des arcs (orientés),
- Φ_V est l'ensemble fini des caractéristiques de sommet (classes RDF-Schema),
- Φ_A est l'ensemble fini des caractéristiques d'arc (propriétés RDF-Schema),
- $\lambda_V : V \longrightarrow \wp(\Phi_V) - \{\emptyset\}$ est la fonction qui associe à chaque sommet un *ensemble* non vide de caractéristiques,
- $\lambda_A : A \longrightarrow \wp(\Phi_A) - \{\emptyset\}$ est la fonction qui associe à chaque arc un ensemble non vide de caractéristiques.[2]

Notons que dans la sémantique de RDF-Schema, une ressource appartenant à une classe appartient également à ses super-classes, et de même une déclaration ayant une propriété pour prédicat admet également pour prédicat toutes ses super-propriétés. Les fonctions λ_V et λ_A sont supposées tenir compte de cette sémantique : un sommet est donc étiqueté par *toutes* les classes auxquelles appartient la ressource qu'il représente, et un arc, par *toutes* les propriétés reliant les deux ressources. Par exemple, tout sommet étiqueté par la classe Assemblage sera également étiquetée par sa super-classe Produit ; tout arc étiqueté par la propriété sous-assemblage sera également étiqueté par sa super-propriété ep:component.

[1] Cette distinction n'est en aucun cas une distinction formelle. Toute caractéristique définit implicitement une classe : l'ensemble des objets possédant cette caractéristique. C'est pourquoi on emploie la relation rdf:type et la notion de classe de RDF-Schema pour représenter les caractéristiques.

[2] Il est également parfois utile de considérer que le graphe est complet par définition ($A = V \times V$), mais que λ_A peut n'affecter aucune étiquette ($\emptyset$) à certains arcs.

On aura également souvent besoin de la relation (en tant qu'ensemble de couples) induite par la fonction λ_V (respectivement λ_A), entre l'ensemble V (resp. A) et l'ensemble des étiquettes Φ_V (resp. Φ_A). On note cette relation $d_V(\mathcal{G})$ (resp. $d_A(\mathcal{G})$).

$$d_V(\mathcal{G}) \doteq \{(v,c) \in V \times \Phi_V |\ c \in \lambda_V(v)\}$$

$$d_A(\mathcal{G}) \doteq \{(a,c) \in A \times \Phi_A |\ a \in \lambda_A(a)\}$$

On utilisera également l'union de ces deux ensembles de couples pour représenter toutes les caractéristiques d'un graphe. On l'appelle *descripteur* du graphe, qui sera noté $d(\mathcal{G})$.

$$d(\mathcal{G}) = d_V(\mathcal{G}) \cup d_A(\mathcal{G})$$

Transition

Un épisode est représenté par son état initial $\mathcal{G}$ et son état final $\mathcal{G}'$. Les représentations de ces deux états contiennent donc toute l'information disponible sur l'épisode. Toutefois, on souhaite en donner une reformulation plus apte à la remémoration et à l'adaptation. Intuitivement, cette reformulation consiste à décrire la *transition* de l'état initial à l'état final, calculée comme la « différence » entre ces deux états. Plus précisément, on utilisera les descripteurs des deux états $\mathcal{G}$ et $\mathcal{G}'$ pour définir cette différence. Notons que deux états de la même trace contiennent des objets communs (des ressources RDF ayant la même URI). Dans le modèle formel cela se traduit par le fait que leurs ensembles de sommets V et V' ont une intersection non vide, et qu'il en est donc de même pour leurs ensembles d'arcs et leurs descripteurs. On définit l'opérateur de différence δ comme suit :

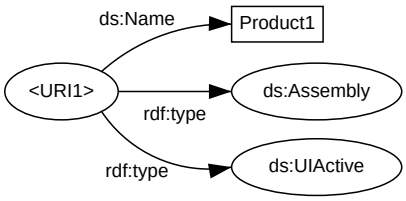
$$\delta_V(\mathcal{G}_i, \mathcal{G}_j) \doteq \{(v,\phi) \in d_V(\mathcal{G}_i) |\ (v,\phi) \notin d_V(\mathcal{G}_j)\}$$
$$\delta_A(\mathcal{G}_i, \mathcal{G}_j) \doteq \{(a,\phi) \in d_A(\mathcal{G}_i) |\ (a,\phi) \notin d_A(\mathcal{G}_j)\}$$
$$\delta(\mathcal{G}_i, \mathcal{G}_j) \doteq \delta_V(\mathcal{G}_i, \mathcal{G}_j) \cup \delta_A(\mathcal{G}_i, \mathcal{G}_j)$$

$\delta(\mathcal{G}, \mathcal{G}')$ contient donc les caractéristiques de sommet et d'arc qui ont été supprimées de $\mathcal{G}$, tandis que $\delta(\mathcal{G}', \mathcal{G})$ contient celles qui ont été ajoutées à $\mathcal{G}'$. Bien sûr, l'intersection des descripteurs $d(\mathcal{G}) \cap d(\mathcal{G}')$ contient les caractéristiques laissées inchangées par l'épisode. Il en découle que $\delta(\mathcal{G}, \mathcal{G}')$, $\delta(\mathcal{G}', \mathcal{G})$ et $d(\mathcal{G}) \cap d(\mathcal{G}')$ constituent une *partition* de $d(\mathcal{G}) \cup d(\mathcal{G}')$. On parlera respectivement des *différences* et des *points communs* entre les deux graphes.

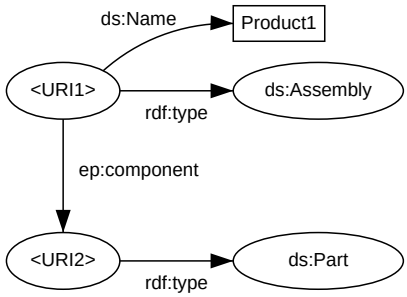
Les transitions sont représentées en RDF simplement par l'ensemble des différences entre les deux états. Chaque différence est représentée par un arc dans le graphe RDF, reliant un sommet à une caractéristique (pour une différence de δ_V), ou deux sommets (différence de δ_A). Dans les deux cas, la propriété étiquetant cet arc est construite à partir de la propriété originale (rdf:type pour les éléments de δ_V) et d'un vocabulaire spécifique. Elle a appartient à l'une des classes ep:PlusProperty ou ep:MinusProperty selon qu'elle correspond à un ajout ou une suppression, et elle possède une propriété ep:diffOf pointant vers la propriété originale (cf. figure 4.4).

Trace

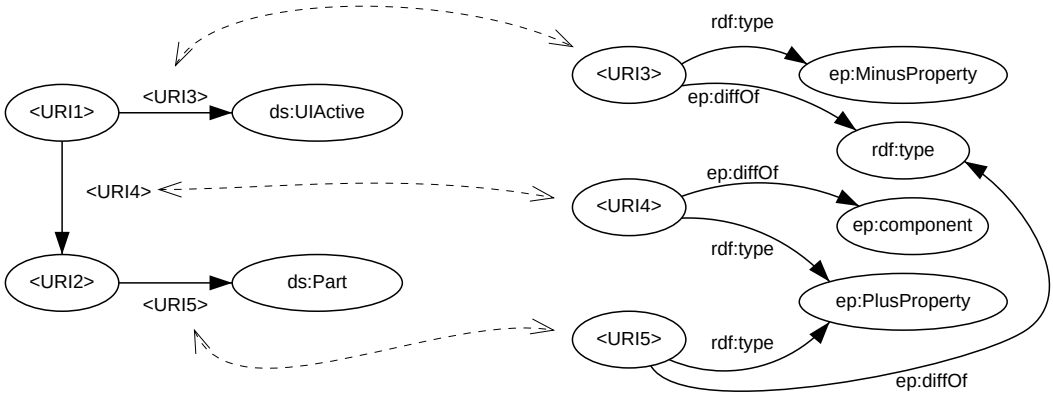
Une *trace* est la représentation, en terme d'épisodes de l'activité de conception. L'activité est ponctuée d'*états* qui sont les bornes des *épisodes*. La trace est donc une séquence alternée d'états et de transitions, qui commence et se termine nécessairement par un état (une transition joint nécessairement deux états).



(a) État initial (RDF)



(b) État final (RDF)



(c) Transition (RDF)

L'épisode entre les deux états représentés (a et b) comprend trois différences : suppression de la caractéristique ds:UIActive pour le sommet noté URI1, ajout de la caractéristique ds:Part pour le sommet noté URI2, et ajout d'une propriété ep:component entre ces deux sommets. Ces trois différences sont représentées par trois arcs dans le graphe RDF de la transition (c, à gauche), avec des propriétés notées URI3, URI4 et URI5 et décrites dans ce même graphe (à droite).

FIG. 4.4 – Représentation d'un épisode en RDF

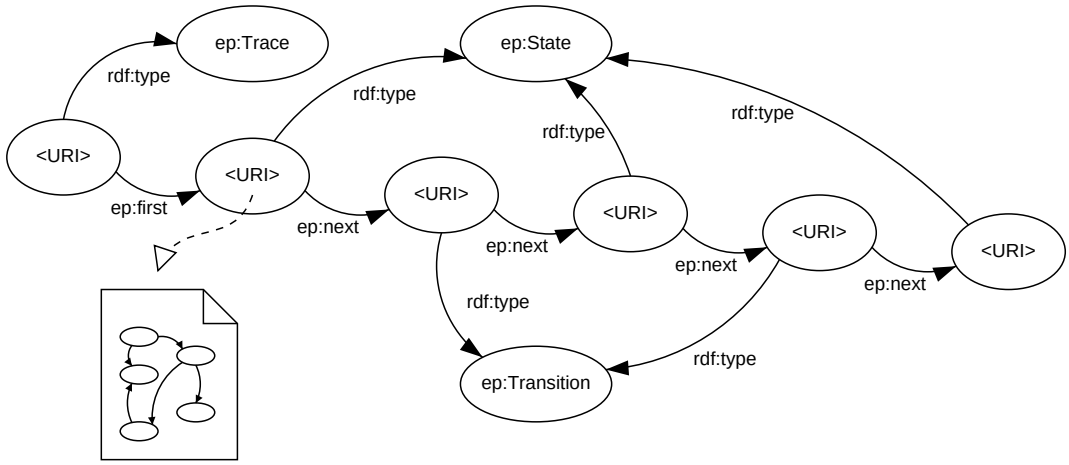


FIG. 4.5 – Représentation RDF d'une trace

À l'instar des états et des transitions, la trace est représentée en RDF. Dans cette représentation, les états et les transitions ne sont pas des graphes mais simplement des ressources dont l'URI permet d'accéder au graphe correspondant. Le schéma RDF définit donc trois classes ep:Trace, ep:State et ep:Transition. La propriété ep:first d'une trace pointe vers son premier état, et la propriété ep:next d'un état ou d'un épisode pointe vers l'élément suivant dans la trace (cf. figure 4.5.

Chapitre 5

Réutilisation d'expérience

L'assistant ne possède pas de connaissances approfondies sur le domaine d'expertise du concepteur. Pour juger de la pertinence à réutiliser un épisode, il se fonde donc simplement sur la similarité du *contexte d'application* de cet épisode avec le contexte courant, partant du principe inspiré de celui du raisonnement à partir de cas, que « des contextes similaires appellent des épisodes similaires », ou en d'autre termes que le contexte contient de façon implicite des informations sur la tâche du concepteur, et donc les raisons d'appliquer tel ou tel épisode. En ce sens, le fonctionnement de l'assistant peut se rapprocher du mécanisme de socialisation tel qu'il est décrit par Nonaka et Takeuchi [1995] : la transmission de connaissances tacites sans passage par l'explicite. Le contexte d'application d'un épisode comprend au minimum son état initial, mais peut également tenir compte de l'épisode qui le précède, voire de plusieurs épisodes précédents.

Une mesure de similarité est donc nécessaire pour la remémoration du contexte d'application d'un épisode réutilisable. Cette mesure de similarité ne doit pas seulement indiquer *à quel point* deux états ou deux épisodes sont similaires, mais également *pourquoi* ils le sont. Répondre à cette deuxième question nécessite de donner une mise en correspondance (ou *appariement*) des éléments de l'un avec les éléments de l'autre, justifiée par le fait que les éléments mis en correspondance sont similaires par eux mêmes, mais également par le fait qu'ils entretiennent le même type de relations, qu'ils *jouent le même rôle* par rapport aux autres éléments de l'état ou de l'épisode auquel ils appartiennent respectivement. Par exemple, les deux états de la figure 5.1 sont similaires car les quatre poutres a, b, c, d correspondent respectivement aux poutres 1, 2, 3, 4, et que les piliers e et f correspondent au mur 5. Les deux états ne sont pas parfaitement similaires, dans la mesure où cette correspondance n'est pas parfaite : d'une part les poutres n'ont pas le même profil (en I à gauche, en U à droite). D'autre part, le nombre de supports

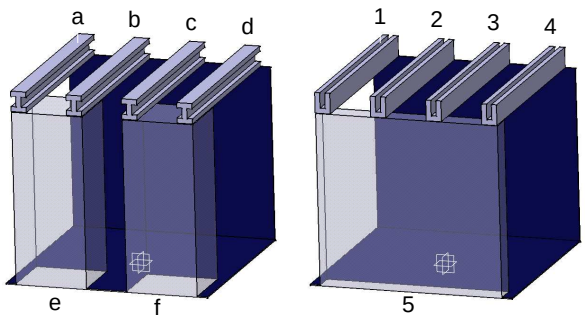


FIG. 5.1 – Deux assemblages similaires

n'est pas le même d'un cas à l'autre : le mur dans le cas de droite joue seul les rôles tenus par les deux piliers dans le cas de gauche.

On pourrait trouver d'autres mises en correspondance des éléments de ces deux états, mais c'est sans doute celle-ci qui rend le mieux compte de leur similarité. Déterminer pourquoi deux états ou épisodes sont similaires, c'est donc trouver la *meilleure* mise en correspondance entre leurs éléments, et c'est la qualité (qui reste à définir) de cette mise en correspondance qui déterminera à quel point ils sont similaires.

La correspondance entre éléments ne doit par ailleurs pas être perdue après la remémoration. Tout d'abord, elle sera d'une grande utilité lors de la phase de réutilisation, mais avant cela, elle servira aussi à justifier le choix du système auprès du concepteur. Ce dernier peut en effet n'être pas satisfait par ce que le système aura jugé être l'état ou l'épisode le plus similaire. Dans ce cas, il devrait avoir l'opportunité de critiquer ou corriger certaines décisions prises par le système. Enfin, ces critiques devront pouvoir être intégrées par le système, avec des portées variées : certaines porteront sur les connaissances générales du domaine, d'autres sur un cas particulier. La similarité doit en effet pouvoir se baser sur des connaissances générales autant que sur des connaissances contextuelles.

Ce chapitre présente la manière dont l'expérience capturée par les épisodes de conception peut être réutilisée. Cette présentation passe par les deux phases complémentaires du cycle du raisonnement à partir de cas : la remémoration (section 5.1) et la réutilisation (section 5.2).

5.1 Une mesure de similarité

5.1.1 Comparaison de graphes

Les graphes offrent un modèle de représentation souple et expressif; ils présentent en outre l'avantage d'avoir été largement étudiés en mathématiques discrètes. C'est pourquoi ils sont utilisés dans de nombreux domaines y compris la représentation des connaissances. Une fois ce type de représentation adopté, il est important de pouvoir comparer deux graphes : déterminer s'ils sont identiques (problème de l'isomorphisme), si l'un est plus général qu'un autre (problème de la subsomption), et dans quelle mesure (quantitativement ou qualitativement) deux graphes sont différents.

Dans le domaine de la représentation de connaissances, l'algorithme de projection de graphes conceptuels, proposé par Sowa [1999], est un exemple classique de recherche de subsomption entre graphes. En RàPC également, les graphes sont utilisés, mais la remémoration d'un cas ne se limite pas à la recherche d'un subsumant pour le problème courant : en général, les cas sont tous du même niveau de spécificité et on recherche le plus similaire au cas cible. La solution proposée par Plaza [1995] consiste à trouver le cas ayant avec le problème un subsumant commun le plus spécifique possible. Lieber et Napoli [1996] utilisent un principe similaire, s'assurant que les généralisations et spécialisations effectuées pour la similarité ont une contrepartie dans la phase d'adaptation : ainsi, ils garantissent que le cas similaire est effectivement adaptable.

Dans les systèmes de représentation de connaissances par objets, la subsomption *exacte* peut également s'avérer trop contraignante. Par exemple, dans les phases de développement itératif d'une base de connaissances, des objets encore mal définis peuvent ne pas entretenir de relation de subsomption *stricto sensu*, mais ce serait le cas s'ils étaient mieux définis. L'utilisation de mesure de similarité, pour élaborer des mécanismes de calcul de subsomption plus souples, est l'objet des travaux de **?** et de **?**. Le même type de relâchement par rapport à un calcul de subsomption, est proposé par **?** dans le domaine de la reconnaissance d'images. Ces derniers

proposent d'employer des graphes conceptuels *flous* (une extension des graphes conceptuels de Sowa, inspirée de la logique floue) auxquels est étendu le principe de la projection.

Des travaux théoriques ont également été menés autour de l'appariement approximatif de graphes (rappelons que pour comparer deux graphes, on doit trouver un appariement entre eux qui rende compte au mieux de leurs similitudes et de leurs différences). L'idée sous-jacente de ces travaux est que le caractère approximatif de la recherche est dû à l'*erreur* dont serait entaché l'un des graphes à comparer : on parle d'isomorphisme *tolérant aux erreurs*, voire même *par correction d'erreurs*. L'idée de ces approches consiste à quantifier les modifications (ou corrections) à apporter à un graphe pour qu'il devienne isomorphe à l'autre [**?**]. La ressemblance entre deux graphes est alors inversement proportionnelle à leur *distance* en terme de modifications.

Tous ces problèmes sont extrêmement complexes : la recherche d'isomorphisme exact est déjà complexe[1], et l'aspect approximatif la complexifie encore. Cette complexité peut être réduite en contraignant le problème (limitation à des arbres pour **???**). Elle peut aussi être contournée en appliquant des algorithmes incomplets, généralement à base de recherche locale, comme **?**. Cette complexité est d'autant plus critique lorsque, comme en RàPC, un graphe doit être comparé à de nombreux autres pour déterminer le plus similaire. Dans ce cas, des algorithmes efficaces servent généralement à filtrer rapidement une grande partie de la base, pour n'appliquer la comparaison qu'à un nombre restreint de graphes. Ce filtrage peut s'effectuer sur les graphes eux-mêmes [**?**], sur des sous-graphes représentatifs [**?**], voire même sur des vecteurs résumant quelques caractéristiques des graphes [**?**].

Dans notre situation, les approches par subsomption sont trop contraignantes : la hiérarchie de types telle que CATIA la fournit ne rend pas suffisamment compte de la sémantique que les concepteurs peuvent prêter aux documents de conception. Par ailleurs ces approches s'intéressent en priorité aux types des sommets, et le plus souvent les types d'arcs (les attributs dans le modèle objet) sont jugés incomparables s'ils ne sont pas identiques. La notion de différence introduite précédemment est assez proche des méthodes de similarité par «correction d'erreurs» (même si, en RàPC, il n'est pas approprié de considérer que les différences entre cas soient des *erreurs*). Hélas, ces méthodes n'admettent pas les fusions/éclatements de sommets (un sommet apparié à plusieurs) alors que même un exemple simple comme celui présenté en début de chapitre requiert un appariement multiple. Les fusions/éclatements sont possibles dans les approches par subsomption, que nous avons déjà écartées, et dans celle de **?**, qui n'exploite absolument pas la notion de différence, ou celle de **?** qui ne s'intéresse qu'aux arbres. Nous avons donc proposé un algorithme fondé sur la notion de différence, et englobant les approches présentées ici dans la mesure où il est plus générique.

À l'instar de travaux précédemment cités (notamment **?**Bouchon-Meunier *et al.* 1996), ce travail s'inspire des résultats en psychologie de Tversky [1977]. Ce dernier s'est attaché à donner un modèle formel générique et cognitivement plausible de la similarité. Il se trouve que la notion de descripteur de graphe définie au chapitre précédent, permet d'appliquer assez simplement les principes développés par Tversky. Pour ce dernier, si deux objets a et b sont représentés par des ensembles de caractéristiques A et B, une mesure de similarité entre a et b s'exprime comme :

$$sim(a,b) = \frac{f(A \cap B)}{f(A \cup B) - \alpha f(A - B) - \beta f(B - A)}$$

La fonction f est une fonction monotone croissante à valeurs positives, c'est-à-dire que

[1] Bien que sa complexité exacte n'aie pas encore pu être déterminée, on s'accorde à penser qu'il est non polynomial et dans NP.

$\forall A, B,\ f(A \cup B) \geq f(A) \geq 0$. Les paramètres α et β sont des valeurs réelles positives. Un choix classique pour la fonction f est de considérer le cardinal des ensembles. Le couple (α, β) prend généralement les valeurs $(0,0)$ pour une mesure de similarité symétrique (ressemblance de a et b) ou $(0,1)$ pour une mesure asymétrique (compatibilité de b avec a). Le résultat de la fonction *sim* est compris entre 0 (absolument dissimilaire) et 1 (parfaitement similaire).

Cette notion d'ensemble de caractéristiques évoque celle présentée précédemment, notamment les descripteurs $d(\mathcal{G})$ des graphes. Cette analogie pose cependant un problème supplémentaire avec les graphes : pour Tversky, les ensembles A et B sont comparables, des objets a et b similaires possédant les *mêmes* caractéristiques. Il en va de même pour des graphes représentant des états de la même trace. En revanche, avec deux graphes quelconques, les caractéristiques sont attachées à des sommets ou des arcs de chacun des graphes, et ne sont donc comparables qu'à l'aide d'une mise en correspondance de ces éléments, comme suggéré en début de chapitre. Nous verrons toutefois qu'une fois formalisé le problème, le modèle de Tversky s'adapte de façon assez naturelle au cas des graphes.

5.1.2 Définition du problème

Avant d'introduire formellement la notion d'appariement, revenons sur la notion de graphe multi-étiqueté introduite au chapitre précédent (page 48). Nous faisons l'hypothèse que toutes les caractéristiques de sommets et d'arcs sont définies une fois pour toutes par le modèle d'utilisation. On peut donc considérer que les ensembles d'étiquettes Φ_V et Φ_A sont donnés indépendamment des graphes à comparer. Nous faisons également l'hypothèse que les ensembles de sommets des deux graphes à comparer (et par conséquent leurs ensembles d'arcs) sont disjoints. La donnée du problème est donc le quadruplet :

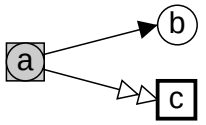
$$\langle \Phi_V, \Phi_A, \mathcal{G}_1, \mathcal{G}_2 \rangle$$
$$\text{tel que } \mathcal{G}_1 = \langle V_1, A_1, \lambda_{V1}, \lambda_{A1} \rangle, \quad \mathcal{G}_2 = \langle V_2, A_2, \lambda_{V2}, \lambda_{A2} \rangle, \quad V_1 \cap V_2 = \emptyset$$

Dans le cas où $\mathcal{G}_1$ et $\mathcal{G}_2$ ont des sommets ayant même URI, l'hypothèse que V_1 et V_2 sont disjoints peut sembler contradictoire avec ce qui a été présenté au chapitre précédent à propos de la fonction δ. Il est important de remarquer que cette dernière visait à comparer des graphes dont on *sait* qu'ils ont des éléments communs. À l'inverse, on cherche ici à déterminer *quels* éléments sont communs afin de pouvoir comparer les graphes. Or les éléments les plus ressemblants peuvent ne pas être ceux ayant le même URI. On reviendra sur les liens entre la fonction δ et la mesure de similarité en section 5.1.3.

Pour les exemples de ce chapitre, on préférera aux modèles d'utilisations liés à la conception un ensemble de caractéristiques géométriques, représentées directement sur les figures afin de simplifier leur lecture. Pour les sommets, on utilisera les caractéristiques *cercle*, *carré*, *gris*, *épais*. Pour les arcs, on utilisera les caractéristiques (représentées par la tête de la flèche) *noire*, *blanche*, *double*. Ces caractéristiques peuvent être combinées dans une certaine mesure : *cercle+carré+gris* ou *blanche+double* par exemple (cf. figure 5.2).

Appariement

Un appariement entre les deux graphes $\mathcal{G}_1$ et $\mathcal{G}_2$ est une *relation* $m \subseteq V_1 \times V_2$. L'appariement m associe donc à chaque sommet de $\mathcal{G}_1$ zéro, un ou plusieurs sommet(s) de $\mathcal{G}_2$, et réciproquement. On l'emploiera donc également sous forme fonctionnelle pour désigner l'ensemble des sommets appariés par m à un sommet donné.



$$d(\mathcal{G}) = \{ \quad (a, \textit{cercle}), (a, \textit{carré}), (a, \textit{gris}), \\ (b, \textit{cercle}), (c, \textit{carré}), (c, \textit{épais}), \\ ((a,b), \textit{noire}), ((a,c), \textit{blanche}), \\ ((a,c), \textit{double}) \quad \}$$

FIG. 5.2 – Représentation des caractéristiques géométriques

$$\forall v_1 \in V_1, \quad m(v_1) \doteq \{v_2 \in V_2 |\ (v_1, v_2) \in m\}$$
$$\forall v_2 \in V_2, \quad m(v_2) \doteq \{v_1 \in V_1 |\ (v_1, v_2) \in m\}$$

Si un appariement m fait correspondre les sommets des graphes, on peut aussi considérer qu'il fait correspondre leurs arcs, en définissant intuitivement qu'un arc a_1 est apparié à un arc a_2 si les sommets sources de a_1 et a_2 sont appariés, et si leurs sommets destinations sont également appariés. On utilise là encore la notation fonctionnelle de m.

$$\forall a_1 = (v_1, v_1') \in A_1, \quad m(a_1) \doteq \{(v_2, v_2') \in A_2 |\ v_2 \in m(v_1) \wedge v_2' \in m(v_1')\}$$
$$\forall a_2 = (v_2, v_2') \in A_2, \quad m(a_2) \doteq \{(v_1, v_1') \in A_1 |\ v_1 \in m(v_2) \wedge v_1' \in m(v_2')\}$$

Par la suite, ces notations fonctionnelles seront employées pour simplifier l'écriture. Il convient cependant de garder à l'esprit que l'appariement est d'abord défini comme une relation, donc comme un ensemble de couples de sommets. De ce fait, on peut appliquer aux appariements les opérateurs ensemblistes, et notamment l'inclusion $\subseteq$.

Graphiquement, on représentera l'appariement entre deux graphes par des traits pointillés reliant les sommets appariés. (cf. par exemple la figure 5.3)

Similarité relative à un appariement

La qualité d'un appariement m découle de la similarité entre les deux graphes du point de vue de cet appariement (la similarité «absolue» entre les deux graphes étant celle de leur meilleur appariement). Tversky suggère que le calcul de similarité s'appuie sur l'ensemble des caractéristiques possédées par l'un ou l'autre des objets à comparer ($A \cup B$), et sur l'ensemble des caractéristiques communes aux deux objets ($A \cap B$) (on considérera pour l'instant des valeurs nulles pour les paramètres α et β). Puisque nous assimilons les ensembles A et B aux descripteurs $d(\mathcal{G}_1)$ et $d(\mathcal{G}_2)$, l'union ensembliste des deux descripteurs correspond naturellement à $A \cup B$. En revanche, les caractéristiques communes dépendent de l'appariement. Intuitivement, une caractéristique ϕ, attachée à un sommet ou un arc x de $\mathcal{G}_1$, est commune aux deux graphes par rapport à m si m associe à x un élément y de $\mathcal{G}_2$ la possédant également. Plus formellement, on définit la fonction de *ressemblance* $ress_m$ comme suit[2] :

$$\begin{aligned} ress_m(\mathcal{G}_1, \mathcal{G}_2) \quad \doteq \quad & \{(v, \phi) \in d_V(\mathcal{G}_1) |\ \exists v' \in m(v),\ (v', \phi) \in d_V(\mathcal{G}_2)\} \\ \cup \quad & \{(a, \phi) \in d_A(\mathcal{G}_1) |\ \exists a' \in m(a),\ (a', \phi) \in d_A(\mathcal{G}_2)\} \\ \cup \quad & \{(v, \phi) \in d_V(\mathcal{G}_2) |\ \exists v' \in m(v),\ (v', \phi) \in d_V(\mathcal{G}_1)\} \\ \cup \quad & \{(a, \phi) \in d_A(\mathcal{G}_2) |\ \exists a' \in m(a),\ (a', \phi) \in d_A(\mathcal{G}_1)\} \end{aligned}$$

[2] Cette définition *existentielle* des ressemblances admet une définition duale, dite *universelle*. Elle sera discutée en section A.3.1.

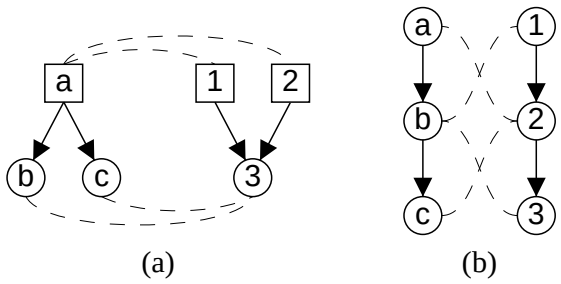


FIG. 5.3 – Deux appariements «parfaits» du point de vue de $sim1$

L'ensemble $ress_m(\mathcal{G}_1, \mathcal{G}_2)$ correspond bien à l'ensemble des caractéristiques de $\mathcal{G}_1$ et $\mathcal{G}_2$ qui sont conservées par l'appariement m. Elle joue donc bien le même rôle que $A \cap B$ dans le modèle de Tversky. On peut donc proposer comme première mesure de similarité :

$$sim1_m(\mathcal{G}_1, \mathcal{G}_2) = \frac{f(ress_m(\mathcal{G}_1, \mathcal{G}_2))}{f(d(\mathcal{G}_1) \cup d(\mathcal{G}_2))}$$

Cette fonction n'est cependant pas totalement satisfaisante. En effet, elle donne pour les deux appariements présentés sur la figure 5.3, une valeur de similarité de 1 puisque toutes les caractéristiques des graphes sont communes. En ce qui concerne l'appariement (a), les graphes ne sont pas isomorphes, ce qui rend ce résultat inattendu. Il n'est pas ici question d'affirmer que deux graphes non isomorphes ne peuvent pas être considérés comme parfaitement similaires : cela dépend évidemment du domaine d'application. On souhaite cependant que la mesure de similarité *puisse* rendre compte de la différence de ces deux graphes, ce qui n'est pas le cas ici. Si les graphes impliqués dans l'appariement (b) sont effectivement isomorphes, l'appariement lui même n'est pas celui qui semble le plus approprié pour justifier d'une similarité de 1. Ces résultats contre-intuitifs proviennent du fait que la fonction $sim1$ ne tient pas compte de l'appariement multiple de certains sommets. Or le fait que le rôle tenu par un sommet dans un graphe se trouve distribué sur plusieurs sommets de l'autre graphe contribue à faire considérer que ces graphes sont dissimilaires. Pour répondre à cette exigence supplémentaire, on définit l'ensemble (en fait un multi-ensemble[3]) des *éclatements* de sommets par la fonction suivante :

$$\begin{aligned} mult_m(\mathcal{G}_1, \mathcal{G}_2) &\doteq \{\langle v, n\rangle |\ v \in V_1,\ n = \max(|m(v)| - 1, 0)\} \\ &\sqcup \{\langle v, n\rangle |\ v \in V_2,\ n = \max(|m(v)| - 1, 0)\} \end{aligned}$$

Ce multi-ensemble contient tous les sommets appariés à plusieurs autres, et leur cardinalité dans le multi-ensemble donne le nombre de sommets auxquels ils sont appariés, au delà du premier — un sommet apparié à un seul autre ou aucun aura une cardinalité de zéro, ce qui revient à ne pas appartenir au multi-ensemble des éclatements. Comme il a été remarqué plus haut, il semble normal que ce multi-ensemble participe à faire décroître la similarité entre $\mathcal{G}_1$ et $\mathcal{G}_2$. Une mesure de similarité plus satisfaisante est donc :

$$sim_m(\mathcal{G}_1, \mathcal{G}_2) = \frac{f(ress_m(\mathcal{G}_1, \mathcal{G}_2)) - g(mult_m(\mathcal{G}_1, \mathcal{G}_2))}{f(d(\mathcal{G}_1) \cup d(\mathcal{G}_2))}$$

[3] Un multi-ensemble peut contenir le même élément plusieurs fois. On notera $\langle x, n\rangle$ un élément d'un multi-ensemble, où x est l'élément à proprement parler, et n est le nombre de fois où il apparaît dans le multi-ensemble. Pour plus de détails sur les notations pour les multi-ensembles, cf. annexe A.1.

avec une fonction g ayant les mêmes propriétés que f (mais sur les multi-ensembles). Cette fonction permet d'appliquer le modèle de Tversky aux graphes, au prix de quelques libertés prises avec ce modèle :

- La mesure ne dépend plus uniquement des objets à comparer, mais d'un appariement entre eux. La similarité entre deux graphes est celle du meilleur appariement. C'est la façon de palier l'absence de structure des objets dans le modèle original.
- Nos ensembles de caractéristiques semblent correspondre à ceux de Tversky. En revanche, nous introduisons l'ensemble des éclatements, rendu nécessaire, là encore, par le fait que les objets sont comparés par rapport à un appariement.

La similarité entre deux graphes $\mathcal{G}_1$ et $\mathcal{G}_2$ est la meilleure valeur possible de $sim_m(\mathcal{G}_1, \mathcal{G}_2)$. Calculer cette similarité revient donc à trouver l'appariement m entre $\mathcal{G}_1$ et $\mathcal{G}_2$ qui maximise $sim_m(\mathcal{G}_1, \mathcal{G}_2)$, ce qui revient à maximiser la fonction suivante :

$$score(m) \doteq f(ress_m(\mathcal{G}_1, \mathcal{G}_2)) - g(mult_m(\mathcal{G}_1, \mathcal{G}_2))$$

La fonction $score(m)$ correspond en effet au numérateur de l'expression de la fonction sim_m, car le dénominateur ne dépend pas de l'appariement. Il n'est là que pour normaliser la valeur de sim_m entre zéro et un[4].

La prise en compte de la fonction $mult_m$ permet bien de résoudre les problèmes posés par les appariements de la figure 5.3. L'appariement (a) comporte deux éclatements (sommets a et 3), ce qui peut faire diminuer sa valeur de similarité. L'appariement (b) contient également des éclatements (sommets b et 2), mais les deux graphes admettent un appariement meilleur : $\{(a,1),(b,2),(c,3)\}$ donne la même valeur à $ress_m$ (à savoir $d(\mathcal{G}_1) \cup d(\mathcal{G}_2)$), mais ne contient pas d'éclatements. On notera que la mesure sim_m *permet* de considérer que ces appariements ne sont pas parfaits, mais qu'elle permet également de considérer que les éclatements qu'ils impliquent sont acceptables et que la similarité entre les graphes vaut effectivement un. Tout dépend du choix des fonctions f et g.

Connaissances de similarité

La définition de la fonction $sim_m(\mathcal{G}_1, \mathcal{G}_2)$ fait appel aux deux fonctions de mesure d'ensembles f et g. Ce sont elles qui permettent de capturer les connaissances de similarité spécifiques au domaine. Le modèle de Tversky impose qu'elles soient monotones par rapport à l'inclusion. Pour faciliter la modélisation des connaissances de similarité, nous imposons les contraintes supplémentaires suivantes :

$$f(\emptyset) = 0,\ f(A \cup B) = f(A) + f(B - A)$$

$$g(\emptyset) = 0,\ g(M \sqcup N) = g(M) + g(N)$$

ce qui revient à associer un *poids* $w(e)$ positif ou nul, à chaque élément e des ensembles mesurés, puis à faire la somme des poids de tous les éléments :

[4] Il est évidemment possible de construire un appariement de score négatif (entraînant peu de ressemblances et beaucoup d'éclatements), donnant donc à sim_m une valeur inférieure à zéro. Cependant, ce type d'appariement n'est pas rationnel, puisqu'on peut l'améliorer en lui retirant des couples — dans tous les cas, l'appariement vide a un score de zéro. On peut donc garantir que la similarité entre deux graphes (celle du meilleur appariement) sera toujours comprise entre zéro et un.

$$f(ress_m(\mathcal{G}_1, \mathcal{G}_2)) = \sum_{(x,\phi)} w_f(x, \phi)$$

$$g(mult_m(\mathcal{G}_1, \mathcal{G}_2)) = \sum_{\langle v,n \rangle} n \times w_g(v)$$

En l'absence de connaissances particulières, on pourra utiliser simplement une fonction affectant la valeur 1 à tout élément, ce qui revient à valuer les ensembles *ress* et *mult* par leur cardinal. Des connaissances générales portant sur les caractéristiques de sommets et d'arcs peuvent facilement être introduites, par exemple sous la forme d'une fonction h :

$$h : \Phi_V \cup \Phi_A \longrightarrow \mathbb{R}^+$$

$$w_f(x, \phi) = h(\phi)$$

La définition de la fonction w_f permet également de capturer des connaissances contextuelles, c'est-à-dire propres à un graphe donné, puisque sa valeur dépend d'une caractéristique mais aussi du sommet (ou de l'arc) précis qui la possède. De telles connaissances peuvent être déduites à l'aide de connaissances plus précises sur le domaine, mais également inscrites directement dans le graphe sous forme d'annotations (de type «sur ce sommet, cette caractéristique est facultative» ou «cette relation est primordiale»). D'autres variantes pour les fonctions w_f et w_g seront discutées dans la section 5.1.4.

Rappelons que les connaissances d'adaptation sont duales des connaissances de similarité. Ainsi, les fonctions w_f et w_g ne rendent-elles pas seulement compte de l'importance des caractéristiques dans le graphe, mais également de la difficulté à adapter un épisode lorsque cette caractéristique est manquante. Par exemple, la distinction suggérée précédemment entre types et autres caractéristiques peut se répercuter par des poids plus forts sur les types : un assemblage sélectionné est plus similaire à un assemblage non sélectionné qu'à une contrainte sélectionnée, car il est plus facile de changer l'état de sélection d'un assemblage que de changer cet assemblage en une contrainte.

5.1.3 Algorithme

Ici sont présentés deux algorithmes pour trouver le meilleur appariement entre deux graphes. Ils ont été conçus et implantés avec l'aide de Sébastien Sorlin lors de son stage de maîtrise. Ce travail est présenté en détail par **?**.

Ressemblances et différences

Historiquement, l'algorithme n'a pas été développé en utilisant la fonction de ressemblance *ress* proposée précédemment, mais une fonction de différence *diff*, qui est sa complémentaire par rapport à l'ensemble de toutes les caractéristiques $d(\mathcal{G}_1) \cup d(\mathcal{G}_2)$.

$$diff_m(\mathcal{G}_1, \mathcal{G}_2) \doteq \Delta_m(\mathcal{G}_1, \mathcal{G}_2) \cup \Delta_m(\mathcal{G}_2, \mathcal{G}_1)$$

avec

$$\begin{aligned} \Delta_m(\mathcal{G}_i, \mathcal{G}_j) &\doteq \{(v, \phi) \in d_V(\mathcal{G}_i) |\ \forall v' \in m(v),\ (v', \phi) \notin d_V(\mathcal{G}_j)\} \\ &\cup \{(a, \phi) \in d_A(\mathcal{G}_i) |\ \forall a' \in m(a),\ (a', \phi) \notin d_A(\mathcal{G}_j)\} \end{aligned}$$

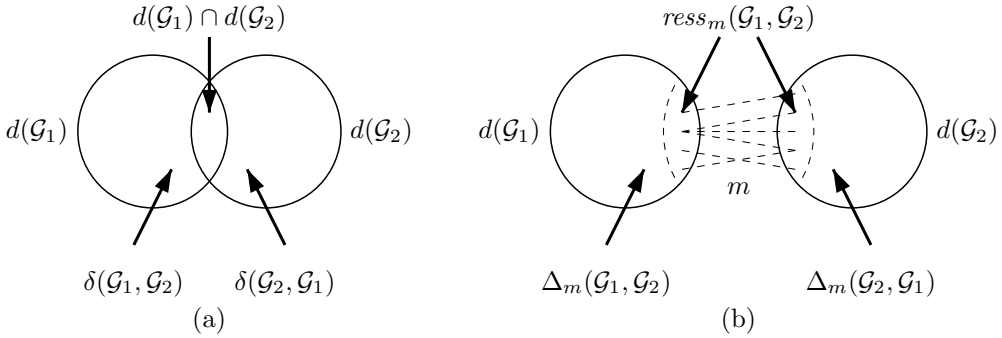


FIG. 5.4 – Différences entre graphes : δ et Δ_m

La fonction $\Delta_m(\mathcal{G}_i, \mathcal{G}_j)$ représente les caractéristiques de $d(\mathcal{G}_i)$ n'ayant pas de correspondance par m dans $d(\mathcal{G}_j)$. Elle étend donc la fonction δ du chapitre précédent (page 49) au cas où les graphes sont comparés sur la base d'un appariement m plutôt que sur la base de leurs sommets communs (ce qu'illustre la figure 5.4). La fonction $\mathit{diff}_m(\mathcal{G}_1, \mathcal{G}_2)$ représente l'ensemble des caractéristiques de l'un ou l'autre graphe n'ayant pas de correspondance par m ; c'est donc bien l'ensemble des différences entre les deux graphes.

De la complémentarité des ensembles $\mathit{ress}_m(\mathcal{G}_1, \mathcal{G}_2)$ et $\mathit{diff}_m(\mathcal{G}_1, \mathcal{G}_2)$ il découle que maximiser le score $\mathit{score}(m)$ d'un appariement équivaut à minimiser son coût défini comme :

$$\mathit{coût}(m) \doteq \mathit{diff}_m(\mathcal{G}_1, \mathcal{G}_2) + \mathit{mult}_m(\mathcal{G}_1, \mathcal{G}_2)$$

Espace de recherche

L'espace des solutions à explorer est l'ensemble de tous les appariements possibles, structuré en treillis par la relation d'inclusion entre appariements. On recherche dans cet espace un appariement minimisant la fonction *coût*. Cette dernière n'est en général pas monotone : le coût d'un appariement peut aussi bien augmenter ou diminuer lorsqu'on lui ajoute un couple. Cependant on peut démontrer que les fonctions *diff* et *mult* ont, elles, un comportement monotone par rapport à l'inclusion. Plus précisément :

$$\begin{aligned} \forall m_1, m_2 \subseteq V_1 \times V_2 \quad m_1 \subseteq m_2 \quad &\Rightarrow \quad \mathit{diff}_{m_1}(\mathcal{G}_1, \mathcal{G}_2) \supseteq \mathit{diff}_{m_2}(\mathcal{G}_1, \mathcal{G}_2) \\ &\wedge \quad \mathit{mult}_{m_1}(\mathcal{G}_1, \mathcal{G}_2) \sqsubseteq \mathit{mult}_{m_2}(\mathcal{G}_1, \mathcal{G}_2) \end{aligned}$$

d'où il découle, d'après la monotonie de f et g :

$$\begin{aligned} \forall m_1, m_2 \subseteq V_1 \times V_2 \quad m_1 \subseteq m_2 \quad &\Rightarrow \quad f(\mathit{diff}_{m_1}(\mathcal{G}_1, \mathcal{G}_2)) \geq f(\mathit{diff}_{m_2}(\mathcal{G}_1, \mathcal{G}_2)) \\ &\wedge \quad g(\mathit{mult}_{m_1}(\mathcal{G}_1, \mathcal{G}_2)) \leq g(\mathit{mult}_{m_2}(\mathcal{G}_1, \mathcal{G}_2)) \end{aligned}$$

En d'autres termes, les poids (au sens de f et g) des ensembles diff_m et mult_m sont monotones, respectivement non croissant et non décroissant. Or *coût* est la somme des poids de ces deux ensembles. La valeur de *coût* est donc composée d'une partie *récupérable*, due à *diff*, qui ne peut aller qu'en diminuant lorsqu'on ajoute des couples à l'appariement, et d'une partie *irrécupérable*, due à *mult*, qui ne peut aller qu'en augmentant. On exploitera cette propriété pour réduire l'espace de recherche : lorsque la partie irrécupérable du coût d'un appariement dépasse le coût du meilleur appariement connu, il est inutile d'aller plus loin, puisque même en réduisant la partie récupérable à zéro, on ne pourra obtenir mieux. On peut donc éliminer de la recherche tous les successeurs dans le treillis, de l'appariement courant.

Algorithme glouton

Ce premier algorithme décrit sur la figure 5.5 effectue une recherche gloutonne dans l'espace des appariements, en partant d'un appariement vide, et en suivant le plus fort gradient de coût décroissant, sans retour en arrière : à chaque itération, il choisit un couple qui améliore l'appariement (qui fasse diminuer son coût), et tel qu'aucun autre couple ne l'améliore plus. Si plusieurs couples sont *ex-aequo* selon ces critères, une heuristique est appliquée pour en choisit un. Si aucun couple ne satisfait ces critères l'algorithme s'arrête.

Cet algorithme est relativement efficace (complexité polynomiale en fonction de la taille des ensembles V_i) mais ne garantit bien sûr pas l'optimalité. L'absence de retour arrière l'empêche notamment de remettre en cause un couple semblant localement prometteur mais ne faisant pas partie d'une solution optimale. L'heuristique appliquée au choix d'un couple parmi les *ex-aequo* tente d'anticiper l'intérêt d'un couple en tenant compte, non seulement de la variation de coût qu'il entraîne, mais également des arcs entrants et sortants des sommets. Par exemple, considérons le problème de la figure 5.3-b : le premier couple, si on ne considère que la variation de coût, pourrait être n'importe lequel, car tous les sommets sont identiques. C'est cependant le couple $(b, 2)$ qui sera choisi, car ses deux sommets ont des arcs entrants et sortants en même quantité et de même type, et ce sont eux qui en ont le plus. Ce couple est donc *susceptible* de faire disparaître les différences concernant ces arcs, puisque la moitié du travail est déjà faite. Il sera en conséquence préféré à d'autres couples entraînant la même variation *effective* de coût. Cette heuristique donne d'assez bons résultats : dans nombre des cas qui ont servi de tests, l'algorithme glouton avec cette heuristique fournit une solution optimale ou proche.

Algorithme complet

Bien que la recherche d'un meilleur appariement soit un problème très complexe ($2^{|V_1|\times|V_2|}$ possibilités à considérer), il est utile d'expérimenter une approche complète, ne serait-ce que pour déterminer précisément les limites d'applicabilité de cette approche, et obtenir une référence absolue pour pouvoir ensuite évaluer des approches incomplètes.

L'algorithme complet explore le treillis en parcourant un arbre particulier, représenté sur la figure 5.6, et que nous appelons «arbre fractal». Sa construction peut en effet se décrire ainsi : en chaque fils de la racine est enracinée une copie du sous-arbre composé de tous les nœuds situés à sa gauche[5]. Ce processus de construction est représenté sur la figure 5.6-a : les traits pleins représentent la relation de parenté dans l'arbre, et les traits pointillés la relation de recopie. Une autre façon de construire cet arbre est de recopier les frères gauches de chaque nœud pour construire ses fils. De la même façon, ce procédé de construction est illustré sur la figure 5.6-b. Les copies d'un nœud sont appelés ses *clones*.

Les liens de parenté dans l'arbre et les liens de recopie correspondent tous à des liens de succession dans le treillis. Lorsqu'un nœud peut être coupé (c'est-à-dire, lorsque la partie irrécupérable de son coût dépasse le meilleur coût connu), non seulement ses fils peuvent être coupés grâce à la monotonie de *mult*, mais également ses clones. Il paraît donc pertinent de garder trace des liens de recopie, et ce pour les deux modes de copie, puisqu'ils fournissent des relations différentes (cf figure 5.6-c). Cette solution a pour avantage de permettre l'*héritage* des coupes d'un nœud à l'autre, sans pour autant maintenir *toutes* les relations de succession du treillis, qui peuvent être très nombreuses (de l'ordre de $|V_1| \times |V_2|$ à mi-hauteur du treillis). Elle n'en conserve au maximum que trois (parenté, recopie de sous-arbres, recopie de voisins), avec

[5] En d'autres terms : les nœuds rencontrées avant lui dans un parcours en profondeur ordonné de gauche à droite.

Fonction RechercheGloutonne
  entrée : deux graphes $\mathcal{G}_1$, $\mathcal{G}_2$
  sortie : un appariement $m \subseteq V_1 \times V_2$
début
$m \leftarrow \emptyset$
$candidats \leftarrow V_1 \times V_2$
$couples \leftarrow \{c_i \in candidats |$
  $co\hat{u}t(\{c_i\}) < co\hat{u}t(\emptyset) \wedge$
  $\forall c_j \in candidats,\ co\hat{u}t(\{c_i\}) \leq co\hat{u}t(\{c_j\})\ \}$
tant que $couples \neq \emptyset$
  $c \leftarrow$ ChoisirCouple($couples$)
  $m \leftarrow m \cup \{c\}$
  $candidats \leftarrow candidats - \{c\}$
  $couples \leftarrow \{c_i \in candidats |$
    $co\hat{u}t(m \cup \{c_i\}) < co\hat{u}t(m) \wedge$
    $\forall c_j \in candidats,\ co\hat{u}t(m \cup \{c_i\}) \leq co\hat{u}t(m \cup \{c_j\})\ \}$
fin-tant que
retourne m
fin

Fonction ChoisirCouple
  entrée : un ensemble de couples *ex-aequo couples*
  sortie : un couple c
début
retourne le couple c dont les sommets ont le plus d'arcs en commun
fin

Fig. 5.5 – Algorithme glouton de recherche d'un meilleur appariement

la contrepartie de pouvoir manquer certaines coupes. Par exemple, sur la figure 5.6-c, le sommet gris foncé est un prédécesseur du sommet gris clair, mais si le premier est coupé, le second sera exploré malgré tout, faute de relation entre eux explicitée dans l'arbre fractal.

Notons qu'on pourrait moduler ce compromis en explicitant plus de trois liens de succession. En effet, le parcours de l'arbre fractal garantit qu'un appariement n'est exploré qu'après que tous ses sous-appariements (au sens de l'inclusion) ont déjà été explorés. En ce sens, on peut le qualifier de parcours en profondeur «prudent» du treillis. Ainsi on peut, sans changer l'ordre du parcours, consulter autant de prédécesseurs d'un nœud qu'on le souhaite, éventuellement tous, afin de détecter d'éventuelles coupes.

Un autre point critique de cet algorithme est l'ordre dans lequel les nœuds de l'arbre sont explorés. Pour cela, nous utilisons l'algorithme glouton présenté ci-dessus. Celui-ci ordonne les couples (puisqu'il construit un appariement en choisissant toujours le couple le plus prometteur), et nous gardons cet ordre pour la recherche complète. Par ailleurs, il fournit une première solution dont le coût permet de couper plus rapidement certains nœuds.

Résultats expérimentaux

Les résultats expérimentaux de l'algorithme complet plaident définitivement pour une approche incomplète. L'approche complète n'est utilisable que sur de très petits graphes (sept sommets au maximum), au delà de quoi elle sature la mémoire (testé avec 384 méga-octets) et prendrait de toute façon un temps prohibitif. Une telle approche ne paraît donc envisageable que dans un domaine qui permettrait de développer des heuristiques *ad hoc*. On notera que si ces heuristiques peuvent s'exprimer à l'aide des fonctions w_f et w_g, elles sont applicables sans changer les fondements de l'algorithme. Nous envisageons donc d'étudier des méthodes de recherche locale, comme la méthode TABOU, employée avec succès sur ce type de problème par **?**.

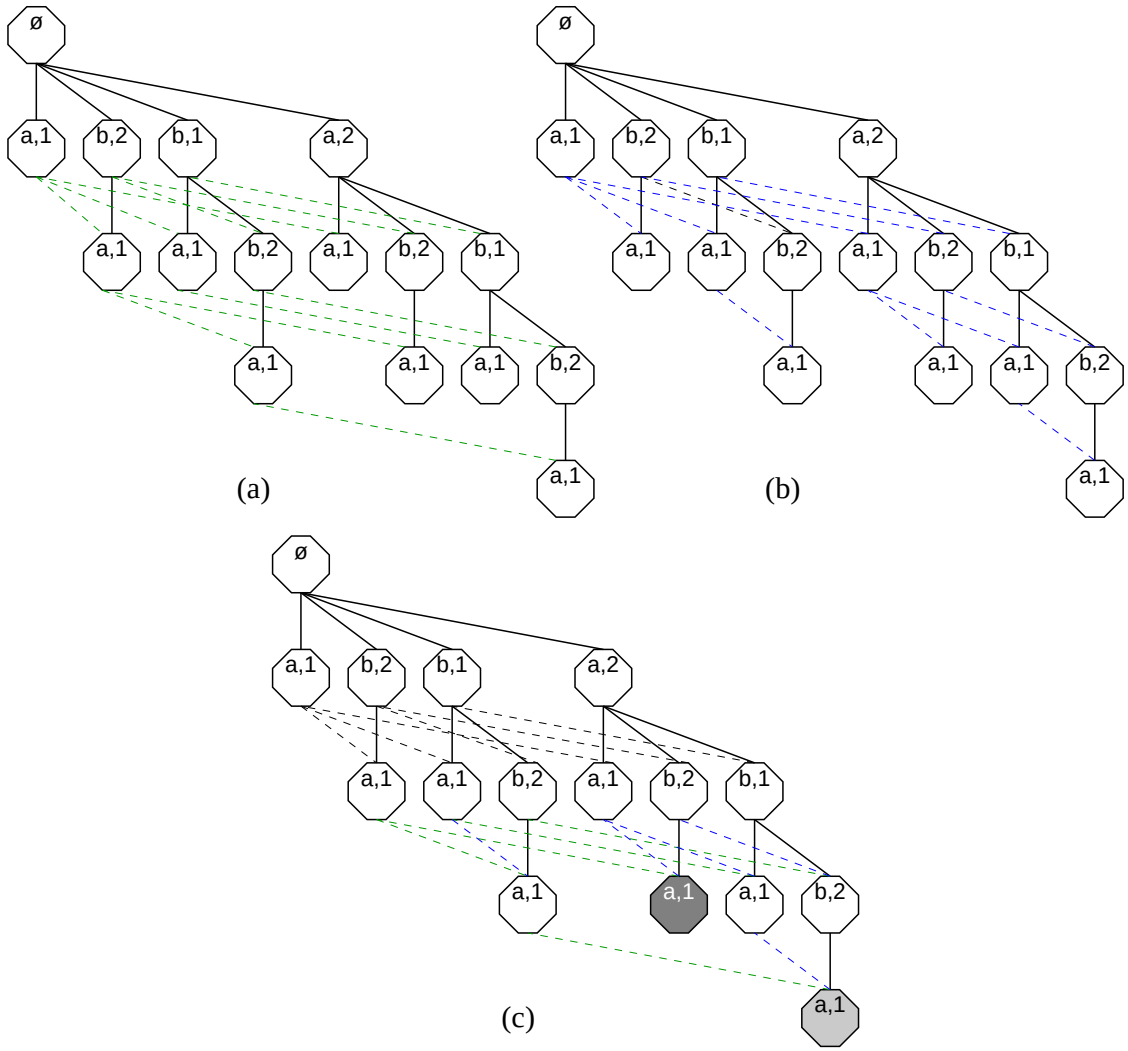
5.1.4 Discussion et comparaison à d'autres approches

Un modèle général

Le modèle proposé ici est extrêmement général (ce qui explique d'ailleurs les faibles performances de l'algorithme complet et le recours nécessaire aux méthodes incomplètes). Bien que nous ayons jusqu'à maintenant négligé les paramètres α et β du modèle de Tversky, leur rôle peut en fait être joué par les fonctions w_f et w_g, en altérant les poids des caractéristiques d'un graphe relativement à l'autre. Les fonctions proposées précédemment ne tiennent pas compte du graphe, elles sont donc symétriques ($\alpha = \beta = 0$). Au contraire, les fonctions $w\text{-}asym_f$ et $w\text{-}asym_g$ définies ci-après (à l'aide de la fonction h de la page 60) ne comptabilisent que les caractéristiques de l'un des graphes, et correspondent donc à une mesure asymétrique de type «mesure de compatibilité» ($\alpha = 1$, $\beta = 0$).

$$w\text{-}asym_f(x,\phi) = \begin{cases} h(\phi) & \text{si } (x,\phi) \in d(\mathcal{G}_1) \\ 0 & \text{si } (x,\phi) \in d(\mathcal{G}_2) \end{cases} \qquad w\text{-}asym_g(x) = \begin{cases} 1 & \text{si } (x,\phi) \in V_1 \\ 0 & \text{si } (x,\phi) \in V_2 \end{cases}$$

D'une façon similaire, on peut ramener à notre modèle la plupart des méthodes de comparaison de graphes qui ont été proposées dans la littérature et présentées en section 5.1.1. La recherche d'isomorphisme de graphe revient à affecter un poids infini à tous les éclatements, et à ne s'intéresser en outre qu'aux solutions de coût nul. La recherche d'isomorphisme de sous-graphe est l'équivalent asymétrique de la précédente. Dans les approches par subsomption, les



Chaque nœud de l'arbre fractal est étiqueté par un couple ; l'appariement qu'il représente est celui représenté par son père, auquel on ajoute le couple qui lui correspond (la racine de l'arbre correspondant à l'appariement vide). On peut donc l'obtenir en remontant jusqu'à la racine. Par exemple, le nœud gris foncé représente l'appariement $\{(a,1),(b,2),(a,2)\}$; le nœud gris clair, l'appariement $\{(a,1),(b,2),(b,1),(a,2)\}$.

FIG. 5.6 – Arbre fractal : construction par copie d'arbre (a), par copie de voisins (b), mixte (c)

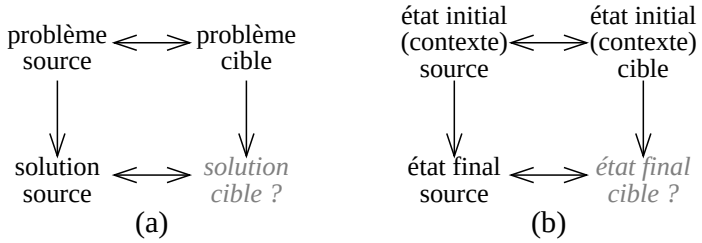


FIG. 5.7 – Le carré d'analogie (a) [??] adapté aux épisodes (b)

différences concernant les sommets, et les éclatements du graphe spécifique sont acceptés (poids nul) tandis que ceux du graphe général, ainsi que toutes les différences concernant les arcs, ont un poids infini. Contrairement aux recherches d'isomorphisme, les sommets ont dans ce dernier cas plusieurs caractéristiques (représentant tous les types de la hiérarchie auxquels ils appartiennent), et seules certaines de ces caractéristiques peuvent trouver une correspondance dans l'autre graphe (ce qui correspond à apparier un sommet de type spécifique avec un sommet de type plus général).

Appariement interactif

L'ambition de ce travail n'est pas de capturer une fois pour toutes les connaissances permettant de déterminer si un épisode de conception est réutilisable dans un contexte donné. Au contraire, le système n'est là que pour *suggérer* des choses à l'utilisateur, et doit pouvoir tirer profit de ses interactions avec lui afin d'améliorer ses connaissances du domaine en altérant les fonctions de pondération w_f et w_g.

La vocation interactive de cette mesure de similarité modère également les premiers résultats décevants de l'algorithme. En effet, il est assez facile de modifier l'algorithme complet pour le faire fonctionner de manière *anytime*, c'est-à-dire de façon à pouvoir être interrompu à tout moment et donner malgré tout une solution (la meilleure trouvée jusqu'alors). Le fait que cette solution ne soit pas «optimale» n'est pas particulièrement gênant dans l'optique de fonctions f et g ne représentant pas parfaitement le domaine. Au contraire, mieux vaudra sans doute interrompre régulièrement la recherche pour permettre au concepteur de la «recentrer» en précisant les poids w_f et w_g.

5.2 Le processus d'adaptation

En RàPC classique, la phase de réutilisation consiste à appliquer la solution du cas source au problème cible. Si l'on reprend le carré d'analogie de **?** cité par **?**, la situation courante tient lieu de problème cible, tandis que le contexte d'application de l'épisode réutilisable tient lieu de problème source. L'épisode réutilisable lui même constitue bien la «solution» du problème source, mais comme le contexte d'application comprend déjà l'état initial, on peut limiter son expression à l'état final de l'épisode. Réutiliser l'épisode dans la situation courante revient donc à construire un nouvel état qui résulterait de cet épisode, adapté à la situation courante (cf. figure 5.7). Le rôle de l'assistant n'est bien sûr pas d'adapter cet épisode de façon définitive, mais au moins d'en *proposer* une adaptation qui rende compte au mieux des ressemblances et différences mises en évidence par la mesure de similarité, entre la situation courante et son contexte d'application original.

5.2.1 Rôle des différences pour l'adaptation

Comme il a déjà été remarqué, les ensembles $\delta(\mathcal{G}_1, \mathcal{G}_2)$ et $\mathit{diff}_m(\mathcal{G}_1, \mathcal{G}_2)$ sont de même nature (ensembles de caractéristiques issues des $d(\mathcal{G}_i)$), et ont des interprétations semblables : ils contiennent tous deux les différences entre les deux graphes, c'est-à-dire les caractéristiques non partagées entre ces graphes. Rappelons que dans le cas des graphes comparés à l'aide d'un appariement s'ajoutent des différences d'éclatement représentées par l'ensemble $\mathit{mult}_m(\mathcal{G}_1, \mathcal{G}_2)$.

Les relations verticales de la figure 5.7-b correspondent aux différences de $\delta(\mathcal{G}_1, \mathcal{G}_2)$ et $\delta(\mathcal{G}_2, \mathcal{G}_1)$, tandis que les relations horizontales correspondent aux différences de $\mathit{diff}_m(\mathcal{G}_1, \mathcal{G}_2)$ et $\mathit{mult}_m(\mathcal{G}1, \mathcal{G}_2)$. Puisque l'état final cible est à construire, les ensembles de différences qui l'impliquent ne sont pas connus, mais doivent être analogues aux relations « parallèles » qui elles sont connues : l'épisode adapté est similaire à l'épisode source (relations verticales), et conserve les spécificités du cas cible par rapport au cas source (relations horizontales).

L'analogie entre les relations horizontales est obtenue en construisant l'état final cible à partir d'une copie de l'état initial cible, entretenant le même appariement avec l'état final source que les états initiaux entre eux. Rappelons que l'appariement choisi par la mesure de similarité découle directement de l'ensemble des différences entre les deux états initiaux. Ensuite, chacune des différences décrivant l'épisode source est interprétée comme un ajout ou une suppression dans le graphe initial, qui est appliquée au graphe en cours de construction chaque fois qu'elle garde un sens à travers l'appariement (on ne peut pas, par exemple, supprimer une caractéristique déjà manquante). L'algorithme est décrit plus en détail dans la section suivante.

5.2.2 Algorithme

L'algorithme général d'adaptation présenté sur la figure 5.8 prend en entrée trois graphes représentant les états initial et final de l'épisode source ($\mathcal{S}_i$ et $\mathcal{S}_f$), ainsi que l'état courant, qui sera l'état initial de l'épisode cible ($\mathcal{T}_i$), et qui est mis en correspondance avec $\mathcal{S}_i$ par un appariement m. Cet algorithme construit l'état final cible $\mathcal{T}_f$ issu de l'application à l'état courant de l'épisode source adapté ; il construit également un appariement m' entre $\mathcal{S}_f$ et $\mathcal{T}_f$. Initialement, $\mathcal{T}_f$ et m' sont des copies de $\mathcal{T}_i$ et m respectivement.

L'algorithme peut alors se décomposer en trois phases : marquage des différences, vérification de l'intégrité sémantique, application effective de l'épisode. Lors de la première phase, les différences supprimées ou ajoutées par l'épisode source sont marquées pour suppression ou création respectivement, dans le graphe $\mathcal{T}_f$. On remarque que le marquage pour création requiert un traitement particulier : si l'élément x auquel s'applique la caractéristique est un sommet de $\mathcal{S}_f$ (et non un arc), s'il a été créé par l'épisode (il n'appartient pas à $\mathcal{S}_i$), et si sa création n'a pas encore été répercutée dans $\mathcal{T}_f$ (m' ne l'apparie à aucune sommet), alors un sommet correspondant est ajouté à l'état final cible.

On remarquera qu'en revanche les sommets supprimés par l'épisode source ne le sont pas dans l'épisode cible. La suppression d'un sommet passe de toute façon par la suppression de tous ses arcs. Mieux vaut garder ce sommet isolé dans le graphe, et laisser à l'utilisateur le soin de le « sauver » en le reliant à nouveau au reste du graphe, ou de le supprimer pour de bon. On remarquera aussi qu'en cas d'appariement multiple, la création d'une caractéristique est distribuée à tous les éléments appariés, alors que seul l'un d'eux nécessite l'ajout de la caractéristique pour satisfaire les différences (cf. figure 5.9). Cependant, l'algorithme n'a pas les connaissances nécessaires pour sélectionner un des éléments. Plutôt que de forcer un choix non justifié, le système applique la solution redondante, quitte à ce que cette redondance soit supprimée ensuite par la vérification d'intégrité (voir ci-après) dans le cas particulier d'une caractéristique pour laquelle

Fonction AdapterÉpisode
  entrée : trois graphes $\mathcal{T}_i, \mathcal{S}_i, \mathcal{S}_f$
    un appariement m entre $\mathcal{T}_i$ et $\mathcal{S}_i$
  sortie : un graphe $\mathcal{T}_f$
début
$\mathcal{T}_f \leftarrow \mathcal{T}_i$
$m' \leftarrow m$
pour tout $(x, \phi) \in \delta(S_i, S_f)$
  pour tout $y \in m'(x)$
    si $(y, \phi) \in d(\mathcal{T}_f)$ **alors** marquer (y, ϕ) pour suppression
  fin-pour
fin-pour
pour tout $(x, \phi) \in \delta(S_f, S_i)$
  si $x \in V_{S_f} \wedge x \notin V_{S_i} \wedge m'(x) = \emptyset$
  alors
    $x' \leftarrow$ nouveau sommet de $\mathcal{T}_f$
    $m' \leftarrow m' \cup \{(x, x')\}$
  fin-si
  pour tout $y \in m'(x)$
    si $(y, \phi) \notin d(\mathcal{T}_f)$ **alors** marquer (y, ϕ) pour création
  fin-pour
fin-pour
VérifierIntégrité$(\mathcal{S}_i, \mathcal{S}_f, \mathcal{T}_i, \mathcal{T}_f, m')$
pour tout (y, ϕ) marqué pour suppression
  $d(\mathcal{T}_f) \leftarrow d(\mathcal{T}_f) - \{(y, \phi)\}$
fin-pour
pour tout (y, ϕ) marqué pour création
  $d(\mathcal{T}_f) \leftarrow d(\mathcal{T}_f) \cup \{(y, \phi)\}$
fin-pour
retourne $\mathcal{T}_f$ **fin**

Fig. 5.8 – Algorithme d'adaptation d'épisode

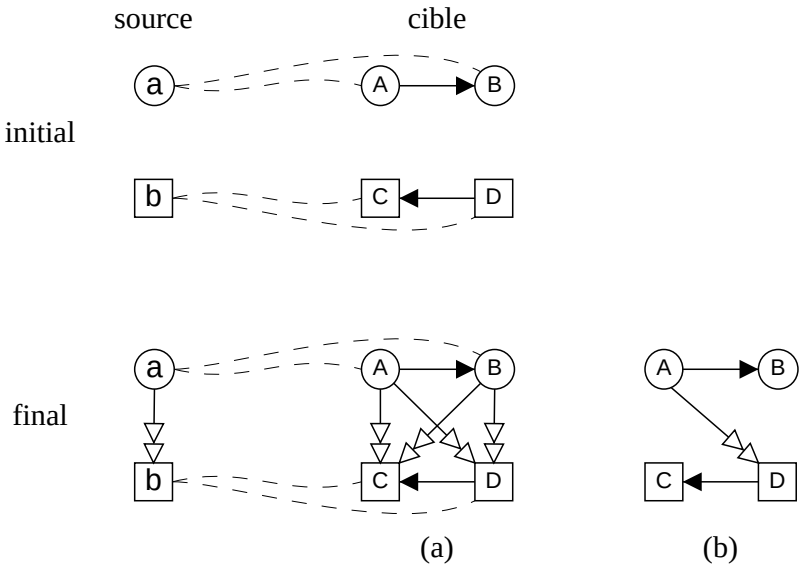


FIG. 5.9 – Création d'arcs redondants (a) alors que l'ajout d'un seul peut être suffisant (b)

des connaissances supplémentaires sont effectivement disponibles.

La deuxième phase de l'algorithme consiste à vérifier, étant données certaines connaissances particulières liées à certains types de caractéristique, que les suppressions et créations marquées sont bien cohérentes avec les règles du domaine d'application. Ces vérifications sont faites par la procédure VÉRIFIERINTÉGRITÉ qui doit pouvoir être surchargée afin d'y inclure les connaissances spécifiques à un modèle d'utilisation particulier. Au minimum, cette procédure vérifie l'intégrité liée à la description RDF-Schema du modèle d'utilisation (intégrité des domaines et des portées des relations), ainsi qu'à la sémantique particulière de la relation de composition (cf. figure 5.10), en s'assurant qu'un objet ne devienne pas un composé de plusieurs autres. Les ajouts d'une caractéristique d'arc ne respectant pas ses contraintes de domaine ou de portée son ignorés. Dans les cas où plusieurs arcs de composition sont marqués pour création en direction d'un sommet, et si un unique candidat parmi les candidats pères admet tous les autres pour ancêtres, alors c'est celui-là qui sera retenu (la relation de composition avec tous les autres sera vérifiée par transitivité). Dans le cas contraire, tous ces arcs de composition ne sont plus marqués pour création mais pour *suggestion*. Là encore, l'algorithme ne fait aucun choix arbitraire, mais évite ainsi de corrompre l'intégrité de la relation de composition dans l'état final cible.

Dans la dernière phase de l'algorithme d'adaptation, les caractéristiques marquées pour suppression sont effectivement détruites, celles marquées pour création sont effectivement créées.

5.2.3 Guider la similarité par l'adaptation

On a vu au chapitre 1 que les connaissances d'adaptation sont duales des connaissances de similarité, et que la similarité doit être guidée par l'adaptation (puisque son rôle est de permettre la remémoration d'un cas adaptable). L'algorithme général d'adaptation présenté ici suggère plusieurs façons de guider la mesure de similarité. Ce guidage est naturellement possible en influant sur les fonctions de pondérations w_f et w_g.

D'une part, on a déjà fait remarquer que l'ajout de caractéristiques sur un élément (sommet ou arc) apparié à plusieurs autres pose un problème, puisque cela requiert d'introduire des redondances ou de faire un choix sans forcément avoir les connaissances adéquates. Il est donc

Fonction VérifierIntégrité.DomainePortée
  entrée/sortie : quatre graphes $\mathcal{T}_i, \mathcal{T}_f, \mathcal{S}_i, \mathcal{S}_f$
  un appariement m entre $\mathcal{T}_i$ et $\mathcal{S}_i$
  un appariement m' entre $\mathcal{T}_f$ et $\mathcal{S}_f$
début
pour tout $((s,d),\phi)$ marqué pour création
  si $s \notin$ domaine de ϕ **ou** $d \notin$ portée de ϕ **alors**
    retirer de $((s,d),\phi)$ la marque pour création
  fin-si
fin-pour
fin

Fonction VérifierIntégrité.Composition
  entrée/sortie : quatre graphes $\mathcal{T}_i, \mathcal{T}_f, \mathcal{S}_i, \mathcal{S}_f$
  un appariement m entre $\mathcal{T}_i$ et $\mathcal{S}_i$
  un appariement m' entre $\mathcal{T}_f$ et $\mathcal{S}_f$
début
pour tout $s \in V_{\mathcal{T}_f}$
  $p \leftarrow$ père de s
  si $((p,s),$ ep:component$)$ est marqué pour suppression **alors**
    $candidats = \{q \mid ((q,s),$ ep:component$)$ marqué pour création$\}$
    si $|candidats| > 1$ **alors**
      si $\exists q \in candidats, \forall q' \in candidats$, q' est un ancêtre de q **alors**
        pour tout $q' \in candidats - \{q\}$
          retirer de $((q',s),$ ep:component$)$ la marque pour création
        fin-pour
      sinon
        pour tout $q \in candidats$
          retirer de $((q,s),$ ep:component$)$ la marque pour création
          marquer $((q,s),$ ep:component$)$ pour suggestion
        fin-pour
      fin-si
    fin-si
  fin-si
fin-pour
fin

Fig. 5.10 – Algorithmes de vérification d'intégrité

utile d'augmenter le poids pour les éclatements (w_g) des sommets de l'état initial source qui sont impliqués dans un ajout de caractéristique (sur le sommet lui même ou un de ses arcs). Ceci permet d'assurer la réutilisabilité des épisodes remémorés en évitant, dans une certaine mesure, à l'algorithme d'adaptation de se retrouver dans cette situation problématique.

D'autre part, les caractéristiques de l'état initial source qui sont destinées à être supprimées (dans l'épisode source) pourraient voir leur poids diminué, voire annulé. Ceci permettrait la remémoration d'un épisode dans un contexte courant où il a déjà été partiellement appliqué. Cependant, ceci n'est qu'une heuristique qu'il convient d'employer avec prudence : une caractéristique de l'état initial, même si elle doit être supprimée dans l'épisode correspondant, peut être extrêmement représentative de cet état initial, et donc rester nécessaire pour la remémoration pertinente de cet épisode.

Chapitre 6

Prototype

Ce chapitre présente le prototype mettant en œuvre les principes présentés jusqu'ici. La figure 6.1 représente l'architecture du système, avec deux variantes possibles (à gauche et à droite). La variante de gauche est celle qui a été implantée : l'utilisateur interagit avec l'application CATIA instrumentée pour produire une description en RDF de ses états, sous forme de fichiers RDF. L'assistance à la réutilisation doit s'effectuer dans une application auxiliaire baptisée *Moniteur* d'épisodes. Ce dernier capture les états (fichiers RDF) produits par l'application, et est capable d'en donner une représentation synthétique et de proposer une adaptation à la situation courante d'un épisode réutilisable.

Pour le stockage et la remémoration d'épisodes en revanche, le Moniteur s'appuie sur les services du *Gestionnaire* d'épisodes. C'est lui qui conserve toutes les traces d'utilisation, et peut comparer un état ou un épisode à tous ceux contenus dans cette base de traces, afin de déterminer quel épisode est réutilisable dans une situation donnée. Moniteur et Gestionnaire ont été développés dans un souci de généricité et de réutilisabilité : tous les aspects spécifiques à l'application CATIA résident dans l'instrumentation de cette dernière et dans le modèle d'utilisation utilisé.

La variante de droite consisterait à pousser plus loin l'instrumentation de l'application afin que l'aide à la réutilisation d'épisode y soit totalement *intégrée*. Elle s'appuierait également sur les services du Gestionnaire d'épisodes pour le stockage et la remémoration (ne serait-ce que pour permettre le partage de la base de traces entre plusieurs utilisateurs). Une telle variante présenterait des avantages évidents du point de vue ergonomique, mais demanderait un travail conséquent sur CATIA. Quant à la réutilisabilité du système, elle s'en trouverait nécessairement diminuée. C'est pourquoi nous avons opté pour la première variante, dont les trois composants sont décrits dans ce chapitre : l'instrumentation de CATIA (section 6.1), le Gestionnaire d'épisodes (section 6.2) et le Moniteur (section 6.3)

6.1 Instrumentation de CATIA

L'instrumentation de CATIA a consisté à rendre l'application capable d'exporter son état au format RDF, afin de pouvoir construire les traces nécessaires à la capture et à la réutilisation d'expérience. Une instrumentation complète aurait demandé que cet export se fasse de manière automatique en réaction aux indices comportementaux mis en évidence par les expérimentations des psychologues travaillant sur le projet ARDECO. Ce dernier point aurait cependant requis une connaissance technique pointue des couches d'interface de CATIA, alors que le centre d'intérêt

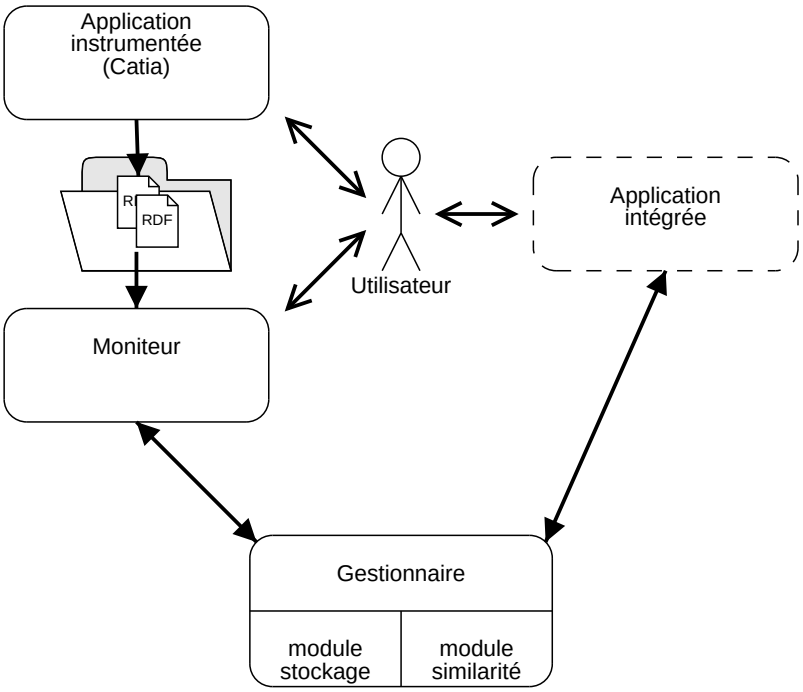


FIG. 6.1 – Architecture globale de la gestion d'épisodes

durant le projet était plutôt la représentation interne des documents de conception. La détection automatique n'a donc pas été implantée. L'export en RDF des états est pour l'instant le fait d'une opération explicite du concepteur, et se matérialise par un fichier RDF créé dans le même répertoire que le document CATIA correspondant.

Par ailleurs, on a vu que les états sont décrits à l'aide d'un modèle d'utilisation, et que différents MUs peuvent coexister pour décrire les états selon différents points de vue. On présente dans cette section l'infrastructure mise en place pour permettre le développement de MUs spécifiques au sein de CATIA, ainsi qu'un MU développé à l'aide de cette infrastructure afin de tester le système, et portant sur l'atelier d'assemblage.

6.1.1 Infrastructure de développement de MUs

Comme il a déjà été souligné, les objets (au sens de la programmation) dans CATIA offrent de multiples points de vue, qui permettent d'envisager autant de MUs différents. Le travail d'un développeur souhaitant implanter un MU consiste donc d'une part à parcourir l'ensemble des objets constituant l'état du travail du concepteur, au travers des interfaces pertinentes dans ce contexte, et d'autre part à transcrire en RDF la structure ainsi parcourue. Le rôle de l'infrastructure mise en place est de faciliter au maximum le deuxième aspect, réduisant la nécessité d'une connaissance approfondie du modèle RDF et de la syntaxe RDF/XML.

Au centre de cette infrastructure de situe la classe `CATXmlRdfSyntax`, destinée à être spécialisée par toute classe produisant une description RDF d'un état de l'application, et qui fournit les méthodes suivantes :

`doHeader` et `doFooter` marquent respectivement le début et la fin de la description d'un état. Elles servent à ouvrir et fermer l'élément rdf:RDF, en y incluant les déclaration d'espaces de noms nécessaires.

`startDescription` **et** `endDescription` marquent respectivement le début et la fin de la description d'un objet. Elles servent à ouvrir et fermer l'élément rdf:Description, en y incluant l'URI de l'objet décrit (cf. ci-après). Elles peuvent apparaître directement entre `doHeader` et `doFooter`, ou entre `startProperty` et `endProperty` (cf. ci-après).

`addType` peut apparaître entre `startDescription` et `endDescription`. Elle permet d'exprimer une caractéristique de l'objet en cours de description, en ajoutant un élément rdf:type.

`addProperty` peut apparaître entre `startDescription` et `endDescription`. Elle permet d'exprimer un attribut littéral ou une relation entre l'objet en cours de description et un autre objet (décrit par ailleurs), en ajoutant un élément correspondant.

`startProperty` **et** `endProperty` s'utilisent de la même manière que `addProperty`, mais peuvent à leur tour encadrer la description de l'objet valuant la propriété.

Une autre classe, `CATRdfServices`, fournit un certain nombre de méthodes utiles à la description en RDF des états, la plus importante étant `GetURI` : cette méthode prend en paramètre n'importe quel objet CATIA (classe `CATBaseUnknown`) et retourne un URI pour cet objet. On notera que les objets CATIA sont pour la plupart munis d'un identificateur universellement unique (UUID). Cet identificateur ne change jamais au court de la vie de l'objet, contrairement à son nom affiché par l'interface ou à celui du fichier qui le contient, qui peuvent être modifiés par l'utilisateur. Or l'opérateur δ utilisé pour calculer les différences induites par un épisode, doit être capable de reconnaître l'identité des objets malgré ces changements. Les URIs données aux objets sont donc des URNs de type urn:uuid:⟨uuid-de-l'objet⟩.

6.1.2 Un exemple de MU : l'atelier d'assemblage

Afin de constituer un jeu de tests, un modèle d'utilisation a été implanté. Nous avons choisi le domaine de l'assemblage mécanique. Celui-ci utilise deux espaces de noms, notés ds: (pour Dassault Systèmes) et type: (qui correspond en fait aux éléments empruntés au MU générique CATITYPE proposé par KNOWLEDGEWARE et auquel il a été fait allusion en section 4.2.4). Le schéma RDF décrivant ce MU est donné en annexe B.1.2.

Les différents types d'objets impliqués dans ce MU sont décrits ci-dessous, avec leur URI (sous forme abrégée). Ils correspondent tous à des objets visibles dans la vue hiérarchique de CATIA.

Les produits (type:Product) correspondent aux objets physiques qui constituent l'artefact. Ils se divisent en deux catégories : les assemblages (type:Assembly) qui sont composés d'autres produits, et les pièces (type:Part), éléments terminaux de l'arbre de composition. Les liens de composition entre produits sont représentés par la relation ds:proChild.

Les contraintes mécaniques (ds:Cst) sont définies au niveau des assemblages (auquel elles sont reliées par la propriété ds:connection) et s'appliquent à leurs composants (*via* la propriété ds:connector). Elles appartiennent également à un certain nombre de sous-catégories : concentricité, distance, angle, etc. Elles possèdent enfin une caractéristique indiquant si elles sont validées ou non dans l'état décrit (ds:Cst/Status/Valid et ds:Cst/Status/Invalid).

Les paramètres (ds:Parameter) sont des grandeurs nommées liées à un produit par la relation ds:parameter. Les pièces et les contraintes comportent intrinsèquement un certain nombre de paramètres, mais seuls ceux créés ou renommés par l'utilisateur apparaissent. Les paramètres ont

pour caractéristique le type de grandeur qu'ils représentent : longueur (ds:CkeType/LENGTH), angle (ds:CkeType/ANGLE), etc.

Les formules (ds:Relation) sont associées à des produits par la propriété ds:relation. Elles servent à calculer la valeur de certains paramètres (qui leur sont liés par la propriété ds:CkeRelation/out) sur la base d'autres paramètres (liés par ds:CkeRelation/in).

6.2 Gestionnaire d'épisodes

Cette section décrit les trois composants du Gestionnaire d'épisode : son protocole de communication, le module de stockage des épisodes, et le module de remémoration. Le Gestionnaire a été écrit en JAVA afin d'assurer sa portabilité.

6.2.1 Protocole

Le Moniteur (ou l'application intégrée) communique avec le Gestionnaire à travers le réseau selon le modèle client-serveur, afin de permettre une centralisation des traces et un partage d'expérience entre plusieurs utilisateurs. Le protocole de communication est décrit ici de façon abstraite. Il a fait l'objet d'une implantation *ad hoc*, mais pourrait aussi bien être implanté à l'avenir comme une extension du protocole HTTP, voire même en se basant sur des mécanismes standards de communication entre objets comme JAVA-RMI[1] ou CORBA[2].

La plupart des requêtes ne nécessite pas d'état persistant de la connexion (seule exception : la remémoration de graphes similaires, cf. ci-après). Le client peut donc préciser pour chaque requête, à l'aide de l'attribut keepAlive, s'il souhaite que le Gestionnaire maintienne la connexion après sa réponse.

StartTrace Cette requête ne prend aucun paramètre. Elle demande au Gestionnaire la création d'une nouvelle trace. La réponse à cette requête contient l'URI de la trace nouvellement créée. Cet URI sera utilisé pour tout ajout d'état ou de transition dans cette trace (cf. ci-après).

EndTrace Cette requête prend comme paramètre l'URI d'une trace. Complémentaire de la précédente, elle indique au serveur que la trace en question est terminée. Aucun état ou transition ne pourra y être ajouté. Une trace ainsi terminée sera dite fermée, dans le cas contraire, elle sera dite ouverte.

SendState Cette requête prend comme paramètres l'URI d'une trace ouverte, un graphe RDF, ainsi qu'éventuellement un nom pour ce graphe. Le Gestionnaire ajoute à la trace un nouvel état décrit par le graphe. Si le dernier élément de la trace était déjà un état, une transition est calculée (comme décrit au chapitre 4, page 49), et intercalée. L'URI du nouvel état est construit autant que possible en utilisant le nom fourni dans la requête. La réponse à cette requête contient un graphe représentant la portion de la trace ayant été créée, c'est-à-dire : le dernier élément de la trace avant exécution de la requête, éventuellement la transition calculée, l'état créé et les relations de succession entre eux. Le client peut donc savoir quels objets ont été créés, et avec quels URIs.

[1] http://java.sun.com/products/jdk/rmi/
[2] http://www.corba.org/

SendTransition Cette requête prend comme paramètres l'URI d'une trace ouverte, un graphe RDF, ainsi qu'éventuellement un nom pour ce graphe. Si le dernier élément de la trace n'est pas un état, cette requête échoue (le système n'est pas capable de construire un état à partir d'une transition quelconque). Sinon, une transition est ajoutée à la trace, avec un URI construit autant que possible en utilisant le nom fourni dans la requête. La réponse à la requête contient un graphe représentant la portion de la trace ayant été crée, comme pour SendState.

Notons que cette requête n'est pas utilisée dans le système décrit ici, puisque l'application ne produit que des états. Elle a toute de même été implantée dans un souci de généricité (cf. chapitre 7).

Get Cette requête prend comme paramètre l'URI d'une trace, d'un état ou d'une transition. La réponse à cette requête contient un graphe décrivant l'objet requis.

GetSimilar Cette requête prend comme paramètres l'URI d'un état ou d'une transition (le graphe cible). La réponse à cette requête contient l'URI d'un graphe de même type (état ou transition) que le graphe cible, la valeur de la similarité entre ce graphe et le graphe cible, et un graphe décrivant un meilleur appariement entre eux. D'autres graphes similaires peuvent être obtenus ultérieurement grâce à la requête GetOtherSimilar, à condition de maintenir la connexion.

GetOtherSimilar Cette requête ne prend aucun paramètre, mais doit nécessairement succéder (dans la même connexion) à une requête GetSimilar. Elle reçoit le même type de réponse. Les réponses sont généralement données par ordre décroissant de valeur de similarité, mais ceci n'est pas totalement garanti (cf. section 6.2.3).

L'implantation actuelle ne permet pas au client d'influer sur les fonctions w_f et w_g de la mesure de similarité, mais on peut envisager d'ajouter ces informations comme paramètre de GetSimilar et GetOtherSimilar.

6.2.2 Stockage

Le Gestionnaire doit stocker la description de toutes les traces, ainsi que de tous les états et les transitions qui composent ces traces. Tous ces graphes sont dotés d'un URI (affecté par le Gestionnaire), et sont indexés par un unique graphe RDF, qui pour chacun indique leur type (trace ouverte, trace fermée, état, transition), ainsi que les relations qu'il entretient avec les autres graphes (un état ou une transition appartiennent à une trace, et succèdent éventuellement à une transition ou un état, respectivement). De fait, cet index constitue déjà une description de toutes les traces. Peuvent également apparaître dans cet index d'autres propriétés des états et des transitions, utiles notamment lors de la remémoration.

La taille de la base de traces, l'efficacité requise (liée notamment au nombre d'accès concurrents envisagés), peuvent varier d'une utilisation à l'autre du Gestionnaire d'épisode. Ce dernier a donc été doté d'une structure modulaire, permettant de changer le type stockage par simple modification dans un fichier de configuration, et sans nécessiter de recompilation. Le module de stockage doit adhérer à l'interface JAVA `GraphStorage` donnée en annexe B.2.1 et décrite ci-après. Deux implantations de cette interface ont été réalisées : l'une stocke les graphes en mémoire sans aucune persistance (utilisée pour des tests rapides), l'autre utilise l'interface JDBC[3] pour stocker les graphes dans une base de données MYSQL. D'autres solutions, utilisant simplement le système de fichier, ou des systèmes dédiés à RDF (RDFDB[4]) peuvent également être envisagées.

[3] http://java.sun.com/products/jdbc/
[4] http://www.guha.com/rdfdb/

L'interface JAVA `GraphStorage` définit les méthodes utiles pour accéder à une collection de graphes, identifiés par des chaînes de caractères. L'accès se fait par les cinq méthodes suivantes :

`getIDs` ne prend aucun paramètre, et retourne la liste des chaînes de caractères utilisées comme identificateurs de graphe.

`get` prend en paramètre une chaîne de caractères, et renvoie (s'il existe) le graphe identifié par cette chaîne.

`store` prend en paramètres une chaîne de caractères et un graphe. Elle stocke le graphe avec comme identificateur la chaîne donnée. Si cette dernière était déjà utilisée pour identifier un graphe, une exception est soulevée.

`update` prend en paramètres une chaîne de caractères et un graphe. La chaîne doit identifier un graphe déjà stocké, qui sera remplacé par celui passé en paramètre. Sinon, une exception est soulevée.

`remove` prend en paramètre une chaîne de caractère. Le graphe identifié par cette chaîne (s'il existe) est effacé.

Selon le type d'implantation, le remplacement d'un graphe peut être effectué de manière plus efficace qu'en l'effaçant et en stockant le nouveau graphe, d'où la distinction imposée entre `store` et `update`.

Cette interface fournit également un support pour les transactions, au cas ou le système de stockage sous-jacent offre un tel mécanisme (notamment les bases de données). Un objet adhérant à la classe `Transaction` est retourné par la méthode `beginTransaction`, qui possède les cinq méthodes d'accès décrites précédemment, ainsi que les méthodes `cancel` et `commit`, respectivement pour annuler ou valider la transaction.

6.2.3 Remémoration

Lors d'une requête GetSimilar suivie de plusieurs GetOtherSimilar, le Gestionnaire doit trouver dans la base de traces les graphes les plus similaires à celui dont l'URI est passé en paramètre (graphe cible, généralement l'état correspondant à la situation courante). Comme il a été signalé dans la description du protocole, celui-ci ne requiert pas que l'ordre dans lequel les graphes sont retournés soit parfaitement fidèle à la mesure de similarité.

Le Gestionnaire utilise en fait un mécanisme inspiré du filtrage dans l'algorithme MAC/FAC de Gentner et Forbus [1991] : avant d'utiliser une méthode précise mais coûteuse en temps pour comparer un graphe avec ceux de la base, une méthode approximative mais rapide est employée pour déterminer le sous-ensemble de la base le plus prometteur. Cette méthode approximative se base sur des traits de surface, et risque donc de manquer certains graphes pertinents (nuisant ainsi à la complétude du système). Cependant Gentner et Forbus montrent que les humains se basent rarement sur les seuls traits de structure pour leur remémoration, et que les résultats de leur systèmes sont au moins *intelligibles*, et d'une pertinence suffisante.

Le système utilisé par le Gestionnaire s'appuie également sur un vecteur représentant des traits de structure des graphes : nombre de sommets, connexité, profondeur de l'arbre de composition (pour les états). Ces traits ne sont pas utilisés pour filtrer les graphes, mais seulement pour les trier. Une fois ce tri effectué (par ordre décroissant des produits scalaires), les graphes sont comparés un par un avec le graphe cible à la demande du client (GetSimilar, puis GetOtherSimilar).

De la même façon que le module de stockage, le module de similarité a été conçu de façon modulaire, partant du principe que différents types d'application nécessitaient des mesures de similarité plus ou moins précises et plus ou moins rapides. Les différentes mesures doivent être implantées dans une classe adhérant à l'interface JAVA `SimilarityEngine` donnée en annexe B.2.2. Cette interface introduit la notion de session et de comparaison : une comparaison concerne deux graphes (cible et source), mais peut donner lieu à plusieurs mesures (appel à la méthode `getBestMatch`) avec des fonctions w_f et w_g différentes. Une session ne concerne que le graphe cible, et peut donner lieu à plusieurs comparaisons (avec des graphes sources différents). Une instance de `SimilarityEngine` peut bien entendu ouvrir autant de sessions que nécessaires avec des graphes cibles différents. La motivation de ce principe réside dans le fait que la mesure de similarité peut nécessiter des pré-traitements sur les graphes ; ce mécanisme permet d'éviter la répétition intempestive de ces pré-traitements lors de nombreuses comparaisons impliquant le même graphe cible, ou de nombreuses mesures de similarité entre deux graphes avec des variations sur les fonctions de poids.

6.3 Moniteur

Comme le Gestionnaire, le Moniteur a été développé en JAVA. La majeure partie de son développement a fait l'objet du stage de Saurabh Sharma. Le rôle du Moniteur est double : il assure la communication entre l'application instrumentée et le Gestionnaire, et il fournit à l'utilisateur une interface graphique pour l'assistance à la réutilisation d'épisode. Le premier se limite à scruter le contenu d'un répertoire (donné au démarrage de l'application), et à transférer au Gestionnaire un nouvel état pour tout fichier RDF apparaissant dans ce répertoire. L'interface graphique et les interactions avec l'utilisateur sont décrite plus en détail dans cette section.

6.3.1 Aperçu général de l'interface

Comme on le voit sur la figure 6.2, l'interface graphique du Moniteur se divise en quatre parties. Le quadrant supérieur gauche représente l'état courant. Une requête de remémoration a été envoyée au Gestionnaire, dont les résultats apparaissent progressivement dans le cadrant inférieur gauche, sous forme d'une liste d'états triés par ordre décroissant de similarité avec l'état courant. Sur cette liste apparaissent l'URI des états, et leur valeur de similarité. L'état sélectionné (état source) apparaît dans le quadrant supérieur droit, et son état suivant dans le quadrant inférieur droit. La moitié droite de la fenêtre donne donc un aperçu d'un épisode réutilisable (état initial et état final).

Lorsque l'utilisateur est satisfait par l'un des résultats proposés dans la liste, il peut demander au Moniteur de lui proposer une adaptation de l'épisode correspondant à la situation courante. L'état final cible est alors construit grâce à l'algorithme présenté en section 5.2. Une vue de cet état remplace la liste des résultats, la moitié gauche de la fenêtre donnant alors un aperçu de l'épisode adapté. Dans l'état actuel de l'implantation, l'utilisateur doit interpréter cet épisode et l'appliquer manuellement dans l'application. Dans la variante d'une application intégrée, l'état final cible pourrait bien sûr être généré directement dans l'application.

6.3.2 Exploration des graphes

Le Moniteur est indépendant d'un quelconque modèle d'utilisation. La visualisation des états qu'il offre doit donc à la fois être générique et permettre une vue synthétique ou détaillée des états. Ces besoins contradictoires ne peuvent être satisfaits par une vue unique et statique de l'état.

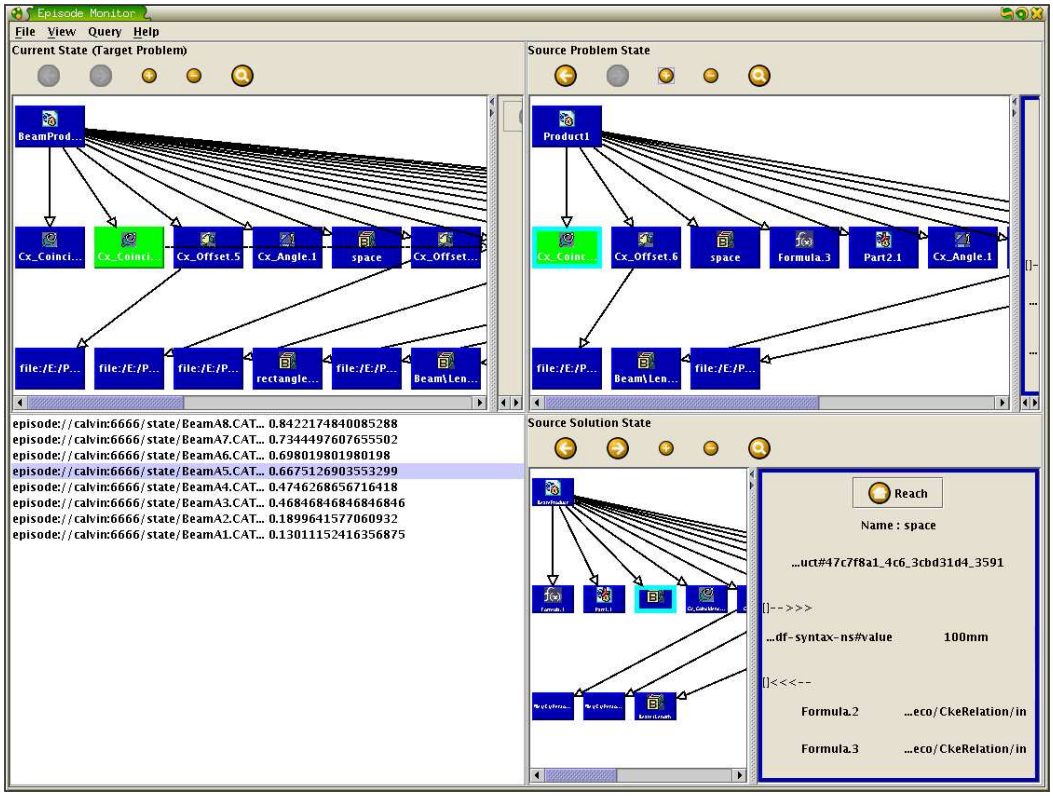


FIG. 6.2 – Interface du Moniteur d'épisodes

C'est donc au travers de ses *interactions* avec le graphe que l'utilisateur en prend connaissance. Le composant de visualisation est séparé en deux parties : le panneau graphique (à droite) et le panneau d'informations (à gauche).

Le panneau graphique donne une représentation simplifiée du graphe : seuls les arcs de composition y apparaissent en permanence. Les autres arcs apparaissent lorsque le pointeur de la souris passe sur l'un de leurs sommets. Les sommets sont affichés avec leur nom (propriété rdfs:label) ou à défaut leur URI, ainsi qu'avec une icône représentative de leur type. Cette dernière permet de faciliter la lecture du graphe, notamment si elle rappelle l'interface graphique de l'application (comme c'est le cas ici). La correspondance entre icônes et types est bien sûr entièrement paramétrable.

Le panneau d'information donne une représentation détaillée de l'élément (sommet ou arc) sélectionné dans le graphe. Un élément est sélectionné en cliquant sur sa représentation dans un des deux panneaux. Pour un sommet, les informations présentées sont son nom, son URI, ses arcs entrants et sortant. Pour un arcs, les informations sont son prédicat (au sens de RDF), son sommet de départ et son sommet d'arrivée. Pour les sommets, ce panneau contient également un bouton permettant de centre la vue, dans le panneau graphique, sur ce sommet.

La barre d'outils au dessus des deux panneaux fournit plusieurs fonctions. Les boutons de navigation (flèches à droite et à gauche) permettent de se déplacer dans l'historique des sélection. Les boutons de zoom (plus et moins) permettent de changer la taille des éléments dans le panneau graphique. Enfin, le bouton représentant une loupe permet d'afficher, lorsqu'elle est disponible, une copie d'écran de l'application dans l'état représenté par le graphe. Cette dernière fonction permet de fournir une vue plus spécifique et donc susceptible d'aider la lecture du graphe.

6.3.3 Similarité

Le Moniteur ne se contente pas d'afficher la valeur de similarité entre l'état initial cible et l'état initial source : il permet également de consulter l'appariement ayant permis de calculer cette valeur. Lorsque le pointeur de la souris passe sur un sommet d'un de ces deux graphes, celui-ci est mis en surbrillance, ainsi que le(s) sommet(s) de l'autre graphe au(x)quel(s) il est apparié. La couleur de la surbrillance dépend de la cardinalité de l'appariement : un sommet apparié une fois sera vert, un sommet non apparié sera rouge, et un sommet apparié plusieurs fois sera orange. Il est également possible de mettre l'intégralité des sommets des deux graphes en surbrillance, ce qui permet une vue globale de la cardinalité des appariements.

Cette possibilité de consulter l'appariement joue d'abord un rôle dans l'intelligibilité de l'assistant : si l'utilisateur est surpris par la valeur de similarité donnée à un couple d'états, il peut éventuellement découvrir de cette façon des mises en correspondances auxquelles il n'avait pas pensé. Mais un autre intérêt de cette fonctionnalité résiderait dans la possibilité de critiquer l'appariement afin d'influer sur les poids utilisés pour la similarité. Ceci n'est cependant pas implanté dans la version actuelle.

Troisième partie

Vers un assistant pour le Web Sémantique

Chapitre 7

Le modèle MUSETTE

Le chapitre 3 a déjà introduit la problématique du Web Sémantique, et présenté les langages développés dans ce contexte. Ces langages (notamment RDF) ont été utilisés dans la première partie de ce travail afin de prendre en compte la dimension *documentaire* des connaissances manipulées par les concepteurs. Inversement, la dimension sémantique que le Web est appelé à prendre rend pertinents les travaux d'intelligence artificielle pour répondre à cette nouvelle problématique.

La section 7.1 montre l'importance que la modélisation d'expérience peut avoir dans le futur Web Sémantique. Les idées développées jusqu'ici dans le domaine de la conception mécanique trouvent donc leur place dans cet autre contexte. La section 7.2 propose un modèle général, le modèle MUSETTE, inspiré de celui proposé autour de CATIA dans le cadre du projet ARDECO, mais également d'autres travaux portant sur la réutilisation d'expérience [Champin et Prié, 2003]. Les liens entre ARDECO et MUSETTE sont discutés plus en détail en section 7.3

7.1 Réutilisation d'expérience sur le Web Sémantique

L'objectif du Web Sémantique est de faire évoluer le Web pour répondre aux besoins toujours plus importants et plus variés suscités par son succès. En effet, pour répondre efficacement à ces besoins, les agents informatiques doivent assumer une partie de la charge cognitive incombant jusqu'ici à l'utilisateur. Pour cela, un certain nombre de connaissances doit être formalisé, afin de mettre en œuvre des mécanismes de raisonnement au sein de ces agents. Nombre de travaux sur le Web Sémantique considèrent les ontologies Heflin *et al.* [1999]; Decker *et al.* [1999]; Fensel *et al.* [2000] comme le moyen de mettre en œuvre ces mécanismes.

Cependant, la conception et le déploiement d'ontologies est un travail difficile et coûteux. Or c'est justement la souplesse et la facilité d'utilisation du Web, notamment la simplicité et la robustesse du langage HTML, qui ont fait son succès, ceci au prix d'une absence de cohérence globale et d'une profusion de liens « morts ». Cela d'ailleurs a fait dire que la fameuse « erreur 404[1] » n'était pas la faiblesse du Web, mais au contraire sa force. Le Web Sémantique doit-il renoncer à cette souplesse pour rendre de plus grands services ? Il semble plutôt que les systèmes fondés sur les ontologies doivent cohabiter avec des systèmes offrant des services plus modestes, mais capables de s'accommoder des incohérences et des données moins formalisées qui composent, et composeront sans doute longtemps, le Web.

Parallèlement, l'idée se développe que l'*annotation* des ressources du Web est une façon de réaliser le Web Sémantique [Handshuh *et al.*, 2001; Handschuh, 2003]. Cela permettra d'abord

[1] page introuvable, dans le protocole HTTP

d'enrichir « sémantiquement » et de façon extrinsèque les ressources existantes dont le format interne ne permet pas cet enrichissement. Ensuite, même pour les ressources dont le format interne autorise un contenu sémantiquement exploitable, l'annotation peut s'avérer utile lorsque la modification de ces ressources n'est pas possible. Enfin, et sans doute le plus important, il est impossible de prédire *a priori* toutes les tâches dans lesquelles une ressource donnée peut être exploitée, et donc de lui attacher d'avance toutes les connaissances nécessaires à son exploitation.

En revanche, tout *usage* d'une ressource fournit des informations sur la manière dont cette ressource peut effectivement être utilisée. Il constitue en ce sens une *signature* des connaissances portées par cette ressource et qu'il contextualise. Il paraît donc pertinent d'annoter une ressource par les utilisations qui en sont faites, et qui évoquent la notion d'épisode introduite dans le projet ARDECO. On parlera donc d'épisodes d'utilisation de la ressource, qui permettront la réutilisation des connaissances capturées en les re-contextualisant dans des situations nouvelles. Cela permettra en outre de mettre en œuvre le principe selon lequel toute activité doit avoir pour effet de bord la création de connaissance.

7.2 Présentation du modèle

MUSETTE propose un cadre général pour la mise en œuvre d'un assistant exploitant des traces et des épisodes d'utilisation des ressources manipulées à l'aide d'un système informatique. Ce cadre est décrit globalement par la figure 7.1. Le système et son utilisateur apparaissent en haut de la figure. Par « système informatique », on peut entendre une application (par exemple CATIA ou un navigateur Web), mais également une fonctionnalité particulière d'un application (un atelier de CATIA, la gestion des signets d'un navigateur) ou à l'inverse un système plus large, comme une suite bureautique ou un système d'exploitation. Enfin, l'interaction entre l'utilisateur et le système peut être médiée par d'autres systèmes non représentés et non considérés par le modèle (par exemple, un moteur de recherche en ligne, dont l'accès est médié par un navigateur).

De façon évidente, l'utilisateur n'est pas observable directement par un agent informatique, mais seulement à travers le système. La *trace d'utilisation* (cf. section 7.2.2) est donc construite grâce à une instrumentation du système. Cette instrumentation est schématisé sur la figure par un *observateur*, mais cette représentation n'impose pas que l'observateur soit une entité unique et distincte. Par exemple dans le prototype développé (cf. chapitre 6), il est réparti entre l'application CATIA (export en RDF), le Moniteur (transfert au Gestionnaire) et le Gestionnaire (stockage de la trace, construction des transitions). L'observateur est lié au *modèle d'utilisation* (cf. section 7.2.1), qui décrit la façon dont la trace sera construite.

Une fois la trace d'utilisation produite, des *épisodes d'utilisation* (cf. section 7.2.4) peuvent en être extraits. Cette extraction se fait par la reconnaissance dans la trace de *signatures de tâche* (cf. section 7.2.3). Chaque épisode représente la partie de l'activité correspondant à l'exécution de cette tâche. Si le modèle d'utilisation est unique (pour un assistant donné), il peut y avoir en revanche plusieurs signatures de tâche. Enfin, ces épisodes sont exploités par l'assistant à proprement parler (à gauche au centre), qui peut alors interagir directement avec l'utilisateur (c'est par exemple le cas avec le Moniteur, cf. chapitre 6), ou en passant par l'application (application intégrée). La suite de cette présentation sera illustrée par un exemple très simple appliqué à l'utilisation d'un navigateur Web.

7.2.1 Modèle d'utilisation

Le modèle d'utilisation décrit la manière dont l'observateur décrit l'utilisation du système. Ce dernier est toujours décrit en termes d'*éléments d'intérêt* qui peuvent être :

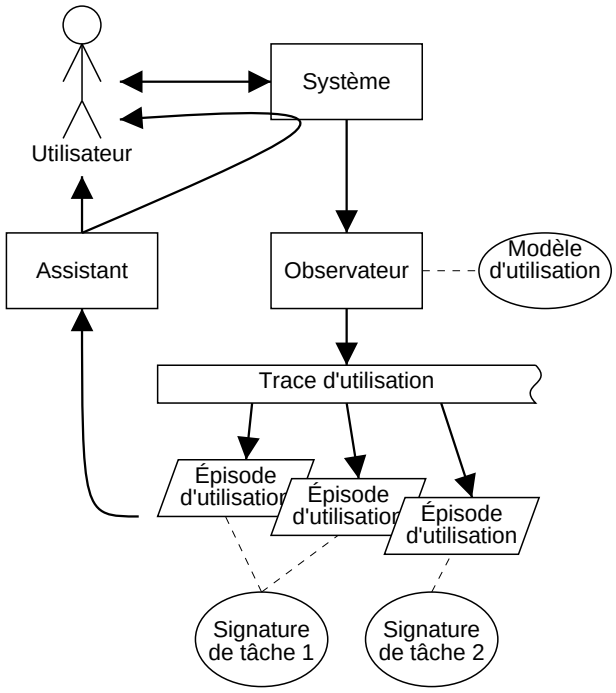


FIG. 7.1 – Architecture MUSETTE

Des objets constitutifs du système et manipulables par l'utilisateur. Cette dernière caractéristique tient au fait que c'est l'*utilisation* du système que l'on cherche à décrire, et non le système lui même. À titre d'exemple, le modèle d'utilisation de l'atelier Assemblage de CATIA, décrit en section 6.1.2, contient (entre autres) les objets Produit, Pièce, Contrainte. Le modèle d'utilisation d'un moteur de recherche contiendra en général les objets Requête, Mot-Clé, Résultat, etc.

Des évènements intervenant dans l'activité de l'utilisateur avec le système. Ces évènements ont en général pour origine l'utilisateur lui même (les opérations qu'il effectue) mais peuvent éventuellement avoir une autre origine (arrivée d'un courrier électronique, déclenchement d'une alarme, etc.).

Le modèle MUSETTE ne contraint pas la description des éléments d'intérêt à un formalisme particulier : cette description peut consister en une véritable ontologie des objets et des évènements, mais également en une description moins formalisée : schéma RDF, strates-IA [Prié, 1999], etc. En général toutefois, le formalisme choisi permettra de structurer la représentation par des relations entre éléments d'intérêt.

Comme cela a déjà été signalé dans le cas de CATIA, il est important de noter qu'il n'existe pas un unique modèle d'utilisation pour un système donné. Les éléments d'intérêt (objets et évènements) n'ont pas vocation à décrire parfaitement l'activité mais simplement, comme leur nom l'indique, la partie intéressante de l'activité dans le contexte où l'assistant doit intervenir. De la même façon, la manière dont ces éléments sont agencés pour créer la trace d'utilisation est spécifiée par le modèle d'utilisation, sans que cette spécification soit la seule possible (cf. ci-après).

Dans l'exemple du navigateur Web, le modèle d'utilisation comporte quatre types d'objet, et quatre types d'évènement (cf. figure 7.2). Les objets sont les liens hypertextes (Lien), les pages Web (Page), les images pouvant être contenues dans les pages (Image), et les options de préférence

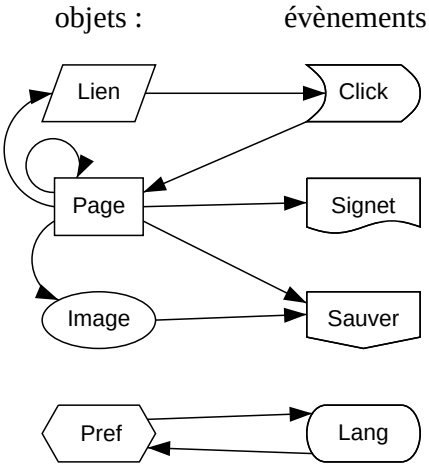


FIG. 7.2 – Un modèle d'utilisation simplifié pour un navigateur Web

du navigateur (Pref). Les évènements correspondent tous à des opérations de l'utilisateur : click sur un lien (Click), pose d'un signet sur une page (Signet), enregistrement d'une page ou d'une image (Sauver) et changement de la langue par défaut (Lang). Ce modèle d'utilisation prescrit aussi quels éléments peuvent être mis en relation (non typée).

7.2.2 Trace d'utilisation

Comme dans ARDECO, la trace d'utilisation dans MUSETTE est décrite par une succession d'*états* et de *transitions*. Les états décrivent le système à un instant donné, grâce aux objets du modèle d'utilisation. Les transitions entre états sont quant à elles décrites grâce aux évènements du modèle d'utilisation. Contrairement à ARDECO en revanche, les traces ne sont pas nécessairement calculées à partir des états (ceci relève du fonctionnement interne de l'observateur).

Comme il a été dit, la manière dont la trace est construite est spécifiée dans le modèle d'utilisation. Cela concerne d'une part les critères conditionnant l'ajout d'un état à la trace. Dans le cas d'ARDECO, ces critères étaient les indices comportementaux liés au changement d'objectif. Dans le cas d'un navigateur, cela pourrait être n'importe quel click de souris, les changements de page, ou encore les changements de site. Cela concerne d'autre part les éléments figurant dans un état ou une transition. On peut par exemple, dans le cas d'un navigateur multi-fenêtré, faire figurer toutes les pages ouvertes à un instant donné, ou uniquement celle qui se trouve au premier plan. Enfin, la question concernant les éléments à représenter se pose également pour les relations.

Il n'est pas ici question de suggérer que l'observateur ne trace pas tout ce qu'il observe, mais de reconnaître que ses capacités d'observation sont nécessairement limitées. On se garde en outre de faire l'hypothèse qu'il existerait un modèle d'utilisation «primitif» et exhaustif servant de point de référence — même si dans de nombreux cas, la vision du concepteur du système (si elle est disponible) s'approche d'un tel idéal. On adoptera plutôt une position selon laquelle le modèle d'utilisation choisi constitue une référence nécessairement arbitraire, mais qu'il pourrait aussi bien être considéré comme une signature de tâche (cf. ci-après) dans un modèle d'utilisation plus général (par exemple, l'utilisation d'un atelier de CATIA est une tâche particulière dans l'utilisation de l'application entière ; l'utilisation d'un gestionnaire de signets est une tâche particulière dans l'utilisation d'un navigateur).

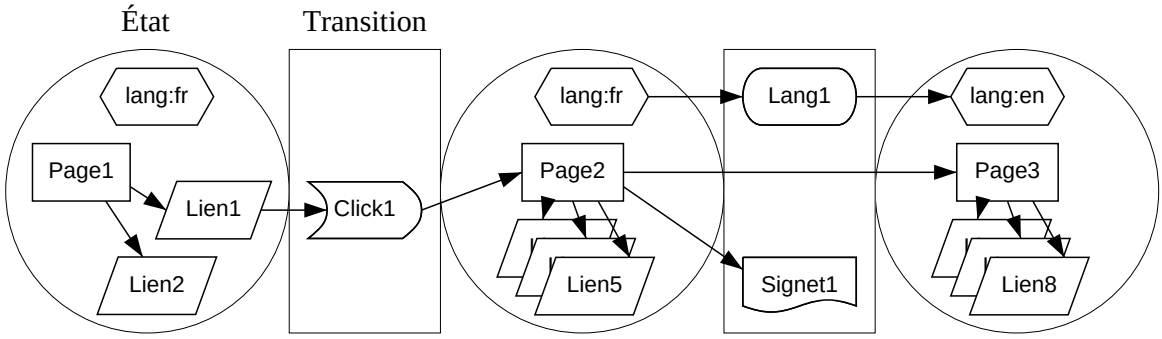


FIG. 7.3 – Un exemple de trace d'utilisation

Dans l'exemple du navigateur Web, le modèle d'utilisation simple proposé spécifie que seule la page affichée au premier plan est représentée dans l'état, et qu'un nouvel état est tracé à chaque changement de page. Les transitions contiennent tous les évènements apparus entre les deux états et représentables dans ce modèle d'utilisation. Un exemple de trace résultante est donnée en figure 7.3. Les cercles représentent des états, et les rectangles des transitions. On peut lire cette trace comme :

> L'utilisateur est en présence d'une page Page1 dotée de deux liens, et son navigateur est configuré en français. Un click sur le premier lien l'amène sur la page Page2, toujours avec la langue française par défaut. Ensuite, l'utilisateur effectue deux opérations : il pose un signet sur Page2 et change la langue par défaut du navigateur pour l'anglais. La relation entre Page2 et Page3 signifie que les deux pages ont même URL (le changement de contenu est en fait dû au changement de langue).

7.2.3 Signatures de tâche

La trace d'utilisation offre une vue globale de l'utilisation du système, sans fournir directement d'information sur les tâches de l'utilisateur. Une hypothèse fondamentale du modèle MUSETTE est que chaque tâche a un ensemble d'invariants qui permet de la détecter dans la trace : on appelle cet ensemble la *signature* de la tâche. Par exemple, dans le modèle d'utilisation d'un moteur de recherche, la tâche «Raffiner une requête» peut se repérer à la succession de deux requêtes dont la seconde contient les mêmes mots-clés que le première ainsi que d'autres mots-clés.

La signature de tâche est également assortie d'*explications* qui permettent d'interpréter, dans le contexte spécifique de cette tâche, la portion correspondante de la trace. Par exemple, dans le cas du moteur de recherche et de la tâche «Raffiner une requête», la première des deux requêtes peut être qualifiée de trop générale, puisque l'utilisateur a eu besoin de la raffiner.

Le formalisme de représentation des signatures de tâche dépend naturellement de celui adopté pour le modèle d'utilisation. Les explications peuvent quant à elles sortir du cadre du modèle d'utilisation : la catégorie «requête trop spécifique» évoquée à propos de l'exemple du moteur de recherche ne fait pas nécessairement partie du modèle d'utilisation initial. La seule contrainte pour les explications est qu'elles puissent être reliées d'une façon ou d'une autre aux éléments de la trace qu'elles sont censées expliquer.

Dans l'exemple du navigateur Web, on peut proposer deux signatures de tâches représentées sur la figure 7.4. La première, «Relever un site intéressant», est caractérisée par un click sur

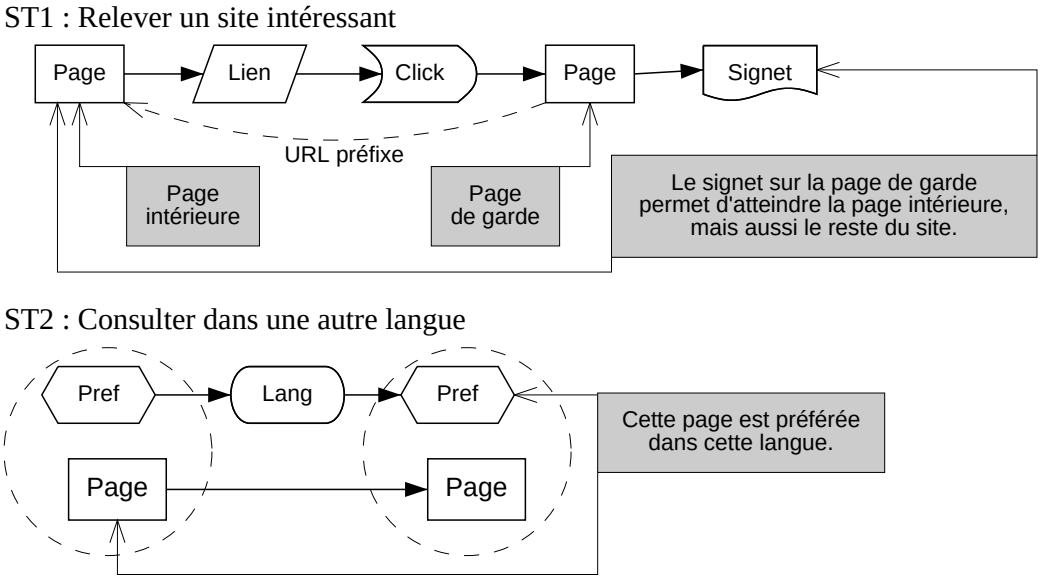


FIG. 7.4 – Deux exemples de signatures de tâche

le lien d'une première page, et de la pose d'un signet sur la page résultante. Une contrainte supplémentaire (représentée par une double flèche en pointillés) stipule que l'URL de la seconde page doit être un préfixe de celle de la première page (exemple : http://www710.univ-lyon1.fr/ est un préfixe de http://www710.univ-lyon1.fr/~champin). Les explications, représentées ici comme des annotations textuelles (en gris), précisent que la première page peut être considérée comme une page *intérieure* d'un site dont la seconde en est la page *de garde*. La pose de signet est expliquée par le fait que la page marquée n'est pas la plus intéressante, mais qu'elle permet d'accéder à une page intéressante, ainsi qu'à d'autres pages du même site, donc potentiellement intéressantes. En ce sens, c'est un signet *de site* plutôt qu'un signet *de page*, distinction qui n'existe par ailleurs pas dans le modèle d'utilisation (la notion de site n'y existe d'ailleurs pas non plus).

La seconde signature de tâche est intitulée «Consulter dans une autre langue». Elle est caractérisée par le fait qu'une page, présente dans le même état (cercle en pointillé) qu'une option de personnalisation, change de contenu sans changer d'URL conjointement à une opération de changement de langue. Les explications indiquent que l'utilisateur préfère consulter la page en question dans la nouvelle langue par défaut.

7.2.4 Épisodes d'utilisation

Un épisode d'utilisation est l'occurrence d'une signature de tâche dans la trace, enrichie par les explications apportées par cette signature de tâche. Pour reprendre l'exemple du navigateur Web, chacune des signatures de tâche proposées (fig. 7.4) a une occurrence dans la trace (fig. 7.3). Ces deux épisodes d'utilisation sont représentés sur la figure 7.5 (pour chacun d'eux la trace est représentée en filigrane afin de faciliter leur identification).

Il apparaît que certains éléments peuvent appartenir à plusieurs épisodes. En effet, l'utilisateur est susceptible d'effectuer plusieurs tâches en parallèle impliquant les mêmes objets, voire les mêmes opérations (en faisant «d'une pierre deux coups»). Inversement, certains éléments de la trace peuvent n'appartenir à aucun épisode. Toutes les tâches possibles ne peuvent en effet être prévues ; l'ensemble des signatures de tâche peut donc ne pas couvrir totalement la

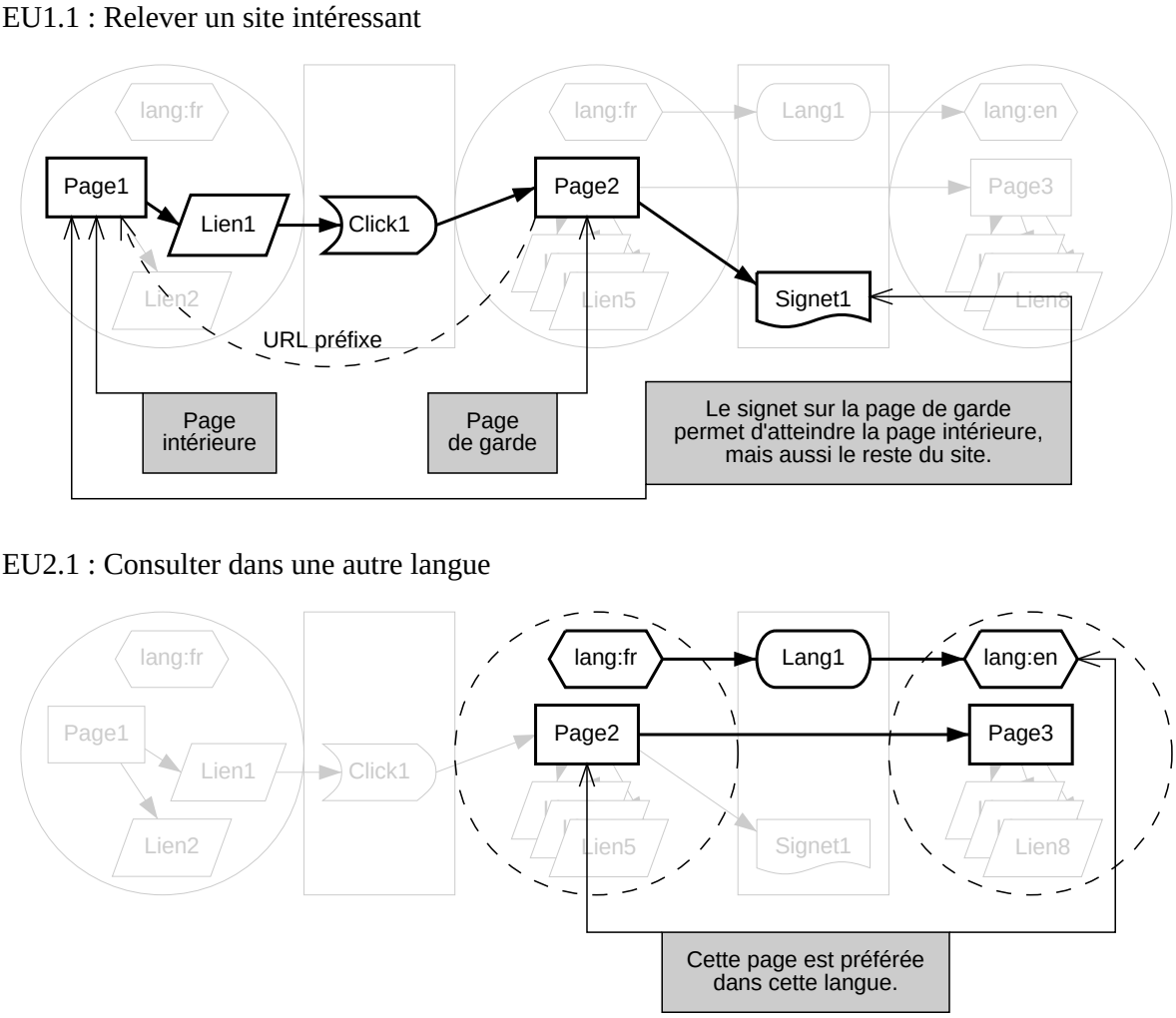


FIG. 7.5 – Deux épisodes d'utilisation

trace. On notera qu'il est cependant possible de proposer des signatures de tâche extrêmement générales, apportant peu d'explication mais ayant une large couverture. C'est notamment le cas dans ARDECO, sur lequel on reviendra en section 7.3.

Ces épisodes peuvent être exploités par l'assistant, par exemple pour en proposer la réutilisation : une occurrence partielle d'une signature de tâche est repérée dans la trace courante, un épisode similaire relevant de la même tâche est remémoré, et les éléments manquants dans la trace courante peuvent être complétés par ceux de l'épisode réutilisable (moyennant adaptation). Par ailleurs, les explications apportées par les signatures de tâche peuvent fournir des connaissances supplémentaires à celles fournies par la trace, dont la validité est limitée au contexte de la tâche mais qui peuvent être exploitées par l'assistant dans ce contexte.

Pour illustrer l'exploitation d'épisodes d'utilisation, on peut reprendre les deux épisodes de la figure 7.5. L'épisode EU2.1 peut être réutilisé dans tout contexte où la page Page2 est affichée avec le français comme langue par défaut. Selon l'autonomie laissée à l'assistant pour la réutilisation, celui-ci changera automatiquement la langue ou se contentera de suggérer à l'utilisateur de le faire. Par ailleurs, si l'utilisateur pose un signet sur une nouvelle page Page4, et qu'il se trouve

que celle-ci se situe sur le site dont Page2 est la page de garde, l'épisode EU1.1 peut être exploité. Il suggère en effet que Page2 permet d'accéder à cette nouvelle page (en fait, à toutes les pages du site), et donc que le signet sur Page4 est inutile, car il en existe un sur Page2. L'assistant pourra donc indiquer à l'utilisateur que ce nouveau signet est superflu.

7.3 ARDECO et MUSETTE

ARDECO est bien une instance du modèle MUSETTE, mais comporte un certain nombre de spécificités qui méritent d'être soulignées.

Comme on l'a déjà fait remarquer, l'observateur dans ARDECO n'est pas un composant distinct, mais son rôle est distribué entre l'application CATIA, le Moniteur et le Gestionnaire. Rappelons cependant que seule la partie liée à CATIA est spécifique à un modèle d'utilisation. Un changement de modèle d'utilisation ne nécessite aucune modification au niveau du Moniteur ou du Gestionnaire.

ARDECO ne spécifie pas de modèle d'utilisation unique pour CATIA ou un de ses atelier CATIA[2] mais requiert que les éléments d'intérêt soient décrits par un schéma RDF (éventuellement étendu par un vocabulaire comme OWL ou DAML). En revanche, la spécification de la façon dont la trace est construite est la même pour tous les modèles d'utilisation : un état est ajouté lorsque les indices comportementaux (notamment changements importants dans la visualisation géométrique) sont repérés. Les transitions sont calculées à partir des états qu'elles relient (les seuls évènements qu'elles contiennent sont les modifications d'objets et de relations entre eux, traduites par les apparitions et disparitions de caractéristiques).

On remarque qu'ARDECO utilise la notion d'épisode sans avoir explicitement défini de signature de tâche. En fait, ARDECO utilise une *unique* signature de tâche implicite qui pourrait s'intituler « Atteindre un objectif identifié » (pour reprendre la définition d'un épisode de conception, page 44). On sait que les indices comportementaux qui régissent la construction de la trace sont caractéristiques des changements d'objectifs. Il en découle que par construction, toute succession de deux états est une occurrence de cette tâche. Faute de plus de connaissances sur l'objectif poursuivi par le concepteur, on gardera dans l'épisode tous les objets et évènements compris entre les deux états.

Cette absence de signatures de tâches spécialisées ne signifie pas que qu'ARDECO n'est pas capable de prendre en compte les spécificités d'un épisode par rapport à un autre. Différentes tâches plus précises que « Atteindre un objectif identifié » auront des signatures différentes qui se répercuteront sur les épisodes, et seront en principe détectées par la mesure de similarité. Dans MUSETTE, les signatures de tâche *identifiées* servent d'index en structurant la mémoire, un peu à la façon des scripts de Schank et Abelson [1977]. Elles évitent de *redécouvrir* à chaque remémoration des régularités entre épisodes. Quoi qu'il en soit ces régularités existent et contribuent à la remémoration d'un épisode traitant une tâche similaire à la tâche en cours, fût-elle implicite.

[2] En ce sens, on peut le voir comme un sous-ensemble de MUSETTE plutôt que comme une instance.

Chapitre 8

Discussion et perspectives

En guise de conclusion, ce chapitre donne en section 8.1 un résumé soulignant la contribution de ce travail. La section 8.2 propose un certain nombre de pistes prolongeant ce qui a été présenté.

8.1 Résumé et contribution

L'objectif de ce travail était, dans le cadre du projet ARDECO, la construction d'un assistant pour le logiciel de conception assistée par ordinateur CATIA de Dassault Systèmes. Cet assistant devait être capable de capturer l'expérience de conception des utilisateurs afin de les aider à réutiliser cette expérience. Dans le chapitre 1, le raisonnement à partir de cas est présenté. Il est discuté de l'adéquation de ce paradigme d'intelligence artificielle avec le modèle de l'assistant, et de sa capacité à réduire les problèmes rencontrés par d'autres modes de raisonnements (notamment à partir de règles). Le chapitre 2 s'attache à l'activité de conception et à ses spécificité par rapport à d'autres activités de résolution de problème. Il en ressort qu'un problème de conception se construit en même temps qu'il se résout, selon un cycle itératif de reformulations. Une revue des systèmes d'assistance à la conception fondés sur le RàPC est également donnée. Les connaissances de conception étant le plus souvent consignées de façon plus ou moins implicite dans des documents de conception, le chapitre 3 s'intéresse aux travaux autour de la notion de document. Parmi ceux-ci, les travaux récents autour de la vision du Web Sémantique sont particulièrement pertinents à l'égard de la problématique de l'assistance.

La deuxième partie de ce mémoire est consacrée à la contribution dans le domaine de l'assistance à la conception. Le chapitre 4 introduit la notion d'*épisode de conception* en la situant dans le cadre du contexte applicatif (CATIA). Sur la base des premiers résultats des psychologues et ergonomes du projet ARDECO, une modélisation des épisodes sous forme de graphes est proposée. Cette modélisation s'appuie sur un *modèle d'utilisation*, et permet d'introduire la notion de *trace d'utilisation*. Une représentation en RDF est proposée pour ces trois types d'entité. Le chapitre 5 décrit une mise en œuvre des mécanismes de remémoration et d'adaptation avec une base d'épisodes de conception. Ces mécanismes s'appuient sur une mesure de similarité guidée par l'adaptation, centrée sur la notion de *différences* entre graphes. Enfin, le chapitre 6 décrit le prototype implantant les mécanismes décrits par une instrumentation de CATIA d'une part, et par deux composants génériques réutilisables d'autre part.

La dernière partie est une ouverture sur la problématique plus générale du Web Sémantique. Il semble en effet que la réutilisation d'expérience, dans des domaines où les connaissances sont mal formalisées, soit une piste porteuse pour la réalisation de cette « vision » de la prochaine étape du Web. Le chapitre 7 présente dans cette optique le modèle MUSETTE, une généralisation

des idées développées dans la deuxième partie. Ce modèle reprend les notions d'épisode, de trace et de modèle d'utilisation. Il introduit également la notion de *signature de tâche*, dont le rôle est d'expliquer des parties de la trace et d'en extraire des épisodes.

8.2 Perspectives

Les perspectives de ce travail s'orientent selon trois directions : la première (section 8.2.1) concerne le prototype développé, sa validation dans le domaine de la CAO et d'autres domaines. La seconde (section 8.2.2) concerne la mesure de similarité, pour laquelle de nombreuses pistes peuvent encore être explorées. Enfin la notion de signature de tâche, introduite par le modèle MUSETTE, ouvre également des pistes de recherches intéressantes présentées en section 8.2.3.

8.2.1 Validation du prototype

Le prototype n'a hélas pas encore pu être testé par des concepteurs utilisant CATIA en situation réelle. De telles expérimentations seraient utiles à la fois pour la validation des modèles psychologiques et celle des modèles informatiques. Cette validation commune est d'ailleurs un des objectifs initiaux de la collaboration interdisciplinaire suscitée par le projet ARDECO.

Une autre piste pour la validation du prototype est son application à un autre domaine de conception : la conception logicielle. Dans ce domaine en effet, les outils de gestion de versions, comme par exemple CVS[1] (*Concurrent Versions System*), conduisent les utilisateurs à marquer explicitement des *étapes* dans leur activité de conception. Les règles préconisées par ce système pour la définition de telles étapes semblent cohérentes avec ce qui a été vu sur la délimitation des épisodes[2]. La succession de versions doit donc pouvoir constituer une trace au sens du modèle MUSETTE. Ajoutons que les étapes, dans CVS, sont généralement accompagnées d'annotations textuelles expliquant la transition y conduisant. Ces annotations, même informelles peuvent constituer autant d'informations utiles (au moins pour le concepteur) lors de la réutilisation d'un épisode.

Enfin, le modèle MUSETTE est applicable à des domaines autres que la conception, et le prototype peut être adapté à ces autres domaines. La seule modification majeure nécessaire est la possibilité de fournir de nouvelles signatures de tâche pour la délimitation des épisodes. Pour cela, les travaux en cours sur MUSETTE visent à préciser la façon dont les signatures de tâche peuvent être exprimées.

8.2.2 Mesure de similarité

Le travail sur la mesure de similarité entre graphes, présenté au chapitre 5, peut être approfondi dans plusieurs directions. La première, qui a déjà été évoquée, est celles des algorithmes de recherche locale. En effet, si l'algorithme glouton a donné des résultats très satisfaisants, il n'en reste pas moins sensible à un certain nombre de « pièges » (du fait de l'absence de retour arrière). Une recherche locale doit permettre, en conservant des performances comparables, d'affiner les résultats de la recherche gloutonne. Sébastien Sorlin poursuit notamment en DEA son travail dans ce domaine en appliquant la méthode de recherche locale TABOU.

[1] http://www.cvshome/org

[2] CVS offre de plus deux niveaux de granularité : `commit` (niveau local) et `tag` (niveau global). Chacun de ces niveaux est donc susceptible de fournir des épisodes, mais il conviendra alors d'identifier le degré d'abstraction pertinent pour les représenter.

Une deuxième direction concerne les heuristiques. Des propriétés remarquables des modèles d'utilisation peuvent probablement donner lieu à des heuristiques particulières qu'il est possible d'exprimer dans les fonctions w_f et w_g. En particulier, la relation de composition a une sémantique suffisamment particulière pour avoir été fournie en standard dans le vocabulaire de description des épisodes. On peut donc considérer dans de nombreux cas les états comme des *arbres* de composition, auxquels sont ajoutés des arcs supplémentaires, dits « de relation » — l'affichage des états dans le prototype est d'ailleurs fondé sur cette considération. Cette structure d'arbre « augmenté » peut sans doute être exploitée au niveau de l'heuristique, d'autant plus que la relation de composition semble jouer un rôle prépondérant dans la remémoration humaine (ce qu'ont montré des travaux au sein du projet ARDECO).

Une troisième façon de compenser la complexité de la mesure de similarité est de faire intervenir l'utilisateur de façon plus précoce et plus importante. On a dit que l'algorithme pouvait facilement être modifié afin d'être rendu *anytime*, c'est-à-dire capable de donner, à n'importe quel moment, la meilleure solution trouvée jusqu'alors. Une telle modification de l'algorithme supposerait également une modification de l'API utilisée par le prototype (`SimilarityEngine`), afin d'y permettre l'exploitation de ces résultats intermédiaires. Cette modification toucherait également l'interface graphique, dans laquelle le statut d'une solution (intermédiaire ou définitive) devrait à tout moment être clair pour l'utilisateur. Un exemple de scénario exploitant un algorithme *anytime* est le suivant : la liste des meilleurs candidats est affichée comme dans le prototype actuel, avec une valeur approchée de leur similarité (par exemple, le résultat trouvé au bout d'un temps fixé). Lorsque l'utilisateur sélectionne un élément de cette liste, le graphe correspondant est affiché, et le résultat est affiné en poursuivant la recherche sur ce graphe. Dans l'interface, l'élément de la liste peut alors changer de rang, et l'appariement être modifié au fur et à mesure que de nouveaux résultats sont fournis. Si ce candidat lui semble insatisfaisant, l'utilisateur peut en afficher un autre (et donc poursuivre la recherche sur cet autre candidat), ou relancer la recherche d'autres candidats.

L'utilisateur peut également estimer que les importances relatives accordées aux différences sont disproportionnées, et ce à un niveau global, ou dans le contexte d'un graphe (état ou transition) particulier. Cette discordance peut provenir du fait que l'heuristique implantée est trop pauvre, mais également des préférences ou de la tâche de l'utilisateur. Par exemple, ce dernier peut préférer un type d'outil (dans l'application) favorisant un certain type d'adaptation, et influant donc subjectivement sur l'adaptabilité des épisodes. Il est donc souhaitable que l'utilisateur puisse influer sur les fonctions w_f et w_g, mais les modalités de cette influence sont loin d'être triviales : accéder individuellement à chaque différence serait beaucoup trop laborieux. Une alternative consisterait à altérer w_f et w_g automatiquement lorsque le graphe source choisi par l'utilisateur n'est pas celui ayant la similarité la plus élevée avec le graphe cible. Cependant, la façon dont les pondérations devraient alors être altérées, constitue en soi un nouveau problème.

8.2.3 Signatures de tâche

Un certain nombre de questions ont été soulevées dans le cadre du projet ARDECO, auxquelles des éléments de réponse peuvent être apportés par la notion de signature de tâche, introduite dans le modèle MUSETTE. ARDECO exploite en effet une unique signature de tâche très générale (cf. section 7.3). Ces pistes de recherche, impliquant d'autres signatures de tâches pour ARDECO, sont exposées dans cette section.

Tout d'abord, le début du chapitre 5 introduit la notion de *contexte d'application* d'un épisode. Dans le prototype, seul l'état initial de l'épisode est considéré comme son contexte d'application, et tous les éléments de ces états sont pris en compte. Cela découle directement de la façon dont

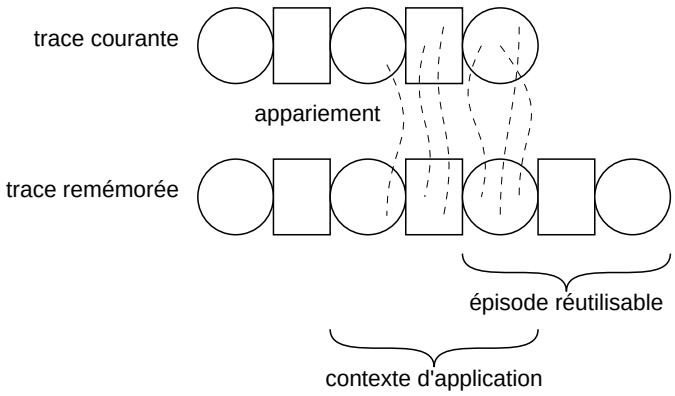


FIG. 8.1 – Contexte d'application complexe pour un épisode

les épisodes sont construits : par leur état initial et leur état final repérés grâce aux indices comportementaux. On imagine cependant que certains éléments de l'état initial sont plus pertinents que d'autres pour expliquer l'épisode, et que certains éléments des états et transitions antérieurs à l'état initial sont également importants (cf. figure 8.1). Déterminer avec plus de précision les éléments qui constituent le contexte d'application d'un épisode, revient en fait à disposer de signatures de tâches plus précises pour délimiter ces épisodes.

La section 4.2.2 souligne par ailleurs que des objectifs complexes, décomposés en objectifs plus simple, sont masqués par ces derniers dans la détection d'épisodes. Dans le cas contraire, les objectifs complexes comportant des sous-objectifs donneraient lieu à des épisodes « abstraits », couvrant plusieurs épisodes. Ici encore, des signatures de tâche permettant d'identifier les invariants des épisodes abstraits pourraient être employés en parallèle du mécanisme de détection actuel.

Enfin, on peut imaginer un type particulier d'épisodes : les *épisodes de réutilisation*, dont la particularité est d'être issus de l'adaptation d'un épisode remémoré. À l'instar de la signature de tâche utilisée dans ARDECO, la signature de tâche donnant lieu aux épisodes de réutilisation est indépendante de toute modèle d'utilisation. Ces épisodes présentent notamment un intérêt pour les adaptations ultérieures du même épisode ou d'un épisode similaire, à la manière des cas d'adaptation proposés par Leake *et al.* [1997]. En modifiant la procédure d'adaptation décrite au chapitre 5 pour exploiter les épisodes de réutilisation, on pourrait donc réutiliser *l'expérience d'adaptation d'épisode*.

Annexes

Annexe A

Notations, définitions et démonstrations

A.1 Rappels et notations mathématiques

Les opérateurs $\wedge$, $\vee$, $\Rightarrow$ et $\Leftrightarrow$ représentent respectivement le *et* logique, le *ou* logique, l'*implication* logique et l'*équivalence* logique. Les symboles $\forall$ et $\exists$ représentent respectivement le quantificateur universel et le quantificateur existentiel.

$\emptyset$ est l'ensemble vide, $\mathbb{N}$ est l'ensemble des nombres entiers naturels, $\mathbb{R}$ est l'ensemble des nombres réels et $\mathbb{R}^+$ est l'ensemble des nombres réels positifs ou nuls.

Si A et B sont deux ensembles, $x \in A$ signifie que x est un élément de A, $A \subseteq B$ signifie que A est un sous-ensemble de B. $A \cup B$ est l'union de ces ensembles, $A \cap B$ en est l'intersection, $A - B$ est la différence ensembliste (ensemble des éléments de A n'appartenant pas à B) et $A \times B$ est le produit cartésien de A par B. $|A|$ est le cardinal de A, c'est-à-dire le nombre de ses éléments. $\wp(A)$ est l'ensemble de toutes les parties (sous-ensembles) de A.

Une *partition* d'un ensemble A est un ensemble de parties de A telles que toutes ces parties sont disjointes (leurs intersections deux à deux sont nulles) et que leur union est égale à A. En d'autres termes, tout élément de A appartient à une et une seule de ces parties.

A.1.1 Relations binaires

Une *relation binaire* R sur $A \times B$ est un sous-ensemble du produit cartésien $A \times B$. Dans le cas particulier ou $B = A$, on parle simplement d'une relation binaire sur A. Une telle relation peut posséder une ou plusieurs des propriétés suivantes :

réflexive $\forall x \in A,\ (x, x) \in R$

symétrique $\forall x, y \in A,\ (x, y) \in R \Rightarrow (y, x) \in R$

antisymétrique $\forall x, y \in A,\ (x, y) \in R \wedge (y, x) \in R \Rightarrow x = y$

transitive $\forall x, y, z \in A,\ (x, y) \in R \wedge (y, z) \in R \Rightarrow (x, z) \in R$

Une relation réflexive et transitive est appelée un *pré-ordre*. Un pré-ordre est dit *partiel* s'il existe des couples d'éléments x et y tels que le pré-ordre ne contienne ni (x, y) ni (y, x). En d'autres termes, il existe des éléments *non comparables* selon le pré-ordre. Dans le cas contraire, le pré-ordre est dit *total*.

A.1.2 Multi-ensembles

La notion de multi-ensemble est une extension de celle d'ensemble : un multi-ensemble peut contenir plusieurs occurrences du même élément. On peut donc voir un multi-ensemble comme un ensemble de couples $\langle x, n\rangle$ ou x est un élément du multi-ensemble, et n est un entier strictement positif représentant le nombre d'occurrences de x dans le multi-ensemble. On notera Pour faciliter les notations, on étend cette définition en convenant que $\langle x, 0\rangle$ est élément de tout multi-ensemble ne contenant pas x.

Soit deux multi-ensembles A et B, on peut redéfinir l'appartenance, l'union et l'intersection ainsi :

- $\langle x, n\rangle \in A$ signifie que A contient *exactement* n occurrences de x (ou n peut être égal à 0)
- $A \sqsubseteq B \Leftrightarrow \forall \langle x, n\rangle \in A,\ \langle x, p\rangle \in B \wedge n \leq p$
- $A \sqcup B \doteq \{\langle x, n\rangle |\ \langle x, p\rangle \in A, \langle x, q\rangle \in B, n = p + q\}$

On remarquera que l'union de deux multi-ensemble, telle qu'elle est définie ici, présente des différences remarquables avec l'union ensembliste classique : notamment, $A \sqcup A \neq A$. Ceci est dû au fait que des éléments identiques appartenant aux deux opérandes ne sont pas considérés comme *communs*, puisque leurs cardinalités sont ajoutées. Selon ce point de vue, l'intersection et la différence ensembliste peuvent également être définies sur les multi-ensembles, mais de façon triviale ($\forall A, B, A \sqcap B = \emptyset, A - B = A$). Elles ne sont de toutes manières pas utilisées dans ce mémoire.

A.1.3 Graphes

Un graphe orienté G est défini par la structure $\langle V, A\rangle$ où :

- S est l'ensemble fini des sommets du graphe,
- $A \subseteq E \times E$ est l'ensemble des arcs du graphes.

Le sous-graphe G' de G engendré par $T \subseteq S$ est $G' = \langle T, A'\rangle$ où A' est la restriction de A à l'ensemble T, c'est-à-dire $A' = A \cap (T \times T)$.

Étant donnés deux graphes $G_1 = \langle V_1, A_1\rangle$ et $G_2 = \langle V_2, A_2\rangle$, le problème de l'*isomorphisme* consiste à trouver une bijection $f : V_1 \to V_2$ telle que :

$$\forall (x_1, x_2) \in V_1 \times V_1,\ (x_1, x_2) \in A_1 \Leftrightarrow (f(x_1), f(x_2)) \in A_2$$

Si une telle bijection existe, on dit que G_1 est isomorphe à G_2 et réciproquement. Le problème de l'*isomorphisme de sous-graphe* consiste à trouver un sous-graphe de G_2 isomorphe à G_1.

A.2 Démonstrations

A.2.1 *diff* et *mult* sont monotones par rapport à l'inclusion

Section 5.1.3, page 61 :

Considérons tout d'abord deux appariements m_1 et m_2 ne différant que par un seul couple :

$$m_2 = m_1 \cup \{(v_1, v_2)\}$$

En d'autres termes, $m_2(v_1) = m_1(v_1) \cup \{v_2\}$ et $m_2(v_2) = m_1(v_2) \cup \{v_1\}$. Pour tout autre sommet v, m_1 et m_2 sont identiques, et donc

$$\forall v \in V_1 \cup V_2 \begin{cases} m_1(v) \subset m_2(v) & \text{si } v \in \{v_1, v_2\} \\ m_1(v) = m_2(v) & \text{sinon} \end{cases}$$

Il en découle que les arcs appariés par m_1 le sont également par m_2, et donc que pour tout arc a, $m_1(a) \subseteq m_2(a)$. Plus précisément, deux arcs a_1 et a_2 non appariés par m_1 (et donc tels que $a_2 \notin m_1(a_1)$) seront appariés par m_2 si et seulement si

$$\begin{array}{rcl} \exists (w_1, w_2) \in m_1 & \text{tels que} & a_1 = (v_1, w_1),\ a_2 = (v_2, w_2) \\ & \text{ou} & a_1 = (w_1, v_1),\ a_2 = (w_2, v_2) \end{array}$$

En revenant à la définition de diff_m, il apparaît que cet ensemble ne peut que décroître en passant de m_1 à m_2, puisque les ensembles $m(v)$ et $m(a)$ vont en augmentant, et donc que les possibilités de trouver une correspondance entre caractéristiques de sommets ou d'arcs se font plus nombreuses. On voit bien que l'ajout d'un couple n'ajoute pas de différence (les correspondances trouvées ne sont pas remises en cause) et permet même d'en faire disparaître (de nouvelles correspondances peuvent être trouvées). Donc $\mathit{diff}_{m_1}(\mathcal{G}_1, \mathcal{G}_2) \supseteq \mathit{diff}_{m_2}(\mathcal{G}_1, \mathcal{G}_2)$.

En ce qui concerne les ensembles mult_m, on a vu que $m_1(v) \subseteq m_2(v)$ pour tout sommet v. Donc les cardinalités des éléments v de mult_{m_1} sont inférieures ou égales aux cardinalités des éléments v de mult_{m_2}, d'où il découle que $\mathit{mult}_{m_1}(\mathcal{G}_1, \mathcal{G}_2) \sqsubseteq \mathit{mult}_{m_2}(\mathcal{G}_1, \mathcal{G}_2)$.

Par induction, puisque ces propriétés sont vraies lorsque m_1 et m_2 diffèrent par un seul couple, elles sont également vraies dans le cas général ou $m_1 \subseteq m_2$.
□

A.2.2 L'algorithme glouton a une complexité polynomiale

Section 5.1.3, page 62 :

À chaque itération dans la boucle «répéter», un couple est retiré de *candidats* pour être ajouté à m. Comme *candidats* est initialisé à $V_1 \times V_2$, on passera dans cette boucle $|V_1| \times |V_2|$ fois au plus (au plus, car on peut en sortir avant d'avoir épuisé *candidats*, si aucun couple ne permet d'améliorer le coût de m).

À l'intérieur de la boucle «répéter», on construit l'ensemble *couples*, qui est un sous ensemble de *candidats*. Cette construction ne nécessite qu'une seule passe sur *candidats*, elle peut donc s'effectuer en $O(|V_1| \times |V_2|)$. Il en va de même pour la fonction ChoisirCouple, qui n'a besoin que d'une seule passe sur *couples* (qui peut, dans le pire des cas, contenir tous les éléments de *candidats*).

Il en résulte que la fonction RechercheGloutonne a une complexité de $O((|V_1| \times |V_2|)^2)$, soit $O(n^4)$ si n est la taille du plus grand des deux graphes.

On notera que la construction de l'ensemble *couples* réclame de calculer les coûts des appariements. Dans l'implantation de l'algorithme qui a été faite, le coût de $m \cup \{c_i\}$ peut se calculer à partir du coût de m, et ce en un temps linéaire par rapport au nombre de caractéristique possédées par c_i. Ce nombre ne dépend pas du nombre de sommets dans le graphe et peut être borné par une constante.
□

A.3 Discussions

A.3.1 Définition universelle des ressemblances

Section 5.1.2, page 57, la ressemblance entre deux graphes par rapport à un appariement est définie par :

$$\mathit{ress}_m(\mathcal{G}_1, \mathcal{G}_2) \quad \doteq \quad \{(v, \phi) \in d_V(\mathcal{G}_1) |\ m(v) \neq \emptyset \wedge \exists v' \in m(v),\ (v', \phi) \in d_V(\mathcal{G}_2)\}$$

$$\begin{aligned} \cup \quad & \{(a,\phi) \in d_A(\mathcal{G}_1) |\ m(a) \neq \emptyset \wedge \exists a' \in m(a),\ (a',\phi) \in d_A(\mathcal{G}_2)\} \\ \cup \quad & \{(v,\phi) \in d_V(\mathcal{G}_2) |\ m(v) \neq \emptyset \wedge \exists v' \in m(v),\ (v',\phi) \in d_V(\mathcal{G}_1)\} \\ \cup \quad & \{(a,\phi) \in d_A(\mathcal{G}_2) |\ m(a) \neq \emptyset \wedge \exists a' \in m(a),\ (a',\phi) \in d_A(\mathcal{G}_1)\} \end{aligned}$$

On notera que la définition donnée page 57 ne comporte pas les termes $m(x) \neq \emptyset$, qui sont en effet superflus : ils découlent des termes suivants, stipulant l'existence d'un élément dans $m(x)$. La distinction entre les deux termes est toutefois importante :

- $m(x) \neq \emptyset$ signifie que la présence d'un appariement est nécessaire à l'identification de ressemblances entre graphes,
- $\exists x' \in m(x)$... exprime les modalités selon lesquelles la ressemblance est établie.

Cette définition *existentielle* des ressemblances admet justement une définition duale, dite *universelle* :

$$\begin{aligned} ress_m(\mathcal{G}_1,\mathcal{G}_2) \quad \doteq \quad & \{(v,\phi) \in d_V(\mathcal{G}_1) |\ m(v) \neq \emptyset \wedge \forall v' \in m(v),\ (v',\phi) \in d_V(\mathcal{G}_2)\} \\ \cup \quad & \{(a,\phi) \in d_A(\mathcal{G}_1) |\ m(a) \neq \emptyset \wedge \forall a' \in m(a),\ (a',\phi) \in d_A(\mathcal{G}_2)\} \\ \cup \quad & \{(v,\phi) \in d_V(\mathcal{G}_2) |\ m(v) \neq \emptyset \wedge \forall v' \in m(v),\ (v',\phi) \in d_V(\mathcal{G}_1)\} \\ \cup \quad & \{(a,\phi) \in d_A(\mathcal{G}_2) |\ m(a) \neq \emptyset \wedge \forall a' \in m(a),\ (a',\phi) \in d_A(\mathcal{G}_1)\} \end{aligned}$$

La motivation pour la première définition, celle utilisée dans ce travail, est que plusieurs éléments (sommets ou arcs) d'un graphe peuvent se partager le rôle tenu par un élément unique dans un autre graphe. C'est par exemple le cas des supports des poutres dans l'exemple de la figure 5.1, page 53. Dans cette optique, il suffit que l'*ensemble* des éléments appariés à un élément x possède les mêmes caractéristiques que x, et donc que chaque caractéristique existe pour au moins un élément apparié. Cette approche tend bien sûr à apparier chaque élément à beaucoup d'autres, afin de «satisfaire» le plus de caractéristiques possibles. C'est pourquoi les appariements multiples sont sanctionnés par ailleurs à l'aide de la fonction $mult_m$.

Dans l'approche universelle, on part du principe que *chaque* élément apparié à x doit posséder les mêmes caractéristiques que x, et donc que chaque caractéristique doit exister pour tous les éléments appariés. Un appariement multiple ne rend donc plus compte du fait que le rôle tenu par x est tenu par un ensemble d'éléments ; il signifie plutôt que plusieurs éléments sont *individuellement* similaires à x.

On peut remarquer que les critères de l'approche existentielle sont logiquement impliqués par ceux de l'approche universelle. La seconde est donc moins tolérante, ce qui se traduit par son incapacité à rendre compte de la ressemblance entre les deux assemblages de la figure 5.1. En l'absence d'un contexte applicatif précis justifiant les restrictions apportées par l'approche universelle, on préférera donc l'approche existentielle étudiée dans ce mémoire. Il convient également de noter que l'approche universelle ne possède pas la propriété de monotonie mise en évidence page 61. Elle nécessite donc d'autres algorithmes que ceux développés dans le cadre de ce travail.

Annexe B

Schémas RDF, Interfaces JAVA

B.1 Représentation d'épisodes

B.1.1 Schéma RDF-Schema

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE rdf:RDF [
 <!ENTITY rdf  'http://www.w3.org/1999/02/22-rdf-syntax-ns#'>
 <!ENTITY rdfs 'http://www.w3.org/2001/01/rdf-schema#'>
 <!ENTITY ep   'http://www710.univ-lyon1.fr/~champin/episode/0.1/'>
]>

<rdf:RDF xml:lang="en"
         xmlns="#"
         xmlns:rdf="&rdf;"
         xmlns:rdfs="&rdfs;"
>

<!--================================

  Vocabulaire de description des traces

  ================================-->

<rdfs:Class rdf:about="&ep;Trace"/>

<!--
 la classe suivante ne sert qu'a definir le domaine et la portee de ep:next
-->
<rdfs:Class rdf:about="&ep;StateOrTransition"/>

<rdfs:Class rdf:about="&ep;State">
  <rdfs:subClassOf rdf:resource="&ep;StateOrTransition"/>
</rdfs:Class>

<rdfs:Class rdf:about="&ep;Transition">
```

```
  <rdfs:subClassOf rdf:resource="&ep;StateOrTransition"/>
</rdfs:Class>

<rdf:Property rdf:about="&ep;first">
  <rdfs:domain rdf:resource="&ep;Trace"/>
  <rdfs:range rdf:resource="&ep;State"/>
</rdf:Property>

<rdf:Property rdf:about="&ep;next">
  <rdfs:domain rdf:resource="&ep;StateOrTransition"/>
  <rdfs:range rdf:resource="&ep;StateOrTransition"/>
</rdf:Property>

<!--=======================================

  Vocabulaire de description des transitions

  =======================================-->

<rdfs:Class rdf:about="&ep;DiffProperty">
  <rdfs:subClassOf rdf:resource="&rdf;Property"/>
</rdfs:Class>

<rdfs:Class rdf:about="&ep;MinusProperty">
  <rdfs:subClassOf rdf:resource="&ep;DiffProperty"/>
</rdfs:Class>

<rdfs:Class rdf:about="&ep;PlusProperty">
  <rdfs:subClassOf rdf:resource="&ep;DiffProperty"/>
</rdfs:Class>

<rdf:Property rdf:about="&ep;diffOf">
  <rdfs:domain rdf:resource="&ep;DiffProperty"/>
  <rdfs:range rdf:resource="&rdf;Property"/>
</rdf:Property>

<!--====================

  Relation de composition

  ====================-->

<rdf:Property rdf:about="&ep;component"/>

</rdf:RDF>
```

B.1.2 Modèle d'utilisation exemple

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE rdf:RDF [
 <!ENTITY rdf  'http://www.w3.org/1999/02/22-rdf-syntax-ns#'>
 <!ENTITY rdfs 'http://www.w3.org/2001/01/rdf-schema#'>
 <!ENTITY ep   'http://www710.univ-lyon1.fr/~champin/episode/0.1/'>
 <!ENTITY ds   'http://www.dsweb.com/ardeco/'>
 <!ENTITY type 'http://www.dsweb.com/ardeco/Type/'>
]>

<rdf:RDF xml:lang="en"
         xmlns="#"
         xmlns:rdf="&rdf;"
         xmlns:rdfs="&rdfs;"
>

<!--======

  Produits

  ======-->

<rdfs:Class rdf:about="&type;Product"/>


<rdfs:Class rdf:about="&type;Assembly">
  <rdfs:subClassOf rdf:resource="&type;Product"/>
</rdfs:Class>

<rdfs:Class rdf:about="&type;Part">
  <rdfs:subClassOf rdf:resource="&type;Product"/>
</rdfs:Class>

<rdf:Property rdf:about="&ds;proChild">
  <rdfs:subPropertyOf rdf:resource="&ep;component"/>
  <rdfs:domain rdf:resource="&type;Assembly"/>
  <rdfs:range rdf:resource="&type;Product"/>
</rdf:Property>


<!--=========

  Contraintes

  =========-->

<rdfs:Class rdf:about="&ds;Cst"/>
```

```
<rdfs:Class rdf:about="&ds;Cst/Type/Angle">
  <rdfs:subClassOf rdf:resource="&ds;Cst"/>
</rdfs:Class>

<rdfs:Class rdf:about="&ds;Cst/Type/On">
  <rdfs:subClassOf rdf:resource="&ds;Cst"/>
</rdfs:Class>

<rdfs:Class rdf:about="&ds;Cst/Type/On">
  <rdfs:subClassOf rdf:resource="&ds;Cst"/>
</rdfs:Class>


<rdfs:Class rdf:about="&ds;Cst/Status/Valid">
  <rdfs:subClassOf rdf:resource="&ds;Cst"/>
</rdfs:Class>

<rdfs:Class rdf:about="&ds;Cst/Status/Invalid">
  <rdfs:subClassOf rdf:resource="&ds;Cst"/>
</rdfs:Class>


<rdf:Property rdf:about="&ds;connection">
  <rdfs:subPropertyOf rdf:resource="&ep;component"/>
  <rdfs:domain rdf:resource="&type;Assembly"/>
  <rdfs:range rdf:resource="&ds;Cst"/>
</rdf:Property>

<rdf:Property rdf:about="&ds;connector">
  <rdfs:domain rdf:resource="&ds;Cst"/>
  <rdfs:range rdf:resource="&type;Assembly"/>
</rdf:Property>


<!--========

  Parametres

  ========-->

<rdfs:Class rdf:about="&ds;Parameter"/>

<rdfs:Class rdf:about="&ds;CkeType/LENGTH"/>
<rdfs:Class rdf:about="&ds;CkeType/ANGLE"/>


<rdf:Property rdf:about="&ds;parameter">
```

```
  <rdfs:subPropertyOf rdf:resource="&ep;component"/>
  <rdfs:domain rdf:resource="&type;Assembly"/>
  <rdfs:range rdf:resource="&ds;Parameter"/>
</rdf:Property>

<!--========

  Parametres

  ========-->

<rdfs:Class rdf:about="&ds;Relation"/>

<rdf:Property rdf:about="&ds;relation">
  <rdfs:subPropertyOf rdf:resource="&ep;component"/>
  <rdfs:domain rdf:resource="&type;Assembly"/>
  <rdfs:range rdf:resource="&ds;Relation"/>
</rdf:Property>

<rdf:Property rdf:about="&ds;CkeRelation/in">
  <rdfs:domain rdf:resource="&ds;Relation"/>
  <rdfs:range rdf:resource="&ds;Parameter"/>
</rdf:Property>

<rdf:Property rdf:about="&ds;CkeRelation/out">
  <rdfs:domain rdf:resource="&ds;Relation"/>
  <rdfs:range rdf:resource="&ds;Parameter"/>
</rdf:Property>

</rdf:RDF>
```

B.2 API d'extension du gestionnaire d'épisodes

B.2.1 Interface `GraphStorage`

```
/*
 * GraphStorage.java
 *
 * Created on January 20, 2002, 4:15 PM
 */

package graph;

import java.util.Collection;

/**
 * A common interface for any system of storage for graphs.
 * <br>Though this interface provides methods for transaction management,
```

```
 * it is not a requirement for implementations to implement them.
 * <br>Note also that it is not required that the GraphModel implementations
 * returned by the <code>get</code> method be the same as the one passed to
 * the <code>store</code> method. Hence some specific properties of that
 * implementation (like order conservation in the collections of nodes or arcs)
 * might be lost in the process.
 *
 * @author  pa
 * @version 0.1
 */
public interface GraphStorage {
    /**
     * Return a Transaction.
     * @see graph.GraphStorage.Transaction
     */
    Transaction beginTransaction() throws GraphStorage.Exception;

    /**
     * Return the stored graph for the given ID.
     * @param graphId the ID of the graph to get
     * @return a graph model or <code>null</code> if the ID is not used.
     */
    GraphModel get(String graphId) throws GraphStorage.Exception;

    /**
     * Return a collection of the used graph IDs.
     * This collection is not bound to be kept up to date when the storage is
     * modified.
     */
    Collection getIDs() throws GraphStorage.Exception;

    /**
     * Store a graph in this storage.
     * This method must not be used for update:
     * if <code>graphId</code> is already in use, an error must occur.
     * @param graphId an ID which is not already in use
     * @param graph the graph to be stored
     */
    void store(String graphId, GraphModel graph) throws GraphStorage.Exception;

    /**
     * Updates a graph in this storage.
     * if <code>graphId</code> is not in use, an error must occur.
     * @param graphId the ID of an existing graph
     * @param graph the new graph to be stored
     */
    void update(String graphId, GraphModel graph) throws GraphStorage.Exception;
```

```
/**
 * Remove a graph from the storage.
 * No behaviour is specified if the ID is not in use.
 */
void remove(String graphId) throws GraphStorage.Exception;

/**
 * The finalizer of a GraphStorage may contain useful things,
 * like closing remaining connections to a database, fo example.
 * So users of this interface SHOULD call it once, just in case...
 * The drawback for implementors is that this method MUST be overriden
 * in order to be public, even if they do not have anything to do in it.
 */
public void finalize() throws Throwable;


/**
 * Provide transaction management when the underlying storage system
 * supports it. A transaction provides the same access methods as the
 * GraphStorage (<code>getIDs get store remove</code>) but their effect
 * must be either commited or cancelled with the appropriate method.
 *
 * <p>In case the underlying storage system has no transaction and the
 * implementation does not emulate it, the <code>cancel</code> method
 * MUST throw a GraphStorage.Exception.
 */
public interface Transaction
{
    /**
     * Return a collection of the used graph IDs.
     * This collection is not bound to be kept up to date when the storage
     * is modified.
     */
    Collection getIDs() throws GraphStorage.Exception;

    /**
     * Return the stored graph for the given ID.
     * @param graphId the ID of the graph to get
     * @return a graph model or <code>null</code> if the ID is not used.
     */
    GraphModel get(String graphId) throws GraphStorage.Exception;

    /**
     * Store a graph in this storage.
     * This method must not be used for update:
     * if <code>graphId</code> is already in use, an error must occur.
     * @param graphId an ID which is not already in use
     * @param graph the graph to be stored
```

```
     */
    void store(String graphId, GraphModel graph)
        throws GraphStorage.Exception;

    /**
     * Updates a graph in this storage.
     * if <code>graphId</code> is not in use, an error must occur.
     * @param graphId the ID of an existing graph
     * @param graph the new graph to be stored
     */
    void update(String graphId, GraphModel graph)
        throws GraphStorage.Exception;

    /**
     * Remove a graph from the storage.
     * No behaviour is specified if the ID is not in use.
     */
    void remove(String graphId) throws GraphStorage.Exception;

    /**
     * Transaction management.
     * After invoking this metod,
     * no other method of the Transaction object should be usable.
     * May be silently unimplemented.
     */
    void commit() throws GraphStorage.Exception;

    /**
     * Transaction management.
     * After invoking this metod,
     * no other method of the Transaction object should be usable.
     * May be unimplemented and then throw an exception.
     */
    void cancel() throws GraphStorage.Exception;
}


/**
 * A GraphStorage.Exception may contain a message or another exception.
 * This allow various implementations to embed theor specific exceptions
 * inside an homogeneous class.
 */
public static class Exception extends java.lang.Exception
{
    private java.lang.Exception exception;

    /**
     * Create a new RDFException.
```

```
 *
 * @param message The error or warning message.
 * @see org.xml.sax.Parser#setLocale
 */
public Exception (String message) {
    super(message);
    this.exception = null;
}

/**
 * Create a new RDFException wrapping an existing exception.
 *
 * <p>The existing exception will be embedded in the new
 * one, and its message will become the default message for
 * the RDFException.</p>
 * <p>If the embeded exception is a SAXParseException,
 * localization data will be added to its message.</p>
 *
 * @param e The exception to be wrapped in a RDFException.
 */
public Exception (java.lang.Exception e)
{
    super(e.getMessage());
    this.exception = e;
}

/**
 * Return the embedded exception, if any.
 *
 * @return The embedded exception, or null if there is none.
 */
public java.lang.Exception getException ()
{
    return exception;
}

/**
 * Overrides stackTrace so that it trace the embeded exception.
 */
public void printStackTrace()
{
    if (exception!=null) exception.printStackTrace();
    else super.printStackTrace();
}

/**
 * Overrides stackTrace so that it trace the embeded exception.
 */
```

```
        public void printStackTrace(java.io.PrintStream ps)
        {
            if (exception!=null) exception.printStackTrace(ps);
            else super.printStackTrace(ps);
        }

        /**
         * Overrides stackTrace so that it trace the embeded exception.
         */
        public void printStackTrace(java.io.PrintWriter pw)
        {
            if (exception!=null) exception.printStackTrace(pw);
            else super.printStackTrace(pw);
        }
    }
}
```

B.2.2 Interface SimilarityEngine

```
/*
 * SimilarityEngine.java
 *
 * Created on January 18, 2002, 5:18 PM
 */

package graph.similarity;

import graph.*;

/**
 * An interface for similarity implementation.
 * A <em>session</em> is dedicated to compare a <i>target</i> graph to a number
 * of <i>source</i> graphs. Each source graph corresponds to a
 * <i>comparison</i>.
 *
 * @author  pa
 * @version 0.1
 */
public interface SimilarityEngine {

    /**
     * Shortcut method to tru a single comparison.
     * @param g1 the target graph
     * @param g2 the source graph
     * @param weights the weights to assign to specific differences
     */
    public Mapping getBestMatch(GraphModel target, GraphModel source,
```

```
                                DiffWeights weights);

    /**
     * Submit a target graph.
     * @param target the target graph
     */
    public Session openSession(GraphModel target);


    /**
     * A session is a sequence of comparisons involving a given graph.
     */
    public interface Session
    {
        /**
         * Submit a source graph, to be compared to the current target graph.
         * @param source the source graph
         */
        public Comparison openComparison(GraphModel source);

        /**
         * Must be invoked for freeing the resources involved in the session.
         */
        public void close();
    }


    /**
     * A comparison is involving two graphs.
     * The result of the comparison can be computed several times with
     * a different DiffWeights each time.
     */
    public interface Comparison
    {
        /**
         * Return the best match and its evaluation, given a set of weights.
         * @param weights the weights to assign to specific differences.
         */
        public Mapping getBestMatch(DiffWeights weights);

        /**
         * Must be invoked for freeing the resources involved in the comparison.
         */
        public void close();
    }
}
```

Bibliographie

Agnar Aamodt et Enric Plaza. Case-Based Reasoning : Foundational Issues, Methodological Variations, and System Approaches. *AI Communications*, 7(1) :39–59, mars 1994.

Christopher Alexander. *Notes on the Synthesis of Form*. Harvard University Press, Cambridge, MA (US), 1964.

Joachim Altmeyer, Stefan Ohnsorge, et Bernd Schürmann. Reuse of Design Objects in CAD Frameworks. Dans *International Conference on Computer Aided Design*, San Jose, CA (US), novembre 1994.

Bruno Bachimont. Bilbiothèques numériques audiovisuelles : des enjeux scientifiques et techniques. *Document Numérique, Numéro Spécial «Les Bibliothèques Numériques»*, 2(3–4) :219–242, 1999.

Bruno Bachimont, éditeur. *Journées française d'Ingénierie des Connaissances*, Institut National des Sciences Appliquées, Rouen (FR), mai 2002.

Dave Beckett, éditeur. RDF/XML Syntax Specification (Revised). Working draft, World Wide Web Consortium, mars 2002.
URL http://www.w3.org/TR/rdf-syntax-grammar.

Ralph Bergmann, Michael M. Richter, Sascha Schmitt, Armin Stahl, et Ivo Vollrath. Utility-Oriented Matching : A New Research Direction for Case-Based Reasoning. Dans Vollrath *et al.* [2001], pages 264–274.

Ralph Bergmann, Ivo Vollrath, et Thomas Wahlmann. Generalized Cases and their Application to Electronic Designs. Dans *7th German Workshop on Cased-Based Reasoning*, 1999.

Tim Berners-Lee. Notation 3 – Ideas about Web architechture. Document Web, 1999.
URL http://www.w3.org/DesignIssues/Notation3.

Tim Berners-Lee, R. Fielding, U.C. Irvine, et L. Masinter. Uniform Resource Identifiers (URI) : Generic Syntax. RFC 2396, Internet Engineering Task Force, août 1998.
URL http://www.ietf.org/rfc/rfc2396.txt.

Tim Berners-Lee et Mark Fischetti. *Weaving the Web*. Harper San Fransisco, San Fransisco, CA (US), 1999.

Bernadette Bouchon-Meunier, Maria Rifqi, et Sylvie Bothorel. Towards general measures of comparison of objects. *Fuzzy sets and systems*, 84(2) :111–116, 1996.

Patrick Bougé. . Thèse de doctorat en psychologie, Université Paris VIII, St Denis (FR), 2003. À paraître.

Patrick Bougé, Françoise Détienne, et Laurent Di Cesare. Épisodes de conception : une étude ergonomique pour le recueil de «cas». Dans Jean Charlet, éditeur, *Journées française d'Ingénierie des Connaissances*, pages 79–94, Grenoble (FR), juin 2001. Presses Universtaires de Grenoble, Grenoble (FR).

Tim Bray, Dave Hollander, et Andrew Layman, éditeurs. Namespaces in XML. Recommendation, World Wide Web Consortium, janvier 1999.
URL http://www.w3.org/TR/REC-xml-names.

Tim Bray, Jean Paoli, et C.M. Sperberg-McQueen, éditeurs. Extensible Markup Language (XML) 1.0. Recommendation, World Wide Web Consortium, février 1998.
URL http://www.w3.org/TR/REC-xml.

Dan Brickley et R.V. Guha, éditeurs. Resource Description Framework (RDF) Schema Specification 1.0. Working draft, World Wide Web Consortium, 2001.
URL http://www.w3.org/2001/sw/RDFCore/Schema/20010913/.

Peter Brusilovsky. Adaptive Hypermedia. *User Modeling and User-Adapted Interaction*, 11 : 87–110, 2001.

John M. Carroll. Dimensions of Participation, Elaborating Herbert Simon's "Science of Design". Dans Perrin [2002].

Pierre-Antoine Champin et Yannick Prié. Modéliser l'utilisateur ou l'utilisation ? Dans Garlatti et Crampes [2002], pages 97–102.
URL http://iasc.enst-bretagne.fr/DVP02/Articles/Actes_DVP2002-web.zip.

Pierre-Antoine Champin et Yannick Prié. Musette *: uses-based annotation for the Semantic Web*. Dans Handschuh [2003], 2003. À paraître.

Balakrishnan Chandrasekaran et John R. Josephson. Representing Function as Effect. Dans Mohammed Modarres, éditeur, *Functional Modeling Workshop*, EDF, Paris (FR), 1997.

Michelene T.H. Chi, P.J. Feltovitch, et Robert Glaser. Categorization of physics problems by experts and novices. *Cognitive Science*, 5 :121–152, 1981.

James Clark, éditeur. XSL Transformations (XSLT). Recommendation, World Wide Web Consortium, novembre 1999.
URL http://www.w3.org/TR/REC-xslt.

Wolfram Conen, Reinhold Klapsing, et Eckhart Köppen. *RDF M&S revisited : From reification to nesting, from containers to lists, from dialect to pure XML*. Dans Cruz *et al.* [2002], 2002.

Dan Connolly, Frank van Harmelen, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, et Lynn Andrea Stein. DAML+OIL (March 2001) Reference Description. Note, World Wide Web Consortium, décembre 2001.
URL http://www.w3.org/TR/daml+oil-reference.

Isabel F. Cruz, Stefan Decker, Jérôme Euzenat, et Deborah L. McGuinness, éditeurs. *The Emerging Semantic Web*, volume 75 de *Frontiers in Artificial Intelligence and Applications*. IOS Press, Amsterdam (NL), 2002.

Paul DE BRA. Adaptive Hypertext & Hypermedia. Conférence invitée, juillet 2002.
URL http://wwwis.win.tue.nl/ah/.

Stefan DECKER, Michael ERDMANN, Dieter FENSEL, et Rudi STUDER. Ontobroker : Ontology Based Access to Distributed and Semi-Structured Information. Dans Robert MEERSMAN , Zahir TARI , et Scott M. STEVENS, éditeurs, *Database Semantics - Semantic Issues in Multimedia Systems, IFIP TC2/WG2.6*, volume 138, pages 351–369, Rotorua (NZ), janvier 1999. Kluwer Academic Publisher, Dordrecht (NL).

Rose DIENG-KUNTZ, Olivier CORBY, Fabien GANDON, Alain GIBOIN, Joanna GOLEBIOWSKA, Nada MATTA, et Myriam RIBIÈRE. *Méthodes et outils pour la gestion des connaissances : une approche pluridisciplinaire du Knowledge Management*. Dunod, Paris (FR), 2001.

Umberto ECO. *Le nom de la rose (Il nome della rosa)*. Grasset, Paris (FR), 1980. Joël Schifano, traducteur.

Tamara ELSNER et Lucienne BLESSING. Designing products for senior citizens – Experience with a participative approach. Dans Perrin [2002].

Jérôme EUZENAT, éditeur. Research Challenges an Perspectives of the Semantic Web. Workshop report and recommendations, European Commission - US National Science Foundation, octobre 2001.
URL http://www.ercim.org/EU-NSF/Semweb.pdf.

Jérôme EUZENAT et François RECHENMANN. SHIRKA, 10 ans, c'est TROPES ? Dans Amedeo NAPOLI, éditeur, *Langages et modèles à objets*, pages 13–34, LORIA, Nancy, octobre 1995.

David C. FALLSIDE, éditeur. XML Schema Part 0 : Primer. Recommendation, World Wide Web Consortium, mai 2001.
URL http://www.w3.org/TR/xmlschema-0.

Boi FALTINGS et Kun SUN. FAMING : supporting innovative mechanism shape design. *Computer-Aided Design*, 28(3) :207–216, 1996.

Dieter FENSEL, Ian HORROCKS, Frank VAN HARMELEN, Stefan DECKER, Michael ERDMANN, et Michel KLEIN. OIL in a Nutshell. Dans Rose DIENG-KUNTZ et Olivier CORBY, éditeurs, *Knowledge Acquisition, Modeling, and Management*, volume 1937 de *Lecture Notes in Computer Science*, pages 1–16, Juan-les-pins (FR), octobre 2000. Springer Verlag, Berlin (DE).

Jon FERRAIOLO, éditeur. Scalable Vector Graphics (SVG) 1.0 Specification. Recommendation, World Wide Web Consortium, septembre 2001.
URL http://www.w3.org/TR/SVG.

R. FIELDING, U.C. IRVINE, G. GETTYS, J. MOGUL, H. FRYSTYK, L. MASINTER, P. LEACH, et Tim BERNERS-LEE. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, Internet Engineering Task Force, 1999.
URL http://www.ietf.org/rfc/rfc2616.txt.

David FLANAGAN. *Java in a Nutshell*. O'Reilly, Cambridge, MA (US), 2002.

Pierre FLENER, Alan M. FRISCH, Brahim HNICH, Zeynep KIZILTAN, Ian MIGUEL, Justin PEARSON, et Toby WALSH. Breaking Row and Column Symettry in Matrix Models. Dans

Thierry VIDAL et Paolo LIBERATORE, éditeurs, *Starting Artificial Intelligence Researchers Symposium*, volume 78 de *Frontiers in Artificial Intelligence and Applications*, pages 207–216, Lyon (FR), juillet 2002. IOS Press, Amsterdam (NL).

Béatrice FUCHS, Jean LIEBER, Alain MILLE, et Amedeo NAPOLI. Un algorithme pour la phase d'adaptation du raisonnement à partir de cas. Dans Andreas HERZIG, éditeur, *Journées nationales sur les modèles de raisonnement*, pages 79–92, Arras (FR), mai 2001.

Erich GAMMA, Richard HELM, Ralph JOHNSON, et John VLISSIDES. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley, Boston, MA (US), 1995.

Serge GARLATTI et Michel CRAMPES, éditeurs. *Documents Virtuels Personnalisables*, École Nationale Supérieure des Télécommunications de Bretagne, Brest (FR), juillet 2002.
URL http://iasc.enst-bretagne.fr/DVP02/Articles/Actes_DVP2002-web.zip.

Dedre GENTNER et Kenneth D. FORBUS. MAC/FAC : A Model of Similarity-based Retrieval. Dans *Annual Conference of the Cognitive Science Society*, pages 504–509. Cognitive Science Society, Cincinnati, OH (US), 1991.

John S. GERO. Design prototypes : a knowledge representation schema for design. *AI Magazine*, 11(4) :26–36, 1990.

John S. GERO et Udo KANNENGIESSER. The Situated Function-Behaviour-Structure Framework. Dans John S. GERO, éditeur, *International Conference on Artificial Intelligence in Design*, pages 89–104, Cambridge University, Cambridge (GB), juillet 2002. Kluwer Academic Publisher, Dordrecht (NL).

Yolanda GIL, Mark MUSEN, et Jude SHAVLIK, éditeurs. *First International Conference on Knowledge Capture*, Victoria, B.C. (CA), octobre 2001. Association for Computing Machinery, New York, NY (US).

Alain GIRE. *Conception de la conception*, chapitre 3, pages 47–61. Dans Perrin [2001], 2001.

Antony D. GRIFFITHS, Michael D. HARRISON, et Andrew M. DEARDEN. Applying structural similarity to a "knowledge mediation" problem. Dans *UK Case-Based Reasoning Worlkshop*, pages 18–34, Manchester, England (UK), septembre 1999.

Kristian J. HAMMOND. Case-Based Planning : A Framework for Planning from Experience. *Cognitive Science*, 14(3) :385–443, 1990.

Siegfried HANDSCHUH, éditeur. *Annotation for the Semantic Web*. IOS Press, Amsterdam (NL), 2003. À paraître.

Siegfried HANDSHUH, Rose DIENG-KUNTZ, et Steffen STAAB, éditeurs. *Workshop on Knowledge Markup and Semantic Annotation*, Victoria, B.C. (CA), octobre 2001.

Kathleen HANNEY. Learning Adaptation Rules from Cases. Msc thesis, Trinity College, Dublin (IE), 1996.

Patrick HAYES, éditeur. RDF Model Theory. Working draft, World Wide Web Consortium, avril 2002.
URL http://www.w3.org/TR/rdf-mt.

Jeff HEFLIN, James HENDLER, et Sean LUKE. SHOE : A Knowledge Representation Language for Internet Applications. Technical Report CS-TR-4078 (UMIACS TR-99-71), 1999.

Jeff HEFLIN, Raphael VOLZ, et Jonathan DALE, éditeurs. Requirements for a Web Ontology Language. Working draft, World Wide Web Consortium, juillet 2002.
URL http://www.w3.org/TR/webont-req/.

Olivier HERBEAUX et Alain MILLE. *Réutiliser l'expérience en conception : le projet «ACCELERE»*, chapitre 9, pages 159–181. Dans Perrin [2001], 2001.

Douglas L. HINTZMAN. "Schema Abstraction" in a Multiple-Trace Memory Model. *Psychological Review*, 93(4) :411–428, 1986.

Kefeng HUA, Boi FALTINGS, et Ian SMITH. CADRE : Case-Based Geometric Design. *Journal of Artificial Intelligence in Engineering*, 10 :171–183, 1996.

Jean-Mathias HÉRAUD. Pixed : towards the sharing and the re-use of experience to assist training. Dans *World Conference on Educational Multimedia, Hypermedia & Telecomunnication*, Denver, CO (US), juin 2002.

Patrick ION et Robert ROBERT, éditeurs. Mathematical Markup Language (MathML^{TM}) 1.0 Specification. Recommendation, World Wide Web Consortium, avril 1998.
URL http://www.w3.org/TR/REC-MathML.

ISO 8879. Standard Generalized Markup Language (SGML). ISO Standard ISO 8879, International Organization for Standardization, 1986.

Stéphanie JEAN-DAUBIAS. Un système d'assistance au diagnostique de compétences. Dans Catherine GARBAY et Roger MOHR, éditeurs, *Congrès Francophone de Reconnaissance de Formes et d'Intelligence Artificielle*, volume 3, pages 1053–1061, Centre des congrès, Angers (FR), janvier 2002.

Michael KIFER, Georg LAUSEN, et James WU. Logical Foundations of Object-Oriented and Frame-Based Languages. *Journal of the ACM*, 42 :741–843, 1995.

Janet KOLODNER. Reconstructive Memory : A Computer Model. *Cognitive Science*, 7(4) : 281–328, 1983.

Arthur KÖSTLER. *Le cri d'Archimède*. Calmann-Lévy, Paris (FR), 1965.

Pat LANGLEY. User modeling in adaptive interfaces. Dans *International Conference on User Modeling*, pages 357–370, Banff Center, Banff (CA), juin 1999. Springer Verlag, Berlin (DE).

Pat LANGLEY et Michael FEHLING. The Experimental Study of Adaptive User Interfaces. Technical Report 98-3, Institute for the Study of Learning and Expertise, Palo Alto, CA (US), 1998.

David B. LEAKE, Andrew KINLEY, et David WILSON. Case-Based Similarity Assessment : Estimating Adaptability from Experience. Dans *National Conference on Artificial Intelligence*, Rhode Island Convention Center, Providence, RI (US), juillet 1997. American Association for Artificial Intelligence, Menlo Park, CA (US).

Michael LEBOWITZ. Memory-Based Parsing. *Artificial Intelligence*, 21(4) :363–404, 1983.

Jean-Louis LEMOIGNE. Sur l'Épistémologie des Sciences de Conception, Sciences de l'Ingenium – Concevoir des Artefacts Évoluant. Dans Perrin [2002].

Jean LIEBER. Recopier c'est déjà adapter : six types d'adaptation par copie. Dans *Atelier Raisonnement à Partir de Cas*, INSERM, Faculté de Médecine Broussais Hotel Dieu, Paris (FR), mai 2002.

Jean LIEBER et Amedeo NAPOLI. Using Classification in Case-Based Planning. Dans Wolfgang WAHLSTER, éditeur, *European Conference on Artificial Intelligence*, pages 132–136, Budapest (HU), août 1996.

Jean LIEBER et Amedeo NAPOLI. Correct and Complete Retrieval for Case-Based Problem-Solving. Dans Henri PRADE, éditeur, *European Conference on Artificial Intelligence*, pages 68–72, Brighton (GB), août 1998. John Wiley & Sons Ltd, Chichester (GB).

Mary-Lou MAHER et Andrés GÓMEZ. Developing Case-Based Reasoning for Structural Design. *IEEE Expert*, 11(3) :42–52, 1996.

Frank MANOLA et Eric MILLER, éditeurs. Resource Description Framework (RDF) Primer. Working draft, World Wide Web Consortium, avril 2002.
URL http://www.w3.org/TR/rdf-primer.

Deborah L. MCGUINNESS et Frank VAN HARMELEN, éditeurs. Feature Synopsis for OWL Lite and OWL. Working draft, World Wide Web Consortium, juillet 2002.
URL http://www.w3.org/TR/owl-features/.

Alain MILLE. Experience et expertise : les connaissances mobilisées en coopération homme-machine pour la résolution de problème. Habilitation à diriger des recherches, Université Claude Bernard Lyon I, Lyon (FR), 1998.

Alain MILLE, Béatrice FUCHS, et Benoît CHIRON. Raisonnement fondé sur l'expérience : un nouveau paradigme en supervision industrielle. *Revue d'Intelligence Artificielle*, 13 :97–128, 1999.

Amedeo NAPOLI. Une introduction aux logiques de descriptions. Rapport de Recherche RR 3314, INRIA, Nancy (FR), 1997.

Bernhard NEBEL et Jana KÖHLER. Plan Reuse versus Plan Generation : A Theoretical and Empirical Analysis. *Artificial Intelligence (Special Issue on Planning and Scheduling)*, 76(1–2) :427–454, 1995.

Allen NEWELL. The knowledge level. *Artificial Intelligence*, 18 :87–127, 1982.

Ikujiro NONAKA et Hirotaka TAKEUCHI. *The Knowledge Creating Company.* Oxford University Press, Oxford (GB), 1995.

Rivka OXMAN. Precedents in design : a computational model for the organization of precedent knowledge. *Design Studies*, (2) :141–157, 1994.

Rivka OXMAN et Angi VOSS. CBR in design. *AI Communications*, 9 :117–127, 1996.

François PACHET. Représentation de connaissances et langages à objets. Habilitation à diriger des recherches 97-21, LIP6, Paris (FR), 1997.

Steven PEMBERTON, éditeur. XHTML^{TM}1.0 : The Extensible Hyper Text Markup Language. Recommendation, World Wide Web Consortium, janvier 2000.
URL http://www.w3.org/TR/xhtml1.

Jacques PERRIN, éditeur. *Conception, entre science et art.* Presses polytechniques et universtaires romandes, Lausanne (CH), 2001.

Jacques PERRIN, éditeur. *Internationnal Conference on the Sciences of Design*, INSA Lyon, Villeurbanne (FR), mars 2002.

Enric PLAZA. Cases as terms : A feature term approach to the structured representation of cases. Dans Manuela M. VELOSO et Agnar AAMODT, éditeurs, *International Conference on Case-Based Reasonning*, numéro 1010 dans Lecture Notes in Computer Science, pages 265–276, Sesimbra (PT), octobre 1995. Springer Verlag, Berlin (DE).

Yannick PRIÉ. *Modélisation de documents audiovisuels en Strates Interconnectées par les Annotations pour l'exploration contextuelle.* Thèse de doctorat en informatique, Institut National des Sciences Appliquées, Lyon (FR), 1999.

Lena QIAN et John S. GERO. Function-behaviour-structure paths and their role in analogy-based design. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 10 : 289–312, 1996.

Raymond REITER. A Logic for Default Reasoning. *Artificial Intelligence*, 13 :81–132, 1999.

Michael M. RICHTER. The Knowledge Contained in Similarity Measures. Conférence invitée, ICCBR'95, Sesimbra (PT), octobre 1995.
URL http://www.cbr-web.org/documents/Richter95remarks.html.

Maria RIFQI, Vincent BERGER, et Bernadette BOUCHON-MEUNIER. Discrimination power of measures of comparison. *Fuzzy sets and systems*, 110(2) :189–196, 2000.

Thomas ROTH-BERGHOFER et Ioannis IGLEZAKIS. Six Steps in Case-Based Reasoning : Towards a maintenance methodology for case-based reasoning systems. Dans Vollrath *et al.* [2001], pages 198–208.

Stéphane ROUSSET. Les conceptions "système unique" de la mémoire aspects théoriques. *Revue de Neuropsychologie*, 10(1) :27–52, 2000.

Roger C. SCHANK. *Dynamic Memory : a Theory of Learning in Computers and People.* Cambridge University Press, Cambridge (GB), 1982.

Roger C. SCHANK et Robert P. ABELSON. *Scripts, Plans, Goals and Understanding : an Inquiry into Human Knowledge Structures.* Lawrence Erlbaum Associates, Mahwah, NJ (US), 1977.

Karl SCHLECHTA. *Nonmonotonic Logics : Basic Concepts, Results, and Techniques.* Numéro 1187 dans Lecture Notes in Computer Science. Springer Verlag, Berlin (DE), 1997.

Anne-Françoise SCHMID. *Pour une épistémologié de la conception*, chapitre 5, pages 79–97. Dans Perrin [2001], 2001.

Guus SCHREIBER, Bob WIELINGA, Hans AKKERMANS, Walter VAN DE VELDE, et Robert DE HOOG. CommonKADS : A Comprehensive Methodology for KBS Development. *IEEE Expert*, 9(6) :28–37, 1994.

Michèle Sebag et Marc Shoenauer. A Rule Based Similarity Measure. Dans *1st European Workshop On Case-Based Reasoning*, volume 837 de *Lecture Notes in Artificial Intelligence*, pages 65–70, University of Kayserslautern, (DE), 1993. Springer Verlag, Berlin (DE).

Herbert Simon. *The Sciences of the Artificial*. MIT Press, Cambridge, MA (US), 1969.

Barry Smyth et Mark T. Keane. Remembering to Forget : a Competence-preserving Case Deletion Policy for Case Basef Reasoning Systems. Dans C.S. Mellish, éditeur, *International Joint Conference On Artificial Intelligence*, volume 1, pages 377–382, Montréal, Québec (CA), août 1995a. Morgan Kaufmann, San Mateo, CA (US).

Barry Smyth et Matk T. Keane. Retrieval and Adaptation in Déjà Vu, a Case-Based Reasoning System for Software Design. Dans David W. Aha et Ashwin Ram, éditeurs, *AAAI Fall Symposium on Adaptation of Knowledge for Reuse*, MIT Campus, Cambridge, MA (US), novembre 1995b. American Association for Artificial Intelligence, Menlo Park, CA (US).

Barry Smyth et Elizabeth McKenna. Competence Models and the Maintenance Problem. *Computational Intelligence, Special Issue on Maintaining Case-Based Reasoning (à paraître)*, 2002.
URL http://www.cs.ucd.ie/staff/bsmyth/home/papers/compint.zip.

K. Sollins et L. Masinter. Functional Requirements for Uniform Resources Names. RFC 1737, Internet Engineering Task Force, décembre 1994.
URL http://www.ietf.org/rfc/rfc2616.txt.

John Sowa. *Knowledge Representation : Logical, Philosophical, and Computational Foundations*. PWS Publishing Co., Pacific Grove, CA (US), 1999.

Jean-Yves Toussaint et Monique Zimmermann. *De quelques difficultés à prendre en compte les usages dans la conception de produits*, chapitre 12, pages 215–238. Dans Perrin [2001], 2001.

Amos Tversky. Features of Similarity. *Psychological Review*, 84(4) :327–352, 1977.

Leendert W.N. van der Torre et Yao-Hua Tan. Reasoning About Exceptions. Dans Gerhard Brewka, Christopher Habel, et Bernhard Nebel, éditeurs, *German Conference on Artificial Intelligence*, volume 1303 de *Lecture Notes in Computer Science*, Freiburg (DE), septembre 1997. Springer Verlag, Berlin (DE).

Willemien Visser. A Tribute to Simon, and Some —Too Late— Questions, by a Cognitive Ergonomist. Dans Perrin [2002].

Ivo Vollrath, Sascha Schmitt, et Ulrich Reimer, éditeurs. *German Workshop on Case-Based Reasoning*, Baden-Baden (DE), mars 2001. Shaker Verlag, Aachen (DE).

Angi Voss. Principles of Case Reusing Systems. Dans Ian Smith et Boi Faltings, éditeurs, *European Workshop on Case Based Reasoning*, volume 1168 de *Lecture Notes in Computer Science*, Lausanne (CH), novembre 1996. Springer Verlag, Berlin (DE).

Bruce W.A. Whittlesea. Preservation of Specific Experiences in the Representation of General Knowledge. *Journal of Experimental Psychology : Learning, Memory and Cognition*, 13(1) : 3–17, 1987.

Wolfgang WILKE et Ralph BERGMANN. Techniques and Knowledge Used for Adaptation During Case-Based Problem Solving. Dans Angel P. DEL POBIL, Jose MIRA, et Moonis ALI, éditeurs, *International Conference On Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*, volume 1416 de *Lecture Notes in Computer Science*, pages 497–506, Castellón (ES), juin 1998. Springer Verlag, Berlin (DE).

Table des figures

Résumé

L'activité de conception est par nature une activité complexe, difficile à modéliser. Cela explique que la réutilisation tienne une grande place dans cette activité (réutilisation d'objets ou de processus de conception). Ce mémoire propose une modélisation de l'expérience des concepteurs, en vue d'en assister la réutilisation. Dévelopé dans le cadre du projet ARDECO, en collaboration avec des chercheurs en ergonomie cognitive, ce modèle est fondé sur la notion d'épisode de conception. Des processus de remémoration et d'adaptation des épisodes, fondés sur un appariement approximatif de graphes, sont également présentés.

Dans un second temps, une généralisation du modèle d'expérience est proposée. Le modèle MUSETTE étend le précédent à un champ d'application plus large, notamment à celui, en plein développement, du Web Sémantique.

Mots-clés: conception, expérience, modélisation, raisonnement à partir de cas, Web Sémantique

Abstract

Design is by nature a complex activity, which is hard to model. That explains why reusing is usual in that activity (reusing design objects or design processes). This thesis proposes a model of designers' experience, in order to assist its reuse. This model was developed in the context of the ARDECO project, in collaboration with researchers in cognitive ergonomics ; it is based on the notion of design episode. Processes for remembering and adapting episodes are also presented, based on an approximate graph matching algorithm.

Then, a generalization of the experience model is proposed. The MUSETTE model extends the former one to a larger field of application, especially the emerging Semantic Web.

Keywords: case based reasoning, design, experience, model, Semantic Web