Système de règles de production et calcul de réécriture

Hubert Dubois

▶ **To cite this version:**

Hubert Dubois. Système de règles de production et calcul de réécriture. Informatique [cs]. Université Henri Poincaré - Nancy 1, 2001. Français. NNT : 2001NAN10123 . tel-01748119

HAL Id: tel-01748119
https://hal.univ-lorraine.fr/tel-01748119

Submitted on 29 Mar 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

AVERTISSEMENT

Ce document est le fruit d'un long travail approuvé par le jury de soutenance et mis à disposition de l'ensemble de la communauté universitaire élargie.

Il est soumis à la propriété intellectuelle de l'auteur. Ceci implique une obligation de citation et de référencement lors de l'utilisation de ce document.

D'autre part, toute contrefaçon, plagiat, reproduction illicite encourt une poursuite pénale.

Contact : ddoc-theses-contact@univ-lorraine.fr

LIENS

Code de la Propriété Intellectuelle. articles L 122. 4
Code de la Propriété Intellectuelle. articles L 335.2- L 335.10
http://www.cfcopies.com/V2/leg/leg_droi.php
http://www.culture.gouv.fr/culture/infos-pratiques/droits/protection.htm

S.C.D. - U.H.P. NANCY 1
BIBLIOTHÈQUE DES SCIENCES
Rue du Jardin Botanique
54600 VILLERS-LES-NANCY

Département de formation doctorale en informatique **École doctorale IAE + M**
UFR STMIA

Systèmes de règles de production et calcul de réécriture

THÈSE

présentée et soutenue publiquement le 19 septembre 2001

pour l'obtention du

Doctorat de l'université Henri Poincaré – Nancy 1
(spécialité informatique)

par

Hubert Dubois

Composition du jury

Président :	M. François CHARPILLET, Directeur de Recherche, LORIA-INRIA, Nancy
Rapporteurs :	M. Pierre COINTE, Professeur, Ecole des Mines de Nantes M. François FAGES, Directeur de Recherche, INRIA Rocquencourt Mme Marie-Christine HATON, Professeur, Université Henri Poincaré - Nancy I
Examinateurs :	Mme Hélène KIRCHNER, Directeur de Recherche, LORIA-CNRS, Nancy M. Luigi LIQUORI, Chargé de Recherche, LORIA-INRIA, Nancy

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503

S.C.D. - U.H.P. NANCY 1
BIBLIOTHÈQUE DES SCIENCES
Rue du Jardin Botanique
54600 VILLERS-LES-NANCY

Remerciements

Je tiens tout d'abord à remercier Hélène Kirchner pour m'avoir encadré durant tous les travaux qui ont finalement mené à cette thèse. Sa rigueur, ses critiques et ses encouragements ont fait en sorte que mes idées parfois confuses et peu abouties donnent lieu à ces résultats. Je tiens enfin à la remercier pour sa disponibilité.

Je tiens ensuite à remercier l'ensemble des membres de mon jury.

Je commencerais tout d'abord par remercier François Charpillet d'avoir accepté d'être Président du jury. Par ses questions et ses remarques, il m'a témoigné un grand intérêt pour les travaux effectués.

Je remercie ensuite particulièrement Pierre Cointe d'avoir accepté d'être rapporteur de cette thèse. Ses remarques et commentaires m'ont été très utiles et m'encouragent à poursuivre et approfondir le travail mis en place dans cette thèse.

François Fages, a accepté d'être rapporteur de cette thèse. Je le remercie pour ses commentaires et remarques fort judicieux.

Marie-Christine Haton, au cours de discussions et après examen de cette thèse en détail, m'a fait part de nombreuses remarques, de nombreuses questions très pertinentes qui ont fait évoluer le positionnement de cette thèse qui se situe entre deux domaines qui nous sont chers à l'un comme à l'autre : les systèmes de règles de réécriture et les systèmes de règles de production.

Je tiens finalement à remercier Luigi Liquori d'avoir été examinateur de cette thèse. Les réunions passées avec lui ont été très enrichissantes tant par le foisonnement d'idées, que par l'aide et l'appui qu'il a manifesté pour ces travaux. Il m'a énormément aidé à y voir plus clair dans la nébuleuse qu'étaient pour moi les calculs objet.

Je tiens ensuite à remercier Claude Kirchner de m'avoir accueilli dans l'équipe PROTHEO durant toute la durée de cette thèse. J'en profite également pour remercier tous les membres de l'équipe. La participation aux travaux de PROTHEO et le fait d'être entouré par toute une équipe m'a permis d'avoir un soutien et un encouragement très utiles.

Je remercie aussi vivement Jean-Christophe, Laurent et Isabelle pour avoir relu des versions préliminaires de cette thèse. Leurs remarques ont fait évoluer ce document dans le bon sens. Merci à eux.

Je remercie ensuite tous les membres du LORIA, chercheurs, enseignants-chercheurs, doctorants et administratifs, avec lesquels j'ai passé (et je passe) de superbes années à travailler ici. Que ce soit pour le travail ou tout simplement pour passer un bon moment de détente, ces quatre années passées au LORIA avec eux ont été très agréables. Je remercie ainsi, en vrac et en en oubliant certainement, Horatiu, Laurent, Sorin, Olivier, Christophe, Raulent, Guillaume, Odile, etc, etc...

Pour finir, je remercie amis et famille pour m'avoir soutenu - sans forcément vous en rendre compte ! - durant toute cette thèse. Soirées, week-ends, vacances ou tout simplement conversations téléphoniques ou électroniques ont été d'importants moments oxygénants et salutaires durant toutes ces années. Je vous adresse un grand merci à vous tous.

S.C.D. - U.H.P. NANCY 1

Table des matières

S.C.D. - U.H.P. NANCY 1
BIBLIOTHÈQUE DES SCIENCES
Rue du Jardin Botanique
54600 VILLERS-LES-NANCY

S.C.D. - U.H.P. NANCY 1
BIBLIOTHÈQUE DES SCIENCES
Rue du Jardin Botanique
54600 VILLERS-LES-NANCY

Introduction

En programmation, il existe de nombreux paradigmes. Parmi ceux-ci, on peut citer la programmation impérative, dont le langage C est peut-être le langage le plus répandu, la programmation fonctionnelle avec les familles Lisp et ML ou encore la programmation logique avec le langage Prolog.

On peut considérer la programmation logique comme une tentative d'utiliser la logique comme langage de programmation. Un programme est donné par un ensemble de fonctions logiques ou prédicats et d'une suite d'assertions et de relations. Parmi les relations utilisables, on peut utiliser l'implication, comme dans Prolog. On peut aussi utiliser la notion de règle, comme cela est proposé dans des langages logiques à base de règles.

La notion de règle de la forme $A \Rightarrow B$ signifiant *"si A alors B"* est un paradigme naturel, facile à lire et très largement utilisé dans différentes communautés informatiques : on utilise des règles en intelligence artificielle, en déduction automatique, en logique, etc. C'est un paradigme qui m'a aussitôt intéressé lorsque je l'ai découvert, il y a de cela plusieurs années maintenant.

Dans la suite, on va s'intéresser à la notion de règles. Les règles des systèmes de règles de production tout d'abord, ainsi que les règles de réécriture utilisées en logique de réécriture. Dans la première approche, les règles sont notamment utilisées pour décrire l'état des connaissances ; dans la seconde, elles permettent d'exprimer la déduction et le calcul.

Les systèmes de règles de production sont une forme de systèmes experts. Les règles de production, d'abord utilisées dans la théorie des automates et les grammaires formelles ont été mises au service des systèmes experts [BF78].

Les langages de programmation basés sur la logique de réécriture sont nés suite aux travaux préliminaires sur le langage OBJ depuis sa première version [GT77] jusqu'à sa dernière OBJ3 [GKK+87]. La logique de réécriture [Mes92], dont les prémisses apparaissent dans [DJ90], permet de décrire une sémantique opérationnelle relativement simple et d'offrir une expressivité agréable à manipuler et puissante.

Dans un premier temps, nous détaillons ces deux concepts de règles : les règles comme description de l'état des connaissances puis les règles comme description de la déduction et du calcul.

Description de l'état des connaissances

Les systèmes de règles de production [DK77] sont une des différentes formes existantes de systèmes experts. Ces derniers, nés dans les années soixantes, viennent d'une branche de l'intelligence artificielle dont le but était de réaliser des programmes capables de simuler la réflexion humaine dans certains domaines comme la résolution de problèmes, la perception visuelle, la compréhension du langage. Ces technologies ont engendré de nombreux systèmes dans des domaines aussi variés que la prospection minière ou encore le diagnostic médical. Parmi les différentes tâches effectuées par un système expert, on peut citer :

- l'interprétation des données ;
- le diagnostic de dysfonctionnements ;
- l'analyse structurelle d'objets complexes ;

- la configuration d'objets complexes ;
- la planification de séquences d'actions.

L'utilisation des règles dans les systèmes de règles de production a permis une large utilisation et une large diffusion de règles de la forme **SI** *conditions* **ALORS** *actions*. Les règles sont ainsi apparues comme un moyen de représenter la déduction suivante : à partir de la présence dans la base de connaissances de l'ensemble des *conditions*, on peut déduire par les *actions* de nouvelles données, de nouveaux faits, qui seront utilisables ultérieurement par le système de production.

L'étude des systèmes de règles de production montre que ces systèmes travaillent sur une base de faits, appelée aussi mémoire de travail, qui est un ensemble dans lequel est représenté l'ensemble des faits caractérisant le domaine dans lequel on travaille à un moment donné. C'est cette base de faits que le système fait évoluer durant tout le processus d'évaluation par les règles de production qui constituent la base de connaissances. Par exemple, ce type de système est couramment utilisé dans des problèmes de planification de tâches.

De plus, la base de faits peut être complétée par un ensemble de contraintes dans lesquelles sont stockées des informations que l'on ne souhaite pas résoudre au moment de la décomposition ou encore des informations qui ne peuvent pas être résolues à ce moment. La résolution des contraintes est alors reportée à une décomposition ultérieure. Le fait de pouvoir combiner réécriture et gestion de contraintes peut être très utile dans de nombreuses applications comme par exemple des processus de planification avec ordonnancement de tâches.

Si l'on cherche à résumer les principales composantes d'un système de règles de production, on peut définir un tel système à partir :

- d'une base de connaissances (ou ensemble de règles),
- d'une base de faits (ou mémoire de travail),
- et d'un moteur d'inférence permettant d'appliquer les règles sur la base de faits.

Description de la déduction et du calcul

Une règle de réécriture de la forme $g \to d$ est une paire de termes (g,d) orientée signifiant que toute occurrence de g est remplacée par d dans un terme composé de symboles préalablement déclarés dans ce qu'on appelle une signature. Les symboles composant les règles respectent de même cette signature.

Afin d'illustrer le concept de règle de réécriture, on peut définir un terme t composé d'une succession des symboles ♣ et ♠ :

$$t = \clubsuit\spadesuit\spadesuit\clubsuit\spadesuit.$$

On considère aussi les deux règles de réécriture suivantes :

$$\clubsuit\spadesuit \to_1 \spadesuit$$
$$\spadesuit\spadesuit \to_2 \spadesuit.$$

On peut alors transformer le terme t par applications successives des deux règles précédentes comme on le montre dans la Figure 1. On peut constater qu'à partir du terme de départ t, on peut appliquer soit la première règle, soit la seconde et ainsi avoir différentes dérivations possibles. On peut aussi appliquer une même règle à des positions différentes sur un même terme de départ. Néanmoins, dans l'exemple proposé ici, le terme résultat est toujours ♠ ; et ceci, quelque soit l'ordre et la position d'application des règles.

Le fait de vouloir définir les règles comme la base d'une sémantique opérationnelle pour les programmes logiques nécessite de définir ce qu'on appelle la logique de réécriture.

Il aura fallu attendre quelques années avant de voir se profiler le concept de logique de réécriture [Mes92] notamment suite aux travaux sur OBJ [Gog78] dès le début des années quatre-vingt et avec lui, la naissance de quelques langages basés entièrement sur des règles comme ELAN [BCD+00],

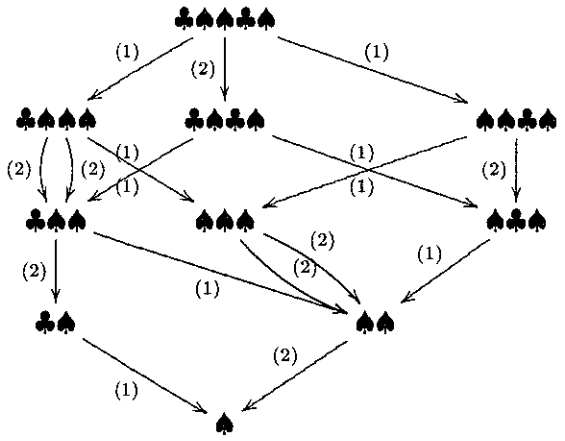
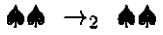


FIG. 1 – *Application de règles de réécriture sur un terme*

Maude [CDE+00] ou encore CafeOBJ [FS94]. La logique de réécriture est un cadre logique défini permettant d'interpréter des systèmes de règles de réécriture. La logique de réécriture a été enrichie dans [KKV93] en un système de calcul afin de pouvoir définir, en plus des systèmes de règles, des systèmes de stratégies permettant de guider l'application des règles. Cela s'avère souvent très utile pour l'efficacité des calculs comme on va le voir en reprenant l'exemple présenté en Figure 1. Sur cet exemple, on se rend compte qu'à partir du terme de départ, la succession d'applications de n'importe quelle règle à n'importe quelle position conduit au même résultat. Par contre, si on avait remplacé la seconde règle par la règle



et en partant toujours du même terme de départ, on aurait alors eu des branches de l'arbre de dérivations qui ne se seraient pas terminées. On n'aurait alors pas eu un unique résultat. On donne une partie de l'arbre de dérivation pour ce cas dans la Figure 2.

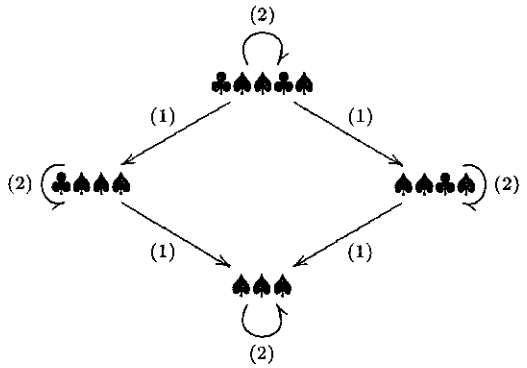


FIG. 2 – *Réécriture non convergente*

Ainsi, la seule utilisation des règles ne permet pas forcément d'avoir un système convergent vers une solution unique. On peut reprendre l'exemple précédent et on se rend compte que chaque terme de l'arbre de dérivation peut être un résultat possible. Cela peut être acceptable et souhaitable dans certains cas ; par contre, si l'on cherche à faire du calcul, on souhaite obtenir un résultat et un seul.

Les règles seules ne suffisant donc pas, l'utilisation des stratégies peut alors parfois permettre de définir un système convergent vers une solution unique. Dans l'exemple précédent, une stratégie définissant l'application répétitive de la première règle permettrait, à partir du terme de départ t, d'obtenir comme

unique résultat le terme ♠♠♠. C .Kirchner, H. Kirchner et M. Vittek ont défini dans [KKV93] un concept de système de calcul permettant de définir un programme comme un ensemble de règles et de stratégies. ELAN est un système basé sur ce concept de système de calcul avec règles et stratégies.

Les objectifs

Tout au long de cette thèse, nous manipulerons les deux concepts de règles suivants : règles comme mode de description de l'état des connaissances et règles comme description de la déduction et du calcul. Le but de cette thèse est de proposer un formalisme permettant d'implanter les systèmes de règles de production en logique de réécriture et plus particulièrement dans un système de calcul comme ELAN dans lequel on trouve la notion de stratégies.

Partant de la définition des systèmes de règles de production, on doit tout d'abord définir un formalisme de règle adapté à la gestion d'une base de faits. On doit alors définir un formalisme adapté à l'utilisation d'une base de faits que l'on souhaite manipuler par ajout, retrait, modification de tout ou partie de cette base.

Ainsi, afin de représenter les faits, on pourrait utiliser une logique des prédicats et voir alors la mémoire de travail comme une conjonction de formules exprimées dans cette logique. Dans notre approche, on préfère travailler avec un formalisme d'objets car ce type de formalisme permet de pouvoir regrouper en un ensemble homogène (la base de faits) un ensemble de données hétérogènes (les objets). De plus, le fait de pouvoir exprimer des concepts propres à la programmation objet en logique de réécriture s'avère être un défi très intéressant.

Parmi les langages basés sur la logique de réécriture, seul le langage Maude a intégré des modules objets à son système dans l'optique de réaliser une version adaptée à des applications liées à l'Internet, Mobile Maude. Mais, seule une sémantique opérationnelle est donnée pour garantir une exécution correcte, par l'interpréteur des règles, des messages et des objets.

On veut montrer que les principales fonctionnalités propres aux langages objets sont implémentables dans un langage basé sur la logique de réécriture. On étudiera donc les concepts de classes, objets, attributs, méthodes, appels de méthodes et héritage et on montrera comment ils peuvent être codés dans le langage ELAN. De plus, on souhaite définir une extension au langage permettant de définir aisément des classes dans une application ELAN.

Ensuite, on souhaite offrir la possibilité de coopérer avec un résolveur de contraintes. En effet, comme nous nous intéressons plus particulièrement à formaliser des problèmes dans les domaines de la planification ou de l'ordonnancement, l'usage de contraintes est très utile afin de pouvoir y délester une partie de l'information qui est non résolvable à un moment donné mais qui peut l'être ultérieurement. Cette information mise de côté dans les contraintes est alors traitée par un résolveur de contraintes. On doit ainsi pouvoir communiquer avec de tels solveurs tant pour lui envoyer des contraintes à résoudre que pour récupérer les solutions qu'il calcule et qu'il nous renvoie.

Quelques formalismes, comme celui présenté par B. Liu, J. Jaffar et H.C. Yap [LJY99], ont déjà cherché à associer la programmation par règles avec la gestion des contraintes. Dans leur proposition, la gestion des contraintes est un processus complètement séparé de l'application des règles sur des termes. Par contre, il permet, en cours de réécriture, l'utilisation de contraintes. Cela est réalisé à partir de primitives telles que le fait de poster une contrainte dans la base de contraintes annexe, de demander une solution, etc.

Une telle approche nous intéresse car elle se base uniquement sur les procédures de communication entre les règles et le résolveur de contraintes sans qu'on ait à se focaliser sur l'écriture d'un solveur. On souhaite donc étudier comment faire le lien entre un résolveur de contraintes et les règles de réécriture dans le formalisme que nous proposons. Le solveur que nous utilisons dans toute notre approche est le système COLETTE [Cas98a] qui est un résolveur de contraintes basés sur règles et stratégies écrit en ELAN.

Dans notre formalisme, on veut donc associer au processus de réécriture la possibilité de travailler conjointement avec une base de contraintes. Ainsi, on doit définir comment représenter une base de contraintes et les liens existants entre la base d'objets et la base de contraintes. Ensuite, on doit définir des primitives de coopération entre ces deux bases permettant de déclencher la résolution d'une contrainte par exemple, ou encore le test de satisfaisabilité de la base de contraintes. Finalement, on doit définir les moyens de communication que l'on utilise entre les règles et le solveur.

Récemment, le ρ-calcul [CK99a, Cir00] a été proposé comme sémantique unifiant à la fois la notion de règle de réécriture et son application sur les termes. Avec cette sémantique, on peut exprimer différents paradigmes et notamment des calculs orientés objet comme celui de K. Fisher, F. Honsell et J.-C. Mitchell [FHM94] ou encore celui de M. Abadi et L. Cardelli [AC96].

On souhaite que l'ensemble du formalisme proposé soit conforme à la sémantique définie dans le ρ-calcul. Les règles proposées dans notre formalisme doivent être étendues à la gestion des objets et des contraintes. Le solveur de contraintes que l'on utilise, à savoir le système COLETTE, comme étant entièrement basé sur le concept de règles et de stratégies standard d'ELAN, respecte par construction la sémantique du ρ-calcul.

Une fois le formalisme défini, on s'intéressera à son utilisation. Le formalisme que l'on développera devra être exécutable dans le système ELAN actuellement diffusé. Pour cela, on définira un mécanisme de transformation de programme qui nous permettra de faire correspondre le formalisme de règles avec objets et contraintes que l'on proposera avec le paradigme de règles et stratégies standard en ELAN.

La définition d'un tel formalisme est intéressante car elle permet d'utiliser toute la puissance des stratégies dans la résolution de problèmes traités dans des systèmes de règles de production. L'utilisation de stratégies est courante pour ce type de problèmes, mais elles sont souvent câblées "en dur" dans le système. ELAN offre de son côté un véritable langage de stratégies permettant à l'utilisateur un contrôle total sur l'application des règles.

De plus, la définition de notre formalisme permettra de définir un paradigme liant règles, objets, contraintes et stratégies dans un cadre logique unique et basé sur la logique de réécriture. Ce formalisme, proche des systèmes de règles de production, devra être adapté à la gestion d'applications telles que la décomposition de tâches ou l'ordonnancement.

Plan de la thèse

Le premier chapitre détaille l'ensemble des concepts logiques utilisés dans les systèmes de réécriture et pour la gestion des contraintes. On y rappelle les notions de termes, de règles, les différentes propriétés des systèmes de réécriture, la notion de formule et l'ensemble des propriétés qu'elle peut vérifier.

Le second chapitre fait un état de l'art sur les systèmes à base de règles. On s'attache tout d'abord à définir ce que sont les systèmes de règles de production, puis on donne une description plus formelle des composants de tels systèmes. On montre alors différentes techniques de vérification qui ont été développées pour ces systèmes et on détaille plus particulièrement quelques systèmes comme CLIPS [GR89], Maude, CRP(FD) [LJY99] et CLAIRE [CL96]. On présente enfin le système ELAN, qui nous intéresse plus particulièrement dans cette thèse. Finalement, on présente les extensions que l'on souhaite apporter au système ELAN.

Dans le troisième chapitre, on explique comment les concepts objets peuvent être intégrés au système ELAN. Pour cela, on détaille tout le ρ-calcul et plus particulièrement le ρ-calcul objet sur lequel on se base afin de donner une sémantique au codage des objets dans le langage ELAN. Ensuite, on définit comment l'extension se déploie en règles de réécriture pouvant être intégrées au système ELAN.

La préservation du type lors de l'application des règles sur des termes mettant en cause des objets est ensuite étudiée. On présente alors un système de règles d'inférence que l'on utilise pour montrer la propriété de préservation du type pour le langage étendu aux concepts objets.

Une fois qu'on a défini comment coder des objets en logique de réécriture, on peut formaliser une mémoire de travail. L'intégration des concepts d'objets et de contraintes aux règles de réécriture est le

sujet du quatrième chapitre. On montre comment on peut définir un formalisme particulier de règles dans lesquelles on peut aisément travailler sur une base d'objets. On montre aussi comment ces règles peuvent être enrichies pour gérer également des contraintes. Les formalismes règles+stratégies+objets et règles+stratégies+objets+contraintes sont chacun illustrés par des applications qui nous permettent de mettre en pratique les concepts et idées présentées tout au long de nos travaux.

Le cinquième chapitre montre comment exécuter une application exprimée dans le formalisme de l'extension au système ELAN proposée dans cette thèse. On présente alors le mécanisme de traduction que nous avons défini et qui est utilisé pour pouvoir obtenir un ensemble de modules exécutables par le système à partir d'un ensemble hétérogène associant des modules ELAN standard, des modules objets et des règles particulières avec objets et contraintes.

Finalement, on donne un ensemble de conclusions et de perspectives à nos travaux.

Chapitre 1

Cadre logique

Dans ce premier chapitre, on s'intéresse à définir les principales notions logiques utilisées tout au long de cette thèse. Nous commençons par rappeler quelques définitions de base: le concept d'algèbre universelle tout d'abord, puis la notion de terme et de substitution. Ensuite, on définit les formules et leurs propriétés. Puis, après avoir présenté les théories équationnelles et les propriétés des relations binaires, nous pouvons finalement présenter le concept de système de calcul fondé sur la notion de logique de réécriture.

1.1 Définitions de base

Les règles de réécriture manipulant des termes, nous présentons dans cette partie les algèbres avant de pouvoir définir ce qu'on appelle les algèbres de termes et la notion de substitution qui lui est associée.

1.1.1 Algèbres

Soit $\mathcal{S}$ un ensemble d'indices. Un ensemble $\mathcal{S}$-indexé est une famille d'ensembles, un pour chaque $s \in \mathcal{S}$ que l'on notera $(A_s)_{s\in\mathcal{S}}$. Une fonction $\mathcal{S}$-indexée entre deux ensembles $\mathcal{S}$-indexés A et B est une fonction $\alpha : A \to B$ telle que $\forall a \in A_s, \alpha(a) \in B_s$. Un élément a d'un ensemble $\mathcal{S}$-indexé A est *indexé* par $s \in \mathcal{S}$ si $a \in A_s$.

Les éléments de $\mathcal{S}$ seront désormais appelés symboles de sortes.

Définition 1.1 *Une signature est un triplet $\Sigma = (\mathcal{S},\mathcal{F},\mathcal{P})$ tel que*

- *$\mathcal{S}$ est un ensemble de symboles de sortes.*
- *$\mathcal{F} = \bigcup_{n\geq 0}(\mathcal{F})_n$ est un ensemble de symboles de fonctions $\mathcal{S}^* \times \mathcal{S}$-indexé tel que tout symbole $f \in (\mathcal{F})_n$ est indexé par $(s_1,\ldots,s_n,s)$, ce qu'on note $f : s_1,\ldots,s_n \to s$ et appelle profil de f. L'entier n est appelé arité de f, notée $|f|$.*
- *$\mathcal{P}$ est un ensemble de symboles de prédicats $\mathcal{S}^*$-indexé tel que tout symbole $p \in \mathcal{P}$ a un unique index $(s_1,\ldots,s_n)$, ce qu'on note $p : s_1,\ldots,s_n$ et appelle profil de p. L'entier n est l'arité de p, notée $|p|$.*

Un symbole de fonction a une arité fixe mais plusieurs profils alors qu'un symbole de prédicat a une arité unique et un seul profil. Nous avons choisi d'introduire immédiatement une information de sorte et la possibilité de surcharger des symboles de fonctions. Cela facilitera ensuite la définition des algèbres à sortes ordonnées.

Les symboles de fonctions d'arité nulle sont notés $a,b,c,\ldots$ et appelés *constantes*, les autres seront notés $f,g,h,\ldots$. Soit $\Sigma = (\mathcal{S},\mathcal{F},\mathcal{P})$ une signature.

Définition 1.2 *Une $(\mathcal{S},\mathcal{F})$-algèbre $\mathcal{A}$ est un domaine $A = (A_s)_{s\in\mathcal{S}}$ muni d'un ensemble de fonctions $\mathcal{F}_\mathcal{A}$ tel que pour tout symbole $f \in \mathcal{F}$ correspond une fonction $f_\mathcal{A} : A^{|f|} \to A$ pour laquelle si f est de profil $f : s_1,\ldots,s_n \to s$ et $(a_1,\ldots,a_n) \in A_{s_1} \times \cdots \times A_{s_n}$ alors $f_\mathcal{A}(a_1,\ldots,a_n) \in A_s$.*

Lorsque $\mathcal{S}$ est un singleton, cet ensemble sera souvent omis et l'on parlera simplement de $\mathcal{F}$-algèbre. Cette remarque est valable pour toutes les notations qui seront définies à l'aide d'un ensemble de sortes $\mathcal{S}$.

Une application qui respecte la structure de $(\mathcal{S},\mathcal{F})$-algèbre est un homomorphisme.

Définition 1.3 *Etant données deux $(\mathcal{S},\mathcal{F})$-algèbres $\mathcal{A}$ et $\mathcal{B}$, une application $\mathcal{S}$-indexée θ de A vers B telle que*

$$\forall f \in \mathcal{F}, \forall a_1,\ldots,a_n \in A, \theta(f_{\mathcal{A}}(a_1,\ldots,a_n)) = f_{\mathcal{B}}(\theta(a_1),\ldots,\theta(a_n))$$

est un homomorphisme de $\mathcal{A}$ vers $\mathcal{B}$.

Définition 1.4 *Une algèbre $\mathcal{A}$ d'une classe de $(\mathcal{S},\mathcal{F})$-algèbres est libre sur un ensemble $\mathcal{S}$-indexé $\mathcal{X}$ si pour toute application $\mathcal{S}$-indexée $\theta : \mathcal{X} \to \mathcal{B}$, il existe un unique homomorphisme $\nu : \mathcal{A} \to \mathcal{B}$ tel que θ et ν coïncident sur $\mathcal{X}$.*

Une algèbre libre est unique à un isomorphisme près si elle existe. On peut donc parler sans ambiguïté de la $(\mathcal{S},\mathcal{F})$-algèbre libre sur $\mathcal{X}$.

Par la suite, les éléments de $\mathcal{X}$ seront appelés variables et notés $x,y,z,\ldots$.

1.1.2 Termes, substitutions

Définition 1.5 *Etant donnés un ensemble de symboles de fonctions $\mathcal{F}$ et un ensemble dénombrable de symboles de variables $\mathcal{X}$, l'ensemble de termes du premier ordre $\mathcal{T}(\mathcal{F},\mathcal{X})$ est le plus petit ensemble qui contient $\mathcal{X}$ et tel que la chaîne $f(t_1,t_2,\ldots,t_n)$ appartient à $\mathcal{T}(\mathcal{F},\mathcal{X})$, où $f \in \mathcal{F}_n$ et $t_i \in \mathcal{T}(\mathcal{F},\mathcal{X})$ pour tout $i \in [1,\ldots,n]$. Un terme sans variable est dit clos et un terme qui contient des variables est dit non-clos ou ouvert. Un terme ouvert est linéaire si chacune de ses variables n'y apparaît qu'une seule fois.*

Les variables sont désignées par x, y, z, ... et les termes par s, t, ... Tous ces symboles peuvent être utilisés avec des indices. $\mathcal{V}ar(t)$ désigne l'ensemble des variables dans t. L'ensemble des termes clos est désigné par $\mathcal{T}(\mathcal{F})$.

Exemple 1.1 *Etant donnés un ensemble de fonctions $\mathcal{F} = \{+,*,a\}$ et un ensemble de variables $\mathcal{X} = \{x\}$, avec arité$(+)$ = arité$(*)$ = 2 et arité$(a) = 0$:*

$$a + a * a$$

est un terme clos et :

$$a * x + a * a$$

est un terme linéaire ouvert.

Définition 1.6 *Soient $\mathcal{S}$ un ensemble de symboles de sorte, $\mathcal{F}$ une signature et $\mathcal{X}$ un ensemble de variables. A chaque symbole f de $\mathcal{F}$ d'arité n est associé une suite de $n+1$ symboles de sorte $(s_1,\ldots,s_{n+1})$, et à chaque variable x de $\mathcal{X}$ est associé un symbole de sorte. La suite de symboles de sortes est appelée le profil du symbole f et on note:*

$$f : s_1 \times \ldots s_n \rightarrowtail s_{n+1}$$

où $f \in \mathcal{F}_n$ et $s_i \in \mathcal{S}$.

Les termes de la $\mathcal{F}$-algèbre hétérogène libre $\mathcal{T}(\mathcal{F},\mathcal{X})$ engendrée par $\mathcal{X}$ et la sorte s d'un terme t, noté $t : s$, sont définis simultanément par :

- *pour toute variable $x \in \mathcal{X}$ ayant associé un symbole de sorte s, $x \in \mathcal{T}(\mathcal{F},\mathcal{X})$ et la sorte de x est s,*
- *pour tout symbole $f : s_1 \times \ldots s_n \rightarrowtail s_{n+1}$ et termes $t_1 : s_1,\ldots,t_n : s_n$, $f(t_1,\ldots,t_n) \in \mathcal{T}(\mathcal{F},\mathcal{X})$ et la sorte de $f(t_1,\ldots,t_n)$ est s_{n+1}.*

Le sous-ensemble $\mathcal{T}_s$ de $\mathcal{T}(\mathcal{F},\mathcal{X})$ défini par l'ensemble des termes de sorte s est appelé une sorte. Une algèbre hétérogène est aussi désignée sous le nom d'algèbre de termes multi-sortée.

S.C.D. - U.H.P. NANCY 1
BIBLIOTHÈQUE DES SCIENCES
Rue du Jardin Botanique
54600 VILLERS-LES-NANCY

Définition 1.7 *Une substitution est une application sur $\mathcal{T}(\mathcal{F},\mathcal{X})$ déterminée uniquement par l'image d'un ensemble fini de variables. Elle est notée $\{x_1 \mapsto t_1,\ldots,x_n \mapsto t_n\}$. L'application d'une substitution $\sigma = \{x_1 \mapsto t_1,\ldots,x_n \mapsto t_n\}$ sur un terme t est définie récursivement de la manière suivante :*

- *si t est une variable x_i pour un certain $i \in [1,\ldots,n]$, alors $\sigma(t) = t_i$;*
- *si t est une variable $x \neq x_i$, pour tout $i \in [1,\ldots,n]$, alors $\sigma(t) = t$;*
- *si t est un terme $f(u_1,\ldots,u_k)$, alors $\sigma(t) = f(\sigma(u_1),\ldots,\sigma(u_k))$.*

Exemple 1.2 *Etant donnés un ensemble de fonctions $\mathcal{F} = \{+,*,a,b\}$ et un ensemble de variables $\mathcal{X} = \{x,y\}$, avec arité$(+)$ = arité$(*)$ = 2 et arité(a) = arité(b) = 0. Etant donnés le terme $t = a*x+a*y$ et la substitution $\sigma = \{x \mapsto a+a, y \mapsto b\}$, l'application de σ sur t donne comme résultat le terme :*

$$\sigma(t) = a*a+a+a*b$$

Définition 1.8 *Soit $\mathbb{N}_+$ l'ensemble des entiers strictement positifs, $\mathbb{N}_+^*$ le monoïde libre engendré par $\mathbb{N}_+$, ϵ le mot vide et . l'opération de concaténation. Pour tous $p,q \in \mathbb{N}_+^*$, p est un préfixe de q, ce que l'on note $p \leq q$, s'il existe $q' \in \mathbb{N}_+^*$ tel que $q = p.q'$. p est un préfixe strict de q, noté $p < q$, si $p \leq q$ et $p \neq q$. Si $p \not\leq q$ et $q \not\leq p$, p et q sont disjointes ou incomparables, ce qu'on note $p \bowtie q$.*

Un arbre sur $\mathcal{F} \cup \mathcal{X}$ est une application t d'une partie non vide $\mathcal{P}os(t)$ de $\mathbb{N}_+^*$ dans $\mathcal{F} \cup \mathcal{X}$ telle que :

1. *$\mathcal{P}os(t)$ est clos par préfixe.*
2. *Pour tout $p \in \mathcal{P}os(t)$ et tout $i \in \mathbb{N}_+$, $p.i \in \mathcal{P}os(t)$ si et seulement si $t(p) = f \in \mathcal{F}$ et $1 \leq i \leq |f|$.*

$\mathcal{P}os(t)$ est appelé ensemble des positions de t, et t est fini si $\mathcal{P}os(t)$ l'est. La taille $|t|$ d'un terme t est dans ce cas le cardinal de $\mathcal{P}os(t)$.

L'ensemble des arbres finis sur $\mathcal{F} \cup \mathcal{X}$ peut être muni naturellement d'une structure de $(\mathcal{S},\mathcal{F})$-algèbre isomorphe à $\mathcal{T}(\mathcal{S},\mathcal{F},\mathcal{X})$. On parlera donc dorénavant indifféremment d'arbre ou de terme.

Définition 1.9

- *Pour tout terme t et toute position $p \in \mathcal{P}os(t)$, $t(p)$ est appelé symbole à la position p dans t. $t(\epsilon)$ est également appelé symbole de tête de t.*
- *On appelle sous-terme de t à la position $p \in \mathcal{P}os(t)$, le terme noté $t_{|p}$, et défini par $t_{|p}(q) = t(p.q)$. $t_{|p}$ est un sous-terme strict de t si $p \neq \epsilon$.*
- *Si $t(\epsilon) = f \in \mathcal{F}$, on notera t sous la forme $f(t_{|1},\ldots,t_{|n})$ où $n = |f|$.*
- *Une position p de t est variable si $t(p) \in \mathcal{X}$. L'ensemble des positions variables de t est noté $\mathcal{VP}os(t)$ alors que l'ensemble des positions non variables de t est noté $\mathcal{FP}os(t)$.*
  Une position p de t est constante si $t(p) \in (\mathcal{F})_0$. L'ensemble des positions constantes de t est noté $\mathcal{CP}os(t)$.

La notation $t[s]_\omega$ est utilisée pour signifier que t contient s comme sous-terme à la position ω et la notation $t[\omega \hookleftarrow s]$ pour faire remarquer que le sous-terme $t_{|\omega}$ a été remplacé par s dans t.

1.2 Les formules

Lors de la gestion des contraintes, on est amené à utiliser des formules et à en montrer certaines propriétés. Cette partie nous permet de définir l'ensemble des notions utilisées dans ce cadre.

Définition 1.10 *Soit p un symbole de prédicat muni d'une arité. Soit $\mathcal{P}_n$ l'ensemble de tous les symboles de prédicats d'arité n. L'ensemble de tous les symboles de prédicats est défini par $\mathcal{P} = \bigcup_{n\geq 0} \mathcal{P}_n$. La notion de signature peut être étendue afin d'inclure des symboles de prédicats. Etant donnés un ensemble de fonctions $\mathcal{F}$ et un ensemble de symboles de prédicats $\mathcal{P}$, une signature Σ est l'ensemble de tous les symboles de fonctions de $\mathcal{F}$ et de tous les symboles de prédicats de $\mathcal{P}$. Cette signature est désignée par $\Sigma = (\mathcal{F},\mathcal{P})$.*

Les symboles de prédicats sont désignés par les lettres p, q, ... Tous ces symboles peuvent avoir des indices.

Définition 1.11 *Etant donnés une signature $\Sigma = (\mathcal{F},\mathcal{P})$ et un ensemble dénombrable de symboles de variables $\mathcal{X}$, une Σ-formule du premier ordre est dite bien formée si elle est définie inductivement de la manière suivante :*

- *si $t_1, \ldots t_n$ sont des termes et p est un symbole de prédicat d'arité $n > 0$, alors $p(t_1, \ldots t_n)$ est une Σ-formule (appelée une formule atomique ou simplement un atome) ;*
- *si A et B sont des Σ-formules, alors $(\neg A)$, $(A \wedge B)$, $(A \vee B)$, $(A \rightarrow B)$ et $(A \leftrightarrow B)$ sont aussi des Σ-formules ;*
- *si A est une Σ-formule et x est une variable, alors $\forall x\ A$ et $\exists x\ A$ sont des Σ-formules.*

Lorsqu'il n'y a pas de confusion, le mot *formule* est utilisé pour désigner une Σ-formule. La portée de $\forall x$ dans $\forall x\ A$ et $\exists x$ dans $\exists x\ A$ est A.

Une *occurrence* de la variable x est dite liée si elle apparaît dans la portée d'un quantificateur $\forall x$ ou $\exists x$. Toute autre occurrence d'une variable est *libre*. Une formule est *complètement quantifiée* si toutes ses variables sont liées. Une Σ-théorie est un ensemble de Σ-formules complètement quantifiées.

Exemple 1.3 *Etant donnés $\Sigma = (\mathcal{F},\mathcal{P})$ et $\mathcal{X} = \{x,y\}$, où $\mathcal{F} = \{x, *, a\}$ et $\mathcal{P} = \{=, \geq\}$ avec arité$(+)$ = arité$(*)$ = arité$(=)$ = arité$(\geq) = 2$ et arité$(a) = 0$.*

- *$a + x = a * a$ est une formule atomique ouverte.*
- *$\exists x\ (y + x) \wedge (x * a)$ est une formule où les occurrences de x sont liées, mais l'occurrence de y est libre ;*
- *$\exists x,y\ (y + a) \vee (x * a)$ est une formule complètement quantifiée et donc elle appartient à la Σ-théorie.*

Définition 1.12 *Etant donnée une signature Σ, une Σ-structure $\mathcal{D}$ est une paire (D,I), où D est un ensemble non vide, dit le domaine de la structure, et I est une fonction, dite d'interprétation, qui affecte les fonctions et les relations dans D aux symboles respectivement en $\mathcal{F}$ et $\mathcal{P}$. Cette affectation des fonctions et des relations est définie de la manière suivante :*

- *pour tout symbole de fonction f d'arité $n > 0$, une interprétation $I(f)$ dans D est une fonction d'arité n de D^n vers D ;*
- *pour tout symbole de fonction f d'arité $n = 0$, i.e., une constante, une interprétation $I(f)$ dans D correspond à l'affectation d'un élément de D ;*
- *pour tout ensemble de symboles de fonctions $\mathcal{F}$, une interprétation $I(\mathcal{F})$ de $\mathcal{F}$ dans l'ensemble D est une application qui associe à chaque symbole de fonction dans $\mathcal{F}$ une interprétation dans D ;*
- *pour tout symbole de prédicat p d'arité $n \geq 0$, une interprétation $I(p)$ dans D est une relation d'arité n dans D^n ;*
- *pour tout ensemble de symboles de prédicats $\mathcal{P}$, une interprétation $I(\mathcal{P})$ de $\mathcal{P}$ dans l'ensemble D est une application qui associe à chaque symbole de prédicat dans $\mathcal{P}$ une interprétation dans D.*

Les interprétations $I(f)$, $I(\mathcal{F})$, $I(p)$ et $I(\mathcal{P})$ sont aussi désignées respectivement par $f_{\mathcal{D}}$, $\mathcal{F}_{\mathcal{D}}$, $p_{\mathcal{D}}$ et $\mathcal{P}_{\mathcal{D}}$.

Exemple 1.4 *Soient $\Sigma = (\mathcal{F},\mathcal{P})$ une signature et $\mathcal{X} = \{x\}$ un ensemble de symboles de variables où $\mathcal{F} = \{+, *, a\}$ et $\mathcal{P} = \{=\}$ avec arité$(+)$ = arité$(*)$ = arité$(=) = 2$ et arité$(a) = 0$. Soit $\mathcal{D} = (\mathbb{N},I)$ une Σ-structure où $\mathbb{N}$ est l'ensemble des nombres naturels, on définit $+_{\mathcal{D}}$ comme l'addition, $*_{\mathcal{D}}$ comme la multiplication, $a_{\mathcal{D}}$ comme le nombre naturel 1 et $=_{\mathcal{D}}$ comme la relation d'égalité. Toutes les formules construites à partir de $\mathcal{D}$ sont des équations sur les nombres naturels, comme par exemple :*

$$a_{\mathcal{D}} +_{\mathcal{D}} x = a_{\mathcal{D}} *_{\mathcal{D}} a_{\mathcal{D}}$$

A	B	$\neg A$	$A \wedge B$	$A \vee B$	$A \rightarrow B$	$A \leftrightarrow B$
V	**V**	**F**	**V**	**V**	**V**	**V**
V	**F**	**F**	**F**	**V**	**F**	**F**
F	**V**	**V**	**F**	**V**	**V**	**F**
F	**F**	**V**	**F**	**F**	**V**	**V**

TAB. 1.1 – *Valeur de vérité des connecteurs logiques*

Définition 1.13 *Soit $\mathcal{D} = (D,I)$ une Σ-structure et $\mathcal{X}$ un ensemble de symboles de variables.*

- *Une assignation des variables de $\mathcal{X}$ par rapport à I est une fonction α qui affecte à chaque variable dans $\mathcal{X}$ un élément de D. L'assignation d'une variable est notée $\alpha(x)$ et l'ensemble de toutes les assignations de $\mathcal{X}$ dans $\mathcal{D}$ est noté $\alpha_{\mathcal{D}}^{\mathcal{X}}$.*
- *Une assignation d'un terme non variable par rapport à I est définie récursivement par :*

$$\alpha(f(t_1,\ldots t_n)) = f_{\mathcal{D}}(\alpha(t_1),\ldots,\alpha(t_n))$$

où $f_{\mathcal{D}}$ est l'interprétation du symbole de fonction f d'arité n. L'assignation d'un terme t par rapport à I et α est désignée par $\alpha(t)$.
- *La valeur de vérité, vrai (**V**) ou faux (**F**), d'une formule dans $\mathcal{D}$ est obtenue de la manière suivante :*
  - *si la formule est un atome $p(t_1,\ldots,t_n)$, alors sa valeur de vérité est obtenue récursivement par :*

$$\alpha(p(t_1,\ldots,t_n)) = p_{\mathcal{D}}(\alpha(t_1),\ldots,\alpha(t_n))$$

  où $p_{\mathcal{D}}$ est l'interprétation du symbole de prédicat p d'arité n.
  - *si la formule a la forme $(\neg A)$, $(A \wedge B)$, $(A \vee B)$, $(A \rightarrow B)$ ou $(A \leftrightarrow B)$, alors sa valeur de vérité est donnée par le Tableau 1.1.*
  - *si la formule a la forme $\exists x\ A$, alors sa valeur de vérité est **V** s'il existe $d \in D$ tel que A a la valeur de vérité **V** par rapport à I et $\alpha_{|x\mapsto d}$, où $\alpha_{|x\mapsto d}$ est α sauf que x est affecté à d ; sinon, sa valeur de vérité est **F**.*
  - *si la formule a la forme $\forall x\ A$, alors sa valeur de vérité est **V** si, pour tout $d \in D$, A a une valeur de vérité **V** par rapport à I et $\alpha_{|x\mapsto d}$; sinon, sa valeur de vérité est **F**.*

L'interprétation d'une formule A par rapport à I et α est désignée par $\alpha(A)$.

Définition 1.14 *Soient Σ une signature et A une formule.*

- *Etant données une Σ-structure $\mathcal{D}$ et une affectation α, $\mathcal{D}$ satisfait A avec α si $\alpha(A) = \mathbf{V}$. Ceci est aussi noté $(\mathcal{D},\alpha) \models A$.*
- *Etant donnée une Σ-structure $\mathcal{D}$, la formule A est satisfaisable dans $\mathcal{D}$ s'il existe une affectation α telle que $\alpha(A) = \mathbf{V}$.*
- *La formule A est satisfaisable s'il existe une Σ-structure $\mathcal{D}$ dans laquelle A est satisfaisable.*
- *Etant donnée une Σ-structure $\mathcal{D}$, la formule A est valide dans $\mathcal{D}$ (ou vrai en $\mathcal{D}$) si, pour toute affectation α, $\alpha(A) = \mathbf{V}$. Ceci est désigné par $\mathcal{D} \models A$. Dans ce cas, la Σ-structure $\mathcal{D}$ est appelée un modèle de A.*
- *La formule A est valide (ou universellement valide) si elle est valide dans chaque Σ-structure $\mathcal{D}$. Ceci est désigné par $\models A$.*
- *Etant donnée une Σ-théorie T, T est satisfaisable s'il existe une structure $\mathcal{D}$ et une affectation α telle que $(\mathcal{D},\alpha) \models A$ pour toute formule $A \in T$.*
- *Etant données une Σ-structure $\mathcal{D}$ et une Σ-théorie T, $\mathcal{D}$ est un modèle de T si $\mathcal{D}$ est un modèle de chaque formule en T. Ceci est désigné par $\mathcal{D} \models T$.*

- *Etant donnée une Σ-théorie T, T est valide si $\mathcal{D} \models T$ pour toute Σ-structure $\mathcal{D}$. Ceci est désigné par $\models T$.*
- *Etant données une Σ-théorie T et une formule B, B est une conséquence sémantique de T, désignée par $T \models B$, pour chaque Σ-structure $\mathcal{D}$, si $\mathcal{D} \models T$ alors $\mathcal{D} \models B$.*

1.3 Théories équationnelles

Selon le cadre dans lequel on se place, un terme comme $1 + 2$ peut être équivalent au terme $2 + 1$. Dans un tel cas, cela est vérifié si le symbole "+" désignant ici l'addition des entiers est commutatif. Par contre, si l'on se place dans un cadre purement syntaxique, le terme $1 + 2$ n'est pas égal au terme $2 + 1$. Au travers de cet exemple, ce sont les notions de théories équationnelles qui sont en jeu. On les définit dans cette partie.

Une paire de termes (l,r) est appelé *égalité* ou encore *axiome équationnel* ou *égalitaire* suivant le contexte, et notée $l = r$.

Définition 1.15 *Etant donné un ensemble de variables $\mathcal{X}$, une algèbre $\mathcal{A}$ et une assignation $\alpha : \mathcal{X} \to \mathcal{A}$, on note $\underline{\alpha}$ l'unique homomorphisme de $\mathcal{T}(\mathcal{F},\mathcal{X})$ vers l'algèbre $\mathcal{A}$ étendant α tel que*

$$\forall f \in \mathcal{F}, \underline{\alpha}(f(t_1,\ldots,t_n)) = f_{\mathcal{A}}(\underline{\alpha}(t_1),\ldots,\underline{\alpha}(t_n))$$

Définition 1.16 *Une $\mathcal{F}$-algèbre $\mathcal{A}$ valide une égalité $s = t$, noté $\mathcal{A} \models s = t$ ou plus simplement $s =_{\mathcal{A}} t$ si pour toute assignation $\alpha : \mathcal{X} \to \mathcal{A}$, $\underline{\alpha}(s) = \underline{\alpha}(t)$. L'algèbre $\mathcal{A}$ satisfait une égalité $s = t$ s'il existe une assignation α telle que $\underline{\alpha}(s) = \underline{\alpha}(t)$. Une $\mathcal{F}$-algèbre $\mathcal{A}$ est un modèle d'un ensemble d'égalités E si elle valide toutes les égalités de E.*

On note $\mathcal{T}h(\mathcal{A})$ l'ensemble des égalités valides dans une $\mathcal{F}$-algèbre $\mathcal{A}$ et $\mathcal{M}od(E)$ la classe des $\mathcal{F}$-algèbres qui sont modèles de E.

Soit E un ensemble d'égalités de $\mathcal{T}(\mathcal{F},\mathcal{X})$, appelées dans ce contexte, *axiomes*.

Définition 1.17 *Etant donnée une signature $\mathcal{F}$, une présentation équationnelle est un couple $(\mathcal{F},E)$ telle que E est un ensemble d'axiomes de $\mathcal{T}(\mathcal{F},\mathcal{X})$.*

Le problème de validité dans $\mathcal{M}od(E)$ consiste à décider si une égalité $s = t$ est valide dans tout modèle de E. Ce problème peut se ramener à des considérations syntaxiques.

Définition 1.18 *Etant donnée une présentation équationnelle $(\mathcal{F},E)$, on appelle théorie équationnelle engendrée par $(\mathcal{F},E)$ ou E-égalité et on note $=_E$ la plus petite congruence sur $\mathcal{T}(\mathcal{F},\mathcal{X})$ contenant toutes les égalités $\sigma(l) = \sigma(r)$ où $l = r$ est un axiome de E et σ une substitution quelconque.*

Le théorème suivant est le fondement de la logique équationnelle. Il relie le problème sémantique de la validité d'une égalité dans une classe de modèles au problème syntaxique de la E-égalité.

Théorème 1.1 *(Birkhoff [Bir35]), Complétude du raisonnement équationnel pour un ensemble E d'axiomes équationnels)*
$s = t$ est valide dans $\mathcal{M}od(E)$ si et seulement si $s =_E t$.

La E-égalité peut encore être obtenue par le *remplacement* d'égal par égal décrit ci-après.

Définition 1.19 *Etant donné un ensemble d'axiomes E, on note $\leftrightarrow_E$ la relation binaire symétrique sur $\mathcal{T}(\mathcal{F},\mathcal{X})$ définie par $s \leftrightarrow_E t$ s'il existe un axiome $l = r$ de E, une position ω de s et une substitution σ tels que $s_{|\omega} = \sigma(l)$ et $t = s[\sigma(r)]_\omega$.*

Remarque $s =_E t \Leftrightarrow s \overset{*}{\longleftrightarrow}_E t$.

Par abus de langage et de notation, on confond souvent la théorie équationnelle $=_E$, la présentation équationnelle $(\mathcal{F},E)$ et l'ensemble des axiomes équationnels E.

L'ensemble des classes de congruence de E dans $\mathcal{T}(\mathcal{F},\mathcal{X})$ peut être muni naturellement d'une structure d'algèbre, notée $\mathcal{T}(\mathcal{F},\mathcal{X})/=_E$, qui est l'algèbre libre sur $\mathcal{X}$ de la classe des $\mathcal{F}$-algèbres modèles de E.

1.4 Propriétés des relations binaires sur un ensemble

Les termes peuvent être connectés entre eux par des relations ou par des transformations des uns vers les autres. Nous donnons quelques propriétés abstraites liées aux relations binaires dont nous aurons besoin par la suite.

Définition 1.20 *Une relation binaire $\longrightarrow$ sur un ensemble de termes T construits en utilisant un ensemble d'opérateurs Φ est compatible (avec les opérateurs) si pour tous les termes $u_i, v_i \in T$, $i = 1, \ldots, n$ et tout opérateur ϕ_n d'arité n*

$$u_i \longrightarrow v_i,\ i = 1, \ldots, n \implies \phi_n(u_1, \ldots, u_n) \longrightarrow \phi_n(v_1, \ldots, v_n)$$

Définition 1.21 *Etant donnée une relation binaire $\longrightarrow$ sur un ensemble T :*

- *la relation inverse de $\longrightarrow$ est notée $\longleftarrow$,*
- *la fermeture symétrique de $\longrightarrow$, notée $\longleftrightarrow$, est la plus petite relation symétrique contenant $\longrightarrow$.*
- *la fermeture transitive de $\longrightarrow$, notée $\xrightarrow{+}$, est la plus petite relation transitive contenant $\longrightarrow$.*
- *la fermeture réflexive et transitive de $\longrightarrow$ est notée $\xrightarrow{*}$.*
- *la fermeture réflexive, symétrique et transitive de $\longrightarrow$ est notée $\xleftrightarrow{*}$.*
- *la fermeture compatible (ou fermeture par contexte) de $\longrightarrow$ est la plus petite relation contenant $\longrightarrow$ et fermée par rapport aux règles de formation de termes de T.*

La composition des relations $\longrightarrow_1$ et $\longrightarrow_2$ est notée $\longrightarrow_2 \circ \longrightarrow_1$ ou $\longrightarrow_1 \longrightarrow_2$.

Une relation binaire $\sim$ réflexive, symétrique et transitive est une relation d'équivalence. Un ordre $>$ est une relation binaire irréflexive, antisymétrique et transitive. Un préordre $\geq$ est une relation binaire réflexive et transitive.

Définition 1.22 *Un ordre $>$ sur T est nœthérien s'il n'existe pas de suite infinie $(t_i)_{i \geq 1}$ d'éléments de T telle que $t_1 > t_2 > \ldots$*
Un ordre $>$ sur T est total si $\forall s,t \in T$ on a $s > t$ ou $t > s$.

La construction d'ordres nœthériens peut éventuellement se faire par extension. L'extension lexicographique permet par exemple de comparer des uplets. Pour comparer des ensembles de termes, on introduit la notion de *multi-ensemble* sur T qui est une application de T vers $\mathbb{N}$. Un ordre sur T peut être facilement étendu à un ordre sur les multi-ensembles sur T.

Définition 1.23 *Pour une relation $\longrightarrow$, un élément t de T est réductible par $\longrightarrow$ s'il existe t' dans T tel que $t \longrightarrow t'$. Dans le cas contraire, il est irréductible. On appelle forme normale de t tout élément t' irréductible tel que $t \xrightarrow{*} t'$. Lorsque un terme t a une unique forme normale, celle-ci est notée $t\!\downarrow$.*

La question que l'on se pose est de savoir si $t \xrightarrow{*} t'$. L'idéal serait de calculer une forme normale de chacun des éléments et de tester si elles sont égales. Cela n'est possible que si d'une part une forme normale existe pour tout élément, et si d'autre part elle est unique. Les formes normales existent dès que $\longrightarrow$ *termine*, c'est-à-dire qu'il n'existe pas de suite infinie $(t_i)_{i \geq 1}$ d'éléments de T telle que $t_1 \longrightarrow t_2 \longrightarrow \cdots$. Dans le cas où une forme normale existe, son unicité est assurée par la propriété de Church-Rosser ou par la confluence qui est une propriété équivalente.

Définition 1.24

1. *$\longrightarrow$ a la propriété de Church-Rosser si*

$$\xleftrightarrow{*} \subseteq \xrightarrow{*} \circ \xleftarrow{*}$$

2. *$\longrightarrow$ est confluente si*

$$\xleftarrow{*} \circ \xrightarrow{*} \subseteq \xrightarrow{*} \circ \xleftarrow{*}$$

3. $\longrightarrow$ *est localement confluente si*

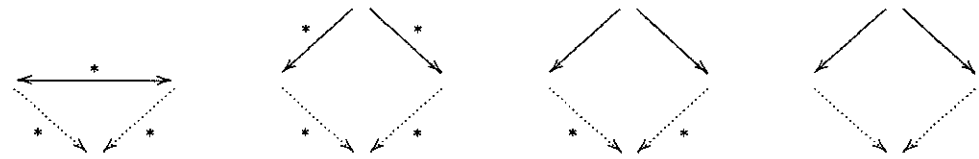
$$\longleftarrow \circ \longrightarrow \subseteq \xrightarrow{*} \circ \xleftarrow{*}$$

4. $\longrightarrow$ *est fortement confluente si*

$$\longleftarrow \circ \longrightarrow \subseteq \longrightarrow \circ \longleftarrow$$

5. $\longrightarrow$ *est convergente si* $\longrightarrow$ *termine et a la propriété de Church-Rosser.*

Ces différentes définitions se représentent chacune par un diagramme. Dès que ce sera possible, nous adopterons cet artifice typographique pour exprimer les propriétés des relations. Une flèche pleine figure une hypothèse et une flèche en pointillé une conclusion.



Church-Rosser *confluence* *confluence locale* *confluence forte*

Si une relation est fortement confluente alors elle est confluente. Si une relation est confluente alors elle est localement confluente. Une relation est confluente si et seulement si elle satisfait la propriété de Church-Rosser.

La confluence est une propriété difficile à tester. En pratique, le test de confluence se fait localement grâce au théorème suivant:

Théorème 1.2 *(Newman [New42]) Si* $\longrightarrow$ *termine, alors les propriétés suivantes sont équivalentes:*

1. $\longrightarrow$ *a la propriété de Church-Rosser,*
2. $\longrightarrow$ *est confluente,*
3. $\longrightarrow$ *est localement confluente,*
4. $\forall t, t' \in T : t \xleftrightarrow{*} t' \implies t\downarrow = t'\downarrow$.

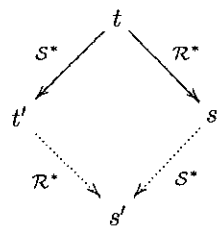
La normalisation (forte ou faible) est la seconde des deux propriétés importantes pour une relation. Si nous considérons une relation comme un calcul sur un ensemble, la normalisation forte assure que ce calcul est toujours fini; la normalisation faible assure qu'il y a un moyen de terminer tout calcul.

Définition 1.25 *Soit une relation binaire* $\longrightarrow$ *sur un ensemble* T.

- *On dit que* $t \in T$ *est une forme normale s'il n'existe pas de* $u \in T$ *tel que* $t \longrightarrow u$ *et on dit que* $v \in T$ *a une forme normale* t *s'il existe une forme normale* t *tel que* $v \longrightarrow t$.
- *La relation* $\longrightarrow$ *est faiblement normalisable (weakly normalizing) si tout terme* $t \in T$ *a une forme normale.*
- *La relation* $\longrightarrow$ *est fortement normalisable (strongly normalizing) ou normalisable s'il n'existe pas de suite infinie* $(t_i)_{i\geq 1}$ *d'éléments de* T *telle que* $t_1 \longrightarrow t_2 \longrightarrow \cdots$.

Dans la pratique, on est souvent amené à analyser les propriétés d'une relation obtenue en composant deux (ou plusieurs) relations. Plusieurs méthodes ont été développées pour démontrer la confluence d'une telle relation en fonction des propriétés des deux relations.

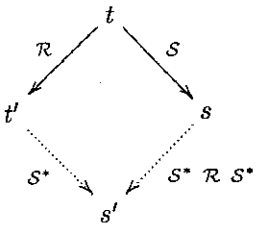
Lemme 1.1 *(Hindley-Rosen [Ros73]) Etant données deux relations confluentes* $\longrightarrow_{\mathcal{R}}$ *et* $\longrightarrow_{\mathcal{S}}$ *telles que le diagramme suivant est satisfait:*



Alors la relation $\longrightarrow_{\mathcal{R}} \cup \longrightarrow_{\mathcal{S}}$ *est confluente.*

Si le diagramme du Lemme 1.1 est satisfait on dit que les relations $\longrightarrow_{\mathcal{R}}$ et $\longrightarrow_{\mathcal{S}}$ commutent.

Lemme 1.2 *(Yokouchi [YH90]) Etant données deux relations* $\longrightarrow_{\mathcal{R}}$ *et* $\longrightarrow_{\mathcal{S}}$ *telles que* $\longrightarrow_{\mathcal{S}}$ *est confluente et terminante,* $\longrightarrow_{\mathcal{R}}$ *est fortement confluente et le diagramme suivant est satisfait :*



Alors la relation $\xrightarrow{*}_{\mathcal{S}} \longrightarrow_{\mathcal{R}} \xrightarrow{*}_{\mathcal{S}}$ *est confluente.*

Si le diagramme du Lemme 1.2 est satisfait on dit que les relations $\longrightarrow_{\mathcal{R}}$ et $\longrightarrow_{\mathcal{S}}$ sont *cohérentes.*

On peut aussi analyser les propriétés de confluence et Church-Rosser modulo une relation d'équivalence. On considère $\longrightarrow_R$ et $\longleftrightarrow_E$ deux relations binaires sur l'ensemble T, dont l'une, $\longleftrightarrow_E$, est une relation d'équivalence. On note $\longrightarrow_{R/E}$ la relation $\xleftrightarrow{*}_E \circ \longrightarrow_R \circ \xleftrightarrow{*}_E$ simulant la relation induite par $\longrightarrow_R$ sur les classes d'équivalence de $\xleftrightarrow{*}_E$.

On simule souvent la relation $\longrightarrow_{R/E}$ par une relation $\longrightarrow_S$ plus faible satisfaisant $\longrightarrow_R \subseteq \longrightarrow_S \subseteq \longrightarrow_{R/E}$. On a alors une propriété de Church-Rosser modulo E pour $\longrightarrow_S$, ainsi qu'une notion de confluence qui n'implique plus la propriété de Church-Rosser. Pour une présentation détaillée des propriétés des relations définies sur des classes d'équivalence le lecteur peut se référer à [Hue80], [JK86] et [KK99].

1.5 Les systèmes de réécriture

L'idée centrale de la réécriture [DJ90, Klo90, BN98] est d'imposer une direction dans l'utilisation des axiomes, qui sont alors orientés en règles de réécriture.

Définition 1.26 *Une règle de réécriture est une paire de termes orientée, notée* $l \to r$*, où* l *est le membre gauche de la règle et* r *son membre droit.*

Un système de réécriture sur les termes est un ensemble de règles de réécriture.

Deux conditions sont imposées habituellement sur la construction des règles de réécriture :

1. le membre gauche d'une règle de réécriture n'est pas une variable $l \notin \mathcal{X}$,
2. l'ensemble des variables du membre droit est inclus dans l'ensemble des variables du membre gauche $(\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l))$.

L'ensemble des variables d'une règle $l \to r$, noté $\mathcal{V}ar(l \to r)$, est défini par $\mathcal{V}ar(l) \cup \mathcal{V}ar(r)$ et si la condition précédente est satisfaite alors $\mathcal{V}ar(l \to r) = \mathcal{V}ar(l)$.

Une règle de réécriture est *linéaire à gauche* si son membre gauche est linéaire. Un système de réécriture est linéaire à gauche si toutes ses règles le sont.

Une règle de réécriture $l \to r$ est *régulière* si $\mathcal{V}ar(l) = \mathcal{V}ar(r)$. Un système de réécriture est régulier si toutes ses règles le sont.

La relation *de réécriture* $\longrightarrow_R$ associée à un système de réécriture R est définie par : $t \longrightarrow_R t'$ s'il existe une position p dans t, une règle $l \to r$ dans R et une substitution σ telles que $t_{|p} = \sigma l$ et $t' = t[\sigma r]_p$. Si on veut préciser la position, la règle et la substitution, alors on écrira $t \longrightarrow_{p,l\to r,\sigma} t'$.

Par application du Théorème 1.2, si la relation de réécriture $\longrightarrow_R$ est convergente, alors pour décider de l'égalité $t \xleftrightarrow{*}_R t'$, il suffit de calculer les formes normales $t \downarrow_R$ et $t' \downarrow_R$ puis de les comparer.

Définition 1.27 *Un système de réécriture* R *est convergent (resp. est confluent, termine) si la relation de réécriture* $\longrightarrow_R$ *est convergente (resp. est confluente, termine).*

1.5.1 Terminaison des systèmes de réécriture

La convergence requiert la terminaison. Cette propriété est indécidable en général même pour un système de réécriture réduit à une seule règle linéaire à gauche [HL78, Dau89]. On peut néanmoins prouver la terminaison dans certains cas. La condition suffisante la plus utilisée est l'existence d'ordres sur les termes, possédant des propriétés particulières.

Définition 1.28 *Un ordre $\succ$ est appelé ordre de réécriture si :*

1. *$\succ$ est monotone : étant donnés deux termes s et t tels que $s \succ t$, alors $w[s]_p \succ w[t]_p$ pour tout terme w et toute position p dans w,*
2. *$\succ$ est stable par instanciation : pour toute substitution σ, et pour tous termes s et t tels que $s \succ t$, alors $s\sigma \succ t\sigma$.*

De plus, $\succ$ est un ordre de réduction si :

1. *$\succ$ est un ordre de réécriture,*
2. *$\succ$ est bien-fondé (ou nœthérien) : il n'existe pas de suite infinie de termes $s_1, s_2, \ldots$ telle que $s_i \succ s_{i+1}$ pour tout $i \geq 1$.*

On assure la terminaison de la réécriture en orientant les règles de manière à ce que toute règle $l \to r$ vérifie $l > r$ où $>$ est un ordre de réduction sur les termes.

Théorème 1.3 *(Lankford [Lan77]) Le système de réécriture R termine si et seulement si $\longrightarrow_R$ est contenu dans un ordre de réduction.*

Il est souvent approprié de construire un ordre de réduction par interprétation (polynômiale) en utilisant un homomorphisme τ de termes clos vers une $\mathcal{F}$-algèbre $\mathcal{A}$ équipée d'un ordre bien fondé $>$. On note f_τ l'image de $f \in \mathcal{F}$ par τ et on demande que la contrainte de monotonicité suivante soit satisfaite :

$$\forall a, b \in \mathcal{A}, \forall f \in \mathcal{F}, a > b \;\; \textit{implique} \;\; f_\tau(\ldots,a,\ldots) > f_\tau(\ldots,b,\ldots)$$

Alors, l'ordre $>_\tau$ défini par

$$\forall s, t \in \mathcal{T}(\mathcal{F}), s >_\tau t \;\; \textit{si} \;\; \tau(s) > \tau(t)$$

est bien fondé.

Afin de comparer les termes contenant des variables, les variables sont introduites dans $\mathcal{A}$ menant à $\mathcal{A}(\mathcal{X})$ et aux variables de $\mathcal{X}$ on fait correspondre des variables distinctes dans $\mathcal{A}(\mathcal{X})$. L'ordre $>_\tau$ est étendu en définissant

$$\forall s, t \in \mathcal{T}(\mathcal{F},\mathcal{X}), s >_\tau t \;\; \textit{si} \;\; \alpha(\tau(s)) > \alpha(\tau(t))$$

pour toute assignation α des valeurs dans $\mathcal{A}$ aux variables de $\tau(s)$ et $\tau(t)$. Puisque $>$ est supposé bien fondé, on peut montrer la terminaison d'un système de réécriture si on trouve $\mathcal{A}$, τ et α satisfaisant les conditions précédentes.

Dans la pratique on utilise très souvent l'algèbre des entiers naturels avec l'ordre habituel et des interprétations polynômiales et exponentielles.

Exemple 1.5 *On considère le système de réécriture suivant*

$$\begin{array}{lcl} \ominus \ominus x & \to & x \\ \ominus(x \oplus y) & \to & (\ominus x) \oplus (\ominus y) \\ \ominus(x \otimes y) & \to & (\ominus x) \otimes (\ominus y) \\ x \otimes (y \oplus z) & \to & (x \otimes y) \oplus (x \otimes z) \\ (x \oplus y) \otimes z & \to & (x \otimes y) \oplus (x \otimes z) \end{array}$$

En utilisant l'interprétation exponentielle ci-dessous dans les entiers supérieurs à 2

$$\begin{array}{lll} \tau(\ominus x) & \to & 2^{\tau(x)} \\ \tau(x \oplus y) & \to & \tau(x) + \tau(y) + 1 \\ \tau(x \otimes y) & \to & \tau(x) \times \tau(y) \\ \tau(c) & \to & 3 \end{array}$$

pour toute constante $c \in \mathcal{F}$, le système a été montré terminant dans [Fil78].

Par exemple, pour la première règle on a $2^{2^n} > n$ pour tout entier $n > 2$ assigné à la variable x. Utiliser une interprétation dans les entiers positifs ne serait pas suffisant pour montrer les inégalités correspondant aux deux dernières règles et considérer les entiers supérieurs à 1 ne serait pas suffisant pour montrer l'inégalité correspondant à la troisième règle.

De nombreux auteurs ont décrit des ordres de réduction sur les termes. Parmi les plus connus, citons l'ordre de Knuth-Bendix ou *kbo* [KB70], les ordres sur les chemins [Pla78, Der82, BP85, JLR82], les interprétations polynômiales [Lan75, BCL87] ou encore des ordres comme celui défini par D. Kapur, G. Sivakumar et H. Zhang dans [KSZ90] qui est un ordre de réduction AC-compatible basé sur le RPO.

Un ordre de réduction total contient la relation de sous-terme strict. Dans le cas contraire, si $t_{|\omega} > t$ pour un terme t et une position ω, alors il existe une chaîne infinie décroissante $t > t[t]_\omega > t[t[t]_\omega]_\omega > \ldots$.

L'utilisation des ordres de réduction étant en général indécidable, on a introduit la notion d'ordre de simplification.

Définition 1.29 *On dit qu'un ordre $\succ$ est un ordre de simplification si :*

1. *$\succ$ est bien-fondé,*
2. *$\succ$ est monotone,*
3. *$\succ$ est stable par instanciation,*
4. *$\succ$ a la propriété du sous-terme : si s est un sous-terme strict de t, alors $t \succ s$.*

De plus, $\succ$ est total sur l'ensemble des termes clos si, pour tous termes clos s et t, on a soit $s \succ t$, soit $t \succ s$, soit $s = t$.

Théorème 1.4 *(Dershowitz [Der82]) Soit $\mathcal{F}$ un ensemble fini de symboles de fonctions. Un système de réécriture R termine s'il existe un ordre de simplification $>$ tel que pour toute règle $l \to r$ de R, $l > r$.*

Les ordres de simplifications peuvent être construits à partir d'un ordre sur les symboles de fonctions $\mathcal{F}$ appelé *précédence*. Parmi les ordres de simplifications on peut citer l'*ordre multi-ensemble sur les chemins* [Der82] et l'*ordre lexicographique sur les chemins* [KL80]. Pour plus de détails concernant la terminaison, nous renvoyons le lecteur à [Der87].

1.5.2 Les systèmes de réécriture conditionnels

En ajoutant des conditions à l'application des règles de réécriture, on étend naturellement les systèmes de réécriture à des systèmes de réécriture *conditionnels*. Plusieurs définitions des systèmes de réécriture conditionnels ont été proposées et la correspondance entre ces systèmes et leur relation avec les systèmes équationnels a été analysée dans [DO90]. La différence essentielle entre les systèmes conditionnels est l'interprétation des conditions. Nous présentons par la suite quelques unes de ces définitions.

Un système de réécriture *conditionnel naturel* (*natural conditional rewriting system*) a des règles de réécriture de la forme

$$l \to r \ \text{si} \ s_1 \overset{*}{\longleftrightarrow} t_1 \wedge \ldots \wedge s_n \overset{*}{\longleftrightarrow} t_n$$

où les $s_i \overset{*}{\longleftrightarrow} t_i$ sont appelées les conditions de la règle.

La règle $l \to r$ est appliquée dans le sens de la réécriture non-conditionnelle si, pour toute condition $s_i \overset{*}{\longleftrightarrow} t_i$, $i = 1 \ldots n$, instanciée par la substitution appropriée, il existe une preuve pouvant utiliser un nombre quelconque de réécritures dans les deux directions. Si $n = 0$ on obtient une règle non-conditionnelle.

Puisque l'application de telles règles implique des preuves arbitraires d'égalité où la réécriture n'apporte pas beaucoup de bénéfice par rapport aux systèmes équationnels, on peut utiliser une définition plus restrictive de la réécriture conditionnelle.

Un système de réécriture *conditionnel standard* (*standard (join) conditional rewriting system*) a des règles de réécriture de la forme

$$l \to r \quad \text{si} \quad s_1 \downarrow t_1 \wedge \ldots \wedge s_n \downarrow t_n$$

Dans ce cas, une instance $\sigma(l)$ du membre gauche de la règle est réécrite en $\sigma(r)$ seulement si, pour tout $i = 1 \ldots n$, $\sigma(s_i)$ et $\sigma(t_i)$ peuvent être réduits (en utilisant zéro ou plusieurs réécritures) en un même terme.

La condition d'application pour une règle de réécriture peut être encore plus affaiblie. Un système de réécriture *conditionnel normal* (*normal conditional rewriting system*) a des règles de réécriture de la forme

$$l \to r \quad \text{si} \quad s_1 \longrightarrow^! t_1 \wedge \ldots \wedge s_n \longrightarrow^! t_n$$

où $s_i \longrightarrow^! t_i$ indique que t_i est une forme normale de s_i.

Un système standard contenant des règles de la forme

$$l \to r \quad \text{si} \quad s_1 \downarrow t_1 \wedge \ldots \wedge s_n \downarrow t_n$$

peut être transformé dans un système normal où les règles sont remplacées par

$$l \to r \quad \text{si} \quad eq(s_1,t_1) \longrightarrow^! true \wedge \ldots \wedge eq(s_n,t_n) \longrightarrow^! true$$

et la règle

$$eq(x,x) \to true$$

est ajoutée au système. Les réductions des termes ne contenant pas les symboles eq et $true$ sont similaires dans les deux systèmes.

Conclusion

Nous avons donc défini dans cette partie les notions de termes, de substitution, de systèmes de règles de réécriture. Nous pouvons donc maintenant décrire plus précisément des systèmes à base de règles dans leur généralité : les systèmes de règles de production, mais aussi les systèmes basés sur les règles de réécriture.

Chapitre 2

Systèmes à base de règles

La déduction par des raisonnements par cas du type "si" *condition* "alors" *conséquence* est un procédé que l'on utilise régulièrement et de façon quasi intuitive. C'est peut-être grâce à l'aspect intuitif de ce type de raisonnement que les règles de déduction sont utilisées dans de nombreux systèmes.

On s'intéresse dans ce chapitre à différentes utilisations possibles des règles. A la fois comme moyen de représentation des connaissances pour des systèmes experts appelés *Systèmes de Règles de Production*, mais aussi comme mécanisme de base d'un système permettant de faire du calcul basé sur la logique de réécriture. On verra que les règles peuvent aussi être combinées avec d'autres paradigmes tels que les contraintes ou les objets et ainsi permettre de définir des systèmes parfois complexes.

On présente donc dans une première partie les systèmes dits systèmes de règles de production avec une présentation du système CLIPS. Ensuite, on présentera différents systèmes qui, bien que n'étant pas des systèmes de règles de production, nous intéressent car ce sont des systèmes dans lesquels on retrouve les concepts qui ont présenté un intérêt tout particulier dans notre étude, à savoir les règles, les objets et les contraintes. On présente ainsi CLAIRE qui est un système avec règles et objets, CRP(FD) qui est un système avec règles et contraintes puis Maude qui est un langage basé sur la logique de réécriture dans lequel on retrouve objets et règles. Finalement, on présente le système ELAN basé lui aussi sur la logique de réécriture dans lequel on trouve règles et stratégies. Le but est d'adapter ce système à la formalisation de systèmes de règles de production et ainsi de développer un formalisme basé sur la logique de réécriture.

2.1 Les systèmes de règles de production

Dans notre étude, c'est surtout l'aspect de la mécanisation du raisonnement qui nous intéresse et plus particulièrement une classe de programmes que l'on appelle les systèmes experts, ou encore systèmes à base de connaissances. De nombreux ouvrages de référence traitent des systèmes experts. On peut citer les travaux de F. Hayes-Roth [HR85], les travaux de Giarratano et Riley [GR89], ceux de Waterman [Wat86], ceux de Jackson [Jac99] ou, plus récemment, ceux du projet Sachem [LG99]. Dans ce qui va suivre, on présentera le mécanisme général des systèmes experts. Ensuite, on exposera un type particulier de système expert que sont les systèmes de règles de production.

Les premiers systèmes experts sont apparus dans les années soixante-dix. Le but de tels systèmes était de mécaniser des tâches lourdes de raisonnement habituellement dédiées à des experts, telles que le diagnostic ou la planification de tâches. A partir d'un ensemble de connaissances fournies initialement et d'un mécanisme d'inférence, on enrichit un ensemble de faits initiaux par de nouveaux faits, de nouvelles données.

Les parties fondamentales d'un système expert sont la base de connaissance et le système d'inférence qui déduit de nouveaux faits à partir de faits initiaux. Les connaissances utilisées sont obtenues à partir d'experts du domaine étudié, puis formalisées et introduites dans la base de connaissances.

Suivant le problème considéré, le moteur d'inférence cherche soit à saturer la base de faits, soit à déduire un fait bien précis. Par contre, l'utilisateur peut interagir sur le mécanisme d'inférence pour enrichir la base de faits en en entrant de nouveaux ; dans des situations critiques, le système peut aussi

être amené à demander des informations et connaissances complémentaires à l'utilisateur-expert.

De nombreux systèmes experts spécialisés dans des domaines d'application très précis ont été réalisés : la médecine, la prospection minière, etc. On peut citer quelques exemples bien connus de tels systèmes : DENDRAL [Sut67, LBFL80], datant des années soixantes pour le domaine de la chimie, déterminant la structure de composés organiques ; PROSPECTOR [HD77], datant des années soixante-dix, évaluant le potentiel minéral de certaines régions ; MYCIN [Sho76], datant des années soixante-dix, pour le diagnostic médical. Plus récemment, quelques systèmes ont été utilisés dans le domaine militaire.

La base de connaissance peut être représentée à partir de nombreux formalismes [Koc91] incluant par exemple la représentation à base de règles, les représentations basées sur les tableaux - ou "frames" -, la logique des prédicats, les réseaux sémantiques, les classificateurs ou les réseaux de neurones. On peut aussi combiner tout ou partie de ces représentations. Ces méthodes de représentation ne sont pas toutes de même nature. En effet, en utilisant les règles, on définit les connaissances à partir de règles du type :

SI *condition* ALORS *action*

plus ou moins élaborées suivant le langage de règles utilisé. Dans une représentation de type *frame* ou réseaux sémantiques, on définit un réseau de noeuds connectés par des relations. Une hiérarchie est aussi définie dans de tels réseaux et un mécanisme d'héritage lie chaque composant : les noeuds héritent de propriétés de ses ancêtres hiérarchiques.

La représentation des connaissances qui nous intéresse particulièrement est basée sur le formalisme de règles. De nombreux systèmes experts utilisent ce formalisme qui a la particularité d'être naturel et facilement lisible lorsque l'on cherche à décrire un ensemble conséquent de connaissances.

Les systèmes de règles de production sont alors une classe de systèmes experts dans laquelle une partie de la connaissance du système est représentée par un ensemble de règles. Les systèmes de règles de production sont donc constitués de trois éléments :

- La mémoire de travail, qui est une base de données globale constituée d'expressions représentant les faits du problème. Les données sont des instances d'objets, qui peuvent représenter des objets physiques ou des faits liés au domaine d'application.
- Un ensemble de règles qui constitue le programme ou la base de connaissances. Chaque règle a une partie conditionnelle IF qui décrit la configuration dans laquelle la règle peut s'appliquer. Chaque règle a aussi une partie ACTION qui décrit les instructions à exécuter lorsque la règle s'applique sur la mémoire de travail.
- Un outil d'inférence (ou moteur d'inférence) qui permet d'exécuter les règles. On doit alors avoir dans notre outil un mécanisme de sélection des règles qui permet de choisir les règles pouvant s'appliquer ; et aussi, parmi différentes règles possibles, de choisir laquelle sera effectivement appliquée.

On détaille dans la suite comment sont représentés les éléments de la mémoire de travail ainsi que le moteur d'inférence.

2.1.1 Représentation des éléments de la mémoire de travail

La fonction principale de la mémoire de travail est de mémoriser des informations de la forme objet/attribut/valeur. Ces informations doivent être consultables et modifiables par le système de règles de production. Cet ensemble peut être ou non structuré en différents sous-ensembles ; par exemple, un pour les faits, un autre pour les sous-buts à réaliser.

Afin de représenter chaque élément de la mémoire de travail, on peut utiliser différents formalismes allant de simples chaînes à des structures de données complexes. De là, on peut aussi composer de tels éléments via des listes, des ensembles, des multi-ensembles...

Le choix fait pour la représentation des faits doit prendre en compte le fait que l'on cherche à manipuler la mémoire de travail de diverses façons :

- on parcourt la base de faits afin de rechercher un élément pouvant déclencher l'application d'une règle particulière ;
- on modifie un élément en modifiant certaines de ses caractéristiques.

- on efface un élément de la mémoire quand ce dernier n'a plus sa place dans la mémoire de travail : il peut être remplacé par un autre élément ou par plusieurs autres.
- on ajoute de nouveaux éléments à la mémoire suite à l'application d'une règle.

Outre les informations propres au problème traité, de nombreux facteurs peuvent être associés aux éléments de la mémoire de travail. On peut ainsi prendre en compte l'âge des éléments de la base pour des systèmes qui considèrent que les éléments les plus récents sont les plus prioritaires. Certains systèmes travaillent aussi avec des données qui ont une certaine probabilité de véracité : dans des systèmes tels que MYCIN, à partir d'un symptôme donné, on peut diagnostiquer plusieurs maladies, mais chacune a une plausibilité plus ou moins élevée d'être la maladie réellement incriminée.

2.1.2 Un outil d'inférence

On peut avoir deux types de mécanismes dans l'application des règles : un mécanisme de déduction type *forward-chaining* où une règle est choisie lorsqu'un changement dans la mémoire de travail produit une situation qui filtre les prémisses de la règle. On peut alors avoir différentes politiques d'application des règles (on n'autorise pas la même règle à s'appliquer plus d'une fois par exemple) suivant le système utilisé. L'autre mécanisme implanté par le moteur d'inférence est le type *backward-chaining* où on est guidé par le but à atteindre : le mécanisme d'inférence commence avec ce but et, successivement, examine les règles dont les conclusions correspondent au but que l'on souhaite atteindre. Lorsqu'une règle est sélectionnée, ses prémisses deviennent les nouveaux buts à atteindre.

Ainsi, le moteur d'inférence, appelé aussi interpréteur, à partir d'un ensemble de règles de production a un comportement que l'on peut décrire de la façon suivante, en trois pas :

- il filtre les prémisses ou les conclusions des règles avec les éléments de la base de faits,
- s'il y a plusieurs règles pouvant être appliquées, il choisit celle qui sera appliquée,
- il applique la règle et retourne au premier pas.

Les premiers systèmes de règles de production passaient énormément de temps sur le filtrage. La famille de langages à base de règles OPS basé sur un mécanisme de *forward-chaining*, dont le dernier en date est OPS5 [For81, Gro95, BFKM85], utilise par contre un algorithme de filtrage *Rete* mis au point par C.L. Forgy. L'algorithme *Rete* est un algorithme de filtrage dit optimisé par rapport à la génération précédente d'algorithmes au sujet desquels on peut faire les remarques suivantes :

- les membres gauches de règles de production partagent souvent des mêmes conditions et une approche naïve consiste à essayer de filtrer ces conditions à la mémoire de travail autant de fois qu'on trouve d'occurrences,
- la mémoire de travail n'est que très peu modifiée à chaque étape et une approche naïve ne prend pas en compte cette faible modification ; elle essayera à chaque nouvelle étape de filtrer sur l'ensemble de la mémoire de travail.

Rete a été conçu pour corriger ces erreurs.

2.1.3 Un système expert à base de règles et d'objets : CLIPS

On présente dans cette partie le langage CLIPS. Celui-ci est en effet intéressant à plusieurs titres. C'est en effet tout d'abord un langage phare dans le développement de systèmes experts. De plus, on retrouve dans ce langage une partie des aspects qui nous intéressent plus particulièrement : les règles et les objets. A savoir que des contraintes peuvent de plus être représentées par des objets en CLIPS ; de même qu'une forme de contrôle sur l'application des règles peut être lui aussi défini grâce au langage procédural associé au système.

CLIPS [GR89, Jac99] (*C Language Integrated Production System*) est un langage de programmation qui a été développé dans le milieu des années 80 au centre de recherche de la NASA. Initialement, CLIPS était un simple interpréteur de règles de production ; le langage procédural le composant puis les aspects

objets avec l'intégration de COOL (*CLIPS Objet-Oriented Language*) n'ont été intégrés quant à eux au système qu'à partir des années 90.

L'application CLIPS permet un travail interactif avec la base de faits : on peut initialiser la base, ajouter des faits ou des listes de faits, retirer des faits ou des listes, afficher l'ensemble de la base de faits.

Les règles, qui sont des règles nommées, ont un format bien déterminé comprenant, outre des déclarations optionnelles, une liste de prémisses et une liste d'actions. Parmi les déclarations optionnelles, on peut définir un degré de priorité des règles qui permet, lorsque plusieurs règles peuvent s'appliquer à un même moment, de déterminer quelle règle doit être alors appliquée.

Par exemple, considérons une règle appelée `OrganisationSunday` qui fait partie d'un ensemble de règles donnant un programme de tâches à effectuer suivant le jour considéré, dans notre exemple, le jour considéré est le dimanche :

```
(defrule OrganisationSunday
 "Things to do on sunday"
 (salience 10)
 (today is sunday)
 (weather is warm)
 =>
 (assert (wash car))
 (assert (mow lawn))
)
```

La règle `OrganisationSunday` est commentée par `"Things to do on sunday"` qui indique ce qu'elle définit : les tâches à effectuer le dimanche. Une priorité est ensuite définie pour cette règle : elle est mise à 10. Ensuite, les prémisses sont définies : le jour doit être dimanche (`today is sunday`) et le temps doit être au beau fixe (`weather is warm`). La règle peut alors s'appliquer et on ajoute les faits de laver la voiture (`wash car`) et de tondre le gazon (`mow lawn`).

En CLIPS, on peut définir des fonctions grâce à un langage procédural. Chaque fonction a un nom, une liste d'arguments et une suite d'instructions dont la dernière est le résultat retourné par la fonction. Ces fonctions peuvent être notamment utilisées dans les actions des règles.

Les objets sont définis dans CLIPS grâce au langage orienté objet COOL. On définit tout d'abord un ensemble de classes dans COOL. Ensuite, on défini les objets comme des instances de ces classes. On peut alors définir des règles pouvant interagir avec les objets par envoi de message. Les règles contrôlent toujours les calculs mais délèguent certains traitements des données aux objets. Les objets ne font pas partie de la mémoire de travail composés des seuls faits mais les membres gauches des règles peuvent filtrer sur les valeurs des attributs des objets. Les objets ne peuvent pas "appeler" des règles mais peuvent retourner des valeurs utilisées dans les règles.

De tels systèmes que l'on peut qualifier d'hybrides, puisque mélangeant des concepts différents, ont leur propres systèmes de vérification. On peut citer entre autres les travaux de S. Lee et R.M. O'Keefe [LO93] qui proposent des méthodes de détection d'anomalies de subsumption dans le cas de systèmes hybrides combinant des règles et une approche orientée objet.

2.1.4 Vérification de systèmes de règles de production

De part leur complexité, les systèmes à base de connaissances requièrent des méthodes appropriées pour résoudre des problèmes de fiabilité des systèmes de règles de production [GR89, LA91, HR85]. Les méthodes de validation sont difficiles, coûteuses en temps ; néanmoins, plusieurs outils ont été développés afin de répondre à ces attentes. On détaille par la suite quelques approches particulières.

Les travaux pionniers dans les techniques de vérification de bases de connaissances sont ONCOCIN [SSS82] et CHECK [NPLP87] qui furent développés pour des systèmes spécifiques.

Le système ONCOCIN [SSB+81] est un consultant à base de règles permettant de conseiller les médecins d'un service d'oncologie dans la gestion de patients suivant des protocoles basés sur des traitements expérimentaux. Des techniques ont alors été développées dans [SSS82] pour vérifier la correction et la complétude de la base de connaissances afin de tester et raffiner les connaissances du système.

CHECK [NPLP87], développé par Tin A. Nguyen, Walton A. Perkins et Thomas J. Laffey, est un programme vérifiant la consistance et la complétude de bases de connaissances construites pour le système LES (*Lockeed Expert System*) [LPN86] qui est un système à base de connaissances proche de EMYCIN [VM81]. CHECK a néanmoins été conçu pour pouvoir s'appliquer à d'autres systèmes. CHECK combine des principes logiques et les informations spécifiques au formalisme de représentation de la base de connaissances de LES. Le programme vérifie à la fois les règles guidées par le but et celles guidées par les faits. CHECK détecte les inconsistances de la base de connaissances en cherchant les règles redondantes, conflictuelles et subsumées, en détectant les conditions *IF* inutiles et les cycles. Les tests de complétude de la base sont faits par recherche de valeurs d'attributs non référencées, de valeurs d'attributs illégales, de conditions ou buts non atteignables. Ces tests peuvent être utilisés pour suggérer des règles manquantes ou des failles dans la base de connaissances. On permet aussi la génération d'un graphe de dépendance des règles.

Par la suite, des outils basés sur une plus grande variété d'approches différentes furent développés afin de détecter plus de cas subtils d'erreurs potentielles dans les bases de connaissances. On peut citer parmi ceux-ci EVA [SC87] et COVADIS [Rou88]

EVA (*Expert system Validation Associate* a été développé par Rolf A. Stachowitz et Jacqueline B. Combs [SC87]. EVA permet de définir et de développer des outils automatiques afin de valider l'intégrité structurelle (propriétés syntaxiques des règles, des clauses et des faits dans la mémoire de travail), logique et sémantique du système à base de connaissances. Le but d'EVA est d'améliorer le processus de validation d'un système en trouvant des erreurs et omissions dans la base de connaissance, en proposant des modifications par extension et par généralisation de la base de connaissances et en montrant l'impact de ces changements. Les vérifications structurelles et logiques consistent en deux versions chacune : une première version est réalisée par des algorithmes ne prenant pas en compte les informations sémantiques ; la seconde utilise de l'information sémantiques contenue dans la base de méta-connaissances. Les tests sémantiques sont réalisés par des méta-règles qui travaillent sur les méta-règles, méta-faits, règles et faits. La méta-connaissance apparaît sous la forme de connaissance générale, propre au domaine et spécifique à l'application traitée. La méta-connaissance est aussi nécessaire pour l'induction d'extensions ou généralisations possibles de la base de connaissances et aussi dans la vérification dynamique de la base de connaissances en cours d'exécution. L'implémentation d'EVA est réalisée en Lisp pour des bases de connaissances construites avec ART [Ngu87] (*Automated Reasoning Tool*).

Le système COVADIS [Rou88] est une méthode complète permettant de prouver la cohérence de bases de connaissances écrites de la forme attribut/valeur. COVADIS fonctionne de la façon suivante : à partir de tous les faits déductibles, on déduit par une méthode de *forward chaining* les différentes bases de faits permettant de les générer. COVADIS utilise le moteur d'inférence MORSE utilisé aussi comme moteur d'inférence des systèmes à base de connaissances pouvant être vérifiés par COVADIS.

D'autres outils de vérification où ces dernières sont basées sur d'autres approches ont été développés. Ainsi, une méthode de vérification capable de tester l'inconsistance d'une base de connaissances exprimée en logique propositionnelle est basée sur la traduction des règles dans un réseau de Pétri et sur la résolution du système d'équations linéaires associé ; cette méthode a été utilisée et développée dans PREPARE [ZN94]. On peut trouver dans [AL91] un ensemble de travaux et plusieurs autres systèmes de vérification de bases de connaissances utilisant diverses autres approches.

Une caractéristique des systèmes de vérification précédents est que les méthodes qui sont alors développées sont des processus informels souvent très liés aux exemples traités. On peut noter que l'approche de COVADIS [Rou88] est néanmoins plus formelle que les autres. Certains tests sont donc réalisés sur la base de connaissances en considérant plusieurs scénarios possibles que le système peut être amené à traiter. Puisque l'ensemble de tous les scénarios ne peut pas être considéré, tout l'espace du problème n'est pas couvert par les tests. Des techniques de vérification formelles ont alors vu le jour comme celles développées par Gamble, Roman, Ball et Cunningham dans [GRBC94]. Deux types de propriétés sont alors prouvées : des propriétés de sécurité qui garantissent que le programme ne fait rien de faux et des propriétés de progression qui garantissent que le programme fait éventuellement des choses utiles. Toutes les vérifications habituellement traitées dans la littérature (par exemple, la consistance des informations, la génération d'un résultat correct et la terminaison) peuvent être reformulées en terme de propriétés de sécurité et de progression devant être satisfaites par le système. Un des avantages de cette méthode est que ces techniques peuvent être utilisées à la fois sur des systèmes séquentiels ou concurrents.

Plus récemment, un algorithme a été développé par A. Ginsberg et K. Williamson dans [GW93] pour tester consistance et redondance dans des bases de connaissances qui sont composées de règles qui peuvent être représentées par des spécifications logiques du premier-ordre.

James G. Schmolze et Wayne Snyder se sont aussi intéressé à la vérification de systèmes de règles de production. Leurs premiers travaux ont concernés les systèmes parallèles [SS93]. Leur spécificité et ce qui nous intéresse dans leurs travaux est le fait d'avoir utilisé les méthodes de vérification de la réécriture pour pouvoir vérifier des bases de connaissances à base de règles ; cela après avoir défini un lien entre les systèmes de règles de production et les systèmes de réécriture de termes. Ainsi, dans [SS93], ils utilisent les méthodes de confluence pour contrôler les systèmes de règles de production parallèles ; plus généralement, dans [SS94, SS95, SS96], ils étudient la propriété de confluence pour tout système de règle de production. La redondance est aussi étudiée dans [SS97]. L'approche par règles de réécriture développée dans cette thèse permet d'appliquer des techniques de vérification analogues.

2.2 Etat actuel des systèmes experts

On présente tout d'abord des conclusions qui ont été tirées au milieu des années quatre-vingt dix sur le thème des systèmes experts. On estime alors à environ deux mille cinq cents [Dur93] le nombre de systèmes experts qui ont été installés dans des administrations ou des entreprises. On estime aussi que plus de 50% de ces systèmes ne sont pas utilisés [MG91]. Quelles peuvent être les raisons d'une telle sous-exploitation des systèmes experts produits? Quels sont les enseignements que l'on peut tirer à partir de leurs failles et de leurs succès?

Un premier facteur concerne les connaissances du domaine d'application du système. En effet, on ne connaît pas forcément le type de connaissance que l'on peut acquérir et le type de connaissance que l'on veut acquérir n'est pas forcément pris en compte (pas défini, pas de règles permettant l'acquisition, etc...) par le système. Un second problème est dû au manque de considération du contexte des connaissances : certaines connaissances prennent une autre dimension suivant la situation dans laquelle on se trouve. Une des dernières limitations concerne la taille de la base de connaissances qui peut être très importante dans des systèmes réalistes. On est alors confronté aux problèmes de maintenance, de mise à jour, de validation et de vérification de telles bases.

D'autres problèmes concernent les utilisateurs des systèmes et la façon dont ces systèmes sont intégrés dans l'entreprise l'utilisant. Ces problèmes ne sont pas détaillés ici car non directement reliés à notre étude. On peut trouver plus de détails sur ces problèmes dans [Bré98, DCKH95, War96, EHR+95].

Les systèmes experts dans leur généralité répondent aussi difficilement aux problèmes de temps-réel. Certains systèmes doivent pouvoir évaluer en temps réel l'ensemble d'une situation à partir de la connaissance des objets, de leurs relations respectives et de leurs comportements. Un système peut aussi devoir prendre des décisions en temps-réel. Réaliser un tel système suppose donc d'être performant en temps d'exécution sur la machine cible temps-réel ainsi que dans les mécanismes de communication avec les modules d'acquisition ou de traitement des données écrits dans des langages procéduraux. Pour répondre à de tels besoin, des solutions ont été développées parmi lesquelles l'outil XRETE [AF92] par exemple. XRETE est un outil permettant la mise en place de systèmes experts fonctionnant en temps-réel.

Néanmoins, les systèmes actuels ont tirés parti de ces expériences et ont fortement évolués depuis ces années quatre-vingt dix. On voit maintenant des systèmes dans lesquels les notions présentées ici sur les systèmes experts sont réutilisées, mais aussi dans lesquels de nouvelles techniques sont aussi utilisées afin de les rendre plus performants, plus maniables. En effet, maintenant, différents modes de représentation des faits et des connaissances sur le domaine d'application cohabitent. Ainsi, différents types de mécanismes de déduction de nouveaux faits sont utilisables. Les systèmes développés peuvent aussi bien utiliser des objets que des agents plus adaptés au multimédias que d'autres formes de représentations des données. Différents modèles tels que des modèles cognitifs, des modèles appropriés (par exemple, de type connexionniste) pour simuler les fonctions cognitives humaines peuvent aussi être utilisés. Une branche très active dans la conception de tels systèmes est aussi celle du *knowledge management*. Ce domaine cherche à systématiser le processus de recherche, de sélection, d'organisation et de présentation de l'information de telle sorte que la compréhension d'un domaine d'étude spécifique soit améliorée.

2.3 Différents systèmes à base de règles

Partant de la volonté de formaliser les systèmes de règles de production en logique de réécriture, on s'intéresse plus particulièrement dans cette partie à des formalismes utilisant le paradigme de règles mais qui ne s'apparentent pas aux systèmes de règles de production tels qu'ils l'ont été définis précédemment.

Les systèmes présentés dans cette partie utilisent tous le concept de règles et le mélangent avec d'autres formalismes comme les objets ou les contraintes. On présente dans une première partie le langage CLAIRE alliant règles et objets. Ensuite, on présente des travaux sur CRP(FD) où B. Liu, J. Jaffar et R.H.C. Yap présentent un système dans lequel règles et contraintes sont associées. Finalement, on présente le langage Maude basé quant à lui sur la logique de réécriture.

Les systèmes présentés dans cette partie nous ont tous intéressés à différents stades de notre étude. Ainsi, CLAIRE nous a particulièrement intéressé par le fait qu'il combine habillement objets et règles; CRP(FD) nous a aidé à formaliser les liens pouvant exister entre la base de contraintes, la base d'objets et les règles manipulant ces deux ensembles; Maude est quant à lui un système, parmi l'ensemble de ceux basés sur la logique de réécriture, ayant déjà eu une première approche dans la gestion des objets.

2.3.1 Un système basé sur objets et règles: CLAIRE

Par rapport aux autres systèmes présentés plus avant, on présente dans cette partie un langage, CLAIRE [CL96], qui est tout d'abord un langage orienté objet et auquel la possibilité d'utiliser des règles a été ajoutée.

Les spécificités de CLAIRE sont notamment que le système autorise les types intervalles et fonctionnels avec un typage dynamique et statique. Les classes et méthodes peuvent être paramétrées. On y trouve également un système de *version* permettant, lors de l'exploration de l'arbre de recherche, de pouvoir mémoriser un état courant et d'y revenir après exploration d'une branche quelconque. La notion de règles de production est aussi supportée par CLAIRE.

La langage CLAIRE est donc conçu pour réaliser des applications impliquant des structures de données complexes, un calcul à base de règles et de la résolution de problème. CLAIRE est un langage s'intégrant parfaitement bien dans un environnement C++ et ce langage a bénéficié de l'expérience acquise suite aux travaux sur le langage LAURE [Cas91]. Le concept de LAURE était d'ajouter aux notions de langage orienté objet la notion de contraintes.

CLAIRE est un langage objet à héritage simple où tout élément existant est un objet appartenant à une seule classe et avec une identité unique. Les classes permettent de définir les méthodes (ou procédures), les champs et relations. Chacune de ces classes est intégrée dans une hiérarchie de classes.

Une règle de CLAIRE est l'association d'une condition logique avec une expression. La règle est attachée à une ou deux entités: chaque fois qu'une condition est vraie, l'expression est évaluée pour ces entités. L'intérêt des règles est d'attacher une expression non pas à un appel fonctionnel mais à une condition logique. Ainsi, une règle lie une condition à une conclusion (ou une action). A chaque fois qu'une condition est évaluée à *vrai*, pour une paire d'objets à cause d'un évènement, la conclusion est exécutée. Les conditions sont exprimées comme des formules logiques sur une ou deux variables qui représentent des objets sur lesquels la règle peut s'appliquer. La conclusion est une expression CLAIRE qui utilise les mêmes variables. Un évènement est une évolution d'un de ces objets. Les conditions des règles ne sont évaluées que lorsque l'évènement a eu lieu.

Pour définir une règle, on doit ainsi définir une ou deux variables libres qui sont introduites comme paramètres de la règle, une condition, qui est donnée comme une assertion utilisant les variables définies précédemment et une conclusion qui est précédée par =>. On donne ci-dessous une règle définissant une fermeture transitive:

```
r1(x:person, y:person) :: rule(
       exists(z, x % z.friend & z % y.friend )
       => y.friend :add x )
```

Cette règle nommée `r1` fonctionne comme suit: on prend deux personnes `x` et `y` et on recherche l'existence d'une tierce personne `z` telle que la personne désignée par la variable `x` est une amie de la personne désignée par la variable `z` et la personne désignée par la variable `z` est une amie de la personne

désignée par la variable y. Dans ce cas, on ajoute la personne désignée par la variable x dans la liste des amis de la personne désignée par la variable y.

Le système fonctionne ensuite par application de telles règles tant que des nouveaux évènements ont eu lieu et permettent l'application de l'une ou de l'autre des règles. Les règles peuvent être regroupées en ensembles pour gérer plus facilement l'application de tel ou tel groupe de règles.

Le système global permet ainsi de déclencher l'évaluation par règles après déclenchement d'un évènement initial. On peut aussi mélanger l'utilisation des règles avec des chargements de définition de classes ou fonctions, ainsi qu'avec la mémorisation d'état (ou de version) afin de permettre de revenir à un état précédent du système si la branche explorée ne convient pas.

2.3.2 Un système à base de règles et de contraintes : CRP(FD)

Partant du fait que les systèmes à base de règles sont très largement utilisés et diffusés mais que de tels systèmes répondent mal à des notions de choix et de disjonctions, B. Liu, J. Jaffar et R.H.C. Yap ont proposé dans [LJY99] un formalisme appelé CRP(FD) permettant de faire des choix suivant l'état de la base de connaissances et de faire marche arrière lorsqu'un choix effectué n'est pas en accord avec des critères que l'on a préalablement fixé.

Afin de répondre à cette attente, ils ont décidé d'intégrer les mécanismes de contraintes aux systèmes à base de règles. L'environnement qu'ils ont mis en place est composé des éléments suivants :

- une mémoire de travail : un ensemble de faits représentant l'état courant de l'application développée. Ces faits peuvent être composé de variables présentes aussi dans la contrainte,
- une mémoire de règles : un ensemble de règles de la forme **SI** *conditions* **ALORS** *actions*. La partie conditionnelle de la règle peut comporter des tests sur la contrainte et la partie *actions* de la règle peut elle aussi comporter des opérations propres aux contraintes,
- le moteur d'inférence : il applique les règles sur la mémoire de travail avec un mécanisme de *forward-chaining*,
- une base de contraintes : c'est la représentation de l'état courant des contraintes du système,
- un solveur de contraintes : il utilise les tests de consistance et la recherche par *backtrack* pour résoudre la contrainte. Il met aussi à notre disposition un ensemble de services permettant de connaître l'état courant de la base de contraintes.

Les membres gauches des règles peuvent comporter des tests à effectuer sur la contrainte. Un test de contrainte, noté *cst-test* est de la forme :

cst-test(démon,fonction)

où *démon* est un démon vérifiant tout changement d'état de la contrainte. Son utilisation est décrite plus loin. *fonction* est une fonction qui retourne *vrai*, *faux* ou *possible*. Les définitions des valeurs *vrai* et *faux* que peut retourner la fonction sont triviales. La fonction retourne *possible* lorsqu'elle ne peut pas actuellement déterminer si le test peut échouer ou réussir. En fonction de ces définitions, le comportement du système pour réaliser les tests sur la contrainte est le suivant :

- lorsque le système doit procéder à un test *cst-test* une première fois, la fonction *fonction* est alors exécutée. Si elle retourne *vrai* (ou *faux*), le test réussit (ou échoue). Si elle retourne *possible*, le test est mis en attente et le démon *démon* est alors mis en place en attente d'une modification de la base de contraintes,
- lorsque qu'un changement intervient, le test *cst-test* est réveillé et la fonction *fonction* est à nouveau réexécutée. Si la fonction retourne alors *vrai* ou *faux*, le démon est supprimé.

Les membres droits des règles peuvent quant à eux comporter des opérations de la forme :

- *cst-tell-eq(!x,!y)* qui permet de vérifier que les variables x et y sont toujours égales,
- *cst-tell-not-eq(!x,!y)* qui permet de vérifier que les variables x et y sont toujours différentes,
- *cst-tell-not-in(v,!x)* qui permet de vérifier que les valeurs de l'ensemble v ne sont pas des valeurs possibles pour la variable x,

- *cst-ask-dom(!x)* qui permet d'obtenir le domaine de la variable x,
- *cst-ask-value(!x)* qui permet d'obtenir la valeur de la variable x lorsqu'elle est unique, sinon, il y a échec puis *backtrack*.

Le système CRP(FD) a été testé et implanté en Common Lisp avec un formalisme de règles proche de celui utilisé dans CLIPS.

2.3.3 Un système de réécriture : Maude

Maude est un langage permettant les calculs équationnels et utilisant la logique de réécriture. Maude [CDE+00] est issu des travaux préliminaires de OBJ3 [GKK+87, KKM88]. Ainsi, le langage équationnel de OBJ3 est contenu dans celui de Maude.

Maude est un des systèmes dont les calculs sont basés sur la logique de réécriture. La logique de réécriture [Mes92] est une logique qui décrit de façon naturelle les états et les calculs non déterministes. D'autres systèmes basés sur la logique de réécriture sont ELAN [BCD+00] et CafeOBJ [FS94, FN97].

Le système Maude est basé sur l'interpréteur Core Maude qui accepte une hiérarchie de modules systèmes ou fonctionnels non paramétrés avec une syntaxe définie par l'utilisateur. Il est implanté en C++ et consiste en deux parties : un moteur de réécriture et un interpréteur.

Full Maude est l'extension de Core Maude en un langage plus riche permettant, notamment, de définir des modules orientés objet. C'est surtout cet aspect qui nous intéresse et que l'on développe dans cette partie.

Dans un système orienté objet concurrent, l'état concurrent, appelé *configuration*, a la structure d'un multi-ensemble constitué d'objets et de messages. L'état évolue suivant les règles utilisant la réécriture Associative-Commutative et les messages. De façon intuitive, les messages entrent en contact avec les objets auxquels ils sont destinés causant ainsi un *évènement de communication* par application d'une règle de réécriture. Les calculs concurrents orientés objet peuvent être vus comme des déductions en logique de réécriture puisque les configurations S qui sont atteignables à partir d'une configuration initiale S_0 sont exactement celles telles que le séquent $S_0 \rightarrow S$ est prouvable en logique de réécriture en utilisant les règles de réécriture qui spécifient le comportement du système orienté objet.

Un objet dans un état donné est représenté par le terme :

$$< O : C \mid a_1 : v_1 \ , \ \ldots \ , \ a_n : v_n >$$

où O est l'identificateur de l'objet, C est sa classe, les a_i sont les noms des attributs et les v_i sont les valeurs correspondantes. Un objet sans attribut est représenté par :

$$< O : C \mid >$$

Les messages n'ont pas une forme syntaxique précise et peuvent être définis par l'utilisateur pour chaque application. L'état concurrent d'un système orienté objet est un multi-ensemble d'objets et de messages, appelé une *Configuration*, avec une union décrite avec la syntaxe vide _ _.

En Full Maude, les systèmes orientés objet concurrents peuvent être définis dans des modules orientés objet, introduits par le mot-clé `omod`, qui utilise une syntaxe plus adaptée que les modules systèmes : dans ces modules, les notions d'objet, de message, de configuration sont connues.

Dans ces modules objet, les classes sont définies avec le mot clé `class` suivi par le nom de la classe C et par une liste de déclarations d'attributs séparées par des virgules. Chaque déclaration a la forme $a \ : \ S$ où a est l'identificateur de l'attribut et S est la sorte des valeurs possibles de l'attribut. Une définition de classe a ainsi la forme :

$$classC \mid a_1 : S1 \ , \ \ldots \ , \ , a_n : S_n \ .$$

On peut définir une classe sans attribut de la même façon mais avec une liste de déclarations d'attributs vide ; la syntaxe est alors :

$$classC \ .$$

On peut ainsi définir une classe de comptes en banque appelée `Accnt` avec un attribut `bal` désignant le solde du compte. La classe `Accnt` est définie par :

```
class Accnt | bal : MachineInt .
```

où `MachineInt` désigne le type des entiers. Plusieurs messages peuvent être envoyés à tout objet de cette classe : pour créditer, débiter ou transférer de l'argent sur un compte.

Par exemple, le message de débit d'un compte est déclaré par :

```
msg debit : Oid MachinInt -> Msg .
```

Débiter un compte est défini par le message `debit` qui prend un identificateur d'objet de sorte `Oid` et un entier correspondant au montant débiter.

La règle correspondant au message `debit` est définie par :

```
crl [debit] : debit(A, M) < A : Accnt | bal : N >
    => < A : Accnt | bal : (N - M) >
       if N > M .
```

Le message s'applique à un objet de la configuration nommé `A` et réécrit cet objet (et fait ainsi évoluer la configuration) avec le nouveau montant débité. Cet règle est contrainte par le test qu'un débit ne peut pas être supérieur au montant disponible.

On peut ainsi, grâce à l'interpréteur de commandes de Full Maude, envoyer un message à une configuration d'objets comme on le montre ci-dessous avec une configuration ne contenant ici que trois objets :

```
Maude> (rew < 'Paul : Accnt | bal : 5000 >
            < 'Peter : Accnt | bal : 2000 >
            < 'Mary : Accnt | bal : 9000 >
         debit('Peter, 1000)
         credit('Paul, 1300)
         credit('Mary, 200) . )

Result Configuration : < 'Peter : Accnt | bal : 1000 >
                       < 'Paul : Accnt | bal : 6300 >
                       < 'Mary : Accnt | bal : 9200 >
```

L'héritage est géré en Full Maude par une structure de types avec sortes ordonnées : une définition de sous-classe `C < C'` est juste une déclaration de sous-sorte. Les effets d'une telle déclaration sont que les attributs, messages et règles de la super-classe caractérisent aussi la structure et le comportement de la sous-classe. L'héritage multiple est permis (c'est-à-dire qu'une même classe peut hériter de plusieurs autres classes) mais cela suppose que l'utilisateur définisse correctement les classes qu'il utilise. Ainsi, considérons une classe `A` avec un attribut `val` de sorte `MachinInt` et une autre classe `B` définissant le même attribut mais avec le type `bool` ; si une classe `C` hérite des deux classes `A` et `B`, le système ne gère pas le conflit qui existe entre les deux types différents pour un même attribut. Le type de résultat que l'on peut attendre en pareil cas n'est pas plus explicité et dépend totalement de la machine.

Afin de faire coexister des modules objets avec des modules systèmes standards, Full Maude opère de la manière suivante : même si la syntaxe d'un module objet diffère de celle d'un module système comme défini dans Core Maude, la sémantique des modules objet est réduite à celle d'un module système. En fait, chaque module objet peut ainsi être traduit en un module système correspondant.

Le mécanisme de traduction est basé sur la signification des concepts objets définis dans les modules objet dans la logique de réécriture qui elle est prise en compte dans les modules systèmes standards. La traduction associe donc à chaque message une règle similaire et à chaque définition de classe et d'objet les déclarations correspondantes.

Le système Full Maude a été intégré dans la dernière version de Maude appelée Maude2.0 [CDE+00]. Le système a été utilisé pour le développement de quelques applications, notamment dans la gestion de protocoles de communication dans [DMT00], dans la définition d'agents mobiles dans [DELM00].

S.C.D. - U.H.P. NANCY 1
BIBLIOTHÈQUE DES SCIENCES
Rue du Jardin Botanique
54600 VILLERS-LES-NANCY

2.4 Le système ELAN

On présente dans cette partie le système dans lequel l'implantation du formalisme présenté au cours de cette thèse est réalisé.

Le langage ELAN a été conçu au sein du projet PROTHEO à Nancy au début des années quatre-vingt-dix. Sa première version est décrite dans la thèse de M. Vittek [Vit94] et son implantation a été initialement détaillée dans [KKV95b]. Au cours des années, le langage a évolué et depuis le début de l'année 2001 la version 3.5 est disponible avec le manuel associé [BCD+00]. C'est cette version du langage que nous utiliserons tout au long de ce travail. ELAN a été conçu comme un cadre logique pour le prototypage de systèmes de calcul.

Une logique est définie en général par une syntaxe, un système de déduction, une classe de modèles et une relation de satisfaisabilité. Ces quatre composantes ont été étudiées dans le cadre de la logique de réécriture qui a été proposée par J. Meseguer dans [Mes92] comme manière d'interpréter les systèmes de réécriture.

Les systèmes de calcul furent introduits par C. Kirchner, H. Kirchner et M. Vittek dans [KKV95a], où ils présentent une version plus élaborée des idées qu'ils avaient proposées originalement dans [KKV93]. Un système de calcul enrichit le formalisme de la logique de réécriture avec une notion de *stratégie*: un *système de calcul* est composé d'une théorie de réécriture et d'un *système de stratégies*. Un programme est vu comme un ensemble de règles de réécriture contrôlées par des stratégies:

$$\textbf{Programme} = \textbf{Règles} + \textbf{Stratégies}$$

Ce cadre, utilisé initialement dans le domaine de la déduction automatique, permet de lier des règles de transformation à des règles de réécriture, et, en plus, il nous donne la possibilité d'exprimer leur enchaînement en utilisant la notion de stratégie. Les stratégies contrôlent l'application des règles de réécriture en spécifiant des parcours dans l'arbre de toutes les dérivations possibles et de cette façon décrivent quels sont les nœuds considérés comme des résultats d'un calcul. Elles sont utilisées, d'une part, pour décrire le déroulement d'arbres de dérivation, aussi appelés preuves, qui nous intéressent et, d'autre part, pour restreindre l'espace de recherche de ces preuves.

Du point de vue de la programmation, le langage ELAN offre la possibilité de spécifier des systèmes de calcul composés de théories de réécriture multi-sortées, chacune décrite par une signature et par un ensemble de règles de réécriture et de stratégies d'exécution.

La signature définit les sortes et les symboles de fonctions utilisés dans la description de la théorie. ELAN permet d'utiliser des symboles libres et associatifs-commutatifs, qui peuvent être spécifiés en utilisant une notation *mix-fix*.

L'ensemble de règles de réécriture est composé de règles non nommées et de règles nommées (ou étiquetées).

- Les règles non-nommées sont utilisées pour la normalisation de termes. Leur application n'est pas contrôlée par l'utilisateur, elles sont exécutées avec une stratégie prédéfinie dans le langage. Cette stratégie pré-définie est la stratégie de normalisation *leftmost-innermost*. Puisque la stratégie de normalisation est pré-définie dans ELAN, elle n'est pas spécifiée dans la théorie de réécriture de l'utilisateur. L'ensemble de règles non-nommées doit être confluent et terminant.
- L'ensemble de règles nommées, qui n'est pas nécessairement confluent et terminant, peut être contrôlé par des *stratégies élémentaires*. Les deux raisons principales pour leur utilisation sont:
  - si l'ensemble de règles n'est pas terminant, l'utilisateur a la possibilité de restreindre l'ensemble de dérivations à un sous-ensemble de dérivations finies, afin d'éviter des dérivations infinies;
  - si l'ensemble de règles n'est pas confluent, l'utilisateur a la possibilité de spécifier certains sous-ensembles de toutes les dérivations possibles et obtenir ainsi un sous-ensemble de tous les résultats possibles.

Les stratégies sont utilisées en ELAN dans trois buts différents :

- pour séparer dans un programme la partie calcul de la partie contrôle ;
- pour exprimer des dérivations non-déterministes et concurrentes ;
- pour spécifier des procédures de normalisation particulières.

Un des avantages de la séparation entre le calcul et le contrôle est mis en valeur surtout au niveau de la compréhension des programmes. Cela veut dire que la partie du calcul est souvent technique et difficile à présenter en détail, tandis que le contrôle, exprimé en utilisant les constructions de concaténation, d'itération et de choix, est souvent plus facile à comprendre.

Parmi plusieurs caractéristiques, le calcul non-déterministe d'ELAN le différencie d'autres systèmes basés sur la réécriture. L'avantage de cette option est que cela permet de travailler avec des systèmes de réécriture non-confluents.

Le style de programmation en ELAN, basé sur le paradigme des systèmes de calcul (règles+stratégies), unifie certaines caractéristiques de la programmation fonctionnelle et logique. La programmation par réécriture est similaire à l'approche fonctionnelle restreinte au premier ordre. Cependant, la possibilité de spécifier des sous-ensembles de dérivations par un langage de stratégies joue le rôle du non-déterminisme de la programmation logique.

La variété des applications qui ont été implantées en ELAN illustre la généralité du paradigme des systèmes de calcul et montre l'expressivité et la puissance du langage. Parmi elles, on peut citer :

- deux implantations de la procédure de complétion de Knuth-Bendix [KM95, KLS96] ;
- une implantation du prouveur de prédicats B [CK97] ;
- la combinaison d'algorithmes d'unification dans des théories arbitraires [Rin97] ;
- un algorithme d'unification d'ordre supérieur [Bor95] ;
- la SLD-résolution [Vit94] ;
- résolution de CSP [Cas98a] ;
- CLP [KR98] ;
- la réécriture du premier ordre [KM96] et d'ordre supérieur ;
- une implantation de la procédure de résolution de contraintes d'ordre pour la preuve de terminaison basée sur l'ordre général sur les chemins (ordre GPO) [GG97] ;
- une implantation de la terminaison *innermost* par induction [Fis00] ;
- l'étude de la terminaison par induction pour différentes stratégies de réécriture [GKF01] ;
- une implantation d'automates temporisés [BBKK01] ;
- l'implantation de procédures de décision en utilisant le raisonnement équationnel de Coq [BBC+97] dans [AN00] ;
- l'implantation de ENAR (*Extended Narrowing And Resolution*) [Stu00] ;
- la spécification de protocoles d'authentification [CK01].

La première version d'ELAN avait offert un interpréteur implanté en C++ choisi comme langage d'implantation surtout pour des raisons d'efficacité et de portabilité. De nouvelles techniques de compilation de systèmes de calcul furent étudiées et maintenant il existe un compilateur du langage implanté en Java qui permet d'utiliser des symboles associatifs-commutatifs [MK97, MK98].

Dans le reste de cette section, nous présentons brièvement le langage ELAN. Nous illustrons la syntaxe des trois composantes d'un système de calcul : signatures, règles de réécriture et stratégies. Nous décrivons aussi informellement la sémantique opérationnelle du langage. Notre intérêt est seulement de présenter les diverses constructions du langage et de les illustrer par des exemples liés à la construction d'une configuration composée d'une mémoire d'objets et d'un ensemble de contraintes. Une description formelle et détaillée du langage est donnée dans [Bor98], une sémantique du point de vue fonctionnel est présentée dans [BKK98] et une autre basée sur le ρ-calcul est présentée dans [CKL00]. Tous les détails nécessaires pour l'utilisation du langage peuvent être trouvés dans [BCD+00].

2.4.1 Signatures ELAN

ELAN permet de définir des signatures multi-sortées où les sortes sont spécifiées par un ensemble S. L'exemple 2.1 présente la déclaration de sortes utilisées pour traiter un problème de définition d'un état basé sur des bases d'objets et de contraintes.

Exemple 2.1 *Dans le cadre de la formalisation de problèmes de planification de tâches, on s'intéresse à la gestion d'une mémoire de travail composée d'objets à laquelle est jointe une base de contraintes. De façon synthétique, on considère donc des objets de sorte* `Object`*, l'ensemble de ces objets formant une configuration de sorte* `configuration`*. Les contraintes sont de sorte* `CSP` *pour Problème de Satisfaction de Contraintes (Constraint Satisfaction Problem). L'ensemble composé d'une configuration et de contraintes est de sorte* `state`*.*

La déclaration des sortes `Object`*,* `configuration`*,* `CSP` *et* `state` *est faite en* ELAN *de la manière suivante :*

```
sort Object configuration CSP state;
end
```

Une fois déclarées les sortes de la signature, on peut définir les symboles de fonctions indexés qui appartiennent à l'ensemble de symboles de fonctions $\mathcal{F}$ de la signature. Chaque symbole, défini par son profil en notation *mix-fix*, peut être décoré par des attributs sémantiques comme étant un symbole libre ou associatif-commutatif "`(AC)`" et il peut aussi être décoré par des attributs syntaxiques tels que :

- sa priorité syntaxique (e.g. `pri 10`) ;
- sa visibilité dans d'autres modules (e.g. `global/local`) ;
- son associativité syntaxique à gauche par défaut par `assocLeft` ou à droite par `assocRight` ;
- le fait d'être synonyme avec un autre symbole (e.g. `alias`).

Dans le langage, il existe également des symboles pré-définis attachés à des procédures écrites en C++ et décorés par leurs identificateurs (e.g. `code 18`).

L'exemple 2.2 montre la définition de symboles de fonctions avec des attributs sémantiques.

Exemple 2.2 *A partir de la déclaration de sortes de l'exemple 2.1, on peut définir des opérateurs sur ces sortes, notamment les opérateurs de construction et d'initialisation.*

Afin de construire un terme de sorte `configuration`*, on déclare tout d'abord que tout terme de sorte* `Object` *est également de sorte* `configuration`*. On définit également que toute combinaison de deux termes de la sorte* `configuration` *par un opérateur invisible est aussi de sorte* `configuration`*, cet opérateur invisible étant déclaré comme étant Associatif-Commutatif. De plus, on définit que la constante* `nobj` *est elle aussi de sorte* `configuration`*, cette constante permettant de définir une configuration vide.*

Définissons maintenant les opérateurs d'initialisation de ces structures. Un opérateur `InitConfiguration` *ne prenant aucun argument est défini et rend un terme de sorte* `configuration`*, cet opérateur permettant de créer une nouvelle configuration. Les derniers opérateurs* `InitCSP(_,_)` *et* `INIT(_,_)` *combinent quant à eux deux termes de sortes* `int` *et produisent un terme, respectivement de sorte* `CSP` *et* `state`*.* `InitCSP(_,_)` *permet d'initialiser une nouvelle contrainte de type CSP de la façon suivante : le domaine de toute variable de la nouvelle contrainte est celui compris entre les deux entiers donnés en paramètres.* `INIT` *permet quant à lui d'appeler successivement* `InitConfiguration` *et* `InitCSP(_,_)` *pour créer un nouvel ensemble configuration/contrainte de sorte* `state`*.*

Toutes ces déclarations sont faites de la manière suivante[1] *:*

1. On remarque dans cet exemple que le symbole "`@`" utilisé en ELAN dénote la place des arguments. Cet emplacement est par contre noté "`_`" dans le texte. Ce sera la même convention qui sera utilisée dans toute la suite de la thèse.

```
operators
global

        @ : (Object) configuration;
        @ @ : (configuration configuration) configuration (AC);
        nobj : configuration;

        InitConfiguration : configuration;
        InitCSP(@,@)  : (int int) CSP;
        INIT(@,@)  : (int int) state;
end
```

Le langage permet aussi de spécifier un ensemble de *sélecteurs* pour chaque symbole de fonction $f \in \mathcal{F}$. Soit $f : (s_1 \dots s_n) \mapsto s$ un profil du symbole f. La construction syntaxique suivante spécifie un ensemble de noms de champs optionnels $field_1 \ \dots \ field_n$ du symbole f

$$f(@,\dots,@) : (field_1 : s_1 \ \dots \ field_n : s_n) \mapsto s$$

Pour cette définition du symbole f, ELAN offre n sélecteurs $@.field_i : (s) \mapsto s_i$ et n modificateurs $@[.field_i \leftarrow @] : (s\ s_i) \mapsto s$ définis par les règles suivantes :

$$\begin{array}{lcl} f(t_1,\dots,t_n).field_i & \Rightarrow & t_i \\ f(t_1,\dots,t_n)[.field_i \leftarrow t] & \Rightarrow & f(t_1,\dots,t_{i-1},t,t_{i+1},\dots,t_n) \end{array}$$

Exemple 2.3 *A partir des déclarations de sortes de l'exemple 2.1 et des définitions de symboles de fonctions de l'exemple 2.2, on définit les symboles de fonctions nécessaires pour représenter les symboles de prédicats nous permettant de construire un état de sorte* `state` *constitué d'une base d'objets de sorte* `configuration` *associée à une base de contraintes de sorte* `CSP` *:*

```
operators global

        @ with @  : (ListObjects:configuration ListConstraint:CSP) state;
end
```

Ainsi, on peut utiliser les sélecteurs pour obtenir, par exemple, à partir de l'état `S` *de sorte* `state` *composé d'une configuration de deux objets* `O1` *et* `O2` *et du CSP* `X+3 =? 5` *et donné par :*

```
S = O1 O2 nobj with X+3 =? 5
```

l'ensemble de la base d'objets par :

```
S.ListObjects
```

ce qui donne le terme `O1 O2 nobj` *de sorte* `configuration`*. On peut aussi modifier les valeurs des champs de l'état* `S` *par :*

```
S[.ListObjects <- O1 nobj]
```

ce qui donne comme résultat l'état suivant :

```
S = O1 nobj with X+3 =? 5
```

toujours de sorte `state`*.*

2.4.2 Règles de réécriture

Il existe deux types de règles souvent introduites dans une théorie de réécriture: les règles non-conditionnelles et les règles conditionnelles. De plus, à partir de la toute première version du langage ELAN, la notion d'*affectation locale* a été introduite pour des variables non-instanciées pendant le filtrage [BCD+00]. Les affectations locales à une règle permettent d'appliquer une stratégie sur un terme autre que le terme en membre gauche de la règle. Le terme résultat, une fois calculé, est mémorisé une fois pour toute dans une variable qui peut alors être utilisée plusieurs fois dans la suite de la règle. On peut ainsi économiser de nombreuses réécritures si ce terme calculé dans l'affectation locale doit être utilisé plusieurs fois: une simple référence à la variable locale mémorisant cette valeur suffit.

La syntaxe des règles conditionnelles avec des affectations locales est la suivante:

$$[\ell] \quad l \Rightarrow \quad r \qquad if \ - \ where$$

où

- $\ell \in \mathcal{L}$ est l'étiquette de la règle (qui est vide dans le cas d'une règle non-nommée);
- l et r sont des termes de $\mathcal{T}(\mathcal{F},\mathcal{X})$ représentant les membres gauche et droit de la règle;
- $if \ - \ where$ est de la forme:

$$\{\textbf{if} \quad v \mid \textbf{where} \quad y \ := (S)u \mid \textbf{where} \quad y \ := ()u\}^*$$

où

- **if** v est une condition booléenne;
- **where** $y := (S)u$ est une affectation de la variable $y \in \mathcal{X}$ par le résultat de l'application de la stratégie S sur le terme $u \in \mathcal{T}(\mathcal{F},\mathcal{X})$;
- **where** $y := ()u$ est une affectation de la variable $y \in \mathcal{X}$ par le résultat de la normalisation du terme u.

L'application d'une règle de réécriture à un terme clos commence par une étape de filtrage permettant de calculer la substitution associée au problème de filtrage du membre gauche vers le terme à réduire. Les affectations locales et les conditions sont alors évaluées les unes à la suite des autres (de haut en bas) jusqu'à atteindre la dernière; c'est seulement s'il n'y a pas d'échec dans cette suite d'évaluations que la règle peut s'appliquer et que le membre droit est construit. Chaque condition v est mise en forme normale puis comparée à la valeur de vérité `true` pré-définie par le système. En cas d'égalité, on dit que la condition est satisfaisable et le calcul des évaluations locales se poursuit. L'affectation locale **where** $y := (S)u$ permet de déclencher l'application d'une stratégie. Dans un premier temps, le terme u est mis en forme normale en n'utilisant que des règles non nommées; la stratégie S est ensuite appliquée sur le terme en forme normale. Une normalisation se déclenche automatiquement après chaque application d'une règle de réécriture. En cas d'échec d'une condition ou (de la stratégie) d'une évaluation locale, un mécanisme de *retour arrière* (*backtracking*) est déclenché: les affectations locales précédentes sont réévaluées pour en extraire d'autres solutions. Si aucune autre solution n'est trouvée, l'application de la règle courante échoue et une autre règle est sélectionnée.

Exemple 2.4 *A partir des sortes et des symboles de fonctions définis dans les exemples 2.1 et 2.2, les deux règles ci-dessous, la première non-nommée et la seconde nommée* `Init`*, définissent l'initialisation d'un état, terme de sorte* `state`*.*

```
rules for configuration
global
[] InitConfiguration => nobj end
end
```

```
rules for state
  N1,N2 : int;
  C : configuration;
  csp : CSP;
global
[Init] INIT(N1,N2) => C with csp
       where C   := () InitConfiguration
       where csp := () InitCSP(N1,N2)
end
end
```

Dans la première règle, aucune affectation n'est réalisée.

Dans la seconde règle, la variable C est affectée au résultat de la normalisation appliquée au terme `InitConfiguration` *et ainsi utilise la définition de la règle donnée en premier lieu. L'affectation de la variable csp utilise quant à elle la définition de* `InitCSP` *qui se trouve dans un module le définissant.*

Pour des raisons de confort et d'efficacité d'exécution, le langage dispose de différentes extensions pour la construction de règles et notamment celle des *affectations généralisées.*

L'*affectation généralisée* est une construction syntaxique

$$\textbf{where} \quad (sort) \quad p := (S)u$$

où p est un terme non-clos de sorte $sort \in \mathcal{S}$.

Le terme p, dit *motif*, est composé de constructeurs et de variables, où un constructeur est un symbole de fonction qui n'apparaît pas comme opérateur de tête dans un membre gauche d'une règle de réécriture.

Toutes les variables dans le motif p non encore instanciées, sont instanciées par le filtrage de ce motif p avec le résultat de l'application de la stratégie S au terme u, ou au cas où une stratégie S n'est pas spécifiée, le résultat de la normalisation du terme u.

Exemple 2.5 *Afin de pouvoir faire évoluer les états des bases d'objets et de contraintes suivant les désirs de l'utilisateur, on peut définir une règle SBS qui, à partir d'un état S donné, réalise son affichage par l'opérateur* `print` *puis, suite à l'application de* `SucceedInput` *récupère un entier N donné par l'utilisateur et un état S2 susceptible d'avoir été modifié. On récupère un nouvel état S3 après application d'une dernière affectation locale.*

```
rules for state
    S,S1,S2,S3 : state;
    N : int;
global

[SBS] StepByStep(S) => S3
          where S1 := () print(S)
          where (pair[int,state]) [N,S2] := () SucceedInput(S)
          where S3 := (NewState) selection(S2,N)
end
```

2.4.3 Stratégies élémentaires d'ELAN

Le langage de *stratégies élémentaires* d'ELAN permet de contrôler l'application des règles nommées, de définir des exécutions non-déterministes et de spécifier des dérivations simultanées. Le langage permet aussi d'introduire le typage de stratégies construites à partir de règles de réécriture. Le typage de stratégies permet de vérifier si leur usage est correct. En général, la sorte $\langle s_1 \mapsto s_2 \rangle$ d'une stratégie élémentaire exprime le fait que cette stratégie, appliquée à un terme de sorte s_1, donne des résultats de sorte s_2.

Une stratégie de sorte $\langle s \mapsto s \rangle$ est bien typée si toutes ses sous-stratégies sont de la même sorte $\langle s \mapsto s \rangle$ et, également, toutes les règles dans cette stratégie ont des membres gauches et droits de sorte

s. Une stratégie de sorte $\langle s_1 \mapsto s_3 \rangle$ est bien typée si elle peut être décomposée en deux sous-stratégies S_1 et S_2 où S_1 est de sorte $\langle s_1 \mapsto s_2 \rangle$ et S_2 est de sorte $\langle s_2 \mapsto s_3 \rangle$.

Le langage de stratégies élémentaires offre notamment :

- une construction pour la concaténation de stratégies : ";" ;
- trois constructions de choix : **dk**, **dc** et **first** ;
- deux constructions d'itération : **repeat*** et **iterate*** ;
- deux constructions pour les stratégies identité et échec qui sont respectivement **id** et **fail**.

La syntaxe et la sémantique opérationnelle des stratégies élémentaires sont définies par la suite de façon informelle.

- **Construction de concaténation**

  ; La concaténation de stratégies S_1 ; S_2 correspond à l'axiome de transitivité de la logique de réécriture. Pour typer une concaténation, il faut que les deux stratégies S_1 et S_2 soient respectivement de sortes $\langle s_1 \mapsto s_2 \rangle$ et $\langle s_2 \mapsto s_3 \rangle$. On se restreint au cas où $s_1 = s_2 = s_3 = s$ et ainsi, S_1 et S_2 sont de sorte $\langle s \mapsto s \rangle$, qui devient également la sorte de cette concaténation. S_1 ; S_2 échoue si S_1 échoue ou si S_2 échoue.

- **Constructions de choix**

  dk La stratégie **dk**(S_1 , ... , S_n) donne tous les résultats de l'application de toutes les stratégies $S_1, \ldots, S_n$. Si toutes les stratégies S_i échouent alors la stratégie **dk** échoue.

  dc La stratégie **dc**(S_1 , ... , S_n) donne tous les résultats de l'application d'une des stratégies $S_1, \ldots, S_n$ laquelle est choisie de manière aléatoire. Si toutes les stratégies S_i échouent, alors la stratégie **dc** échoue.

  first La stratégie **first**(S_1 , ... , S_n) donne tous les résultats de l'application de la première stratégie $S_1, \ldots, S_n$ qui est applicable (en ordre textuel). Si toutes les stratégies S_i échouent, alors la stratégie **first** échoue.

- **Constructions d'itération**

  repeat* La stratégie **repeat** $*(S)$ correspond à $S^i = \overbrace{S \ ; \ \ldots \ ; \ S}^{i}$ si S^{i+1} n'est plus applicable. Si la stratégie S échoue à l'étape à l'étape $i+1$ alors la stratégie **repeat*** retourne le terme obtenu à l'étape i. La stratégie **repeat*** n'échoue donc jamais.

  iterate* La stratégie **iterate** $*(S)$ correspond à **dk**(**id** , S , S ; S , S ; S ; S , ...). Si la stratégie S échoue à une étape donnée $i+1$ alors la stratégie **iterate*** retourne comme dernier résultat le terme obtenu à l'étape i. Tous les termes intermédiaires sont retournés comme des résultats de cette stratégie. **iterate** $*(S)$ n'échoue jamais.

- **Constructions d'identité et d'échec**

  id La stratégie **id** correspond à l'identité, elle retourne le même terme que celui donné en entrée et peut ainsi toujours être appliquée.

  fail La stratégie **fail** correspond à un échec, elle échoue toujours.

L'exemple 2.6 montre la définition d'une stratégie en ELAN.

Exemple 2.6 *Dans la règle SBS définie dans l'exemple 2.4, la stratégie* `NewState` *est utilisée afin de retourner un nouvel état suite à un choix de l'utilisateur. La stratégie* `NewState` *utilise les règles nommées S1 à S7 et est définie de telle sorte que le résultat de son application est le premier ensemble de résultats de l'application des règles S1 à S7, dans l'ordre spécifié, qui n'a pas échoué.*

```
stratop global NewState : < state -> state > bs; end

strategies for term
implicit

[] NewState => first (S1 , S2 , S3 , S4, S5 , S6 , S7)
end
```

2.4.4 Pré-processeur

Une caractéristique originale proposée par ELAN est l'utilisation d'une phase de pré-*processing* qui autorise un codage plus simple de la description de la logique.

Le pré-processeur réalise des remplacements textuels en partant d'informations données par le programmeur dans le module et par l'utilisateur dans la spécification. La phase de pré-processing en ELAN peut être vue comme un mécanisme d'expansion qui étend le processus de paramétrisation décrit précédemment.

Il y a trois formes différentes de pré-processing autorisées en ELAN :

- La duplication simple : on autorise alors une simple recopie du texte désigné. La syntaxe de l'expression *exp* à dupliquer un nombre n de fois est donnée par :

$$\{\ exp\ \}\ n$$

  Par exemple, le texte `bool{,int}  3` est remplacé par `bool,int,int,int`.

- La duplication avec argument : il est parfois nécessaire d'autoriser la description d'objets comme $f(t_1,\ldots,t_5)$. Si l'on souhaite dupliquer une expression *exp* contenant un identificateur *ident* en autant d'expressions dans lesquelles l'identificateur *ident* est remplacé successivement par chaque élément de l'énumération de exp_1 à exp_n, la syntaxe suivante est alors utilisée :

$$\{\ exp\ \}\_\ ident{=}\ exp_1 \ldots\ exp_n$$

  Par exemple, le texte `{s_I=t_I &}_{I=1...3} true` est réécrit en `s_1=t_1 & s_2=t_2 & s_3=t_3 & true`

  Une forme spéciale de la duplication avec argument est la construction explicite d'une liste d'identificateurs indexés id_i, pour i de $N1$ à $N2$, et séparés par le caractère c avec la syntaxe suivante :

$$id_N1\ c\ \ldots\ c\ id_N2$$

  Le texte `s_2,...,s_5` est donc remplacé par `s_2,s_3,s_4,s_5`.

- L'énumération : cette construction permet de faire le lien entre la spécification donnée par l'utilisateur et la logique décrite par le programmeur. Une motivation naturelle pour cette construction est donnée par l'utilisation des règles d'inférence. Considérons la règle de transformation **Décompose** de l'unification donnée par :

$$\textbf{Décompose}\quad P \wedge f(s_1,\ldots,s_n) =^? f(t_1,\ldots,t_n) \rightarrow P \wedge s_1 =^? t_1 \wedge \ldots \wedge s_n =^? t_n$$

  Cette règle est générique dans le sens où l'opérateur f et son arité ne sont pas spécifiés. Cette règle est donc valable que f soit d'arité 2 (si f est l'opérateur d'addition + par exemple) ou 1 (si f est l'opérateur de négation booléenne *not* par exemple).

  ELAN permet donc d'être générique par la construction d'énumération du pré-processeur. Ainsi, dans ce type d'énumération, on cherche à recopier un texte *exp* dans lequel certaines variables *vars* apparaissent, ces variables étant successivement remplacées par les différentes valeurs qu'elles peuvent prendre, ces valeurs étant définies par *affectation*. On donne ici la syntaxe générale sans donner le détail de la syntaxe des déclarations de variables et de leurs affectations :

$$\textbf{FOR EACH}\ vars\ \textbf{SUCH THAT}\ affectations :\{\ exp\ \}$$

  Par exemple, considérons la règle **Décompose** précédemment présentée et une définition générique utilisant l'énumération :

```
FOR EACH SF:pair[identifier,int];F:identifier;N:int
SUCH THAT SF:=(listExtract)elem(L) AND F:=()first(SF) AND N:=()second(SF) :{
 rules for constraint
  s_1,...,s_N,t_1,...,t_N:term;
 local
 [decompose] P & F(s_1,...,s_N)=F(t_1,...,t_N) => P {& s_I=t_I}_I=1...N end
 end}
```

où l'on considère que `L` contient la définition des symboles de fonctions sous la forme `[f,2].[g,1].nil` avec f et g respectivement déclarés d'arités 2 et 1 ; `F` va désigner le symbole de fonction et `N` son arité. Le texte précédent est alors transformé en le groupe de règles suivant :

```
rules for constraint
 s_1,s_2,t_1,t_2 : term;
local
[decompose] P & f(s_1,s_2)=f(t_1,t_2) => P & s_1=t_1 & s_2=t_2 end
end
rules for constraint
 s_1,t_1 : term;
local
[decompose] P & g(s_1)=g(t_1) => P & s_1=t_1 end
end
```

2.4.5 Modularité, visibilité et encapsulation

Un programme ELAN peut être composé de plusieurs modules paramétrés qui définissent des sortes, des symboles de fonctions, des règles et des stratégies. Toutes les règles de réécriture et les stratégies des modules d'un programme ELAN sont dans le système de calcul et sont mises ensemble au même niveau ; les définitions des sortes, des symboles de fonctions, des règles et des stratégies sont mises à plat et la structure de modules disparaît.

Cependant, pour des applications de taille moyenne, il est souhaitable d'avoir la possibilité de restreindre la visibilité de certains opérateurs, règles ou stratégies définis dans un module ou bien de paramétrer certains modules. De ce point de vue, la possibilité d'organiser un programme en modules et de les structurer en ensembles d'opérateurs, de règles et de stratégies est une facilité syntaxique pour spécifier des systèmes de façon modulaire. Un module peut alors importer d'autres modules dont tout ou partie des définitions de ceux-ci lui sont nécessaires.

Dans ELAN, les déclarations de visibilité et d'encapsulation peuvent être faites à deux niveaux :

- dans l'importation de modules (`local` ou `global`) ;
- dans la déclaration d'opérateurs, règles ou stratégies dans un module (`local` ou `global`).

où le mot clé `local` indique que les importations ou les définitions qui suivent ont une visibilité restreinte au module courant ; le mot clé `global` indique quant à lui une visibilité de l'ensemble de l'application.

Etant donnée la définition d'un symbole de fonction, d'une règle ou d'une stratégie O dans un module A,

- si O est défini comme étant `local` dans le module A, alors O n'est pas visible dans un module B, quel que soit le type d'importation du module A dans le module B ;
- si O est défini comme étant `global` dans le module A et le type d'importation du module A dans un module B est `local`, alors O est vu comme étant défini localement dans le module B ;
- si O est défini comme étant `global` dans le module A et le type d'importation du module A dans un module B est `global`, alors O est vu comme étant défini globalement dans le module B.

Exemple 2.7 *La définition de l'affichage de termes par l'opérateur* `print` *peut être faite d'une manière paramétrée de la façon suivante :*

```
module prompt[X]
import local io[string] io[X]; end

operators global
    print(@)      : (X) X;
    printnl(@)    : (X) X;
end
end
```

Ainsi, le module `prompt` *peut être invoqué avec comme paramètre une instance spécifique de la sorte* `X`*. Ce module utilise lui aussi l'importation locale du module* `io` *paramétré avec* `string` *d'une part et avec le même* `X` *d'autre part.*

Le module `Initialisation` *définissant la règle SBS définie dans l'Exemple 2.5 doit alors ainsi comporter, entre autres, l'importation suivante :*

```
module Initialisation

import
global prompt[state];
end

end
```

De cette façon, dans le module `Initialisation` *on connaît les opérateurs* `print` *et* `printnl` *définis pour les termes de sortes* `state`*.*

En pratique, la gestion des modules paramétrés est faite grâce au même moteur que celui utilisé pour le pré-processeur. En effet, paramétrer un module revient à vouloir faire de nouvelles déclarations, à créer de nouvelles règles de réécriture ou stratégies.

Conclusion

L'ensemble des travaux exposés dans ce chapitre montre une grande richesse dans le développement de systèmes à base de règles.

Chacun des systèmes présentés ici est basé sur le fait que les règles sont des objets syntaxiques manipulés par des moteurs d'inférence travaillant sur une structure de données bien définie. Un système de règles de production comme CLIPS ou le système CLAIRE permettent par exemple d'allier les concepts de règles et ceux des langages objet. CRP(FD) est un système permettant de lier l'évaluation par règles avec l'utilisation d'une base de contraintes annexe.

La définition de la logique de réécriture et le développement de certains langages basés sur cette logique permet alors de voir les règles non plus comme de simples objets syntaxiques mais plus encore comme la définition d'une sémantique opérationnelle pour de tels systèmes. Quelques langages comme Maude, ELAN ou encore CafeOBJ reposent sur la logique de réécriture et sont entièrement basés sur les règles : de la représentation des connaissances au mécanisme de déduction. Deux de ces langages ont été plus particulièrement décrits, Maude tout d'abord, développé au SRI, puis ELAN développé au LORIA.

Maude est un langage héritier des travaux sur OBJ3. Une des spécificités de Maude est d'offrir, par sa version appelée Full Maude, une extension orientée objet. ELAN est le langage auquel nous nous sommes intéressé tout au long de cette thèse. Une des spécificités d'ELAN est d'offrir quant à lui un mécanisme de contrôle d'application des règles par les stratégies. La déduction et les calculs sont contrôlés de façon précise, ce qui offre un outil puissant dans le développement d'applications.

Un des buts de ce chapitre est de montrer les différents composants d'un système de règles de production afin d'orienter notre approche dans notre volonté de formaliser les systèmes de règles de production en logique de réécriture. On a ainsi vu que la base de connaissances d'un système de règles de production est composée d'un ensemble de règles travaillant sur une mémoire de travail. Différents formalismes peuvent être utilisés pour décrire les éléments composant cette mémoire de travail et le formalisme objet nous parait être intéressant à étudier dans le cadre d'une formalisation en logique de réécriture. Faire cohabiter des contraintes avec les objets est très intéressant lorsque l'on cherche à modéliser des problèmes de type ordonnancement comme dans notre cas. L'approche faite alors dans CRP(FD) nous semble intéressante dans la façon dont mémoire de travail et base de contraintes cohabitent et sont modifiables par les règles.

Chapitre 3

Codage des objets en ELAN

Le langage ELAN présenté en fin de chapitre précédent est basé sur des ensembles de règles et de stratégies. Afin de pouvoir traiter des problèmes de systèmes de règles de production, on doit pouvoir étendre le langage. Pour cela, considérons dans un premier temps comment lier à ELAN certains concepts de la programmation par objets.

Dans les systèmes de règles de production, on cherche à faire évoluer une base de faits au fur et à mesure de l'application des règles. Cette base de faits (ou mémoire de travail) se modélise sous la forme d'un ensemble, voire plus simplement de structures, d'objets. On peut citer certains systèmes, comme Oz [Smo95], ILOG Rules [SA97] et CLAIRE [CL96], dans lesquels on peut implanter des bases de connaissances et de faits.

Une première solution consiste à ne considérer que des objets structurés comme cela est déjà défini dans la syntaxe ELAN standard [BCD+00] et comme présenté dans la partie 2.4.1. Mais, cette syntaxe sur laquelle nous nous sommes appuyé dans un premier temps, même si elle nous permet de définir des objets structurés, ne nous offre qu'une petite partie des mécanismes objets qui peuvent nous intéresser.

On va donc chercher à coder en ELAN un nouveau formalisme donnant accès à des fonctionnalités "objet" en particulier les mécanismes d'appel de méthodes par messages et d'héritage. L'enjeu est de concevoir ce nouveau formalisme de telle sorte qu'il reste cohérent avec la sémantique d'ELAN.

Ce chapitre est décomposé comme suit : après avoir brièvement présenté les notions caractéristiques des langages à objets, nous présenterons les fonctionnalités objet que nous manipulerons en ELAN ainsi que la syntaxe des modules objets. Ensuite, nous définirons le calcul de réécriture qui nous permettra par la suite de définir une sémantique pour l'implémentation en ELAN du système de règles $\mathcal{R}$ définissant les manipulation des objets par des règles. Finalement, on montrera un résultat sur la préservation du type pour le système de codage des objets que nous considérons.

3.1 Les langages objets

Les langages objets sont nés de différents besoins dans le développement de logiciels : la réutilisation des modules, la hiérarchie et la structuration des données. De nombreux langages basés sur les objets ont vu le jour, comme Simula [BDMN79], Smalltalk [GR83], C++ [Str86, Str94], Eiffel [Mey92] et Java [AG96].

Parmi les langages objets disponibles, on peut distinguer deux classes : les langages objets basés sur le concept de classe (Simula, C++, Eiffel, Java) et ceux basés sur le concept d'objet (Smalltalk).

Les concepts des langages objets basés sur les classes sont apparus avec le langage Simula [BDMN79]. Les principes généraux des langages de classe sont les suivants :

- Les classes sont des descriptions d'objets. Dans une définition de classe, on donne toute la structure qui compose chaque objet de cette classe. Les classes sont composées de champs et de méthodes.
- Un objet peut être créé à partir d'une classe C par l'invocation de la méthode `C.new`. Un tel objet est appelé *objet de la classe C* ou *instance de la classe C*.
- Dans la représentation des classes, les définitions de méthodes ne font pas partie intégrante de la définition des classes mais elles sont désignées par des pointeurs vers une zone où les corps de ces

méthodes sont définis. Les définitions des méthodes sont donc partagées par l'ensemble des objets d'une classe. Lors de l'invocation d'une méthode, les objets délèguent l'invocation de méthode à la méthode proprement dite. L'invocation d'une méthode est notée $o.m$ pour l'invocation de la méthode m sur l'objet o. Dans l'implantation d'une méthode, une référence à `self` réfère l'objet qui a reçu originellement l'invocation de la méthode. Cela correspond à ce qui est aussi appelé `this` dans certains langages objet.

- Les classes peuvent être hiérarchisées par ce qu'on appelle des relations de sous-classe héritant d'une autre classe. Une sous-classe hérite donc des définitions de sa *super*-classe et peut l'enrichir par de nouvelles définitions ou la modifier en redéfinissant méthodes ou champs. Dans une méthode qu'une sous-classe c' hérite d'une classe c, l'utilisation du `self` réfère un objet de la sous-classe c' et non un objet de la classe c où la méthode est définie originellement. On parle d'héritage multiple lorsqu'une classe hérite de plusieurs classes en même temps ; l'héritage simple désigne alors l'héritage d'une seule classe. Si un langage permet l'héritage multiple, on doit donner des règles, parfois délicates, concernant les conflits et duplications qui peuvent alors survenir. Lorsque l'on hérite d'une méthode, on peut aussi surcharger une méthode en redéfinissant une méthode héritée avec des paramètres et/ou un corps différents.

Les concepts des langages à base d'objets sont apparus plus progressivement. On peut néanmoins citer le traité d'Orlando [SLU89] qui donne une description des principes généraux des langages basés sur les objets. En règle générale, on attend d'un langage basé sur les objets qu'il soit plus simple d'utilisation et plus flexible qu'un langage de classes. Nous en détaillons ici les concepts les plus fondamentaux :

- La principale caractéristique des langages à base d'objets est l'absence de notion de classe et la présence de constructeurs spéciaux pour la création d'objets. La définition d'objets passe alors par la définition d'une interface où les types des champs et des méthodes sont donnés. Ensuite, l'implantation d'un objet est décrite en référence à l'interface précédemment définie. La création d'un objet peut être faite de façon procédurale ; c'est-à-dire qu'une procédure peut être écrite afin d'initialiser un objet conformément à l'interface décrite, l'appel répété de cette procédure permettant de créer un ensemble d'objets.
- Certains langages appelés langages à prototypes adoptent une approche différente pour engendrer les objets, le clonage. A partir d'un prototype d'objet, le clonage permet d'engendrer un ensemble d'objets différenciables mais identiques au prototype. La différence majeure entre le clonage et l'opération `new` des langages à base de classe est que le clonage s'effectue sur un objet et `new` est une méthode s'appliquant à une classe.
- Les langages basés sur les objets peuvent utiliser le mécanisme des procédures afin de générer des objets partageant un même comportement ; les langages à prototype ont le mécanisme de clonage qui permet d'engendrer de nouveaux objets et une possibilité de mise à jour d'objets qui peut s'apparenter à la surcharge.
- Pour l'héritage, on a dans le cadre des langages objets différentes appellations : l'héritage implicite ou explicite, l'héritage par plongement ou par délégation. L'héritage implicite est l'héritage où les objets qui sont désignés comme sources sont intégralement recopiés de façon implicite. L'héritage explicite permet quant à lui de différencier certains groupes d'attributs et de n'hériter que des morceaux d'objets dûment explicités. L'héritage par plongement consiste à rendre les attributs de l'objet fils identiques aux attributs hérités du père. L'héritage par délégation laisse les attributs hérités propriété du père.

Bien que le concept de classe soit utilisé par la suite dans le codage des objets que nous proposons pour le langage ELAN, ce codage définit en fait un langage à prototype. En effet, afin de définir une représentation interne pour les classes et les objets d'un tel langage, le mécanisme qui a été choisi est une représentation à base d'objets. Les objets sont donc un unique moyen de représentation commun aux objets eux-mêmes tout comme aux classes. Ceci est détaillé dans les parties qui suivent.

3.2 Implantation des objets en ELAN

On se propose dans cette partie de définir un codage des concepts objets dans le langage ELAN. On va ainsi faire coexister dans une application ELAN des modules standard dans lesquels sont définis des opérateurs avec règles et stratégies associées et d'autres modules dans lesquels sont définies des classes d'objets. Ces derniers modules seront appelés OModules. Dans chaque module objet sera définie une classe composée d'attributs et de méthodes. Tout objet de cette classe sera composé de ces attributs, auxquels des valeurs seront associées, ainsi que de références aux méthodes accessibles par l'objet. Un programme basé sur les objets consiste ainsi à invoquer des méthodes : on parle d'envoi de message à un objet où le message est le nom de la méthode que l'on cherche à appliquer. L'objet ciblé répond au message en exécutant la méthode associée si celle-ci est accessible.

La syntaxe générale d'un module objet est décrite dans une première partie. Ensuite, quelques points particuliers sont développés comme l'héritage, l'importation de modules. La syntaxe générale complète est donnée en Annexe A.

3.2.1 Syntaxe générale d'un module objet

A toute définition de classe d'objets correspond un module spécifique dans lequel la classe est définie. Afin de différencier ces modules des autres modules ELAN, une syntaxe particulière pour la définition de modules orientés objets est introduite.

La définition d'une classe se fait dans un module objet avec la syntaxe[2] suivante :

<module objet>::= **class** *<nom de la classe>*
<héritage>
<importations>
<attributs>
<méthodes>
End

dans lequel sont successivement déclarés :

- les classes dont elle hérite,
- les modules importés,
- les attributs composant chaque objet de la classe,
- les méthodes associées à la classe.

Les deux dernières parties sont détaillées avant l'héritage et l'importation de modules.

3.2.2 Définition des attributs

La définition des attributs se fait de la manière suivante :

<attributs>::= **attributes** *<liste attributs>*

<liste attributs>::= *<attribut>* |
<attribut> *<liste attributs>*

où les définitions d'attributs sont faites successivement. Chaque déclaration suit la syntaxe suivante :

<attribut>::= *<nom de l'attribut>*:*<type>* = *<valeur initiale>*

A chaque attribut sont associés un type définissant l'ensemble des valeurs pouvant être prises par l'attribut ainsi qu'une valeur initiale qui sera la valeur de l'attribut lors de la création d'un nouvel objet de la classe.

2. Cette définition de syntaxe est dans le style BNF défini comme suit : les symboles terminaux sont mis en gras (**terminal**) et les symboles non terminaux sont mis entre chevrons (*<non terminal>*). Un choix entre deux syntaxes possibles se note *choix 1* | *choix 2*. Une option est notée entre crochets (*[option]*).

Exemple 3.1 *Considérons par exemple la classe* `Point` *définissant les points dans un espace à deux dimensions. Deux attributs sont définis dans cette classe, un attribut appelé* `X` *définissant l'abscisse de tout point et un attribut* `Y` *pour l'ordonnée. Une telle classe est déclarée dans le module objet suivant :*

```
class Point
attributes X:int = 0
           Y:int = 0
End
```

Tout objet de cette classe est créé avec une abscisse et une ordonnée initialisées à 0.

3.2.3 Définition des méthodes

Une méthode est une fonction qui s'applique à un objet donné. Cette méthode, qui peut être plus ou moins complexe, peut modifier l'objet cible. Dans tous les cas, l'appel d'une méthode `m` sur un objet `o` se note : `o.m`. On parle aussi dans ce cas d'envoi de message à un objet.

`Get` et `Set` : des méthodes particulières

A chaque attribut sont associées deux méthodes particulières qui permettent de les manipuler : la première méthode permet de récupérer la valeur courante de l'attribut et la deuxième permet de remplacer la valeur de l'attribut par une nouvelle valeur donnée en paramètre.

Ces méthodes n'ont pas besoin d'être explicitées, elles sont implicites pour chaque déclaration d'attribut.

Définition 3.1 *Soit une classe C dans laquelle un attribut A de type t_A est déclaré. A cette classe sont associées deux méthodes $GetA$ et $SetA$.*

- *Le message $o.GetA$ appelle la méthode $GetA$ sur l'objet o. Le résultat de cet appel de méthode est de type t_A et est la valeur associée à l'attribut A au moment de l'appel sur o.*
- *Le message $o.SetA(V)$ appelle la méthode $SetA$ sur l'objet o avec un paramètre V de type t_A. Le résultat de cet appel de méthode est l'objet o dont la valeur de l'attribut A est modifiée en V.*

Exemple 3.2 *Dans la classe Point précédemment définie, prenons un objet O_1 de cette classe avec deux attributs X et Y initialisés à 0. Cet objet O_1 est noté :*

$$O_1[X(0)\ ,\ Y(0)]$$

- *$O_1.SetY(34)$ est l'appel de la méthode $SetY$ sur O_1 avec la nouvelle valeur de Y mise à 34. Le résultat est l'objet O_1 mis à jour : $O_1[X(0)\ ,\ Y(34)]$.*
- *$O_1.GetY$ est l'appel de la méthode $GetY$ sur O_1. Considérant O_1 comme l'objet précédemment modifié par $SetY$, le résultat est l'entier associé à l'attribut Y : 34.*

Les méthodes

La syntaxe générale des déclarations de méthodes est :

<méthodes>::= *<méthode>* |
<méthode> *<méthodes>*

Une méthode est définie par son nom, une liste d'arguments si la méthode nécessite des paramètres, le type du résultat et enfin le corps de la méthode :

<méthode>::= **method** *<nom de la méthode>*[(*<arguments>*)] **for** *<type du résultat>*
< *<corps de méthode>*>

Les arguments sont déclarés comme suit :

<arguments>::= *<argument>*: *<type argument>*|
<argument>: *<type argument>*, *<arguments>*

Les arguments déclarés ici sont des paramètres de la méthode. Notons que l'objet auquel est appliquée une méthode est un paramètre particulier qui n'a pas à être déclaré dans cette partie mais qui peut être référencé dans le corps de la méthode via le mot clé **self** qui désigne l'objet auquel le message est envoyé par `objet.méthode`.

Les corps de méthodes sont composés de différentes *instructions*. La première instruction est l'appel de méthode :

<appel de méthode>::= *<identificateur>* . *<nom de méthode>*[(*<paramètres>*)]

Exemple 3.3 *Enrichissons la classe* `Point` *précédente et définissons la classe* `PointReinitX` *qui, en plus des déclarations d'attributs* `X` *et* `Y`*, définit une méthode* `ReinitX` *réinitialisant à* `0` *l'abscisse de l'objet auquel la méthode est appliquée. Une telle classe est définie comme suit à l'aide des méthodes* `GetX`*,* `SetX`*,* `GetY` *et* `SetY` *implicitement définies :*

```
class PointReinitX
attributes X:int = 0
           Y:int = 0
method ReinitX for PointReinitX <self.SetX(0)>
End
```

De même, la classe PointReinitXY définissant une méthode réinitialisant X et Y est définie par :

```
class PointReinitXY
attributes X:int = 0
           Y:int = 0
method ReinitX for PointReinitXY <self.SetX(0)>
method ReinitY for PointReinitXY <self.SetY(0)>
End
```

Un appel de méthode peut aussi être l'appel à une fonction préalablement définie dans un module standard. Si l'on considère une fonction f avec les paramètres x_1 , ... , x_n, l'appel de méthode est alors de la forme : $f(x_1,\ldots,x_n)$.

<appel de méthode>::= *<identificateur>*[(*<paramètres>*)]

Néanmoins, les corps de méthodes doivent pouvoir être plus complexes afin de permettre de définir des méthodes dont la syntaxe enrichie peut s'avérer être plus appropriée au problème traité. C'est pourquoi nous introduisons dans la suite la concaténation d'instructions, les tests booléens et les affectations de variables locales à une méthode.

Pour enchaîner plusieurs instructions dans un même corps de méthode, le mot clé ";" est introduit afin de séparer les instructions les unes des autres.

Exemple 3.4 *Considérons la classe* `PointReinitXY` *précédente et définissons à la place la classe* `PointReinit` *qui, en plus des méthodes précédentes, définit aussi la méthode* `Reinit` *qui consiste à appeler successivement les deux autres méthodes :*

```
class PointReinit
attributes X:int = 0
           Y:int = 0
method ReinitX for PointReinit <self.SetX(0)>
method ReinitY for PointReinit <self.SetY(0)>
method Reinit for PointReinit <self.ReinitX ; self.ReinitY>
End
```

L'enchaînement des différentes instructions composant une méthode se fait de gauche à droite et l'échec d'une des instructions entraîne l'échec global de la méthode, avant même que les instructions suivantes ne soient exécutées.

La possibilité de faire des tests booléens est aussi une autre fonctionnalité ajoutée au formalisme des méthodes. La syntaxe des tests est la suivante :

<test>::= **if** *<expression booléenne>*

On doit différencier le test booléen effectué ici du "if-then-else" peut-être plus courant. En effet, le test booléen tel qu'il est défini ici permet de vérifier la validité d'une expression booléenne en cours d'exécution. Si cette expression booléenne est évaluée en *vrai*, on poursuit l'exécution du corps de la méthode. Sinon, il y a échec de la règle.

Exemple 3.5 *Considérons la classe* `PointReinit` *précédente avec la méthode de réinitialisation modifiée afin de ne la faire que si les valeurs de* `X` *ou de* `Y` *sont différentes de* `0` *:*

```
class PointReinitWithCondition
attributes X:int = 0
           Y:int = 0
method ReinitX for PointReinitWithCondition <self.SetX(0)>
method ReinitY for PointReinitWithCondition <self.SetY(0)>
method Reinit  for PointReinitWithCondition <if self.GetX != 0 or self.GetY != 0 ;
                                             self.ReinitX ; self.ReinitY>
End
```

Les expressions booléennes telles que la négation (`not`), la conjonction (`and`) et la disjonction (`or`) sont disponibles dans l'implantation réalisée.

La dernière opération est la possibilité d'affecter des variables locales à la définition de la méthode, ces variables ayant été précédemment déclarées dans une liste de variables locales. La syntaxe des affectations locales est la suivante :

<affectation locale>::= *<nom de variable>*:= *<appel de méthode>*

Exemple 3.6 *Considérons la classe* `PointTranslation` *qui définit deux méthodes : une première* `TranslateX` *qui modifie la valeur de l'attribut* `X` *en ajoutant la valeur d'un paramètre* `N` *à l'ancienne valeur de l'attribut* `X`*. La deuxième,* `Translate`*, n'appelle la translation sur* `X` *que si une condition sur les valeurs de* `X` *et* `Y` *est vérifiée ; si tel est le cas, une valeur* `N` *est calculée grâce à l'utilisation d'une variable locale pour enfin appeler* `Translate` *paramétrée avec* `N`*.*

```
class PointTranslation
attributes X:int = 0
           Y:int = 0
method TranslateX(N:int) for PointTranslation <self.SetX(self.GetX + N)>
method Translate for PointTranslation N:int <if self.GetX > self.GetY ;
                                             N := self.GetX - self.GetY ;
                                             self.TranslateX(N)>
End
```

Nous avons ainsi décrit comment définir des attributs et des méthodes associés à une classe.

`new` : méthode de création

De la même façon, à chaque définition de classe est associée une méthode de création de classe que l'on appelle `new`. Son appel se fait de la façon suivante : *nom_de_classe*`.new`.

Le but de cette méthode associée à toute classe est de pouvoir générer un nouvel objet de la classe. Tout nouvel objet `O` créé avec pour type le nom de la classe est composé de :

- l'ensemble des attributs associés à la classe, chaque attribut étant initialisé avec la valeur initiale donnée lors de la création de la classe ;

S.C.D. - U.H.P. NANCY 1
BIBLIOTHÈQUE DES SCIENCES
Rue du Jardin Botanique
54600 VILLERS-LES-NANCY

- l'ensemble des noms de méthodes associées à la classe. Cet ensemble non vide de noms de méthodes indique qu'à sa création, tout objet peut appeler n'importe quelle méthode disponible pour sa classe. Ces accès peuvent ensuite évoluer en cours d'exécution : une méthode peut ainsi devenir inaccessible lors de l'exécution puis devenir de nouveau accessible.

3.2.4 L'importation de modules

Chaque module objet peut, grâce au mot clé `imports` importer d'autres modules ELAN dans lesquels sont définis des opérateurs utilisables par tout objet de la classe. On doit bien distinguer une importation de modules de la notion d'héritage décrite plus tard : en effet, une importation de module ne concerne que des modules ELAN et non des modules objets. Par exemple, une importation de module permet d'avoir accès à un opérateur f qui est défini ailleurs.

Exemple 3.7 *En se basant sur la version de la classe Point donnée dans l'exemple 3.1, on cherche maintenant à l'enrichir afin de définir des points colorés. Les couleurs définies pour les points le sont dans un autre module* ELAN *qui s'appelle* `color.eln` *et dans lequel le type* `color` *est défini comme le type énumérant toutes les valeurs possibles de couleurs.*

Si l'on appelle la nouvelle classe de points colorés la classe ColoredPoint, que l'attribut désignant la couleur est l'attribut `Color`*, de type* `color` *et initialisé avec la valeur* `Black` *définie dans le module importé* `color.eln`*, on peut alors définir ColoredPoint par :*

```
class ColoredPoint
imports color
attributes X:int = 0
           Y:int = 0
           Color:color = Black
End
```

3.2.5 L'héritage

Un module objet peut hériter des attributs et méthodes d'une autre classe en utilisant le mot clé **from** comme suit :

<*héritage*>::= **from** <*nom de classe*>

L'héritage permet à une classe `B`, en héritant d'une classe `A`, de se spécialiser en ayant un accès à tous les attributs et méthodes définis en `A`. Les définitions contenues dans la définition de la classe `B` permettent d'ajouter de nouveaux attributs, de nouvelles méthodes ou encore de redéfinir des méthodes de `A` plus appropriées.

L'héritage autorisé dans notre syntaxe est l'héritage simple, c'est-à-dire qu'une classe ne peut hériter que d'une seule autre classe.

Exemple 3.8 *Considérons l'exemple 3.1 définissant la classe Point et l'exemple 3.7 définissant la classe ColoredPoint. Au lieu de redéfinir la classe ColoredPoint ex-nihilo, le mécanisme d'héritage permet de définir simplement la classe ColoredPoint à partir de la classe Point comme suit :*

```
class ColoredPoint
imports color
from Point
attributes Color:color = Black
End
```

Les attributs `X` *et* `Y` *de la classe Point sont alors hérités ainsi que les méthodes* `GetX`*,* `GetY`*,* `SetX` *et* `SetY` *liées à Point.*

Les méthodes héritées sont alors surchargées avec la définition des mêmes méthodes qui sont cette fois-ci liées à la classe qui hérite.

On peut aussi redéfinir une méthode héritée ; de même qu'un attribut peut être défini avec le même nom qu'un attribut hérité mais avec une valeur initiale différente.

Nous avons défini dans cette partie l'ensemble des caractéristiques des modules objets avec les constructions correspondantes. De façon synthétique, nous pouvons ainsi donner une syntaxe générale des modules objets supportés par le langage ELAN. Le lecteur pourra trouver la syntaxe globale en Annexe A.

3.2.6 Evaluation des objets

Après avoir défini les classes, nous sommes maintenant en mesure de définir comment manipuler les objets et comment les évaluer. La manipulation des objets se fait par appel de méthode.

Chaque objet est créé à partir de la méthode **new** qui est associée à chaque classe. Ainsi, pour chaque classe C, l'appel de $C.new$ permet d'obtenir comme résultat un objet o de cette classe tel que la méthode new le définit.

Pour modifier un objet, on procède par appel de méthode. Considérons un objet o et une méthode m avec n paramètres p_i, l'appel de méthode se fait par $o.m(p_1,\ldots,p_n)$. Le résultat est celui de l'évaluation de la méthode avec les arguments donnés (l'objet et les paramètres).

3.3 Le calcul de réécriture

Afin de définir une sémantique aux objets et aux primitives associées en ELAN, nous avons choisi le calcul de réécriture dans lequel ELAN peut lui-même être défini. Le but est ici de montrer que le codage des objets peut être exprimé dans le même formalisme. Dans un premier temps, dans cette partie, on présente le calcul de réécriture.

Le calcul de réécriture, aussi appelé ρ-calcul, a été défini par C. Kirchner et H. Cirstea [CK99a, CK99b, Cir00]. Il répond à une volonté de pouvoir définir dans une sémantique unifiée à la fois la notion de règle de réécriture et la notion d'application d'une règle sur un terme. Ce calcul s'ajoute à quelques autres travaux déjà réalisés sur le thème comme les travaux N. Martì-Oliet et J. Meseguer dans [MOM93].

Lors de leurs travaux, H. Cirstea et C. Kirchner ont défini un calcul uniforme qui leur permet de coder différents paradigmes, notamment le calcul orienté objet de K. Fisher, F. Honsell et J.-C. Mitchell [FHM94] et celui de M. Abadi et L. Cardelli [AC96]. Les codages des calculs objets ont été étudiés dans [CKL00, CKL01b] où une version du ρ-calcul adaptée aux calculs objets est présentée.

Cette section présente dans un premier temps la version du calcul de réécriture qui a été définie dans [CKL01b] et ses principaux résultats. Ensuite, l'extension du calcul de réécriture aux calculs objets et présentée dans [CKL01a] est décrite.

3.3.1 Le ρ-calcul : syntaxe et sémantique

La syntaxe du ρ-calcul est définie par :

$$\begin{array}{lll} t ::= & a \mid Y \mid t \to t \mid t \bullet t \mid & \text{termes non structurés} \\ & null \mid t,t & \text{termes structurés} \end{array}$$

Le symbole t est un terme de l'ensemble de termes $\mathcal{T}$; les symboles $S, Y, Z, \ldots$ dénotent des variables, éléments de l'ensemble de variables $\mathcal{V}$; les symboles $a, b, \ldots, m, 0, 1, \ldots, null, kill, \ldots$ sont des symboles d'arité fixe (constantes ou opérateurs) d'un ensemble de symboles $\mathcal{C}$. Tous ces symboles peuvent être indexés. Le symbole $\triangleq$ dénote l'égalité par définition entre deux expressions.

Une règle de réécriture $t \to t$ est un abstracteur dont la partie gauche détermine les variables liées et le contexte. L'application d'un ρ-terme sur un autre ρ-terme est représentée par l'opérateur $\bullet$ qui est associatif à gauche. Les ρ-termes peuvent être associés dans une structure basée sur l'opérateur ",". Ce symbole peut être lié à différentes théories : la théorie vide, associative, associative-commutative, etc...

Cette théorie notée $\mathbb{T}$ donne le type de structure que l'on associe à l'opérateur ",": une théorie associative définit une structure de liste, une théorie associative-commutative et idempotente définit une structure ensembliste. La constante *null* dénote la structure vide. Des priorités sont associées à ces opérateurs: la priorité la plus élevée est donnée à $\bullet$; ensuite, $\rightarrow$ a une priorité plus élevée que ",".

Comme pour chaque calcul impliquant des lieurs, par exemple le lambda calcul, l'α-conversion est utilisée pour obtenir un calcul des substitutions correct et la substitution du premier ordre n'est pas directement appropriée au ρ-calcul puisque terme et application sont mis au même niveau de représentation. On considère les notations usuelles de l'α-conversion et de la substitution d'ordre supérieur comme définies dans [DHK00, CKL01a].

Définition 3.2 *L'ensemble FV des Variables Libres est défini récursivement par:*

$$\begin{array}{rcl} FV(a) & \triangleq & \emptyset \\ FV(X) & \triangleq & X \\ FV(t_1 \rightarrow t_2) & \triangleq & FV(t_1) \setminus FV(t_2) \\ FV(t_1 \bullet t_2) & \triangleq & FL(t_1) \cup FV(t_2) \\ FV(t_1,t_2) & \triangleq & FV(t_1) \cup FV(t_2) \end{array}$$

Définition 3.3 *Etant donné un ensemble de variables $\mathcal{X}$ tel que $\mathcal{X} \subseteq \mathcal{V}$, l'application $\alpha_{\mathcal{X}}$, aussi appelée α-conversion, est définie par:*

$$\begin{array}{rcl} \alpha_{\mathcal{X}}(a) & \triangleq & a \\ \alpha_{\mathcal{X}}(X) & \triangleq & X \\ \alpha_{\mathcal{X}}(t_1 \bullet t_2) & \triangleq & \alpha_{\mathcal{X}}(t_1) \bullet \alpha_{\mathcal{X}}(t_2) \\ \alpha_{\mathcal{X}}(t_1 \rightarrow t_2) & \triangleq & \alpha_{\mathcal{X}}(t_1) \rightarrow \alpha_{\mathcal{X}}(t_2) \\ & & \quad \textit{si } FV(t_1) \bigcap \mathcal{X} = \emptyset \\ \alpha_{\mathcal{X}}(t_1 \rightarrow t_2) & \triangleq & \{X_i \mapsto Y_i\}_{i=[1,\ldots,n]} \alpha_{\mathcal{X}}(t_1) \rightarrow \{X_i \mapsto Y_i\}_{i=[1,\ldots,n]} \alpha_{\mathcal{X}}(t_2) \\ & & \quad \textit{si chaque } X_i \in FV(t_1) \bigcap \mathcal{X} \textit{ et chaque } Y_i \textit{ sont des variables fraîches et où} \\ & & \quad \{X \mapsto Y\} \textit{ dénote le remplacement de la variable } X \textit{ par la variable } Y \textit{ dans} \\ & & \quad \textit{le terme sur lequel il est appliqué.} \end{array}$$

Cela nous permet de définir l'opération classique de substitution:

Définition 3.4 *L'application d'une substitution $\alpha \equiv \{V_1 \mapsto v_1, \ldots, V_n \mapsto v_n\}$ est définie structurellement par:*

$$\begin{array}{rcl} \alpha(a) & \triangleq & a \\ \alpha(V_i) & \triangleq & v_i \\ \alpha(t_1 \bullet t_2) & \triangleq & (\alpha(t_1)) \bullet (\alpha(t_2)) \\ \alpha(t_1,t_2) & \triangleq & (\alpha(t_1)),(\alpha(t_2)) \\ \alpha(t_1 \rightarrow t_2) & \triangleq & (\alpha(t_1')) \rightarrow (\alpha(t_2')) \end{array}$$

où l'on définit t_1' et t_2' par:

$$\begin{array}{rcl} t_1' & \triangleq & \{X_i \mapsto Z_i\} \alpha_{FV(t_1) \cup Var(\alpha)}(t_1) \\ t_2' & \triangleq & \{X_i \mapsto Z_i\} \alpha_{FV(t_1) \cup Var(\alpha)}(t_2) \end{array}$$

où l'on a $X_i \in FV(t_1)$, Z_i sont des variables fraîches qui n'apparaissent ni dans t_1 ni dans t_2 et telles que $\alpha Z_i = Z_i$ et que pour tout $Y \in FV(t_1)$, $Z_i \notin FV(\alpha Y)$.

La notation $\{V_1 \mapsto v_1, \ldots, V_n \mapsto v_n\}$ dénote la substitution simultanée des variables $V_1 \ldots V_n$ par les termes $v_1 \ldots v_n$ et non la composition $\{V_1 \mapsto v_1\} \ldots \{V_n \mapsto v_n\}$.

On considère un ordre total $\prec$ sur l'ensemble des substitutions. De tels ordres peuvent très facilement être définis et existent toujours. On peut prendre par exemple l'ordre lexicographique sur la représentation aplatie des substitutions.

Définition 3.5 *On définit les abréviations suivantes :*

$$\begin{array}{lcll} t(t_1 \dots t_n) & \triangleq & t \bullet t_1 \dots \bullet t_n & \textit{application fonctionnelle} \\ (t_i)^{i=1\dots n} & \triangleq & t_1, \dots, t_n & \textit{structure composée de } n \textit{ éléments } t_i \end{array}$$

où $n \in \mathbb{N}^*$.

Exemple 3.9 *On peut maintenant donner quelques exemples simples de ρ-termes :*

- *Le ρ-terme $(f(X) \to X) \bullet f(a)$ représente l'application de la règle de réécriture $f(X) \to X$ sur le terme $f(a)$.*
- *Le ρ-terme $X \to Y \to X$ représente le λ-terme $\lambda(X).\lambda(Y).X$.*
- *Le ρ-terme $X \to 0, Y \to 0$ dénote une structure binaire avec deux champs, X et Y, valant tous les deux 0.*

Un paramètre important du ρ-calcul est la théorie $\mathbb{T}$ définissant la structure de ρ-termes. $\mathbb{T}$ est une théorie de filtrage qui est définie par différentes règles d'inférence suivant la théorie utilisée. Différentes théories peuvent être considérées et une liste non exhaustive est donnée dans [CKL01a]. Dans la définition suivante, nous ne mentionnons que deux théories qui nous intéressent plus particulièrement par la suite : la théorie vide et la théorie associative-commutative et idempotente liée à la structure ensembliste.

Définition 3.6

- *La théorie vide $\mathbb{T}_\emptyset$ est définie par les règles d'inférences suivantes :*

$$\frac{t_1 = t_2 \quad t_2 = t_3}{t_1 = t_3}\ \textit{Transitivité} \qquad \frac{t_1 = t_2}{t_2 = t_1}\ \textit{Symétrie}$$

$$\frac{t_1 = t_2}{t_3[t_1]_p = t_3[t_2]_p}\ \textit{Contexte} \qquad \frac{\mathcal{X} \subset \mathcal{V}}{t = \alpha_{\mathcal{X}}(t)}\ \textit{Réflexivité modulo } \alpha$$

- *La théorie Commutative $\mathbb{T}_{C(f)}$ pour un symbole binaire donné f est définie par les règles de $\mathbb{T}_\emptyset$ auxquelles on ajoute la règle :*

$$\frac{}{f(t_1\ t_2) = f(t_2\ t_1)}\ \textit{Commutativité}$$

- *La théorie Associative $\mathbb{T}_{A(f)}$ pour un symbole binaire donné f est définie par les règles de $\mathbb{T}_\emptyset$ auxquelles on ajoute la règle d'inférence suivante :*

$$\frac{}{f(f(t_1\ t_2)\ t_3) = f(t_1\ f(t_2\ t_3))}\ \textit{Associativité}$$

- *La théorie Idempotente $\mathbb{T}_{I(f)}$ pour un symbole binaire donné f est définie par les règles de $\mathbb{T}_\emptyset$ auxquelles on ajoute la règle :*

$$\frac{}{f(t\ t) = t}\ \textit{Idempotence}$$

- *La théorie Associative-Commutative Idempotente est définie par l'ensemble de ces règles pour tout symbole f déclaré comme tel.*

Une fois définies les différentes règles associées aux théories, on peut maintenant définir la notion de filtrage.

Définition 3.7 *Pour une théorie $\mathbb{T}$ donnée sur les ρ-termes :*

1. *une équation de filtrage suivant la théorie $\mathbb{T}$ est une formule de la forme $t_1 \ll_{\mathbb{T}} t_2$;*
2. *une substitution α est une solution de l'équation de filtrage $t_1 \ll_{\mathbb{T}} t_2$ suivant la théorie $\mathbb{T}$ si $\alpha t_1 =_{\mathbb{T}} t_2$ où $=_{\mathbb{T}}$ dénote l'égalité modulo les règles de la théorie $\mathbb{T}$;*
3. *un système de filtrage suivant la théorie $\mathbb{T}$ est une conjonction d'équations de filtrage suivant la théorie $\mathbb{T}$;*
4. *une substitution α est une solution du système de filtrage suivant la théorie $\mathbb{T}$ si elle est solution pour toute équation de filtrage suivant la théorie $\mathbb{T}$ le composant ;*
5. *un système de filtrage suivant la théorie $\mathbb{T}$ est trivial lorsque toute substitution peut être solution du système. Un système de filtrage suivant la théorie $\mathbb{T}$ n'ayant aucune solution est noté $\mathbb{F}$;*
6. *on définit la fonction Sol sur un système de filtrage S suivant la théorie $\mathbb{T}$ comme la fonction retournant soit une liste ordonnée suivant $\prec$[3] de tous les filtres de S suivant la théorie $\mathbb{T}$ lorsque S est non trivial, soit une liste ne contenant que l'élément σ_{id} lorsque S est trivial, où σ_{id} est la substitution identité.*

Lorsqu'un algorithme de filtrage échoue, la fonction *Sol* retourne la liste vide.

Etant donnés un ordre total $\prec$ et une théorie $\mathbb{T}$, l'évaluation du ρ-calcul est définie par les règles suivantes :

$$\text{(Application - } \rho) \quad (t_1 \to t_2) \bullet t_3 \mapsto_{\mathbb{T}} \begin{cases} null & \text{lorsque } t_1 \ll_{\mathbb{T}} t_3 \text{ n'a pas de solution} \\ \sigma_1 t_2, \ldots, \sigma_n t_2 & \text{lorsque } \sigma_i \in Sol(t_1 \ll_{\mathbb{T}} t_3)\ (n < \infty) \end{cases}$$

$$\text{(Distribution1 - } \epsilon) \quad (t_1, t_2) \bullet t_3 \mapsto_{\mathbb{T}} t_1 \bullet t_3, t_2 \bullet t_3$$

$$\text{(Distribution2 - } \nu) \quad null \bullet t \mapsto_{\mathbb{T}} null$$

Intuitivement, nous pouvons commenter ces quelques règles : la règle principale d'application (aussi notée ρ) décrit l'application en tête d'une règle de réécriture sur un terme t_3 comme le calcul de toutes les solutions de l'équation de filtrage $t_1 \ll_{\mathbb{T}} t_3$ dans la théorie $\mathbb{T}$ suivi de l'application de ces substitutions sur le terme t_2 selon la liste ordonnée suivant un ordre total $\prec$. Lorsqu'il n'y a pas de solution à l'équation de filtrage $t_1 \ll_{\mathbb{T}} t_3$, l'application de la règle rend *null*. Notons que l'on considère dans ce cas un ensemble fini de solutions ; les théories pouvant produire une infinité de résultats [FH86] ne sont donc pas considérées dans notre approche. Néanmoins, on peut citer les travaux d'H. Cirstea, et notamment sa thèse [Cir00], dans lesquels différentes approches sont données afin de pouvoir traiter une infinité de solutions. On peut remarquer que dans le cas où t_1 est une variable, la règle (ρ) correspond à la β-réduction du λ-calcul.

Les règles de distribution (règles ϵ et ν) concernent la distribution de l'application sur les structures dont les opérateurs de base sont "," et *null*.

Dans les travaux sur le ρ-calcul, quelques résultats ont été présentés sur la confluence. Notamment, il a été montré que l'évaluation simple d'un ρ-terme par les règles du ρ-calcul n'est pas confluente si les règles ne sont pas appliquées avec un certaine stratégie. Par contre, la définition de stratégies adéquates impliquant quelques restrictions simples sur les ρ-termes permet d'avoir la confluence du ρ-calcul. Le détail de ces résultats peut être trouvé dans la thèse d'H. Cirstea [Cir00] mais il s'agit d'une version du ρ-calcul [CKL01b] différente de la version utilisée ici.

3.3.2 Le ρ-calcul objet

Dans le ρ-calcul, les objets peuvent être naturellement définis comme des ρ-termes structurés. Des résultats ont été montrés dans [CKL01a] sur la possibilité de représenter dans le ρ-calcul des objets et notamment ceux décrits dans le *Lambda Calcul Objet* de K. Fisher, F. Honsell et J.C. Mitchell [FHM94] et ceux définis dans le *Calcul Objet* de M. Abadi et L. Cardelli [AC96].

3. où $\prec$ est un ordre total sur l'ensemble des substitutions.

Ces deux calculs objets reposent sur le concept d'objet, toute classe étant elle aussi vue comme un objet. Ce sont des langages dits *à prototype*. Ils ont fortement influencé la recherche sur la théorie des types puisque ces calculs ont été étudiés et typés afin de capturer le message d'erreur `message-not-understood` qui est produit lorsque l'on envoie un message m à un objet qui n'a pas de méthode correspondante définie dans son interface.

Le *Lambda Calcul Objet* de K. Fisher, F. Honsell et J.C. Mitchell [FHM94] est un λ-calcul avec constantes, non typé et enrichi avec des primitives objet. Un nouvel objet peut être créé en modifiant et/ou en étendant un prototype d'objet existant ; le résultat est un nouvel objet qui hérite des méthodes et des champs du prototype.

Le *Calcul Objet* de M. Abadi et L. Cardelli [AC96] est un calcul où la seule entité est l'objet. Il existe un nombre important de variantes de ce calcul, notamment basées sur le fait que l'on peut travailler dans un environnement impératif ou fonctionnel qui peut être typé ou non typé.

Néanmoins, le ρ-calcul a dû être étendu et adapté afin de pouvoir représenter tout objet du *Calcul Objet* ou du *Lambda Calcul Objet*.

De ces deux calculs exprimables dans le ρ-calcul, le *Calcul Objet* est le plus proche du codage des objets que nous mettons en œuvre, notamment parce que le *Lambda Calcul Objet* permet de définir des extensions aux objets en cours d'exécution, ce qui n'est pas possible en ELAN.

Par conséquent, considérons l'extension du ρ-calcul permettant de coder le *Calcul Objet* de M. Abadi et L. Cardelli [AC96]. Une traduction notée $[\![\text{-}]\!]_{\varsigma Obj}$ est définie comme suit :

Définition 3.8 *A tout ςObj-terme $t_{\varsigma Obj}$ du Calcul Objet on peut associer un ρ-terme t tel que :* $[\![t_{\varsigma Obj}]\!]_{\varsigma Obj} = t$.

Les définitions des règles associées à cette traduction $[\![\text{-}]\!]_{\varsigma Obj}$ sont données dans [CKL01a].

Pour représenter les termes du *Calcul Objet*, de nouvelles fonctions et abréviations ont été définies pour le ρ-calcul dans [CKL01a]. Ainsi, le ρ-calcul a été enrichi par une fonction $kill_m$ qui permet de supprimer d'une structure de ρ-terme la règle de réécriture dont le membre gauche est m :

$$kill_m \ : \ \mathcal{T} \mapsto \mathcal{T} \ \triangleq \ (X, m \to Y) \to X$$

Quelques abbréviations peuvent alors être adoptées dans ce formalisme de codage des objets :

$t_1 \cdot t_2$	$\triangleq$	$t_1 \bullet t_2 \bullet t_1$	appel de méthode
$t_1 \cdot m := t_2$	$\triangleq$	$kill_m(t_1), m \to t_2$	mise à jour de méthode

Afin de mieux comprendre l'utilisation de ces notations, prenons un objet P du *Calcul Objet* tel qu'il est défini dans le ρ-calcul :

$$\begin{aligned} P \ \triangleq \ & X \to S \to 5, \\ & Y \to S \to 7, \\ & GetX \to S \to S \cdot X, \\ & SetX \to S \to N \to S \cdot X \ := \ S' \to N, \\ & GetY \to S \to S \cdot Y, \\ & SetY \to S \to N \to S \cdot Y \ := \ S' \to N, \\ & ReinitX \to S \to S \cdot SetX(0) \end{aligned}$$

Un tel objet P a deux attributs X et Y valant respectivement 5 et 7 ainsi que cinq méthodes $GetX$, $GetY$, $SetX$, $SetY$ et $ReinitX$. Les quatre premières méthodes sont associées aux attributs. La dernière permet d'appeler $SetX$ avec le paramètre 0 afin d'effectuer une remise à zéro de l'attribut X.

Dans les définitions de chaque méthode ou attribut, l'abstraction par la variable liée S représente l'objet lui-même. Chaque objet est ainsi un paramètre de toute méthode ou attribut le définissant, ce qui permet de pouvoir l'utiliser dans le membre droit d'une règle du ρ-calcul. Ainsi, dans toute définition d'attribut, la variable S, malgré le fait d'être déclarée, n'est pas utilisée. La variable liée S' est un nouveau nom donné à la variable d'abstraction S lors de la définition d'une nouvelle règle composant le ρ-terme. Illustrons le ρ-calcul sur cet exemple en appliquant la méthode $ReinitX$ à cet objet P :

$$
\begin{aligned}
P \cdot ReinitX &\triangleq P \bullet ReinitX \bullet P \\
&\mapsto (S \to S \cdot SetX(0)) \bullet P \\
&\mapsto P \cdot SetX(0) \\
&\triangleq P \cdot SetX \bullet 0 \\
&\triangleq P \bullet SetX \bullet P \bullet 0 \\
&\mapsto (S \to N \to S \cdot X \ := \ S' \to N) \bullet P \bullet 0 \\
&\mapsto (N \to P \cdot X \ := \ S' \to N) \bullet 0 \\
&\mapsto P \cdot X \ := \ S' \to 0 \\
&\triangleq kill_X(P) \ , \ X \to S' \to 0 \\
&\triangleq (Y \to S \to 7, \\
&\quad GetX \to S \to S \cdot X, \\
&\quad SetX \to S \to N \to S \cdot X \ := \ S' \to N, \\
&\quad GetY \to S \to S \cdot Y, \\
&\quad SetY \to S \to N \to S \cdot Y \ := \ S' \to N, \\
&\quad ReinitX \to S \to S \cdot SetX(0)) \ , \ X \to S' \to 0 \\
&\mapsto X \to S \to 0, \\
&\quad Y \to S \to 7, \\
&\quad GetX \to S \to S \cdot X, \\
&\quad SetX \to S \to N \to S \cdot X \ := \ S' \to N, \\
&\quad GetY \to S \to S \cdot Y, \\
&\quad SetY \to S \to N \to S \cdot Y \ := \ S' \to N, \\
&\quad ReinitX \to S \to S \cdot SetX(0)
\end{aligned}
$$

Dans le ρ-calcul, comme les classes sont vues comme des objets, la classe *Point* correspondant à la classe associée à l'objet P ci-dessus est définie par le ρ-terme suivant :

$$
\begin{aligned}
Point \ \triangleq \ & new \ \to S \to \\
& \quad (X \to S' \to (S \cdot X) \bullet S', \\
& \quad Y \to S' \to (S \cdot Y) \bullet S', \\
& \quad GetX \to S' \to (S \cdot GetX) \bullet S', \\
& \quad SetX \to S' \to (S \cdot SetX) \bullet S', \\
& \quad GetY \to S' \to (S \cdot GetY) \bullet S', \\
& \quad SetY \to S' \to (S \cdot SetY) \bullet S', \\
& \quad ReinitX \to S' \to (S \cdot ReinitX) \bullet S'), \\
& X \to S \to S' \to 0, \\
& Y \to S \to S' \to 0, \\
& GetX \to S \to S' \to S' \cdot X, \\
& SetX \to S \to S' \to N \to S' \cdot X \ := \ S'' \to N, \\
& GetY \to S \to S' \to S' \cdot Y, \\
& SetY \to S \to S' \to N \to S' \cdot Y \ := \ S'' \to N, \\
& ReinitX \to S \to S' \to S' \cdot SetX(0)
\end{aligned}
$$

On retrouve dans cette définition de la classe tous les ρ-termes composant tout objet de cette classe et un ρ-terme particulier correspondant à la méthode *new*. C'est l'application de la méthode *new* sur l'objet représentant la classe qui crée un nouvel objet de cette classe.

Le point délicat de cette définition vient du fait que l'on a besoin de déclarer une nouvelle variable S' dans tous les ρ-termes de la classe ainsi que dans le ρ-terme associé à la méthode *new*. Cette variable va servir de référence à chaque nouvel objet créé par la méthode *new* de cette classe.

De façon schématique, la création d'un nouvel objet se fait de la façon suivante : on cherche à appliquer la méthode *new* d'un objet représentant une classe par appel de méthode. Dans l'exemple précédent, ceci est réalisé par l'appel $Point \cdot new$. Le ρ-terme associé à cette méthode permet de créer un nouveau ρ-terme structuré représentant le nouvel objet. La structure de l'objet est donnée dans la définition de la méthode

new ; ainsi, dans l'exemple donné ci-dessus, tout nouvel objet créé est représenté par ρ-terme structuré :

$$\begin{array}{l} X \to S' \to (S \cdot X) \bullet S', \\ Y \to S' \to (S \cdot Y) \bullet S', \\ GetX \to S' \to (S \cdot GetX) \bullet S', \\ SetX \to S' \to (S \cdot SetX) \bullet S', \\ GetY \to S' \to (S \cdot GetY) \bullet S', \\ SetY \to S' \to (S \cdot SetY) \bullet S', \\ ReinitX \to S' \to (S \cdot ReinitX) \bullet S' \end{array}$$

Ensuite, cette structure est transformée grâce aux définitions des méthodes qui sont données par d'autres ρ-termes composant le ρ-terme structuré représentant la classe. En effet, on peut accéder aux déclarations des méthodes grâce à la variable S qui est la première variable d'abstraction de la méthode *new* et qui est instanciée selon la définition $t_1 \cdot t_2 \triangleq t_1 \bullet t_2 \bullet t_1$. Ainsi, pour la classe *Point*, on a $Point \cdot new \triangleq Point \bullet new \bullet Point$ et, ainsi, la variable S de la méthode *new* est remplacée par le ρ-terme représentant l'objet *Point*. On a alors accès aux définitions de toutes les méthodes. En effet, si l'on se réfère à l'exemple, tous les appels de méthodes $(S \cdot X)$, ..., $(S \cdot ReinitX)$ peuvent être instanciés. Puis, conformément aux définitions des méthodes autres que *new*, l'appel de méthode est réalisé. Si l'on considère uniquement le ρ-terme $X \to S' \to (S \cdot X) \bullet S'$ dans le nouvel objet en cours de création, celui-ci devient $X \to S' \to ((S \to S' \to 0) \bullet S) \bullet S'$, puis, après simplification, on obtient le ρ-terme suivant : $X \to S' \to 0$. On obtient ainsi le premier ρ-terme composant le ρ-terme structuré représentant le nouvel objet créé. On peut remarquer qu'on obtient bien la même forme de ρ-terme que lorsqu'on définit directement un objet comme présenté plus avant dans cette partie.

L'exemple suivant concerne l'application de la méthode *new* sur l'objet *Point* défini précédemment et permet de présenter complètement la création d'un nouvel objet par la méthode décrite ci-dessus :

$$\begin{array}{ll} Point \cdot new & \triangleq Point \bullet new \bullet Point \\ & \triangleq (S \to (X \to S' \to (S \cdot X) \bullet S', \\ & \quad Y \to S' \to (S \cdot Y) \bullet S', \\ & \quad GetX \to S' \to (S \cdot GetX) \bullet S', \\ & \quad SetX \to S' \to (S \cdot SetX) \bullet S', \\ & \quad GetY \to S' \to (S \cdot GetY) \bullet S', \\ & \quad SetY \to S' \to (S \cdot SetY) \bullet S', \\ & \quad ReinitX \to S' \to (S \cdot ReinitX) \bullet S')) \bullet Point \\ & \mapsto X \to S' \to (Point \cdot X) \bullet S', \\ & \quad Y \to S' \to (Point \cdot Y) \bullet S', \\ & \quad GetX \to S' \to (Point \cdot GetX) \bullet S', \\ & \quad SetX \to S' \to (Point \cdot SetX) \bullet S', \\ & \quad GetY \to S' \to (Point \cdot GetY) \bullet S', \\ & \quad SetY \to S' \to (Point \cdot SetY) \bullet S', \\ & \quad ReinitX \to S' \to (Point \cdot ReinitX) \bullet S' \\ & \triangleq X \to S' \to ((Point \bullet X) \bullet Point) \bullet S', \\ & \quad Y \to S' \to ((Point \bullet Y) \bullet Point) \bullet S', \\ & \quad GetX \to S' \to ((Point \bullet GetX) \bullet Point) \bullet S', \\ & \quad SetX \to S' \to ((Point \bullet SetX) \bullet Point) \bullet S', \\ & \quad GetY \to S' \to ((Point \bullet GetY) \bullet Point) \bullet S', \\ & \quad SetY \to S' \to ((Point \bullet SetY) \bullet Point) \bullet S', \\ & \quad ReinitX \to S' \to ((Point \bullet ReinitX) \bullet Point) \bullet S' \\ & \mapsto_\alpha X \to S' \to ((S \to S'' \to 0) \bullet Point) \bullet S', \\ & \quad Y \to S' \to ((S \to S'' \to 0) \bullet Point) \bullet S', \\ & \quad GetX \to S' \to ((S \to S'' \to S \cdot X) \bullet Point) \bullet S', \\ & \quad SetX \to S' \to ((S \to S'' \to N \to S \cdot X \ := \ S''' \to N) \bullet Point) \bullet S', \\ & \quad GetY \to S' \to ((S \to S'' \to S \cdot Y) \bullet Point) \bullet S', \\ & \quad SetY \to S' \to ((S \to S'' \to N \to S \cdot Y \ := \ S''' \to N) \bullet Point) \bullet S', \\ & \quad ReinitX \to S' \to ((S \to S'' \to S \cdot SetX(0)) \bullet Point) \bullet S' \end{array}$$

$$
\begin{array}{ll}
\mapsto & X \to S' \to (S'' \to 0) \bullet S', \\
& Y \to S' \to (S'' \to 0) \bullet S', \\
& GetX \to S' \to (S'' \to S' \cdot X) \bullet S', \\
& SetX \to S' \to (S'' \to N \to S \cdot X \ := \ S''' \to N) \bullet S', \\
& GetY \to S' \to (S'' \to S \cdot Y) \bullet S', \\
& SetY \to S' \to (S'' \to N \to S \cdot Y \ := \ S''' \to N) \bullet S', \\
& ReinitX \to S' \to (S'' \to S' \cdot SetX(0)) \bullet S' \\
\mapsto & X \to S' \to 0, \\
& Y \to S' \to 0, \\
& GetX \to S' \to S' \cdot X, \\
& SetX \to S' \to N \to S \cdot X \ := \ S''' \to N, \\
& GetY \to S' \to S \cdot Y, \\
& SetY \to S' \to N \to S \cdot Y \ := \ S''' \to N, \\
& ReinitX \to S' \to S' \cdot SetX(0)
\end{array}
$$

Nous pouvons ainsi donner une forme générale des classes et des objets dans le ρ-calcul :

Définition 3.9 *Une classe est définie par :*

$$
\begin{array}{lll}
classe & \triangleq & new \to S \to (meth_1 \to S' \to (S \cdot meth_1) \bullet S', \\
& & \quad \ldots \\
& & \quad meth_m \to S' \to (S \cdot meth_m) \bullet S'), \\
& & meth_1 \to S \to S' \to body_1, \\
& & \ldots \\
& & meth_m \to S \to S' \to body_m
\end{array}
$$

où new est une méthode particulière permettant de définir un nouvel objet de la classe. Les m méthodes $meth_1 ... meth_m$ composant la classe sont ensuite définies.

Un objet est défini dans le même formalisme du ρ-calcul par :

$$
\begin{array}{lll}
objet & \triangleq & meth_1 \to S \to body_1, \\
& & \ldots \\
& & meth_m \to S \to body_m
\end{array}
$$

On a ainsi présenté dans cette section le calcul de réécriture tel qu'il a été présenté par H. Cirstea, C. Kirchner et L. Liquori dans [Cir00, CKL01a] qui nous permet de donner une sémantique unifiée pour les notions de règles de réécriture et d'application de cette règle sur un terme. On a aussi montré comment, dans le calcul de réécriture, on peut naturellement représenter des objets.

3.4 Un codage en ELAN

Après avoir présenté le calcul objet, nous sommes en mesure de définir les objets en ELAN dans ce calcul tels que nous les avons implantés. Le but de cette partie est ainsi d'établir le lien existant entre la définition d'objets et classes dans les OModules et leur implantation réalisée en ELAN. Nous basons notre implantation sur la sémantique du calcul objet.

Dans le codage des objets en ELAN, nous avons choisi de distinguer les attributs des méthodes comme c'est le cas dans une grande majorité de langages objet. Nous résumons ici les différentes raisons qui nous ont amené à faire cette différence et à adapter la représentation des ρ-termes objets :

- Dans les problèmes que nous abordons et qui nous ont amenés à vouloir formaliser des objets en ELAN, la manipulation des objets consiste essentiellement en celle des attributs par la lecture et la modification de leurs valeurs.

- La différenciation entre attributs et méthodes nous permet de distinguer ce qui représente l'état d'un objet à un moment donné (les attributs) et les fonctions qui peuvent lui être appliquées.
- Dans l'implantation qui consiste à représenter les méthodes comme des règles de réécriture, le fait de pouvoir modifier ou créer une règle en cours d'exécution n'est pas possible en ELAN.
- La définition des attributs permet d'avoir des types plus ou moins complexes pouvant englober les règles de réécriture par exemple. Cette solution a été envisagée dans les travaux d'H. Cirstea [Cir00] : un objet est défini à partir de méthodes qui sont des règles de réécriture et il est représenté par un terme dont les sous-termes représentent eux-mêmes des règles. L'inconvénient majeur de cette implantation réside dans le fait que l'on doit redéfinir en ELAN tous les mécanismes de filtrage, d'unification, etc. Ce codage très souple qui permet de représenter tout ρ-terme ne prend pas en compte les mécanismes dont on dispose directement en ELAN.

La structure de cette partie est la suivante : après avoir défini comment les objets sont représentés dans le langage ELAN, nous expliquons comment, à partir des déclarations contenues dans les OModules, nous construisons les déclarations d'opérateurs et les règles de réécritures associées. Finalement, on définit le lien existant entre le codage en ELAN et le ρ-calcul, ce qui permet de donner une sémantique dans le ρ-calcul à une évaluation d'un terme par les règles de réécriture définissant le codage des objets en ELAN.

3.4.1 La structure des objets

On choisit de définir la représentation d'un objet en ELAN de la façon suivante :

Définition 3.10 *Un objet o constitué de n attributs a_1,..., a_n prenant les valeurs v_1,..., v_n respectivement dans les domaines D_1,..., D_n et de m méthodes nommées $meth_1$,..., $meth_m$ est représenté en* ELAN *par :*

$$[a_1(v_1)\ ,\ \ldots\ ,\ a_n(v_n)\ ,\ meth_1\ ,\ \ldots\ ,\ meth_m]$$

Cette représentation correspond au ρ-terme :
$o \triangleq a_1 \to S \to v_1,$
$\ldots$
$a_n \to S \to v_n,$
$meth_1 \to S \to body_1,$
$\ldots$
$meth_m \to S \to body_m$

où les corps $body_1$,..., $body_m$ associés à chacune des méthodes sont définis par des règles de réécriture.

Définition 3.11 *Une classe c constituée de n attributs $a_1 \ldots a_n$ prenant les valeurs initiales $vi_1 \ldots vi_n$ dans les domaines $D_1 \ldots D_n$, de m méthodes nommées $meth_1 \ldots meth_m$ et de la méthode new permettant de créer un nouvel objet de cette classe est représentée en* ELAN *par :*

$$[a_1(vi_1),\ldots,a_n(vi_n),new]$$

Cette représentation correspond au ρ-terme :
$c \triangleq new \to S \to (a_1 \to S' \to (S \cdot a_1) \bullet S',$
$\ldots$
$a_n \to S' \to (S \cdot a_n) \bullet S',$
$meth_1 \to S' \to (S \cdot meth_1) \bullet S',$
$\ldots$
$meth_m \to S' \to (S \cdot meth_m) \bullet S'),$

$a_1 \rightarrow S \rightarrow S' \rightarrow vi_1,$
...
$a_n \rightarrow S \rightarrow S' \rightarrow vi_n,$
$meth_1 \rightarrow S \rightarrow S' \rightarrow body_1,$
...
$meth_m \rightarrow S \rightarrow S' \rightarrow body_m$

Les corps des méthodes sont définis par des règles de réécriture.

3.4.2 Une représentation à base d'opérateurs et de règles

On peut maintenant donner les idées générales du codage algébrique permettant la représentation des objets.

Chaque objet est composé d'attributs et de méthodes auxquelles il a accès. Les objets sont représentés à partir de termes ; quant aux méthodes, elles sont représentées à l'aide de règles de réécriture. Comment faire le lien entre les deux?

Nous donnons ici l'ensemble des composants d'une définition d'objet :

- Pour représenter les attributs et les valeurs qui leurs sont associées, on utilise un ensemble de paires *(attribut,valeur)*. Chaque attribut aura une constante le désignant et on pourra associer à cette constante un terme qui est du type de cet attribut.
- Pour représenter les méthodes, on utilise des règles de réécritures. A chaque méthode, on associe un opérateur du même nom lui correspondant. Cet opérateur a le même profil que la méthode à laquelle il correspond. A chaque corps de méthode correspond un membre droit de règle et des affectations et tests locaux qui lui sont propres.
- Chaque objet est composé d'une liste de constantes donnant accès aux méthodes. En effet, pour lier les règles associées aux méthodes aux objets, on déclare pour chaque méthode une constante qui permet, lorsqu'elle est présente dans la liste d'accès propre à chaque objet, de faire le lien avec l'opérateur associé à la méthode, et donc, à la règle correspondante.

On définit donc les objets par :

- un ensemble de déclarations de constantes dont celles associées à chaque attribut et celles associées à chaque méthode,
- un ensemble de déclarations d'opérateurs dont ceux associés à chaque définition de méthode,
- un ensemble de règles dont celles permettant de définir les corps des méthodes.

A ces déclarations et définitions s'ajoutent les règles et déclarations permettant de manipuler les objets : ajout d'attribut, modification de valeur d'attribut, accès à une méthode, etc.

On peut donc schématiser la représentation d'un objet comme on le montre en Figure 3.1.

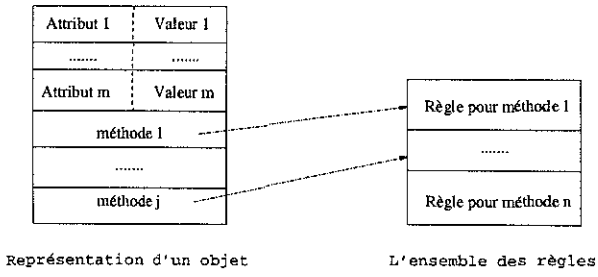


FIG. 3.1 – *Représentation des objets*

On peut ainsi voir que tout objet est représenté par un ensemble de paires *(nom d'attribut,valeur)* et par un ensemble de constantes désignant des méthodes qui sont elles implantées par des règles.

On donne la signature des opérateurs permettant de construire la représentation des objets dans la Figure 3.2.

$$
\begin{array}{lcll}
[\_] & : & Methods & \mapsto Object \\
\_,\_ & : & Methods \times Methods & \mapsto Methods \quad (AC) \\
\_ & : & Method & \mapsto Methods \\
\_(\_) & : & MName \times MBody & \mapsto Method \\
\_ & : & MName & \mapsto Method
\end{array}
$$

FIG. 3.2 – *Signature pour la représentation des objets*

Dans cette signature, `[_]` est l'opérateur de construction des objets. Chaque objet o a la forme $[LM]$ où LM est un ensemble composé de paires *(attribut,valeur)* et de références à des méthodes. Cet ensemble est construit par l'opérateur `_,_` qui est associatif et commutatif; chaque élément étant soit la paire *(attribut,valeur)* (opérateur `_(_)`), soit la référence à une méthode (opérateur `_` défini en dernier).

On donne maintenant dans la Figure 3.3 les profils des opérateurs manipulant ces objets.

$$
\begin{array}{lcll}
add(\_,\_) & : & Object \times Method & \mapsto Object \\
kill(\_,\_) & : & Object \times MName & \mapsto Object \\
access(\_,\_) & : & Object \times MName & \mapsto MBody \\
new(\_) & : & Object & \mapsto Object
\end{array}
$$

FIG. 3.3 – *Signature des opérateurs manipulant les objets*

Dans la Figure 3.3, on définit quatre opérateurs de base permettant de manipuler les objets : l'opérateur `add(_,_)` permet d'ajouter à l'ensemble des paires *(attribut,valeur)* et des références à des méthodes un nouvel élément (une nouvelle paire ou une nouvelle référence) ; l'opérateur `kill(_,_)` permet quant à lui d'ôter un élément donné en paramètre de l'ensemble des paires et références composant l'objet ; l'opérateur `access(_,_)` permet d'accéder au champ valeur d'une paire *(attribut,valeur)* pour un attribut donné d'un objet ; l'opérateur `new` permet quant à lui de créer un nouvel objet à partir de l'objet représentant la classe.

Parmi l'ensemble des méthodes que l'on peut définir, on en définit deux particulières : `Get(_,_)` permet d'accéder, pour un attribut donné, au champ valeur de la paire *(attribut,valeur)* correspondante ; `Set(_,_)` permet de modifier, pour un attribut donné, le champ valeur de la paire *(attribut,valeur)* correspondante, avec une nouvelle valeur donnée en paramètre. Les profils de ces deux opérateurs sont donnés en Figure 3.4.

$$
\begin{array}{lcll}
Get(\_,\_) & : & Object \times MName & \mapsto MBody \\
Set(\_,\_,\_) & : & Object \times MName \times MBody & \mapsto Object
\end{array}
$$

FIG. 3.4 – *Signature des opérateurs Get et Set*

Exemple 3.10 *On considère un objet o d'une classe C avec deux attributs `X` et `Y` de type entier ayant les valeurs respectives `1` et `2` et pour lequel les méthodes `GetX`, `SetX` et `GetY` sont accessibles. Quatre méthodes sont par contre définies pour la classe C et elles sont définies dans le système de règles : les trois méthodes `GetX`, `SetX` et `GetY` ainsi que la méthode `SetY` qui est par contre inaccessible pour l'objet présenté dans cet exemple.*

On a alors la structure de représentation de l'objet o qui est présentée en Figure 3.5.

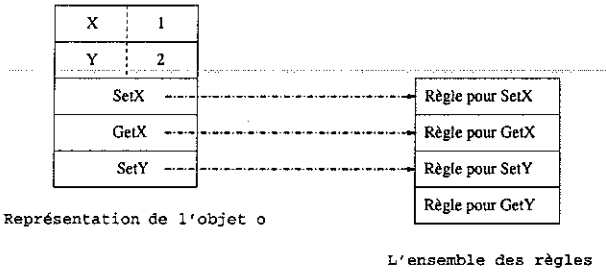


FIG. 3.5 – *Représentation des objets*

3.4.3 $\mathcal{R}$: un système de règles pour manipuler les objets

Nous sommes maintenant en mesure de définir un système de règles que nous utilisons pour manipuler les objets. On appelle $\mathcal{R}$ ce système défini par des règles de réécriture et on le donne dans la Figure 3.6.

Ce système de règles est basé sur la représentation des objets que nous avons présentée dans la section précédente et s'inspire de la définition du ρ-calcul objet.

Détaillons le système de règles $\mathcal{R}$:

- La première règle, **Ajout d'un composant**, permet d'ajouter un composant *me* à la structure $[LM]$ représentant un objet. Cette composante que l'on ajoute peut être soit un couple donnant un attribut et sa valeur *(nom d'attribut,valeur)*, soit le nom d'une méthode. Par contre, dans la pratique, cette règle est conditionnée. En effet, tel que *add* est défini, c'est-à-dire sans condition, nous permettrions des ajouts de composants déjà présents ou alors des ajouts de références à des méthodes ou de couples *(nom d'attribut,valeur)* qui ne seraient pas forcément possibles pour l'objet désigné. Les membres gauche et droit de la règle **Ajout d'un composant** ne sont donc pas modifiés, mais l'application de la règle sera conditionnée dans son implémentation.
- Les règles **Suppression d'un composant** permettent de supprimer des composants de la structure d'un objet. On peut ainsi supprimer un attribut M avec sa valeur B dans la première règle ou alors un accès M à une méthode dans la seconde règle.
- L'accès à la valeur B d'un attribut M est défini dans la quatrième règle **Accès à la valeur d'un attribut**.
- La cinquième règle **Accès à une valeur par** `Get` définit la fonction $Get(\_,\_)$ pour un objet o et un attribut M. L'accès à la valeur associée à l'attribut est définit par un appel à la quatrième règle définissant *access*.
- La modification de la valeur d'un attribut est définie par la sixième règle appelée **Modification de valeur par** `Set`. Pour modifier la valeur d'un attribut M avec une nouvelle valeur B pour un objet o, on fait appel aux constructeurs *kill* et *add*. Après avoir supprimé la référence à l'attribut M par *kill*, on ajoute de nouveau la référence au même attribut, mais avec la nouvelle valeur B.
- La septième règle, **Création d'un nouvel objet**, permet de créer un nouvel objet d'une classe représentée par l'objet o. Ce nouvel objet a la structure que l'on définit dans notre formalisme avec les n attributs a_i, leurs valeurs initiales vi_i et les m accès aux méthodes m_j.
- Les deux règles définissant les opérateurs **Geta** et **Seta** permettent de faire le lien entre les définitions des opérateurs génériques `Set` et `Get` et leur utilisation par des opérateurs propres à chaque attribut. On considère par la suite que les notations Get_a et `Geta` sont équivalentes.
- Les trois règles, **Appel de la méthode** `Geta`, **Appel de la méthode** `Seta` et **Appel de la méthode** `new` permettent de définir les notations usuelles d'appel de méthode pour les objets. Elles consistent à vérifier tout d'abord par filtrage que la méthode appelée fait bien partie des méthodes autorisées pour l'objet et ensuite, puis font appel par le mécanisme d'évaluation de la règle aux fonctions définies précédemment. Les notations *Geta* et *Seta* font référence à tout attribut appelé a composant l'objet.

Ajout d'un composant	$add([LM],me) \rightarrow_{\mathcal{R}} [LM,me]$
Suppression d'un composant-1	$kill([M(B),LM],M) \rightarrow_{\mathcal{R}} [LM]$
Suppression d'un composant-2	$kill([M,LM],M) \rightarrow_{\mathcal{R}} [LM]$
Accès à la valeur d'un attribut	$access([M(B),LM],M) \rightarrow_{\mathcal{R}} B$
Accès à une valeur par `Get`	$Get(o,M) \rightarrow_{\mathcal{R}} access(o,M)$
Modification de valeur par `Set`	$Set(o,M,B) \rightarrow_{\mathcal{R}} add(kill(o,M),M(B))$
Création d'un nouvel objet	$new(o) \rightarrow_{\mathcal{R}} [a_1(vi_1),\ldots,a_n(vi_n),m_1,\ldots,m_m]$
L'opérateur `Geta`	$Get_a(o) \rightarrow_{\mathcal{R}} Get(o,a)$
L'opérateur `Seta`	$Set_a(o,v) \rightarrow_{\mathcal{R}} Set(o,a,v)$
Appel de la méthode `Geta`	$[LM,Get_a].Get_a \rightarrow_{\mathcal{R}} Get_a([LM,Get_a])$
Appel de la méthode `Seta`	$[LM,Set_a].Set_a(v) \rightarrow_{\mathcal{R}} Set_a([LM,Set_a],v)$
Appel de la méthode `new`	$[LM,new].new \rightarrow_{\mathcal{R}} new([LM,new])$
Appel d'une méthode `m`	$[LM,m].m(p_1,\ldots,p_m) \rightarrow_{\mathcal{R}} m([LM,m],p_1,\ldots,p_m)$

FIG. 3.6 – *Le système $\mathcal{R}$*

- La dernière règle, **Appel d'une méthode `m`**, permet quant à elle l'appel d'une méthode définie par l'utilisateur. Cette méthode peut être paramétrée (ce sont les p_i arguments) et on voit explicitement le lien existant entre la vérification de la présence de la constante m désignant cette méthode dans l'objet et l'opérateur associé à chaque méthode $m(o,p_1,\ldots,p_n)$. Les arguments de cet opérateur sont tout d'abord l'objet auquel s'applique l'appel de méthode, puis viennent les paramètres p_i passés lors de l'appel.

Afin d'implémenter ce système de règles $\mathcal{R}$, on va tout d'abord montrer que ce système est nœthérien, c'est-à-dire qu'il termine. On peut ainsi définir la propriété suivante pour l'ensemble des opérateurs et des règles composant la théorie de réécriture $\mathcal{R}$ telle que définie ci-dessus à partir des constructeurs objets, des définitions d'opérateurs et des règles associées concernant les opérateurs *access*, *add*, *kill*, *Get*, *Set* et *new* ainsi que ceux concernant tout attribut a et les méthodes *Geta* et *Seta*:

Proposition 3.1 *La théorie de réécriture $\mathcal{R}$ constitue un système confluent et nœthérien.*

Preuve : *Nous considérons un ordre RPO AC-compatible basé sur la précédence suivante : .Geta $\succ$ Geta $\succ$ a et .Seta $\succ$ Seta $\succ$ a ; Geta $\succ$ Get $\succ$ access $\succ$ [] et Seta $\succ$ Set $\succ$ add $\succ$ kill $\succ$ [] où [] est le constructeur d'objet. On définit aussi une précédence sur les constructeurs d'objet new $\succ$ [] $\succ$, $\succ$ (). Dans la suite de la preuve, l'opérateur associatif et commutatif "," est représenté sous forme aplatie et on le considère comme ayant un statut multi-ensemble. On utilise comme ordre de réduction l'ordre AC-compatible basé sur l'ordre RPO défini par D. Kapur, G. Sivakumar et H. Zhang dans [KSZ90]. On note $t \succ_{RPO_{AC}} s$ pour désigner que t est plus petit que s pour cet ordre.*

On considère chacune des règles présentées précédemment et on montre que chaque membre gauche de règle est supérieur au membre droit correspondant en appliquant la définition de l'ordre RPO AC-compatible.

- *Pour la règle :*

$$add([LM],me) \rightarrow [LM,me]$$

on a bien $add([LM],me) \succ_{RPO_{AC}} [LM,me]$ *car* $add \succ []$ *et* $add([LM],me) \succ_{RPO_{AC}} LM$ *et* $add([LM],me) \succ_{RPO_{AC}} me$ *par la propriété de sous-terme de l'ordre.*

- *Pour les règles :*

$$kill([M(B),LM],M) \rightarrow [LM]$$

$$kill([M,LM],M) \rightarrow [LM]$$

on a aussi $kill([M(B),LM],M) \succ_{RPO_{AC}} [LM]$ *car* $kill \succ []$ *puis par propriété de sous-terme on obtient que* $kill([M(B),LM],M) \succ_{RPO_{AC}} LM$. *Le même raisonnement est utilisée pour montrer que* $kill([M,LM],M) \succ_{RPO_{AC}} [LM]$.

- *Pour la règle :*

$$access([M(B),LM],M) \rightarrow B$$

il suffit d'utiliser la propriété de sous-terme et on obtient bien $access([M(B),LM],M) \succ_{RPO_{AC}} B$.

- *Pour la règle :*

$$Get(o,M) \rightarrow access(o,M)$$

comme $Get \succ access$, *il suffit là aussi de montrer que* $Get(o,M) \succ_{RPO_{AC}} o$ *et que* $Get(o,M) \succ_{RPO_{AC}} M$ *par propriété de sous-terme.*

- *Pour la règle :*

$$Set(o,M,B) \rightarrow add(kill(o,M),M(B))$$

comme $Set \succ add$, *on doit tout d'abord montrer que* $Set(o,M,B) \succ_{RPO_{AC}} kill(o,M)$ *qui se vérifie car* $Set \succ kill$ *et par propriété de sous-terme de l'ordre, on a* $Set(o,M,B) \succ_{RPO_{AC}} o$ *et* $Set(o,M,B) \succ_{RPO_{AC}} M$. *On doit ensuite montrer que* $Set(o,M,B) \succ_{RPO_{AC}} M(B)$, *ce qui est vrai car* $Set \succ ()$ *et par propriété de sous-terme de l'ordre.*

- *Pour les règles :*

$$[LM,Get_a].Get_a \rightarrow Get_a([LM,Get_a])$$

$$[LM,Set_a].Set_a(v) \rightarrow Set_a([LM,Set_a],v)$$

on considère les précédences $.Geta \succ Geta$ *et* $.Seta \succ Seta$, *et par propriété de sous-terme de l'ordre, on obtient les comparaisons recherchées.*

- *Pour les règles :*

$$Get_a(o) \rightarrow Get(o,a)$$

$$Set_a(o,v) \rightarrow Set(o,a,v)$$

on utilise la précédence $Geta \succ Get$ *et* $Seta \succ Set$. *On montre ainsi que* $Geta(o) \succ_{RPO_{AC}} Get(o,a)$ *par la propriété de sous-terme appliquée à* o *et par le fait que* $Geta \succ a$. *De même, on montre que* $Seta(o,B) \succ_{RPO_{AC}} Set(o,a,B)$ *par la propriété de sous-terme appliquée à* o *et* B *et par le fait que* $Seta \succ a$.

- *Pour la règle :*

$$[LM,new].new \rightarrow new([LM,new])$$

on utilise la précédence .new $\succ$ new et la propriété du sous-terme.

- *Pour la règle :*

$$new(o) \rightarrow [a_1(vi_1),\ldots,a_n(vi_n),m_1,\ldots,m_m]$$

on enrichit la précédence donnée en début de preuve par () $\succ a_i$ pour tout attribut a_i déclaré ; on donne aussi , $\succ m_j$ pour toute méthode définie et () $\succ vi_i$ pour toute valeur initiale vi donnée. Cette dernière précédence concorde avec le fait que l'on a la déclaration

```
@ : (D_a) MBody;
```

pour chaque domaine D_a d'attribut a_i.
Comme new $\succ$ [], on doit montrer que new(o) $\succ_{RPO_{AC}}$,($a_1(vi_1)$,...,$a_n(vi_n)$,m_1,....,m_n) comme l'opérateur "," est vu sous forme aplatie. On a bien new $\succ m_j$, new $\succ$ () $\succ a_i$ et new $\succ vi_i$; ainsi, on a new(o) $\succ_{RPO_{AC}}$ [$a_1(vi_1)$,...,$a_n(vi_n)$,m_1,....,m_n].

- *Finalement, pour la règle :*

$$[LM,m].m(p_1,\ldots,p_m) \rightarrow_{\mathcal{R}} m([LM,m],p_1,\ldots,p_m)$$

on doit enrichir la précédence par .m $\succ$ m pour chaque méthode m définie. Comme .m $\succ$ m, on montre par la propriété de sous-terme de l'ordre que $[LM,m].m(p_1,\ldots,p_m) \succ_{RPO_{AC}} [LM,m]$ et que $[LM,m].m(p_1,\ldots,p_m) \succ_{RPO_{AC}} p_i$ pour chaque $p_{i=[1,\ldots,m]}$.

On peut donc montrer, avec la précédence $\succ$ définie tout au long de cette preuve, que pour tout membre gauche t et tout membre droit s de règle, on a bien $t \succ_{RPO_{AC}} s$. □

Dans le cas où l'utilisateur définit lui-même des méthodes, on ne peut pas prétendre que toute définition de méthode engendrera un ensemble de règles confluentes et terminantes.

Note : On supposera par la suite chaque fois qu'il le sera nécessaire, que l'ensemble $\mathcal{R}$ considéré est confluent et terminant.

3.4.4 Des règles de réécriture pour les méthodes

Afin de comprendre comment les corps des méthodes définies par l'utilisateurs sont déduits à partir des déclarations dans les modules objet, nous définissons un mécanisme de traduction permettant, à partir d'une définition de méthode dans un module objet, d'associer une règle de réécriture correspondante.

Dans cette section, seules sont concernées les méthodes définies par l'utilisateur dans les OModules ; en effet, les méthodes particulières *Get*, *Set* et *new* sont construites différemment comme cela a été présenté dans la section précédente et cela de façon générique.

Nous appelons $build - rule(\_)$ la fonction qui prend en argument la définition d'une méthode dans le formalisme décrit dans la partie 3.2.3 et retourne la règle de réécriture associée dans la syntaxe ELAN.

Nous allons décrire le comportement de $build - rule(\_)$ à partir de la syntaxe générale des méthodes que l'on rappelle ici :

<*méthode*>::= **method** <*nom de méthode*>[(<*arguments*>)] **for** <*type du résultat*>
[<*déclarations de variable*>] < <*corps de méthode*>>

Compte tenu des options possibles sur les déclarations d'arguments et de variables locales pour une méthode, la fonction $build - rule(\_)$ est définie de la façon suivante :

```
build − rule( method name (args) for t vars corps)  =
                    rules for t
                     S : class_name;
                     var − decl(vars,args,corps)
                     [] name(S,get − args(args)) => build − rhs(corps,list_vars) end
                    end
build − rule( method name (args) for t corps)       =
                    rules for t
                     S : class_name;
                     var − decl(args,corps)
                     [] name(S,get − args(args)) => build − rhs(corps,list_vars) end
                    end
build − rule( method name for t vars corps)         =
                    rules for t
                     S : class_name;
                     var − decl(vars,corps)
                     [] name(S) => build − rhs(corps,list_vars) end
                    end
build − rule( method name for t corps)              =
                    rules for t
                     S : class_name;
                     var − decl(corps)
                     [] name(S) => build − rhs(corps,list_vars) end
                    end
```

Dans cette définition de $build - rule$, on donne les quatre cas de figure qui peuvent apparaître et qui dépendent ou non de la présence d'arguments *args* et de déclarations de variables *vars*. Les règles produites suivent la syntaxe ELAN et on y retrouve :

- le type t de la règle qui correspond au type du résultat de la méthode,
- les déclarations de variables locales à la règle qui sont obtenues par une fonction $var - decl$. Celle-ci déduit à partir du corps de la méthode *corps* de nouvelles variables qui peuvent être engendrées du simple fait de la traduction du corps de méthode en membre droit de règle de réécriture. La fonction $var - decl$ ajoute à ces variables les variables locales à la méthode lorsqu'elles existent et les arguments de la méthode lorsqu'ils existent. On donne aussi une première variable locale toujours utilisée, c'est la variable S désignant l'objet auquel la méthode est appliquée. Cette variable est donc déclarée comme étant du type *class_name*, c'est-à-dire du nom de la classe à laquelle les méthodes se rapportent,
- une définition de règle non nommée dont le membre gauche est la fonction correspondant à la méthode avec ses paramètres : S représentant l'objet de la classe auquel la méthode est appliquée et $get - args(args)$ permettant d'obtenir les autres paramètres à partir de la déclaration des arguments de la méthode. Le membre droit de la règle est quant à lui obtenu par la fonction $build - rhs(corps,list\_vars)$ qui est détaillée ci-après.

La définition de $build - rhs(\_,\_)$ dépend tout d'abord de la forme du corps de la méthode dont on rappelle le format ci-après et qui est donné comme premier argument. Le second argument est une liste de variables qui se construit au fur et à mesure de la génération du membre droit. Chaque nouvelle variable ajoutée à la liste correspond à une variable locale instanciée par une affectation locale de la règle. Cette liste de variables va notamment nous servir à mémoriser les variables locales utilisées lors de la modification des objets ; en effet, on génère alors un nouveau terme représentant l'objet modifié et à ce terme est associée une variable locale qui désignera par la suite l'objet initial dans le reste de la règle. Lors de son initialisation, la liste de variables ne comprend que la variable S utilisée pour représenter

l'objet passé en paramètre de toute méthode. S est donc déclarée comme une variable locale à la règle de réécriture. En association avec cette liste de variables, on définit deux fonctions :

- une fonction $get-var-name(\_,\_)$ qui prend une variable et donne le nom de la dernière variable locale lui correspondant à partir de l'état courant de la liste qui est donnée en second paramètre.
- une fonction $update-var-list(\_,\_)$ qui prend une nouvelle variable et une liste de variables locales et rend comme résultat la liste mise à jour.

On rappelle ici la forme d'un corps de méthode :

<corps de méthode>::= *<appel de méthode>* |
<test> ; *<appel de méthode>* |
<affectation locale> ; *<corps de méthode>* |
<corps de méthode> ; *<corps de méthode>*

Ainsi, on peut définir la fonction $build-rhs(c,l)$ où c est le corps de la méthode que l'on cherche à transformer en membre droit de règle et l, la liste courante de variables locales :

- si c est un appel de méthode de la forme $o.m'(p'_1,\ldots,p'_n)$ alors :

$$build-rhs(o.m'(p'_1,\ldots,p'_n),l) \;=\; m'(get-var-name(o,l),p'_1,\ldots,p'_n)$$

- si c est une méthode, impliquant un opérateur donné f, de la forme$f(p'_1,\ldots,p'_n)$, alors :

$$build-rhs(f(p'_1,\ldots,p'_n),l) \;=\; f(p'_1,\ldots,p'_n)$$

- si c est un test suivi d'un appel de méthode de la forme $if\ t\ ;\ o.m'(p'_1,\ldots,p'_n)$, alors :

$$build-rhs(if\ t\ ;\ o.m'(p'_1,\ldots,p'_n),l) \;=\; m'(get-var-name(o,l),p'_1,\ldots,p'_n)\ \textbf{if}\ t$$

- si c est une affectation locale suivie d'un corps de méthode de la forme :

$$variable\ :=\ f(p'_1,\ldots,p'_n)\ ;\ corps$$

alors on a :

$$build-rhs(variable\ :=\ f(p'_1,\ldots,p'_n)\ ;\ corps,l) \;=$$

$$v\ \textbf{where}\ variable\ :=\ ()\ f(p'_1,\ldots,p'_n)\ \textbf{where}\ v\ :=\ ()\ build-rhs(corps,l)$$

- si c est la composition de deux corps de méthodes $c_1\ ;\ c_2$, on a alors :

$$build-rhs(c_1\ ;\ c_2,l) \;=$$

$$v_2\ \textbf{where}\ v_1\ :=\ ()\ build-rhs(c_1,l)\ \textbf{where}\ v_2\ :=\ ()\ build-rhs(c_2,update-var-list(v_1,l))$$

Nous avons ainsi présenté dans cette section la façon dont les règles de réécriture associées aux méthodes définies par l'utilisateur sont construites.

3.4.5 Des programmes ELAN pour la théorie algébrique des objets

Comme le système $\mathcal{R}$ considéré est un système terminant et confluent, nous pouvons l'implémenter au moyen de règles de réécriture non nommées en ELAN. A partir des codages des objets et des méthodes, nous définissons maintenant la théorie algébrique des objets en ELAN.

Afin de définir un codage algébrique des objets, nous définissons une théorie de réécriture associée à $\mathcal{R}$ composée d'une signature Σ et d'un ensemble de règles de réécriture R. La signature Σ est donnée en ELAN par :

```
operators global
  [@]    : (Methods) Object;
  @,@    : (Methods Methods) Methods      (AC);
  @      : (Method) Methods;
  @(@)   : (MName MBody) Method;
  @      : (MName) Method;

  add(@,@)     : (Object Method) Object;
  kill(@,@)    : (Object MName) Object;
  access(@,@)  : (Object MName) MBody;
  Get(@,@)     : (Object MName) MBody;
  Set(@,@,@)   : (Object MName MBody) Object;
  new(@)       : (Object) Object;
end
```

On peut séparer ces opérateurs en deux parties : les premières déclarations permettent de définir les constructeurs des objets ainsi que des attributs et méthodes les composant ; les suivantes sont des déclarations d'opérateurs manipulant les objets.

Le premier opérateur construit à partir d'un terme de type `Methods` un terme de type `Object`. Un terme de type `Methods` est un multi-ensemble de termes de type `Method` avec le constructeur "`,`" déclaré comme étant associatif et commutatif. Un objet est ainsi composé de plusieurs méthodes ; l'ordre dans lequel les méthodes sont déclarées n'est pas important. Comme nous l'avons vu précédemment dans la définition du formalisme des objets, nous pouvons distinguer :

- les attributs qui sont représentés par le nom de l'attribut et la valeur associée :

```
@(@) : (MName MBody) Method;
```

- des méthodes, qui ne sont représentées dans les objets que par leur nom :

```
@ : (MName) Method;
```

  Le corps de chaque méthode est quant à lui défini par des règles de réécriture, soit selon la méthode présentée dans la partie précédente pour les méthodes utilisateurs, soit de façon plus automatique comme présenté dans la suite de cette section.

Ainsi, pour chaque attribut a prenant ses valeurs dans un domaine D_a, les déclarations suivantes doivent être effectuées :

```
a    : MName;
@    : (D_a) MBody;
```

afin de déclarer a comme nom d'attribut et D_a comme étant un domaine possible de valeurs.

Pour une méthode donnée, seul son nom m doit être déclaré comme nom de méthode par :

```
m    : MName;
```

C'est ce type de déclaration qui est effectué pour chaque méthode Get_a et Set_a.

Le premier opérateur manipulant les objets est `add(_,_)` qui prend comme paramètres un terme de type `Object` et un autre de type `Method` et dont le but est d'étendre l'objet initial avec la nouvelle

méthode passée en paramètre. Dans la règle de R associée à cet opérateur, la méthode me est ajoutée à l'objet initial [LM] :

```
rules for Object
 LM  : Methods;
 me  : Method;
global
[] add([LM],me)  => [LM , me] end
            if not(in(me,[LM]))
            if allowed(me,class\_name_i)
end
```

On peut remarquer dans l'implémentation de l'opérateur add(_,_) la présence de deux conditions, comme nous l'avions préalablement mentionné dans la présentation du système $\mathcal{R}$ donné en Figure 3.6. En effet, on ne peut pas permettre n'importe quel type d'ajout à un objet, on doit le conditionner avec :

- Un test sur l'absence de la méthode *me* passée en paramètre de l'objet $[LM]$ considéré : on ne souhaite pas avoir deux fois les mêmes références à une même méthode ou avoir une double définition d'un même attribut. Ce test est réalisé par $if\ not(in(me,[LM]))$ où l'opérateur in(_,_) permet de tester si le premier argument *me* est présent dans l'objet $[LM]$. On conditionne la règle avec le test de la négation de ce calcul.
- Un test sur la possibilité d'ajouter *me* à l'objet $[LM]$. On désigne chaque objet représentant une classe par une constante appelée $class\_name_i$. Cet objet est connu pour chaque classe et on peut donc connaître pour chaque classe, quelles sont les méthodes accessibles et quels sont les attributs définis. On teste à ce moment si l'élément *me* que l'on cherche à ajouter à l'objet $[LM]$ est bien un élément composant la classe $class\_name_i$ dont l'objet $[LM]$ est un issu.

L'opérateur kill(_,_) prend quant à lui en paramètres une méthode M et l'élimine de l'objet passé lui aussi en paramètre. *kill* peut donc éliminer soit un attribut M(B) soit une référence à une méthode M ; d'où les deux règles associées à cet opérateur :

```
rules for Object
 M  : MName;
 LM : Methods;
 B  : MBody;
global
[] kill([M(B),LM],M) => [LM] end
[] kill([M,LM],M)    => [LM] end
end
```

L'opérateur access(_,_) prend en paramètres un nom d'attribut et un objet et rend la valeur B associée à l'attribut dans l'objet :

```
rules for MBody
 M  : MName;
 B  : MBody;
 LM : Methods;
global
[] access([M(B),LM],M) => B end
end
```

Les opérateurs Get(_,_) et Set(_,_,_) sont définis de façon générique pour les attributs et permettent, respectivement, d'accéder à la valeur de l'attribut et de modifier cette valeur. Ces opérateurs prennent comme arguments l'objet concerné, le nom de l'attribut et aussi, dans le cas de l'opérateur Set, la nouvelle valeur devant être associée.

S.C.D. - U.H.P. NANCY 1
BIBLIOTHÈQUE DES SCIENCES
Rue du Jardin Botanique
54600 VILLERS-LES-NANCY


```
rules for MBody
 o : Object;
 M : MName;
global
[] Get(o,M)   => access(o,M) end
end

rules for Object
 o : Object;
 B : MBody;
 M : MName;
global
[] Set(o,M,B) => add(kill(o,M),M(B)) end
end
```

Dans ces définitions d'opérateurs d'accès et de modification d'attributs, nous n'avons à aucun moment mentionné la possibilité pour un objet d'avoir accès à telle méthode ou telle autre. Cette possibilité d'exécuter une méthode est réalisée par l'appel de méthode `_._` des langages objets. Nous l'implémentons de la même manière par la définition d'opérateurs `_.Geta` et `_.Seta(_)` pour chaque attribut a défini dans la classe. Le but de ces opérateurs est de vérifier que la méthode appelée sur l'objet est bien disponible pour cet objet au moment de l'appel, c'est-à-dire que l'on vérifie que la méthode est bien dans la liste des méthodes accessibles par l'objet.

Ainsi, pour chaque attribut a défini dans la classe C, les déclarations suivantes sont effectuées :

- deux constantes `Geta` et `Seta` sont déclarées. Elles correspondent aux noms des deux méthodes associées à l'attribut et qui sont disponibles pour un objet de cette classe.
- deux opérateurs `_.Geta` et `_.Seta(_)` sont aussi déclarés : ils permettent l'appel de méthode et vérifient si l'accès à la méthode est possible au moment de l'appel. `_.Geta` est un opérateur prenant un paramètre de type `C` et rend un résultat de type `D_a`. `_.Seta(_)` prend quant à lui deux arguments, l'un de type `C` et l'autre de type `D_a`. Le résultat est de type `C`.
  Les règles associées à ces deux opérateurs sont définies par :

```
rules for D_a
 LM : Methods;
global
[] [Geta , LM].Geta   => Geta([Geta , LM]) end
end

rules for C
 B : D_a;
 LM : Methods;
global
[] [Seta , LM].Seta(B) => Seta([Seta , LM],B) end
end
```

- deux opérateurs `Geta(_)` et `Seta(_,_)` sont aussi déclarés. `Geta(_)` prend un objet de la classe et rend la valeur associée à l'attribut ; son résultat est donc de type `D_a` qui est la sorte de l'attribut a. `Seta(_,_)` prend un objet de la classe et un élément de type `D_a`. Le résultat est l'objet mis à jour.
  Les règles associées à ces opérateurs sont aussi définies ; elles sont de la forme :

```
rules for D_a
 o : C;
 a : MName;
global
[] Geta(o)   => Get(o,a) end
end

rules for C
 o : C;
 B : D_a;
 a : MName;
global
[] Seta(o,B) => Set(o,a,B) end
end
```

Parmi l'ensemble des méthodes, on peut tout d'abord distinguer l'opérateur `new(_)` qui prend un objet représentant une classe et permet de générer un nouvel objet de cette classe. Les valeurs initiales de chaque attribut sont celles spécifiées lors de la déclaration du module objet dans la partie 3.2.2. Les méthodes de la classe sont toutes accessibles par l'objet après son initialisation.

```
rules for Object
  o  : Object;
global
[] new(o) => [a_1(vi_1),...,a_n(vi_n),m_1,....,m_n] end
end
```

De même, on présente ici les déclarations correspondant aux méthodes définies par l'utilisateur. Ainsi, pour chaque méthode m définie dans une classe C, où la méthode m prend n paramètres p_i de types D_{p_i} et rend un résultat de type D_m, on a les déclarations suivantes :

- une constante m est définie permettant de construire la liste de méthodes accessibles pour chaque objet de la classe.
- un opérateur `_.m(_,...,_)` est défini. Son premier paramètre est de type `C`, les autres sont de type `D_p_i`. Le résultat est de type `D_m`. La règle associée à cet opérateur est une règle permettant par appel de méthode de vérifier que celui-ci est autorisé ; la règle associée est de la forme :

```
rules for D_m
 LM  : Methods;
 p_1 : D_p_1;
 ...
 p_n : D_p_n;
global
[] [m , LM].m(p_1,...,p_n)   => m([m , LM],p_1,...,p_n) end
end
```

- l'opérateur correspondant à la fonction associée à toute méthode. Il est déclaré par `m(_,_,...,_)` avec comme type de premier paramètre `C`, les autres paramètres étant les types `D_p_i` des paramètres de la méthode ; le résultat est quant à lui de type `D_m`. La règle associée à cet opérateur est définie grâce à la transformation $build - rule(\_)$ définie précédemment.

Exemple 3.11 *Prenons l'objet P de la classe Point précédemment définie :*

$$\begin{aligned} P \triangleq\ & X \to S \to 5, \\ & Y \to S \to 7, \\ & GetX \to S \to S \cdot X, \\ & SetX \to S \to N \to S \cdot X\ :=\ S' \to N, \\ & GetY \to S \to S \cdot Y, \\ & SetY \to S \to N \to S \cdot Y\ :=\ S' \to N, \\ & ReinitX \to S \to S \cdot SetX(0) \end{aligned}$$

L'objet P est ainsi codé en ELAN *par la structure suivante :*

$$[X(5)\ ,\ Y(7)\ ,\ GetX\ ,\ SetX\ ,\ GetY\ ,\ SetY\ ,\ ReinitX]$$

A cet objet sont liées les déclarations suivantes :

```
X       : MName;
Y       : MName;
GetX    : MName;
SetX    : MName;
GetY    : MName;
SetY    : MName;
ReinitX : MName;

@       : (int) MBody;

@.GetX  : (Point) int;
GetX(@) : (Point) int;
@.GetY  : (Point) int;
GetY(@) : (Point) int;

@.SetX(@) : (Point int) Point;
SetX(@,@) : (Point int) Point;
@.SetY(@) : (Point int) Point;
SetY(@,@) : (Point int) Point;

@.ReinitX : (Point) Point;
ReinitX(@): (Point) Point;
```

ainsi que les règles de réécriture suivantes :

```
rules for int
 LM   : Methods;
 V    : int;
global

[] [GetX,LM].GetX   => GetX([GetX,LM]) end
[] GetX([X(V),LM])  => V end
[] [GetY,LM].GetY   => GetY([GetY,LM]) end
[] GetY([Y(V),LM])  => V end
end
```

```
rules for Point
 self : Point;
 LM   : Methods;
 V,V1 : int;
global

[] [SetX,LM].SetX(V)  => SetX([SetX,LM],V) end
[] SetX([X(V),LM],V1) => [X(V1),LM] end
[] [SetY,LM].SetY(V)  => SetY([SetY,LM],V) end
[] SetY([X(V),LM],V1) => [Y(V1),LM] end

[] [ReinitX,LM].ReinitX  => ReinitX([ReinitX,LM]) end
[] ReinitX(self)         => SetX(self,0) end
end
```

Dans cet exemple, on peut constater que les règles correspondant aux opérateurs implémentant des fonctions relatives à chaque méthode, en l'occurence les règles associées aux opérateurs `GetX(_)`, `SetX(_,_)`, `GetY(_)` et `SetY(_,_)` sont simplifiées. On peut retrouver les définitions données dans l'exemple en utilisant les définitions de `access`, `add` et `kill` qui ont été données précédemment.

On a donc présenté dans cette section les règles de réécriture ELAN permettant de coder la théorie algébrique des objets définie par le système $\mathcal{R}$ étendu avec les règles obtenues à partir des méthodes définies par l'utilisateur.

3.4.6 Traduction des objets ELAN en ρ-termes

On définit maintenant le lien existant entre le codage des objets en ELAN et le calcul de réécriture. Cela nous permet de lier l'implémentation des objets au ρ-calcul, et nous conduit ainsi au résultat suivant : à toute évaluation d'un terme suivant un système de réécriture respectant la théorie algébrique des objets correspond une réduction du ρ-terme associé à ce terme.

Dans le cas où $\mathcal{R}$ est confluent et terminant, tout terme t de sorte `Object` a une forme normale notée $FN(t)$. De plus, cette forme normale est un multi-ensemble qui peut contenir, pour chaque attribut m un triplet $[m(b)\ ,\ Getm\ ,\ Setm]$ et pour chaque méthode qui n'est pas un attribut seulement le nom de cette méthode. Une forme générale de terme t_0 de sorte `Object` est donc décrite par :

$$t_0 = [m(b)\ ,\ Getm\ ,\ Setm\ ,\ LM]$$

où LM est un multi-ensemble de termes de sorte `Method` de l'une des formes suivantes : m', $m'(b')$, $Getm'$, $Setm'$.

Considérons la transformation τ qui relie un terme t_0 de la forme

$$t_0 = [m(b)\ ,\ Getm\ ,\ Setm\ ,\ LM]$$

dans la théorie de réécriture $\mathcal{R}$ au ρ-terme t_0' :

$$\begin{aligned}\tau(t_0) = t_0' = (&m \to S \to b,\\ &Getm \to S \to S \cdot m,\\ &Setm \to S \to N \to S \cdot m := S' \to N,\\ &L)\end{aligned}$$

où L est un multi-ensemble de ρ-termes de la forme

$$m' \to S \to body$$

Détaillons la transformation τ en donnant les ρ-termes associés pour toute définition de méthode m', avec le paramètre objet S et n paramètres formels dénotés p_i.

- Si la règle définissant m' est de la forme :

$$[]\ m'(S,p_1,\ldots,p_n) \ \rightarrow\ f(S,p_1,\ldots,p_n)$$

alors, le ρ-terme associé est :

$$m' \rightarrow S \rightarrow p_1 \rightarrow \ldots \rightarrow p_n \rightarrow f(S,p_1,\ldots,p_n)$$

- Si la règle définissant m' est de la forme :

$$[]\ m'(S,p_1,\ldots,p_n) \ \rightarrow\ f(S,p_1,\ldots,p_n,o) \ \textbf{where}\ o := ()b$$

le ρ-terme associé est dans ce cas de la forme :

$$m' \rightarrow S \rightarrow p_1 \rightarrow \ldots \rightarrow p_n \rightarrow (o \rightarrow f(S,p_1,\ldots,p_n,o)) \bullet b$$

- A une règle définissant m' et qui a la forme suivante :

$$[]\ m'(S,p_1,\ldots,p_n) \ \rightarrow\ b \ \textbf{if}\ \ c$$

on associe le ρ-terme suivant :

$$m' \rightarrow S \rightarrow p_1 \rightarrow \ldots \rightarrow p_n \rightarrow (True \rightarrow b) \bullet c$$

On donne alors la proposition suivante qui permet de relier le calcul sur les objets en logique de réécriture et l'évaluation dans le ρ-calcul des ρ-termes correspondants obtenus par la transformation τ :

Proposition 3.2 *Soit $\equiv$ la fermeture réflexive, transitive et symétrique de la relation $\mapsto$ du ρ-calcul. Pour tout termes t, t_0 de sorte `Object` de R, on a :*

- $\tau(FN(add(t_0,m_1))) \equiv \tau(FN(t_0)),\tau(m_1)$
- $\tau(FN(access(t_0,m))) \equiv \tau(FN(t_0)) \cdot m$
- $\tau(FN(kill(t_0,m))) \equiv kill_m(\tau(FN(t_0)))$
- $\tau(FN(Getm(t_0))) \equiv \tau(FN(t_0)) \cdot Getm$
- $\tau(FN(Setm(t_0,V))) \equiv \tau(FN(t_0)) \cdot Setm(V)$
- $\tau(FN(new(t))) \equiv \tau(FN(t)) \cdot new$

Preuve : *Considérons un objet t_0 dont la forme normale est :*

$$FN(t_0) = [m(b)\ ,\ Getm\ ,\ Setm\ ,\ LM]$$

et où $m(b)$ désigne l'attribut m ayant une valeur b, $Getm$ et $Setm$ désignent les méthodes associées à l'attribut m et LM désigne l'ensemble des autres attributs et méthodes déclarés pour cet objet.

A cet objet $FN(t_0)$ correspond un ρ-terme $\tau(FN(t_0))$ tel que :

$$\begin{aligned}\tau(FN(t_0)) = t_0' = (&m \rightarrow S \rightarrow b,\\ &Getm \rightarrow S \rightarrow S \cdot m,\\ &Setm \rightarrow S \rightarrow N \rightarrow S \cdot m := S' \rightarrow N,\\ &L)\end{aligned}$$

où $L = \tau(LM)$. Définissons $\rightarrow_R$ comme la relation de réécriture sur les termes définie par l'application d'une seule règle de R et $\overset{*}{\rightarrow}_R$ comme la fermeture réflexive et transitive de $\rightarrow_R$.

- *Pour toute méthode m_1,*

$$\begin{aligned}add(t_0,m_1) \ &\overset{*}{\rightarrow}_R\ add([m(b)\ ,\ Getm\ ,\ Setm\ ,\ LM],m_1)\\ &\rightarrow_R\ [m(b)\ ,\ Getm\ ,\ Setm\ ,\ LM\ ,\ m_1] = t_1\end{aligned}$$

D'un autre côté, dans le ρ-calcul:

$$\begin{array}{lll} \tau(FN(t_0)),\tau(m_1) & \triangleq & (m \to S \to b, \\ & & Getm \to S \to S \cdot m, \\ & & Setm \to S \to N \to S \cdot m := S' \to N, \\ & & \tau(LM)), \\ & & \tau(m_1) \\ & \mapsto & (m \to S \to b, \\ & & Getm \to S \to S \cdot m, \\ & & Setm \to S \to N \to S \cdot m := S' \to N, \\ & & \tau(LM),\tau(m_1)) \\ & \triangleq & \tau(t_1) \end{array}$$

Ainsi, $\tau(FN(add(t_0,m_1))) \equiv \tau(FN(t_0)),\tau(m_1)$.

- *Pour toute méthode* m,

$$\begin{array}{lll} access(t_0,m) & \stackrel{*}{\to}_R & access([m(b)\ ,\ Getm\ ,\ Setm\ ,\ LM],m) \\ & \to_R & b \end{array}$$

A partir du ρ-terme $t'_0 \cdot m = \tau(t_0) \cdot m$, *l'évaluation suivante est réalisée en appliquant les règles du ρ-calcul:*

$$\begin{array}{lll} t'_0 \cdot m & \triangleq & t'_0 \bullet m \bullet t'_0 \\ & \mapsto & (m \to S \to b, \\ & & Getm \to S \to S \cdot m, \\ & & Setm \to S \to N \to S \cdot m := S' \to N, \\ & & LM) \bullet m \bullet t'_0 \\ & \mapsto & (S \to b) \bullet t'_0 \\ & \mapsto & b \end{array}$$

On en déduit donc que $\tau(FN(access(t_0,m))) \equiv \tau(FN(t_0)) \cdot m$.

- *Pour toute méthode* m,

$$\begin{array}{lll} kill(t_0,m) & \stackrel{*}{\to}_R & kill(FN(t_0),m) \\ & \to_R & kill([m(b)\ ,\ Getm\ ,\ Setm\ ,\ LM],m) \\ & \to_R & [Getm\ ,\ Setm\ ,\ LM] = t_1 \end{array}$$

D'autre part, dans le ρ-calcul:

$$\begin{array}{lll} kill_m(\tau(FN(t_0)) & \triangleq & kill_m(t'_0) \\ & \triangleq & kill_m(m \to S \to b, \\ & & Getm \to S \to S \cdot m, \\ & & Setm \to S \to N \to S \cdot m := S' \to N, \\ & & L) \\ & \mapsto & (Getm \to S \to S \cdot m, \\ & & Setm \to S \to N \to S \cdot m := S' \to N, \\ & & L) \\ & \triangleq & \tau(t_1) \end{array}$$

Donc, on a bien $\tau(FN(kill(t_0,m))) \equiv kill_m(\tau(FN(t_0))$.

- *On a d'un côté avec la réécriture dans* R,

$$\begin{array}{lll} Getm(t_0) & \to_R & Get(t_0,m) \\ & \stackrel{*}{\to}_R & Get(FN(t_0),m) \\ & \to_R & Get([m(b)\ ,\ Getm\ ,\ Setm\ ,\ LM],m) \\ & \to_R & access([m(b)\ ,\ Getm\ ,\ Setm\ ,\ LM],m) \\ & \to_R & b \\ & = & t_1 \end{array}$$

Parallèlement, dans le ρ-calcul :

$$\begin{array}{lll} t'_0 \cdot Getm & \triangleq & t'_0 \bullet Getm \bullet t'_0 \\ & \triangleq & (m \to S \to b, \\ & & Getm \to S \to S \cdot m, \\ & & Setm \to S \to N \to S \cdot m := S' \to N, \\ & & L) \bullet Getm \bullet t'_0 \\ & \mapsto & (S \to S \cdot m) \bullet t'_0 \\ & \mapsto & t'_0 \cdot m \\ & \triangleq & t'_0 \bullet m \bullet t'_0 \\ & \triangleq & (m \to S \to b, \\ & & Getm \to S \to S \cdot m, \\ & & Setm \to S \to N \to S \cdot m := S' \to N, \\ & & L) \bullet m \bullet t'_0 \\ & \mapsto & (S \to b) \bullet t'_0 \\ & \mapsto & b \\ & \triangleq & \tau(t_1) \end{array}$$

On a donc bien $\tau(FN(Getm(t_0))) \equiv \tau(FN(t_0)) \cdot Getm$.

– *On a d'une part :*

$$\begin{array}{lll} Setm(t_0,V) & \stackrel{*}{\to}_R & Setm(FN(t_0),V) \\ & \to_R & Set(FN(t_0),m,V) \\ & \to_R & Set([m(b)\ ,\ Getm\ ,\ Setm\ ,\ LM],m,V) \\ & \to_R & add(kill([m(b)\ ,\ Getm\ ,\ Setm\ ,\ LM],m),m(V)) \\ & \to_R & add([Getm\ ,\ Setm\ ,\ LM],m(V)) \\ & \to_R & [m(V)\ ,\ Getm\ ,\ Setm\ ,\ LM]\ =\ t_1 \end{array}$$

D'autre part, dans le ρ-calcul :

$$\begin{array}{lll} t'_0 \cdot Setm(V) & \triangleq & t'_0 \bullet Setm \bullet t'_0 \bullet V \\ & \triangleq & (m \to S \to b, \\ & & Getm \to S \to S \cdot m, \\ & & Setm \to S \to N \to S \cdot m := S' \to N, \\ & & L) \bullet Setm \bullet t'_0 \bullet V \\ & \mapsto & (S \to N \to S \cdot m := S' \to N) \bullet t'_0 \bullet V \\ & \mapsto & (N \to t'_0 \cdot m := S' \to N) \bullet V \\ & \mapsto & t'_0 \cdot m := S' \to V \\ & \triangleq & kill_m(t'_0),m \to S' \to V \\ & \mapsto & (Getm \to S \to S \cdot m, \\ & & Setm \to S \to N \to S \cdot m := S' \to N, \\ & & L),m \to S' \to V \\ & \mapsto & (m \to S' \to V, \\ & & Getm \to S \to S \cdot m, \\ & & Setm \to S \to N \to S \cdot m := S' \to N, \\ & & L) \\ & \triangleq & \tau(t_1) \end{array}$$

Ainsi, $\tau(FN(Setm(t_0,V))) \equiv \tau(FN(t_0)) \cdot Setm(V)$.

– *D'un côté, on a :*

$$new(t) \quad \stackrel{*}{\to}_R \quad [m(vi_m)\ ,\ Getm\ ,\ Setm\ ,\ LM]$$

De l'autre, dans le ρ-calcul,

$$
\begin{array}{rcl}
t' \cdot new & \triangleq & (m \to S \to S' \to vi_m, \\
& & Getm \to S \to S' \to S \cdot m, \\
& & Setm \to S \to S' \to N \to S \cdot m := S' \to N, \\
& & L, \\
& & new \to S \to (m \to S' \to (S \cdot X) \bullet S', \\
& & \qquad Getm \to S' \to (S \cdot GetX) \bullet S', \\
& & \qquad Setm \to S' \to (S \cdot SetX) \bullet S', \\
& & \qquad L1)) \cdot new \\
& \triangleq & (m \to S \to S' \to vi_m, \\
& & Getm \to S \to S' \to S \cdot m, \\
& & Setm \to S \to S' \to N \to S \cdot m := S' \to N, \\
& & L, \\
& & new \to S \to (m \to S' \to (S \cdot X) \bullet S', \\
& & \qquad Getm \to S' \to (S \cdot GetX) \bullet S', \\
& & \qquad Setm \to S' \to (S \cdot SetX) \bullet S', \\
& & \qquad L1)) \bullet new \bullet t' \\
& \mapsto^* & m \to S' \to vi_m, \\
& & Getm \to S \to S \cdot m, \\
& & Setm \to S \to N \to S \cdot m := S' \to N, \\
& & LM
\end{array}
$$

La dernière réduction suit le même schéma que la réduction donnée pour l'exemple de création d'un objet de la classe Point dans la section 3.3.2.

Finalement, on obtient bien $\tau(FN(new(t)) \equiv \tau(FN(t)) \cdot new$ □

Nous étendons maintenant la définition de la traduction τ en posant, pour tout t_0 de sorte `Object` et tous m et m_1 de sorte `Method`, les équivalences suivantes :

- $\tau(add(t_0,m_1)) = \tau(t_0),\tau(m_1)$
- $\tau(access(t_0,m)) = \tau(t_0) \cdot m$
- $\tau(kill(t_0,m)) = kill_m(\tau(t_0))$
- $\tau(Getm(t_0)) = \tau(t_0) \cdot Getm$
- $\tau(Setm(t_0,V)) = \tau(t_0) \cdot Setm(V)$
- $\tau(new(t_0)) = \tau(t_0) \cdot new$

Ceci nous amène au théorème suivant qui nous permet de relier une réduction dans le ρ-calcul et une réécriture suivant les règles de R, au moyen de la relation τ :

Théorème 3.1 *Pour chaque terme t dans R où R est un système terminant et confluent, $\tau(FN(t)) \equiv \tau(t)$.*

Preuve : *Cette preuve se fait par induction sur la structure du terme t et utilise la Proposition 3.2. Nous détaillons ici la preuve pour un terme t de la forme $add(t_0,m_1)$. On a :*

$$
\begin{array}{rcll}
\tau(FN(add(t_0,m_1))) & \equiv & \tau(FN(t_0)),\tau(m_1) & \textit{par la Proposition 1} \\
& \equiv & \tau(t_0),\tau(m_1) & \textit{par induction} \\
& = & \tau(add(t_0,m_1)) & \textit{par la définition de } \tau
\end{array}
$$

□

3.4.7 La gestion de l'héritage

Dans le ρ-calcul, une classe A est définie par un ensemble de n attributs att_A avec les valeurs initiales vi_A et par un ensemble de m méthodes $meth_A$ auxquelles il faut ajouter la méthode new_A pour la création

d'un objet de la classe. La classe A est représentée par le ρ-terme suivant :

$$
\begin{aligned}
classe_A \triangleq\ & new_A \to S \to (att_{A_1} \to S' \to (S \cdot att_{A_1}) \bullet S', \\
& \quad \ldots \\
& \quad att_{A_n} \to S' \to (S \cdot att_{A_n}) \bullet S', \\
& \quad meth_{A_1} \to S' \to (S \cdot meth_{A_1}) \bullet S', \\
& \quad \ldots \\
& \quad meth_{A_m} \to S' \to (S \cdot meth_{A_m}) \bullet S'), \\
& att_{A_1} \to S \to S' \to vi_{A_1}, \\
& \ldots \\
& att_{A_n} \to S \to S' \to vi_{A_n}, \\
& meth_{A_1} \to S \to S' \to body_{A_1}, \\
& \ldots \\
& meth_{A_m} \to S \to S' \to body_{A_m}
\end{aligned}
$$

Une classe B définissant ses n' propres attributs att_B et ses m' propres méthodes $meth_B$ et héritant de la classe A est alors représentée par le ρ-terme suivant :

$$
\begin{aligned}
classe_B \triangleq\ & new_B \to S \to (att_{A_1} \to S' \to (S \cdot att_{A_1}) \bullet S', \\
& \quad \ldots \\
& \quad att_{A_n} \to S' \to (S \cdot att_{A_n}) \bullet S', \\
& \quad att_{B_1} \to S' \to (S \cdot att_{B_1}) \bullet S', \\
& \quad \ldots \\
& \quad att_{B_{n'}} \to S' \to (S \cdot att_{B_{n'}}) \bullet S', \\
& \quad meth_{A_1} \to S' \to (S \cdot meth_{A_1}) \bullet S', \\
& \quad \ldots \\
& \quad meth_{A_m} \to S' \to (S \cdot meth_{A_m}) \bullet S', \\
& \quad meth_{B_1} \to S' \to (S \cdot meth_{B_1}) \bullet S', \\
& \quad \ldots \\
& \quad meth_{B_{m'}} \to S' \to (S \cdot meth_{B_{m'}}) \bullet S'), \\
& att_{A_1} \to S \to S' \to vi'_{A_1}, \\
& \ldots \\
& att_{A_n} \to S \to S' \to vi'_{A_n}, \\
& att_{B_1} \to S \to S' \to vi_{B_1}, \\
& \ldots \\
& att_{B_{n'}} \to S \to S' \to vi_{B_{n'}}, \\
& meth_{A_1} \to S \to S' \to body'_{A_1}, \\
& \ldots \\
& meth_{A_m} \to S \to S' \to body'_{A_m}, \\
& meth_{B_1} \to S \to S' \to body_{B_1}, \\
& \ldots \\
& meth_{B_{m'}} \to S \to S' \to body_{B_{m'}}
\end{aligned}
$$

où l'on peut constater que les méthodes importées peuvent être redéfinies (c'est-à-dire que les corps des méthodes définies dans la classe A par $body_{A_i}$ peuvent être redéfinis en des corps $body'_{A_i}$, de même que les valeurs initiales des attributs vi_{A_j} peuvent être redéfinies en vi'_{A_j}).

Ainsi, toute classe héritant d'une autre classe est représentée par un ensemble d'attributs contenant les siens et ceux de la classe héritée. Les valeurs indiquées sont les valeurs initiales. Les méthodes accessibles sont celles définies par la classe ainsi que celles de la classe héritée. Néanmoins, de nouvelles déclarations et règles correspondent aux méthodes de la classe héritée. En effet, on doit redéfinir les opérateurs avec les profils correspondant à la classe qui hérite. Les mêmes règles sont alors redéfinies, mais elles correspondent aux nouveaux profils.

A cette classe B héritant de la classe A, les mêmes mécanismes de traduction et de correspondances entre les ρ-termes et les termes en ELAN s'appliquent.

3.5 Le typage

Jusqu'ici, nous avons étudié l'extension objet d'ELAN basée sur le calcul de réécriture. Dans cette section, nous nous intéressons plus particulièrement au typage des programmes. La correction du typage est la première façon de vérifier la correction d'un programme et permet d'éviter un grand nombre d'erreurs. On peut typer avant l'exécution : on parle alors de typage statique. On peut aussi prendre en compte la sémantique du programme en vérifiant la propriété de préservation de type par évaluation.

Le but est ici de pouvoir dire que, considérant un terme de départ t_0, si on lui applique l'ensemble des règles du programmes que l'on a développé, le type du terme résultat est le même que le type du terme de départ et qu'ainsi, la transformation a donc bien préservé le type du terme.

Comme les termes que nous manipulons peuvent être représentés par des termes du ρ-calcul, on s'est tout d'abord intéressé aux propriétés de typage définies dans le ρ-calcul. Dans sa thèse [Cir00], H. Cirstea présente un système de types dans le cas du ρ-calcul permettant de prouver que la réduction de tout terme *bien typé* est finie et préserve le type du terme initial. Ces propriétés ont été étendues au ρ^+-calcul qui permet de considérer des ensembles dans le membre gauche.

Par contre, dans le formalisme objet qui nous intéresse, nous ne nous situons pas dans le cadre du ρ-calcul ou du ρ^+-calcul. Nous utilisons en effet pour la représentation des objets une syntaxe dans laquelle la présence d'un symbole associatif et commutatif ne permet pas de se placer dans les théories considérées dans les systèmes de types proposés par H. Cirstea.

Nous définissons donc dans cette section un système de types permettant de montrer que pour tout terme d'un type T défini dans le codage des objets, on a une conservation de son type par les règles d'évaluation d'un système $\mathcal{R}_1$ que nous définissons.

3.5.1 Syntaxe

Dans cette partie, on donne la syntaxe des types que l'on considère, celle de l'environnement de typage et finalement celle des termes que l'on cherche à typer.

Syntaxe des types

On considère un ensemble K de types atomiques K_1, K_2, ... et l'ensemble des types inductivement défini par :

- tout type atomique est un type ;
- si A et B sont des types, alors $A \rightarrowtail B$ en est un aussi.

La flèche $\rightarrowtail$ des définitions de type est associative à droite et ainsi, un type de la forme $A_1 \rightarrowtail A_2 \rightarrowtail \ldots \rightarrowtail A_n$ peut être parenthèsé en $A_1 \rightarrowtail (A_2 \rightarrowtail (\ldots \rightarrowtail A_n)\ldots)$.

On donne alors la syntaxe des types :

$$\begin{array}{lll} T & ::= & K \mid \quad \text{(Type atomique)} \\ & & T \rightarrowtail T \quad \text{(Type composé)} \end{array}$$

Syntaxe de l'environnement

Un environnement est un ensemble de variables typées $x :: K$ et de déclarations des signatures des opérateurs $f :: T$. On donne $x \in \mathcal{X}$ avec $\mathcal{X}$ ensemble de variables et K est un type atomique et $f \in \mathcal{F}$ avec $\mathcal{F}$ ensemble de symboles de fonctions et T un type. On y trouve aussi les coercions de type $K < K'$ où K et K' sont des types atomiques et où la coercion de type $K < K'$ représente l'injection i telle que :

$$\begin{array}{lll} i & : & K \to K' \\ & & x \mapsto x \end{array}$$

On peut alors donner la syntaxe d'un environnement :

E	::=	$\emptyset \mid E,E \mid$	(Construction de l'environnement)
		$x :: K \mid$	(Variable typée)
		$f :: T \mid$	(Type fonctionnel)
		$K < K$	(Coercion de type)

Dans la pratique, l'environnement est constitué de l'ensemble des déclarations de types qui sont vérifiées lorsque l'on type un terme. L'environnement est donc constitué de l'ensemble des déclarations dites "globales" dans lesquelles on trouve toutes les définitions d'opérateurs et les coercions de types. Les déclarations de variables correspondent par contre à des déclarations dites "locales" et elles ne sont ajoutées à l'environnement que lorsque cela est nécessaire (cela est réalisé par les règles de typage).

Syntaxe des termes

Pour définir les règles de typage sur les termes représentant les objets, on doit définir une syntaxe des termes manipulés qui inclut à la fois le codage des objets en un terme constitué d'une liste d'attributs et d'accès aux méthodes, mais aussi l'ensemble des méthodes du système. On définit alors une syntaxe particulière pour les termes dans le cadre du typage. Nous la présentons maintenant.

Les termes t que l'on manipule sont de la forme : `{-}_-` où le premier élément est l'ensemble des méthodes définies et le second élément est l'objet manipulé. A cet objet peut être appliquée une méthode `m` avec ou sans paramètre. Une manipulation d'objet peut aussi consister en la modification de la valeur d'un attribut `att` par une nouvelle valeur pouvant être à son tour un terme. Les termes manipulés peuvent aussi être fonctionnels avec un symbole de tête `f` suivi de ses arguments. On donne la syntaxe des termes dans la suite, sachant que les termes manipulés sont définis dans la première ligne. Suivent ensuite les définitions plus particulières de chaque composant : objet et méthodes.

t	::=	$\{methods\}\_o \mid \{methods\}\_o.m(t_1,\ldots,t_n) \mid$	
		$\{methods\}\_o(att := t) \mid f(t_1,\ldots,t_n)$	(Termes)
o	::=	$[S]$	(Objets)
S	::=	$att(t) \mid m \mid att(t),S \mid m,S$	(Structure des objets)
$methods$	::=	$method \mid method;methods$	(Méthodes)
$method$	::=	$m(arguments) = (variables)corps$	(Méthode)
$arguments$	::=	$self \mid self,liste\ args$	(Arguments)
$liste\ args$	::=	$x \mid x,liste\ args$	(Liste d'arguments)
$variables$	::=	$variable \mid variable,variables$	(Liste de variables)
$variable$	::=	$x : K$	(Variable de méthode)
$corps$	::=	$instruction \mid x := t;corps \mid$	
		$if\ t;corps$	(Corps des méthodes)
$instruction$	::=	$var.att \mid var(att := t) \mid t$	(Instructions)
var	::=	$self \mid x$	(Variable dans corps de méthode)

où $x \in \mathcal{X}$ et $f,m,att \in \mathcal{F}$.

On détaille maintenant la syntaxe des termes :

- Les objets sont composés à partir d'une structure non vide définie par une suite de noms de méthodes `m` ou d'attributs `att` ayant pour valeurs un terme.
- L'ensemble des méthodes est défini comme une succession de définitions de méthodes. Chaque méthode est définie par son nom `m`, ses arguments, ses propres variables locales et son corps. Chacune de ces parties est décrite dans la grammaire des termes. Nous détaillons plus particulièrement la définition des corps de méthodes. Un corps de méthode peut être composé d'une seule instruction : obtention de la valeur d'un attribut (`var.att`), modification de cette valeur (`var(att:=t)`) ou encore tout terme `t` possible. Ces instructions peuvent être composées avec des tests (`if t;corps`) ou encore avec des affectations de variables (`x:=t;corps`).

3.5.2 Règles de typage

Les règles de typage du codage des objets sont définies dans la Figure 3.7.

Définition 3.12 *On considère l'ensemble des formules $E \vdash t :: A$ déduites en utilisant les règles de typage du codage des objets données dans la Figure 3.7, on dit alors que t a le type A dans l'environnement E. On dit que le terme t est typable (ou bien typé) de type A s'il existe un type A tel que t a le type A dans le contexte E. Un terme t est typable s'il existe un contexte dans lequel il est typable.*

(Assertion vraie) $$\frac{E \vdash \diamond}{E,p \vdash p}$$

(Environnement) $$\frac{}{E \vdash \diamond}$$

(Représentation objet) $$\frac{E \vdash o::A \quad E \vdash M::Methods}{E \vdash \{M\}\_o::A}$$

(Construction objet) $$\frac{E,[]::AttsAndMeths \rightarrowtail A \vdash m::AttsAndMeths}{E,[]::AttsAndMeths \rightarrowtail A \vdash [m]::A}$$

(Composition objet) $$\frac{E \vdash m_1::AttsAndMeths \quad E \vdash m_2::AttsAndMeths}{E \vdash m_1,m_2::AttsAndMeths}$$

(Accès aux méthodes) $$\frac{E \vdash m::MName}{E \vdash m::AttsAndMeths}$$

(Définition d'attribut) $$\frac{E \vdash a::MName \quad E \vdash b::MBody}{E \vdash a(b)::AttsAndMeths}$$

(Coercion de type) $$\frac{E,S<S' \vdash a::S}{E,S<S' \vdash a::S'}$$

(Appel de méthode) $$\frac{E' \vdash o::S \quad E' \vdash p_1::S_1 \ \ldots \ E' \vdash p_n::S_n \quad E'' \vdash b::A}{E' \vdash m(v,v_1,\ldots,v_n)=(v'_1:S'_1,\ldots,v'_m:S'_m)b,M\_o.m(p_1,\ldots,p_n)::A}$$
où $E' = E,m :: S \rightarrowtail S_1 \rightarrowtail \ldots \rightarrowtail S_n \rightarrow A$

où $E'' = E',v'_1 :: S'_1,\ldots,v'_m :: S'_m$

(Sélection d'attribut) $$\frac{E' \vdash o::S \quad E' \vdash B<MBody \quad E' \vdash a::MName}{E \vdash \{M\}\_o.a::B}$$

où $E = E',Geta :: S \rightarrowtail B$

(Modification de valeur d'attribut) $$\frac{E' \vdash o::S \quad E' \vdash b::B \quad E' \vdash B<MBody \quad E' \vdash a::MName \quad E' \vdash M::Methods}{E \vdash \{M\}\_o(a:=b)::S}$$

où $E = E',Seta :: S \rightarrowtail B \rightarrowtail S$

(Test) $$\frac{E \vdash t::bool \quad E \vdash b::S}{E \vdash if\ t;b::S}$$

(Affectation) $$\frac{E \vdash b::S}{E \vdash x:=t;b::S}$$

(Enchaînement d'appels) $$\frac{E \vdash m1::S \rightarrowtail S_1 \rightarrowtail \ldots \rightarrowtail S_n \rightarrowtail S' \quad E \vdash m2::S' \rightarrowtail S'_1 \rightarrowtail \ldots \rightarrowtail S'_m \rightarrowtail A}{E \vdash \{M\}\_o.m1(p_1,\ldots,p_n).m2(p'_1,\ldots,p'_m)::A}$$

FIG. 3.7 – *Les règles de typage du codage des objets*

Remarque : Etant donnés deux contextes E et E' tels que $E \subseteq E'$, si $E \vdash t :: A$ alors $E' \vdash t :: A$.

3.5.3 Règles d'évaluation

Définition 3.13 *On définit une substitution σ comme étant étant une substitution qui préserve les types si, étant donné un terme t de type T dans le contexte E, $E \vdash t : T$, alors $E \vdash \sigma(t) : T$.*

On fait l'hypothèse que les substitutions utilisées dans les règles d'évaluation préservent les types. Les règles d'évaluation que nous définissons afin d'évaluer les termes présentés précédemment sont alors données par :

Appel de méthode	$\{m(v,v_1,\ldots,v_n) = (v'_1,\ldots,v'_m)b\ ,\ M\}\_[m,LM].m(p_1,\ldots,p_n)$ où σ est une substitution qui préserve les types et $\sigma = \{v \mapsto o, v_1 \mapsto p_1,\ldots,v_n \mapsto p_n\}$ où $o = \{m(v,v_1,\ldots,v_n) = (v'_1,\ldots,v'_m)b\ ,\ M\}\_[m,LM]$	$\rightarrow_{\mathcal{R}_1}$	$\sigma(b)$
Sélection d'attribut	$\{M\}\_[att(b),LM].att$	$\rightarrow_{\mathcal{R}_1}$	b
Modification de valeur d'attribut	$\{M\}\_[att(b),LM](att := b_1)$	$\rightarrow_{\mathcal{R}_1}$	$\{M\}\_[att(b_1),LM]$
Test	$if\ t\ ;b$ si $t \rightarrow_{\mathcal{R}_1} true$	$\rightarrow_{\mathcal{R}_1}$	b
Affectation	$x := t; b$ où $t \rightarrow_{\mathcal{R}_1} t'$ où σ est une substitution qui préserve les types et $\sigma = \{x \mapsto t'\}$	$\rightarrow_{\mathcal{R}_1}$	$\sigma(b)$
Enchaînement d'appels	$\{M\}\_o.m_1.m_2$ où o_1 est défini par $\{M\}\_o.m_1 \rightarrow_{\mathcal{R}_1} o_1$	$\rightarrow_{\mathcal{R}_1}$	$o_1.m_2$

3.5.4 Préservation du type

Nous montrons maintenant que le système de types que nous avons proposé est cohérent par rapport aux règles d'évaluation que nous avons données dans la section précédente. Nous voulons ainsi montrer que le type d'un terme est le même que le type de tout terme obtenu en le réduisant.

Théorème 3.2 *Pour tout terme t et t', si $t \rightarrow_{\mathcal{R}_1} t'$ et si $E \vdash t :: A$, alors $E \vdash t' :: A$.*

Preuve : En examinant les règles d'évaluation de $\mathcal{R}_1$ une par une, nous montrons que les membres gauche et droit de chaque règle ont le même type dans un contexte donné.

- la règle *Appel de méthode*:
  Considérons que pour le membre gauche de la règle $E \vdash \{m(v,v_1,\ldots,v_n) = (v'_1,\ldots,v'_m)b\ ,\ M\}\_[m,LM].m(p_1,\ldots,p_n) :: A$.
  Par la règle de typage sur l'appel de méthode, on a alors $E' \vdash b :: A$ où $E' = E, v'_1 :: S'_1,\ldots,v'_m :: S'_m$. Comme les informations $v'_1 :: S'_1,\ldots,v'_m :: S'_m$ ne sont pas utilisées par les règles de typage sur les corps des méthodes (règle test et affectation), on peut alors déduire que si $E' \vdash b :: A$, alors $E \vdash b :: A$.
  Comme la substitution σ est une substitution qui préserve les sortes, on peut alors dire que $E \vdash \sigma(b) :: A$. On a alors le membre droit de la règle conséquence du type du membre gauche.
- la règle *Sélection d'attribut*:
  Considérons que le membre gauche de la règle $\{M\}\_[att(b),LM].att$ est de type A dans un environnement E. Par la règle de typage sur la sélection d'attribut, on peut déduire que l'environnement E est composé d'une assertion $Geta :: S \rightarrowtail A$ et qu'alors, on a bien $E \vdash [att(b),LM] :: S$, $E \vdash A < MBody$ et $E \vdash att :: MName$. Par la règle de construction d'objet, on a alors que l'environnement E contient l'assertion $[] :: AttsAndMeths \rightarrowtail S$ et que l'on a $E \vdash att(b),LM :: AttsAndMeths$. Par la règle de composition de l'objet, on a alors $E \vdash att(b) :: AttsAndMeths$ puis, par la règle de définition d'attribut, on a alors $E \vdash b :: MBody$. Finalement, par la règle de coercion de type, comme $E \vdash A < MBody$ et $E \vdash b :: MBody$, on a bien $E \vdash b :: A$.
  Ainsi, on a bien montré que les membres gauche et droit sont de même type A.

- la règle *Modification de valeur d'attribut*:
  Considérons le membre gauche de la règle, $\{M\}\_[att(b),LM](att := b_1)$, qui doit être de type A dans l'environnement E: $E \vdash \{M\}\_[att(b),LM](att := b_1) :: A$. Par la règle de typage sur la modification de valeur d'attribut, on a donc, si on considère que l'environnement E inclut l'insertion $Seta :: A \rightarrowtail B \rightarrowtail A$ que $E \vdash M :: Methods$, $E \vdash [att(b),LM] :: A$, $E \vdash b :: B$, $E \vdash b_1 :: B$, $E \vdash att :: MName$ et $E \vdash B < MName$.
  Pour avoir $E \vdash [att(b),LM] :: A$, en considérant la règle de typage sur la construction d'objet, on a alors que $E \vdash att(b),LM :: AttsAndMeths$ avec l'assertion suivante $[] :: AttsAndMeths \rightarrowtail A$ comprise dans l'environnement E. Ensuite, par la règle de typage sur la composition des objets, on déduit que $E \vdash LM :: AttsAndMeths$.
  Ainsi, on peut déduire puisque $E \vdash b_1 :: B$ et $E \vdash B < MName$ par la règle de coercion de type que $E \vdash b_1 :: MBody$. Ensuite, par la règle de définition d'attribut, comme on a $E \vdash b_1 :: MBody$ et $E \vdash att :: MName$, on a alors $E \vdash att(b_1) :: AttAndMeths$. Puisque $E \vdash att(b_1) :: AttAndMeths$ et $E \vdash LM :: AttsAndMeths$, par la règle de composition d'objet, on a alors $E \vdash att(b_1),LM :: AttAndMeths$. Puis, comme $[] :: AttsAndMeths \rightarrow A$ est dans l'environnement, on peut déduire que $E \vdash [att(b_1),LM] :: A$. Finalement, comme $E \vdash M :: Methods$, par la règle de représentation de l'objet, on déduit que $E \vdash \{M\}\_[att(b_1),LM] :: A$.
  Ainsi, on a bien montré que le membre droit est donc de même type A que le membre gauche.
- la règle *Test*:
  Considérons que le membre gauche doive être de type A, on a ainsi $E \vdash if\ t; b :: A$. Par la règle de typage sur le test, on a alors que $E \vdash t :: bool$ ainsi que $E \vdash b :: A$. On a donc bien pour le membre droit de cette règle d'évaluation le typage $E \vdash b :: A$.
- la règle *Affectation*:
  Considérons le membre gauche de la règle et on a ainsi $E \vdash x := t; b :: A$. Par la règle de typage sur l'affectation, on a donc $E \vdash b :: A$. Appliquons maintenant une substitution σ sur ce terme et on obtient que $E \vdash \sigma(b) :: A$ puisque la substitution σ préserve les sortes.
- la règle *Enchaînement d'appels*:
  Considérons le membre gauche de règle tel que $E \vdash \{M\}\_o.m_1.m_2 :: A$. Par la règle de typage sur les enchaînements d'appels, on a alors que l'environnement E contient une assertion de la forme $m1 :: S \rightarrowtail S_1 \rightarrowtail \ldots \rightarrowtail S_n \rightarrowtail S'$. On applique alors sur le terme $\{M\}\_o.m_1$ la première règle de typage sur l'appel de méthode et on en déduit alors (cf. preuve pour appel de méthode) que si $E \vdash \{M\}\_o.m_1 :: S'$ alors $E \vdash \{M\}\_o_1 :: S'$ aussi. On peut alors appliquer le second appel de méthode puisque l'environnement E contient $m2 :: S' \rightarrowtail S'_1 \rightarrowtail \ldots \rightarrowtail S'_m \rightarrowtail A$. Et on obtient alors de façon similaire que $E \vdash o_1.m_2 :: A$.
  On a donc montré que les membres gauche et droit sont de même type A.

□

Conclusion

Dans ce chapitre, on a expliqué comment on peut utiliser ELAN pour coder un langage à prototype. On a ainsi une représentation propre à chaque objet dans laquelle on retrouve l'ensemble des attributs avec leurs valeurs ainsi qu'un ensemble de noms de méthode. Ces noms de méthodes sont en fait des points d'accès vers des corps de méthodes, les méthodes étant implémentées par des règles de réécriture. La possibilité de supprimer ou d'ajouter un nom de méthode de la liste permet ainsi à chaque objet de modifier en cours d'exécution les accès aux méthodes. Les méthodes sont par contre définies de façon définitive lors de la définition de la classe. Les méthodes peuvent être séparées en deux groupes: celles définies automatiquement par le système et qui concernent les manipulations des attributs et la création d'objet et celles définies par l'utilisateur.

Après avoir défini ce format de représentation, nous avons pu donner un ensemble de règles $\mathcal{R}$ dans lequel nous avons défini les règles de manipulation des objets. Ce système $\mathcal{R}$, qui peut être étendu par les règles correspondant aux méthodes définies par l'utilisateur, est confluent et terminant. Ainsi, on a pu l'implémenter par des règles de réécriture non nommées en ELAN.

S.C.D. - U.H.P. NANCY 1
BIBLIOTHÈQUE DES SCIENCES
Rue du Jardin Botanique
54600 VILLERS-LES-NANCY

On a ensuite montré qu'à tout terme ELAN implémentant un objet correspond un ρ-terme du ρ-calcul et qu'à chaque réécriture de ce terme par le système R implémentant $\mathcal{R}$ correspond une ρ-évaluation. On a ainsi montré que le ρ-calcul définit une sémantique opérationnelle pour l'implémentation du codage des objets en ELAN.

L'ensemble de ces travaux sur l'utilisation d'ELAN pour coder un langage à prototype a été le sujet d'un rapport de recherche [DK00a].

Finalement, on a montré que notre codage permettait de respecter la propriété de préservation de type et que tout terme t de type A pouvait être réécrit par $\mathcal{R}_1$ en un terme t' de même type A.

Chapitre 4

Programmer avec des objets et des contraintes

Dans le chapitre 3, nous avons vu comment on peut coder dans le langage ELAN des concepts objets. Un des buts de nos travaux est de pouvoir formaliser des problèmes de planification dans lesquels les notions d'objets sont importantes, mais dans lesquels nous voulons aussi inclure la notion de contraintes.

Nous devons ainsi étudier comment programmer en ELAN en adaptant les notions de règles de réécriture et de stratégies à la gestion des objets et des contraintes.

En effet, dans le chapitre précédent, nous avons montré comment les objets sont codés en ELAN ; dans ce chapitre, nous montrons comment nous pouvons les utiliser et intégrer la notion d'objet en ELAN. Ainsi, dans un premier temps, nous étudions l'adaptation des règles aux objets. Quelles primitives sont accessibles? Comment les objets sont-ils traités dans ce formalisme? Cette section est illustrée à l'aide d'un exemple de contrôleur d'ascenseurs.

Ensuite, nous étendons ce formalisme de règles avec objets à la notion de contraintes. Dans cette partie, nous montrons comment les mécanismes de communications avec le système COLETTE sont intégrés aux règles pouvant manipuler des contraintes définies dans notre formalisme. Cette section est illustrée au moyen d'un exemple de gestionnaire d'impressions.

4.1 Programmation avec règles, stratégies et objets

Après avoir montré dans le chapitre 3 comment coder et implanter des objets en ELAN, nous pouvons maintenant les intégrer au formalisme ELAN qui est quant à lui basé sur les règles de réécriture et les stratégies.

La définition des classes introduite précédemment nous donne un certain nombre de primitives permettant de manipuler les objets. Les primitives présentées dans 3 servent à coder et à manipuler les objets. Dans cette partie, seule la manipulation des objets nous intéresse. Nous donnons donc un ensemble de primitives autorisées dans la programmation avec règles, stratégies et objets qui sont détaillées dans la première section. Ensuite, nous mettons en place un formalisme particulier qui est adapté à la manipulation d'une mémoire d'objets. Ce formalisme est ensuite illustré par l'écriture d'un contrôleur de plusieurs ascenseurs.

4.1.1 Des primitives objets en ELAN

Tout d'abord, afin de pouvoir manipuler les OModules définissant les classes dans un programme ELAN, nous devons importer dans la partie réservées aux importations de modules (cf. section 2.4.5), pour chaque classe que l'on souhaite utiliser, le OModule la définissant. Cette déclaration permet d'importer les modules ELAN correspondant aux modules objets.

Exemple 4.1 *Par exemple, pour utiliser une classe `C1` dans un module* ELAN, *comme on associe à chaque classe un module* ELAN, *il suffit d'utiliser le mécanisme habituel des importations qui est le suivant :*

```
import global C1;
end
```

Cette importation permet alors au système ELAN *d'importer le module* ELAN *correspondant au codage de la classe C1 qui est définie quant à elle par l'utilisateur dans un OModule.*

Afin de manipuler classes et objets, nous avons à notre disposition différents mécanismes autorisés par le système ELAN. Les primitives que nous présentons ici sont un sous-ensemble de celles présentées dans le chapitre 3. En effet, on ne cherche ici qu'à manipuler les objets et les classes.

- L'envoi d'un message à un objet est fait par "`_._`". Cet opérateur prend comme paramètres un objet et une méthode (qui peut être paramétrée) et vérifie lors de l'exécution que cette méthode est bien accessible par l'objet courant. Si c'est le cas, la méthode est appliquée.

  Exemple 4.2 *Prenons un objet `P` de la classe définie dans l'Exemple 3.6. Demander l'exécution de la méthode `Translate` sur cet objet se note par l'envoi de message `P.Translate`. L'appel de la méthode `TranslateX` à `P` avec comme paramètre 2 se note quant à lui `P.TranslateX(2)`.*

  Parmi tous les envois de messages possibles, nous pouvons détailler quelques messages plus particuliers.

  - La création d'un nouvel objet d'une classe se fait par la méthode `new`. Il consiste à envoyer le message `new` à l'objet représentant la classe concernée. Comme nous l'avons vu précédemment, à toute définition de classe correspond un objet représentant cette classe ; le nom de cet objet est le nom de la classe suffixé par la chaîne de caractères "`Class`".

    Exemple 4.3 *Par exemple, pour la classe `Point`, l'objet représentant cette classe est appelé `PointClass`. Créer un nouvel objet est ainsi noté `PointClass.new`.*

  - L'accès à la valeur d'un attribut A se fait par la méthode `GetA`. Pour un objet `P` d'une classe donnée ayant un attribut `A`, l'appel se fait par `P.GetA`.
    Notation : pour simplifier les notations, au lieu d'écrire `P.GetA`, on note cet envoi de message `P.A`.
  - La modification de la valeur d'un attribut A se fait par la méthode `SetA`. Pour un objet `P` d'une classe donnée où un attribut `A` est défini, l'appel de `SetA` se fait par `P.SetA(V)` où `V` est la nouvelle valeur associée à A.
    Notation : pour alléger l'écriture des programmes, au lieu d'écrire `P.SetA(V)`, on note cet envoi de message `P(A<-V)`.
    Notation : lorsque plusieurs modifications successives d'attributs doivent être faites sur le même objet `P`, par soucis de concision, au lieu de les noter `P.SetA1(V1).....SetAn(Vn)`, on les note `P(A1<-V1)...(An<-Vn)`.

- La suppression d'un attribut ou d'un accès à une méthode par l'opérateur `kill`.
  Pour un objet `P` donné, supprimer un attribut `A` de la liste des attributs de cet objet se note `kill(P,A)`.
  Pour un objet `P` donné, supprimer la référence à une méthode `M` se note `kill(P,M)`.
- La réaffectation après suppression d'un accès à une méthode se fait par `add`. En effet, cela consiste à redonner un accès qui avait été suspendu auparavant par l'opérateur `kill` en ajoutant le nom de cette méthode à la liste des méthodes déjà accessibles par l'objet. Par contre, l'ajout d'une méthode ne veut pas dire que l'on peut définir une nouvelle méthode à *run-time*, cela induirait que l'on puisse ajouter des nouvelles règles de réécriture en cours d'exécution.

Tous ces opérateurs doivent être utilisables dans les règles de réécriture manipulant les objets que l'on va définir par la suite. Une règle de réécriture manipulant un objet défini selon la méthode présentée dans le chapitre précédent peut ainsi le créer puis le manipuler selon les primitives présentées dans cette section.

Néanmoins, dans les problèmes de planification qui nous intéressent, une même règle peut manipuler plusieurs objets différents, ce qui conduit à définir des règles travaillant sur des bases d'objets.

4.1.2 Un formalisme particulier : des règles sur les objets

Dans le cadre de l'utilisation et de la gestion des objets, le formalisme à base de règles et de stratégies que l'on propose consiste à travailler sur une base d'objets que règles et stratégies manipulent et font évoluer.

Définition 4.1 *Une base d'objets représente l'état courant des informations utilisées comme base de faits pour l'application qui est développée. Elle est représentée par un multi-ensemble dont chaque élément est un objet. Ces objets peuvent être des objets de classes différentes. Une base de faits, ici base composée d'objets, est aussi appelée mémoire d'objets ou mémoire de travail.*

Les règles permettent de détruire, modifier ou encore créer des objets de la base et aussi de pouvoir facilement exprimer les concepts de concurrence et de synchronisation.

Les programmes sur les objets sont des ensembles de règles sur des multi-ensembles d'objets. Les modifications de la mémoire d'objets sont décrites par des règles de réécriture conditionnelles de la forme :

$$[lab]\ \ O_1 \ldots O_k \ \Rightarrow\ O'_1 \ldots O'_m\ \ [\textbf{if}\ \ t \mid \textbf{where}\ \ l]^*$$

où $O_1,\ldots,O_k,O'_1,\ldots,O'_m$ sont des objets, t est un terme booléen et l une affectation locale utilisée pour les variables intermédiaires. Cette règle, comme les autres règles ELAN, peut être nommée par le label *lab*.

Les règles peuvent être appliquées sur la base d'objets suivant les stratégies définies par l'utilisateur si les règles sont nommées ou bien sinon selon le processus standard de normalisation qui tente d'appliquer toutes les règles non nommées à la base. Une règle est candidate si son membre gauche filtre un sous-ensemble d'objets dans la base de données. Un objet O_i dans le membre gauche de règle peut avoir l'une ou l'autre des deux formes suivantes :

- une première forme est définie par :

  $$O_i : ClassName_i :: [Att_1(Value_1)\ ,\ \ldots\ ,\ Att_n(Value_n)]$$

  et correspond au fait de rechercher dans la mémoire de travail l'objet de la classe $ClassName_i$ dont les attributs $Att_1,\ldots,Att_n$ ont les valeurs correspondantes $Value_1,\ldots,Value_n$. L'ordre des attributs n'a pas d'importance et certains attributs de la classe peuvent être omis. Lorsqu'un objet est sélectionné, il est associé à une variable locale à la règle que l'on note O_i. La portée de la variable O_i est locale à la règle.
- La seconde forme :

  $$O_i : ClassName_i$$

  correspond au fait de rechercher dans la base d'objets n'importe quel objet de la classe $ClassName_i$. Lorsqu'un objet est sélectionné, une variable locale à la règle O_i est associée à l'objet. La portée de la variable O_i est locale à la règle.

L'application de la règle est conditionnée par l'évaluation de la condition en *true* et par le succès de l'évaluation de chaque affectation locale.

La base d'objets est ensuite mise à jour ; l'ensemble des objets présents dans le membre droit de la règle peut être décomposé en quatre sous-ensembles :

- les objets modifiés : ce sont les objets présents dans le membre gauche de la règle et qui, dans le membre droit, réapparaissent, mais modifiés par l'application d'une méthode (un changement de valeur d'attribut par exemple),

- les objets contextuels : ce sont les objets qui apparaissent dans le membre droit de règle inchangés par rapport au membre gauche. Aucune modification n'a été faite sur eux, ils représentent le contexte d'exécution de la règle,
- les nouveaux objets : ce sont les objets créés lors de l'application de la règle. Ils n'apparaissent pas dans le membre gauche et font partie de la base d'objets du membre droit de cette règle,
- les objets supprimés : ce sont les objets apparaissant dans le membre gauche de règle et qui ne figurent plus dans le membre droit.

Les objets de la base n'apparaissant pas dans le membre gauche de règle sont des objets persistants non pris en compte lors de l'application de la règle et demeurant inchangés après application de cette dernière.

4.1.3 Exemple : un contrôleur d'ascenseurs

Afin d'illustrer ces règles de réécriture traitant les objets, nous considérons un programme modélisant un contrôleur de plusieurs ascenseurs.

Le problème du contrôleur d'ascenseurs a été traité dans la littérature sous différentes formes. Parmi celles-ci, on peut citer quelques approches intéressantes qui ont été utilisées pour la formalisation d'un tel problème. Ainsi, dans [ZM93], une approche basée sur des réseaux contraints est utilisée. Elle permet de spécifier, contrôler et vérifier le système avec un seul ascenseur dans un environnement changeant modélisé par des automates temporisés. La logique temporelle, elle, est utilisée dans les travaux de [Bar85] pour modéliser et vérifier un système cette fois composé de plusieurs ascenseurs.

Une autre approche qui nous intéresse plus particulièrement est celle utilisée dans les ASM (*Abstract State Machine*) [Gur88b, Gur88a]. Un ASM consiste en un ensemble fini de règles de transition avec conditions et actions pouvant s'appliquer sur un état courant et le modifier si les conditions d'application de la règle sont vérifiées. Dans ses travaux, E. Boerger [BH98] présente différents exemples dont un contrôleur de plusieurs ascenseurs inspiré de l'implantation en B réalisée par J.-R. Abrial [Abr96]. Ainsi, si on utilise un formalisme basé sur les ASM, différentes techniques de vérification mathématique et de validation expérimentale sont disponibles.

Le fait de traiter ce genre de problème dans le formalisme de règles avec objets se justifie par le fait que le formalisme présenté ici est basé sur une représentation uniforme à base de règles, que les stratégies permettent de définir un contrôle explicite des règles, que les techniques de preuves sont celles utilisées dans les systèmes de règles : terminaison, confluence, etc.

Pour modéliser le contrôleur d'ascenseurs dans notre formalisme, nous définissons deux classes : une classe `MLift` pour les ascenseurs et une classe `Call` pour le contrôleur. Les règles et stratégies sont ensuite présentées avant de détailler une exécution.

La classe `MLift`

Cette classe décrit la mémoire centrale du contrôleur de plusieurs ascenseurs. Chaque ascenseur est un objet de cette classe.

Un ascenseur est caractérisé par :

- son étage courant qui est désigné par l'attribut `CF` -*current floor*- représenté par un entier,
- son état : est-il en train de monter? de descendre? ou attend-il un appel? C'est l'attribut `State` de sorte `LiftState` qui désigne cet état,
- la liste des étages où il doit s'arrêter : c'est l'attribut `LStop` représenté par une liste d'entiers.

La sorte décrivant l'état de l'ascenseur est appelée `LiftState`. Les termes de cette sorte sont définis par deux opérateurs : une constante `Wait` de sorte `LiftState` et un opérateur `Move(_)` qui prend un terme de sorte `Direction` (`Up` et `Down` sont les deux termes de sorte `Direction`) et qui retourne un terme de sorte `LiftState`. Ceci est défini dans un module ELAN par les déclarations d'opérateurs suivantes :

```
operators global
    Up          : Sense;
    Down        : Sense;
    Move(@)     : (Sense) LiftState;
    Wait        : LiftState;
end
```

Trois autres attributs sont aussi définis :

- `Zone` qui indique la zone où se trouve l'ascenseur. En effet, l'ensemble des étages est divisé en autant de zones qu'il y a d'ascenseurs, une zone étant composée d'étages consécutifs. Ainsi, si un immeuble compte 34 étages numérotés de 1 à 34 et que le nombre d'ascenseurs est de quatre, on aura la zone 1 qui comprendra les étages 1 à 8, la zone 2 de 9 à 17, la zone 3 de 18 à 26 et la zone 4 de 27 à 34. Cet attribut est utile si l'on souhaite vérifier que chaque zone n'a pas plus de deux ascenseurs en activité en même temps afin de garantir une équité de service dans l'ensemble de l'immeuble.
- `F` pour *Flag* dont les valeurs 0 ou 1 indiquent que l'ascenseur est en train de réaliser une instruction (`F` mis à `1`) ou l'ascenseur attend une instruction à réaliser (`F` mis à `0`).
- `I` pour définir une interruption de service pour un ascenseur : si l'ascenseur est disponible, on a `I` = `0` ; sinon, lorsque l'ascenseur est hors service, `I` = `1`.

En plus des attributs, la classe `MLift` définit aussi différentes méthodes :

- une méthode `WhichSense(_)` qui prend en paramètre un entier (représentant le nouvel étage à atteindre) et qui redéfinit le sens de déplacement de l'ascenseur entre l'étage courant et sa nouvelle destination,
- une méthode `UpdateZone` qui modifie l'objet en calculant la zone d'appartenance de l'ascenseur en fonction de l'étage courant,
- une méthode `AddLStop(_)` qui prend en paramètre une liste d'entiers représentant différents étages et rend l'objet avec sa liste d'appels augmentée de cette nouvelle liste. En sortie, la liste d'appels est de plus triée,
- une méthode `RemoveLStop(_)` qui ôte de la liste des appels l'étage passé en paramètre.

Ces déclarations d'attributs et de méthodes se retrouvent dans la définition suivante de la classe `MLift` :

```
class MLift

imports ToolsMLift

attributes CF:int = 0
           State:LiftState = Wait
           LStop:list[int] = nil
           Zone:int = 0
           F:int = 0
           I:int = 0

method WhichSense(N:int) for MLift
 S : Sense;
 <S:=ChooseSense(self.GetCF,N) ; self.SetState(Move(S))>

method UpdateZone for MLift
 <self.SetZone(NewZone(self.GetCF))>

method AddLStop(L:list[int]) for MLift
 <self.SetLStop(AddAndSort(self.GetLStop,L))>
```

```
method RemoveLStop(N:int) for MLift
 <self.SetLStop(RemoveList(self.GetLStop,N))>

End
```

La classe Call

Cette classe décrit la mémoire centrale du contrôleur. Lorsque des personnes ont demandé un ascenseur, cette information est sauvegardée dans l'attribut `LCall` composé d'une liste d'entiers représentant la liste des appels. Afin de distinguer les appels qui sont en cours de traitement (i.e. auxquels le contrôleur a déjà affecté un ascenseur actuellement en route vers cet appel) des appels restant à traiter, un deuxième attribut `AssignedCall` mémorise les étages vers lesquels un ascenseur se déplace.

Différentes méthodes sont associées à cette classe :

- une méthode `AddAssignedCall(_)` qui prend en paramètre un entier représentant un étage et qui l'ajoute dans la liste des appels en cours de traitement.
- une méthode `RemoveAssignedCall(_)` qui prend en paramètre un entier représentant un étage et qui l'enlève de la liste des appels en cours de traitement.
- une dernière méthode `RemoveLCall(_)` qui prend en paramètre un entier représentant un étage et qui l'enlève de la liste des appels restant à traiter.

Cette classe est décrite dans le module objet suivant :

```
class Call

imports Tools

attributes LCall:list[int] = nil
           AssignedCall:list[int] = nil

method AddAssignedCall(N:int) for Call
 <self.SetAssignedCall(AddList(self.GetAssignedCall,N))>

method RemoveAssignedCall(N:int) for Call
 <self.SetAssignedCall(RemoveList(self.GetAssignedCall,N))>

method RemoveLCall(N:int) for Call
 <self.SetLCall(RemoveList(self.GetLCall,N))>

End
```

Les deux classes définies dans les OModules sont maintenant définies. On peut alors utiliser des objets de ces classes dans le formalisme de règles avec objets présenté dans la section 4.1.2.

Les règles

On décrit dans cette partie les principales règles définissant le comportement du contrôleur d'ascenseurs.

Les deux principales règles concernant les mouvements des ascenseurs sont les règles appelées `Up` et `Down` qui sont les deux règles de déplacement les plus basiques. Un ascenseur peut continuer son chemin de montée ou de descente lorsque l'étage où il passe n'est pas dans sa liste des arrêts demandés par les usagers qu'il transporte, ou bien lorsque l'étage courant n'est pas non plus demandé par un appel extérieur non pris en compte par un autre ascenseur. Lorsque l'une ou l'autre de ces deux règles s'applique,

l'étage courant est alors mis à jour et incrémenté (ou décrémenté) de une unité. Les autres conditions d'application de ces règles sont que l'ascenseur n'est pas déjà en train de réaliser une autre tâche (attribut F mis à 0) et que l'ascenseur est bien en état de fonctionnement (attribut I mis à 0).

Les deux règles Up et Down sont les suivantes :

```
[Up] O1:MLift::[State(Move(Up)) , F(0) , I(0)]
     O2:LCall
        =>
      O1(CF<-O1.CF+1).UpdateZone(F<-1)
      O2
        if not(in(O1.CF,O1.Stop))
        if not(in(O1.CF,O2.LCall))

[Down] O1:MLift::[State(Move(Down)) , F(0) , I(0)]
       O2:LCall
         =>
        O1(CF<-O1.CF-1).UpdateZone(F<-1)
        O2
         if not(in(O1.CF,O1.Stop))
         if not(in(O1.CF,O2.LCall))
```

L'application de chacune de ces deux règles implique des modifications de l'ascenseur considéré par la mise à jour de son étage courant, la mise à jour de sa zone et le positionnement de l'attribut F à 1.

Néanmoins, au lieu de continuer son mouvement dans une certaine direction, un ascenseur peut être amené à modifier son sens de déplacement. Les deux règles suivantes définissent un changement de direction du haut vers le bas (règle ChangeToDown) et réciproquement du bas vers le haut (règle ChangeToUp). Un changement de direction a lieu lorsque l'ascenseur a atteint soit l'étage le plus haut (ou le plus bas) ou lorsque son étage courant est égal à l'étage maximum (ou minimum) où il doit s'arrêter. Les deux règles gérant ces actions sont :

```
[ChangeToDown]
    O1:MLift::[State(Move(Up)) , F(0) , I(0)]
      =>
    O1(State<-Move(Down),F<-1)
      if O1.CF > Max(O1.LStop) or O1.CF == MaxLevel

[ChangeToUp]
    O1:MLift::[State(Move(Down)) , F(0) , I(0)]
      =>
    O1(State<-Move(Up),F<-1)
      if O1.CF < Min(O1.LStop) or O1.CF == MinLevel
```

De façon prioritaire par rapport aux déplacements, la tâche principale d'un ascenseur est de laisser sortir ses occupants à l'étage demandé ou encore d'ouvrir ses portes à des utilisateurs ayant appelé un ascenseur.

Premièrement, un ascenseur s'arrête lorsque l'étage où il se trouve est dans la liste des étages où il doit s'arrêter (cette liste est dans l'attribut LStop de chaque ascenseur). C'est la règle OpenDoorsStop :

```
[OpenDoorsStop]
    O1:MLift::[F(0) , I(0)]
        =>
    O1.RemoveLStop(O1.CF)(F<-1)
        if O1.State != Wait
        if in(O1.CF,O1.LStop)
```

Cette règle s'applique si l'ascenseur n'est pas déjà en train de réaliser une autre tâche (attribut F mis à 0) et si l'ascenseur est bien en état de fonctionnement (attribut I mis à 0). Elle vérifie que l'ascenseur est bien en mouvement (attribut State non mis à Wait) et le dernier test permet de vérifier que l'étage courant est bien dans la liste des étages où l'ascenseur doit stopper. Les actions réalisées sont le fait de supprimer l'étage ainsi desservi et d'indiquer que l'ascenseur réalise une action (attribut F mis à 1).

Ensuite, un ascenseur doit s'arrêter si l'étage où il se trouve est dans la liste des étages ayant appelé un ascenseur (attribut LCall du contrôleur) et que cet appel n'a pas encore été assigné. C'est le rôle de la règle OpenDoorsCall :

```
[OpenDoorsCall]
    O1:MLift::[F(0) , I(0)]
    O2:Call
       =>
    O1.AddLStop(L1)(F<-1)
    O2.RemoveLCall(O1.CF)
       if O1.State != Wait
       if in(O1.CF,O2.LCall)
       where L1 := () ObtainNewStops(O1.CF)
```

Les conditions de filtrage sur l'objet ascenseur sont les mêmes que pour la règle précédente. Cette règle vérifie que l'ascenseur est bien en mouvement et que l'étage courant est bien dans la liste des étages où un ascenseur a été appelé. Si c'est le cas, l'affectation locale permet de mettre dans une liste L1 les étages demandés par les nouveaux occupants de l'ascenseur par le biais de l'opérateur ObtainNewStops(_). Les objets mis en jeu sont ensuite modifiés de la façon suivante : l'ascenseur voit sa liste des nouveaux étages où il devra s'arrêter mise à jour en fonction de la liste L1, ainsi que son attribut F mis à 1 ; l'objet contrôleur est mis à jour en ôtant de sa liste LCall l'étage courant.

Par contre, lorsque l'ascenseur se trouve à un étage où il doit s'arrêter et que cet étage est dans la liste LCall du contrôleur, au lieu d'appliquer les deux règles précédentes successivement, on applique une seule règle OpenDoorsStopAndCall :

```
[OpenDoorsStopAndCall]
     O1:MLift::[F(0) , I(0)]
     O2:Call
       =>
     O1.AddLStop(L1).RemoveLStop(O1.CF)(F<-1)
     O2.RemoveLCall(O1.CF)
       if O1.State != Wait
       if in(O1.CF,O1.LStop)
       if in(O1.CF,O2.LCall)
       where L1 := () ObtainNewStops(O1.CF)
```

Cette règle est la composition des deux règles précédentes tant au niveau du filtrage que des tests, des affectations locales et des traitements.

Dans l'implantation, on a préféré composer les deux règles OpenDoorsStop et OpenDoorsCall en une seule règle OpenDoorsStopAndCall plutôt que d'utiliser le langage des stratégies qui nous permet lui aussi la composition des deux règles. En effet, l'écriture d'une seule règle évite, dans ce cas précis, d'avoir à renouveler les opérations de filtrage et de tests qui sont communes aux deux règles.

Une des caractéristiques de ce contrôleur d'ascenseurs est que la priorité est donnée à l'appel d'ascenseurs et que seulement lorsque ceux-ci sont attribués à un ascenseur donné, les demandes d'arrêts de la part des usagers d'un ascenseur donné sont réalisées. Ainsi, une personne attend peu de temps un ascenseur et, lorsqu'elle est dedans, est assurée d'en sortir rapidement avant que d'autres demandes qui ne sont pas sur le trajet soient prises en compte.

Un appel est assigné à un ascenseur dont l'attribut `State` est `Wait`. Ceci est fait grâce à la règle `AssignACall` :

```
[AssignACall]
    O1:MLift::[State(Wait) , F(0) , I(0)]
    O2:Call
       =>
    O1.WhichSense(NextFloor).AddLStop(NextFloor.nil)(F<-1)
    O2.AddAssignedCall(NextFloor).RemoveLCall(NextFloor)
     if O2.LCall != nil
     where NextFloor := () ChooseNextFloor(O1.CF,O2.LCall)
```

L'assignation se fait lorsque l'ascenseur est dans un état d'attente, lorsqu'il n'a alors plus aucun passager à faire sortir, et qu'il est dans des conditions normales de fonctionnement (`F=0` et `I=0`). Cette règle ne s'applique que lorsque la liste contenant les appels (attribut `LCall` du contrôleur) n'est pas vide. Dans ce cas, on affecte alors un appel à cet ascenseur par l'opérateur `ChooseNextFloor(_,_)` qui choisit dans la liste des appels l'étage le plus proche de l'étage courant de l'ascenseur. Les objets contrôleur et ascenseur sont alors modifiés : l'ascenseur ajoute cet appel dans la liste de ses arrêts et décide alors de son sens de déplacement pour l'atteindre ; le contrôleur enlève cet étage de la liste des appels et le met dans la liste des appels en cours de traitement.

Lorsqu'un ascenseur est à un étage qui correspond à un appel, on applique alors la règle `OpenDoors-AssignedCall` :

```
[OpenDoorsAssignedCall]
    O1:MLift::[F(0) , I(0)]
    O2:Call
       =>
    O1.AddLStop(L1).RemoveLStop(O1.CF)(F<-1)
    O2.RemoveAssignedCall(O1.CF)
       if O1.State != Wait
       if in(O1.CF,O2.AssignedCall)
       where L1 := () ObtainNewStops(O1.CF)
```

Cette règle s'applique pour tout ascenseur en état de fonctionnement et lorsque la condition d'appartenance de l'étage courant à la liste des appels est vérifiée. Dans ce cas, par l'opérateur `ObtainNewStops(_)`, on demande les nouveaux stops de l'ascenseur et les objets sont mis à jour : l'ascenseur ajoute ces stops à sa liste et enlève l'étage courant de sa liste d'étages à atteindre tandis que le contrôleur ôte cet étage courant des appels en cours de traitement.

Toutes ces règles s'appliquent avec des conditions fortes sur l'état de l'ascenseur : soit il doit être en mouvement, soit il doit être en attente. Des tests sont donc réalisés sur la valeur de l'attribut `State`. Par contre, aucune des règles précédentes ne modifie cet attribut. La règle `Wait` est alors définie par :

```
[Wait]  O1:MLift::[F(0) , I(0)]
          =>
        O1(State<-Wait)
          if O1.State != Wait
          if O1.LStop = nil
```

En effet, un ascenseur passe d'un état de déplacement à l'attente d'un nouvel appel lorsque sa liste d'arrêts est vide.

Dans cette partie, on vient de définir les règles manipulant la base d'objets de classe `MLift` ou `Call`. Ces règles sont toutes nommées, c'est-à-dire que leur application doit être contrôlée par la définition de stratégies.

En effet, si ces règles n'avaient pas été nommées, elles se seraient appliquées suivant la stratégie de normalisation *innermost* implantée en ELAN. Cela entraîne la pose de points de choix lors de l'évaluation

lorsque plusieurs filtres sont possibles. De plus, aucun contrôle n'est alors faisable et, par exemple, une même règle pourrait s'appliquer en boucle sur la base d'objets, ou alors le moteur d'inférence pourrait chercher à appliquer des règles de prises en charge de nouveaux appels alors que des personnes attendent encore dans un ascenseur pour en sortir. Ce n'est pas ce type de mécanisme d'application que nous souhaitons : nous voulons définir exactement un ordre dans l'application des règles qui correspond à une gestion bien précise du parc des ascenseurs. Ainsi, l'utilisation des stratégies s'avère être un mécanisme fort judicieux.

Les stratégies

Généralement, l'application d'une règle sur la base d'objets peut retourner plusieurs résultats. Ceci arrive notamment lorsque l'on obtient plusieurs filtres lors du filtrage des objets de la base avec le membre gauche de la règle. C'est le cas de bon nombre des règles précédentes. Pour introduire du déterminisme (on souhaite avoir un contrôleur qui réponde rapidement et qui n'hésite pas entre plusieurs solutions) et un contrôle sur l'application des règles, on utilise des stratégies. Les stratégies permettent de contrôler l'application des règles en utilisant l'application séquentielle, le choix, la répétition, etc...

Pour l'exemple du contrôleur d'ascenseurs, plusieurs stratégies ont ainsi été définies afin de permettre un comportement cohérent du système. Elles correspondent à des critères que l'on s'est donnés quant au comportement des ascenseurs et portant sur les priorités que l'on fixe, le mode de fonctionnement des ascenseurs. Il peut être modifié au gré des besoins et l'utilisation des stratégies est un moyen simple et efficace qui nous permet de rapidement mettre au point un nouveau mode opératoire et de le tester.

La première stratégie, `ONELIFT`, est une stratégie qui s'applique à un instant t pour un ascenseur donné. Elle est définie comme suit :

```
[] ONELIFT   => first(  AssignACall ,
                        OpenDoorsAssignedCall ,
                        OpenDoorsStopAndCall ,
                        OpenDoorsCall ,
                        OpenDoorsStop ,
                        ChangeSenseToDown ,
                        ChangeSenseToUp ,
                        Up ,
                        Down)
end
```

Elle tente dans un premier temps d'assigner un appel à un ascenseur en attente ; ensuite, on regarde si à l'étage courant de l'ascenseur, les portes doivent être ouvertes lorsque 1- l'étage est un appel assigné, 2- l'étage est dans la liste des stops et est assigné, 3- l'étage est un appel mais qui n'a pas encore été assigné, 4- l'étage est dans la liste des arrêts ; finalement, c'est seulement une fois que le trafic de passagers est résolu et qu'on a la certitude qu'on ne peut rien faire, que l'on décide d'un mouvement de l'ascenseur, soit on modifie le sens du mouvement, soit on continue dans un sens ou dans l'autre.

Cette stratégie est appliquée sur la base d'objets tant que tous les ascenseurs n'ont pas leur attribut `F` mis à `1` indiquant ainsi que l'ascenseur est déjà en train de réaliser une des neuf actions, chacune représentée par une des règles précédentes. Afin de réitérer cette stratégie `ONELIFT` sur l'ensemble des ascenseurs, on définit la stratégie `ALLLIFTS` :

```
[] ALLLIFTS  => repeat*(Wait) ;
                repeat*(ONELIFT) ;
                repeat*(RemoveFlag)
end
```

Cette stratégie cherche tout d'abord à rendre disponible le maximum d'ascenseurs par une application éventuelle de `Wait` ; ensuite, on applique répétitivement `ONELIFT` jusqu'à ce que ce ne soit plus possible ; finalement, on remet tous les attributs `F` à `0` par l'itération sur la règle `RemoveFlag` afin de pouvoir recommencer en réappliquant la stratégie globale :

```
[RemoveFlag] M1 Z => XX1 Z
                if M1.F == 1
                where XX1 := () M1(F<-0)
end
```

On garantit ainsi que chaque ascenseur tente de réaliser une action à chaque application de la stratégie ALLLIFTS, assurant une répartition équitable du travail entre chaque ascenseur.

Afin de passer d'une situation initiale où certains appels sont à traiter à une situation finale où tous les appels sont traités et où tous les ascenseurs sont vides, la stratégie ALLLIFTS doit être appliquée plusieurs fois. Pour cela, nous définissons la stratégie MAIN qui répète la règle Main jusqu'à ce que la base de données n'évolue plus:

```
[] MAIN => first one (repeat*(Main))
end
```

où la règle Main est donnée par:

```
[Main] ST => ST1
          where ST1 := (ALLLIFTS) ST
          if ST1 != ST
```

L'ensemble ce ces stratégies permet de contrôler l'application des règles sur la base d'objets. Nous avons donc maintenant défini l'ensemble des composants de cette application: les OModules, les règles avec objets et les stratégies. Nous pouvons maintenant expliquer comment l'exécution se déroule.

L'exécution

Considérons un immeuble disposant de trois ascenseurs et de vingt-six étages compris entre le niveau 0 et le niveau 25.

Dans une situation initiale, les trois ascenseurs se trouvent en attente aux étages 2, 11 et 14. Le contrôleur vient de réceptionner quatre appels depuis les étages 3, 9, 17 et 24. Cela se retrouve dans l'état de la base d'objet suivante où nous avons quatre objets: les objets O(1), O(2) et O(3) sont les objets représentant les ascenseurs, l'objet O(4) représente le contrôleur.

```
O(1):MLift::[CF(14) , State(Wait) , Zone(1) , LStop(nil) , F(0) , I(0)]
O(2):MLift::[CF(11) , State(Wait) , Zone(1) , LStop(nil) , F(0) , I(0)]
O(3):MLift::[CF(2)  , State(Wait) , Zone(0) , LStop(nil) , F(0) , I(0)]
O(4):Call::[AssignedCall(nil) , LCall(3.9.17.24.nil)]
```

A partir de cet état initial, on va afficher l'état de la base d'objets après chaque application de la stratégie ALLLIFTS. Nous donnons ici les deux premiers affichages jusqu'à la première sollicitation de l'utilisateur:

```
O(1):MLift::[CF(14) , State(Move(Up))   , Zone(1) , LStop(17.nil) , F(0) , I(0)]
O(2):MLift::[CF(11) , State(Move(Down)) , Zone(1) , LStop(9.nil)  , F(0) , I(0)]
O(3):MLift::[CF(2)  , State(Move(Up))   , Zone(0) , LStop(3.nil)  , F(0) , I(0)]
O(4):Call::[AssignedCall(3.9.17.nil) , LCall(24.nil)]

O(1):MLift::[CF(15) , State(Move(Up))   , Zone(1) , LStop(17.nil) , F(0) , I(0)]
O(2):MLift::[CF(10) , State(Move(Down)) , Zone(1) , LStop(9.nil)  , F(0) , I(0)]
O(3):MLift::[CF(3)  , State(Move(Up))   , Zone(0) , LStop(3.nil)  , F(0) , I(0)]
O(4):Call::[AssignedCall(3.9.17.nil) , LCall(24.nil)]

An elevator is stopped at level 3, please enter the desired stops
as a list of sorted integers separated by . and terminated by end:
```

Expliquons les deux étapes de l'évolution de la base d'objets :

1. A partir de l'état initial, le contrôleur a affecté trois des quatre appels à des ascenseurs : le 17ème est affecté à l'ascenseur initialement au 14ème étage (cet ascenseur entre en mouvement vers le haut) ; le 9ème est affecté à celui au 11ème étage (cet ascenseur entre en mouvement vers le bas) et le 3ème étage est affecté à l'ascenseur positionné au 2ème (cet ascenseur entre en mouvement vers le haut). Les listes des étages attribués (attribut `AssignedCall`) ou non encore attribués (attribut `LCall`) de l'objet `Call` sont elles aussi mises à jour.
2. Les ascenseurs ont ensuite continué leurs déplacements dans les sens qui leur ont été attribués. L'ascenseur désigné par `O(3)` arrive ainsi dès le deuxième état à un étage où il devait s'arrêter. En utilisant le mécanisme des entrées/sorties d'ELAN, on demande alors si de nouveaux étages sont demandés par les nouveaux entrants dans l'ascenseur. On rentre alors une liste de nouveaux étages correspondant au fait que les étages 7 et 10 ont été demandés par des utilisateurs nouvellement entrés dans l'ascenseur :

```
7.10.nil
```

La base d'objet évolue alors en l'état suivant :

```
O(1):MLift::[CF(16) , State(Move(Up))   , Zone(1) , LStop(17.nil)   , F(0) , I(0)]
O(2):MLift::[CF(9)  , State(Move(Down)) , Zone(1) , LStop(9.nil)    , F(0) , I(0)]
O(3):MLift::[CF(3)  , State(Move(Up))   , Zone(0) , LStop(7.10.nil) , F(0) , I(0)]
O(4):Call::[AssignedCall(9.17.nil) , LCall(24.nil)]

An elevator is stopped at level 9, please enter the desired stops
as a list of sorted integers separated by . and terminated by end:
```

Le deuxième ascenseur arrive à l'étage souhaité, le 9ème. Le 3ème ascenseur a bien sa liste des stops mise à jour. L'exécution peut ainsi continuer pas à pas. Nous montrons ici les deux derniers états de la base d'objet pour une exécution donnée :

```
O(1):MLift::[CF(17) , State(Wait)     , Zone(1) , LStop(nil) , F(0) , I(0)]
O(2):MLift::[CF(10) , State(Wait)     , Zone(1) , LStop(nil) , F(0) , I(0)]
O(3):MLift::[CF(24) , State(Move(Up)) , Zone(2) , LStop(nil) , F(0) , I(0)]
O(4):Call::[AssignedCall(nil) , LCall(nil)]

O(1):MLift::[CF(24) , State(Wait) , Zone(2) , LStop(nil) , F(0) , I(0)]
O(2):MLift::[CF(17) , State(Wait) , Zone(1) , LStop(nil) , F(0) , I(0)]
O(3):MLift::[CF(10) , State(Wait) , Zone(1) , LStop(nil) , F(0) , I(0)]
O(4):Call::[AssignedCall(nil) , LCall(nil)]
```

Le processus s'arrête lorsque tous les ascenseurs sont passés dans un état d'attente (attribut `State` mis à `Wait`) et lorsque le contrôleur n'a plus aucun étage à distribuer aux ascenseurs.

On peut remarquer que l'ordre des objets de la base est indifférent. Ainsi, leurs noms peuvent évoluer en cours de traitement puisque les noms des noms des objets sont locaux à chaque règle et lorsqu'ils apparaissent dans les affichages, ils ne sont liés qu'à la façon de programmer cet affichage.

Dans cette partie, nous avons ainsi défini un formalisme de règle manipulant une base d'objets et comment ce formalisme pouvait être utilisé dans la programmation d'applications en ELAN.

4.2 Programmation avec règles, stratégies, objets et contraintes

Afin de définir un environnement permettant d'intégrer les contraintes au formalisme précédent, nous devons tout d'abord définir le concept de contraintes utilisé. Ensuite, nous devons montrer comment toutes ces entités interagissent, quelle est l'architecture du système et où le mécanisme des stratégies intervient.

S.C.D. - U.H.P. NANCY 1
BIBLIOTHÈQUE DES SCIENCES
Rue du Jardin Botanique
54600 VILLERS-LES-NANCY

Pour qu'on puisse formaliser la résolution de contraintes et les tests de satisfaisabilité, la structure d'une base de contraintes (*constraint store*) peut devenir très complexe. On pourrait bien sûr modéliser les contraintes grâce aux classes d'objets : chaque classe correspondrait à une sorte de contraintes. On pourrait alors associer à chaque classe un solveur de contraintes dédié mais alors, apparaîtrait le problème de la combinaison des solveurs de contraintes [NO79, Rin96, Mon99].

Dans le formalisme présenté ici, nous avons choisi de ne pas détailler la structure de base de contraintes et le solveur associé mais plutôt de considérer cette structure comme prédéfinie. Nous sommes donc amené à définir des mécanismes de communication permettant de gérer cette base de contraintes.

4.2.1 Le formalisme des CSP

Le Problème de Satisfaction de Contraintes peut être défini très simplement comme le problème d'affectation d'un ensemble de variables à partir d'un ensemble de valeurs possibles de telle sorte que l'ensemble des contraintes portant sur cette variable soit satisfait.

Dans cette section, on présente une définition formelle du Problème de Satisfaction de Contraintes (*Constraint Satisfaction Problem* ou CSP) basée sur les concepts de l'algèbre universelle. Même si la définition d'un CSP comme étant un triplet $\langle X,D,C\rangle$ est largement utilisée dans la littérature, la formalisation que l'on présente ici facilite sa compréhension d'un point de vue conceptuel.

Définition 4.2 *Une* contrainte élémentaire $c^?$ *est une formule atomique construite à partir d'une signature* $\Sigma = (\mathcal{F},\mathcal{P})$ *et d'un ensemble dénombrable de symboles de variables* $\mathcal{X}$. *Les contraintes élémentaires peuvent être combinées avec le connecteur de conjonction* $\wedge$. *On désigne l'ensemble de contraintes formées à partir de* Σ *et* $\mathcal{X}$ *par* $\mathcal{C}(\Sigma,\mathcal{X})$. *Étant donnés une signature* $\Sigma = (\mathcal{F},\mathcal{P})$, *un ensemble de symboles de variables* $\mathcal{X}$ *et une structure* $\mathcal{D} = (D,I)$, *un* $\langle\Sigma,\mathcal{X},\mathcal{D}\rangle$*-CSP est une contrainte* $C = (c_1^? \wedge \ldots \wedge c_n^?)$ *tel que* $c_i^? \in \mathcal{C}(\Sigma,\mathcal{X}); \forall i = [1,\ldots,n]$ [4].

De la même manière, on peut définir ce qu'est un CSP avec disjonctions :

Définition 4.3 *Etant donnés une signature* $\Sigma = (\mathcal{F},\mathcal{P})$, *un ensemble de symboles de variables* $\mathcal{X}$ *et une structure* $\mathcal{D} = (D,I)$, *un* $\langle\Sigma,\mathcal{X},\mathcal{D}\rangle$*-CSP avec disjonctions est un ensemble formé par des contraintes élémentaires et par leur combinaison avec les connecteurs de conjonction* $\wedge$ *et de disjonction* $\vee$.

On désigne la contrainte C par $C = (c_1^? \wedge \ldots \wedge c_n^?)$ ou bien par $C = \{c_1^?,\ldots,c_n^?\}$. Le nombre de contraintes dans le problème est désigné par n ($n = Card(C)$). L'ensemble des variables libres dans une contrainte $c^?$ est désigné par $\mathcal{V}ar(c^?)$; elles représentent les variables contraintes par $c^?$. L'arité d'une contrainte $c^?$ est définie comme le nombre de variables libres apparaissant dans la contrainte

$$\text{arité}(c^?) = Card(\mathcal{V}ar(c^?))$$

De cette façon on travaille avec un ensemble de contraintes indexées $C = \bigcup_{i\geq 0} C_i$, où C_i est l'ensemble de toutes les contraintes d'arité i. On désigne par e le nombre de variables qui apparaissent dans l'ensemble de contraintes ($e = Card(\mathcal{V}ar(C))$). Finalement, on désigne par a la cardinalité du domaine D ($a = Card(D)$).

Dans ce travail, nous considérerons des CSPs dans lesquels le domaine de la structure est de cardinalité finie. Ce type particulier de CSP est appelé CSP sur des domaines finis.

Définition 4.4 *Une solution d'une contrainte* $c^?$ *est une application* α *de* $\mathcal{X}$ *vers* D *qui associe à chaque variable* $x \in \mathcal{X}$ *un élément dans* D *tel que* $\alpha(c^?)$ *est vraie dans* $\mathcal{D}$. *Une contrainte est satisfaisable dans* D *si elle a au moins une solution dans* D. *L'ensemble de toutes les solutions de* $c^?$ *est défini comme suit*

$$Sol_{\mathcal{D}}(c^?) = \{\alpha \in \alpha_{\mathcal{D}}^{\mathcal{X}} | \alpha(c^?) = \mathbf{V}\}$$

Une solution dans $\mathcal{D}$ *d'un ensemble de contraintes* C *est une solution de toutes les contraintes* $c_i^? \in C$. *L'ensemble de toutes les solutions de* C *est défini par*

4. Pour des raisons de clarté, dans ce travail les contraintes son distinguées syntaxiquement des formules par un point d'interrogation sur leur symbole de prédicat. Le symbole ? explicite le fait que l'on cherche les solutions de la formule c.

$$Sol_{\mathcal{D}}(C) = \bigcap_{c_i^? \in C} Sol_{\mathcal{D}}(c_i^?)$$

Comme une solution est une affectation des variables, les contraintes jouent le rôle de filtres pour les combinaisons d'instanciations des variables qui apparaissent dans les contraintes.

Les tâches typiques concernant les CSPs sont de trouver une ou toutes les solutions ou bien de déterminer l'insatisfaisabilité de l'ensemble de contraintes.

Finalement, la définition suivante nous permettra de conceptualiser d'une manière simple la réduction de l'ensemble de valeurs prises par les variables.

Définition 4.5 *Etant donnés une variable $x \in \mathcal{X}$ et un ensemble non-vide $D_x \subseteq D$, la* contrainte d'appartenance *de x est définie par $x \in^? D_x$. Un $\langle\Sigma,\mathcal{X},\mathcal{D}\rangle$-CSP C' avec* contraintes d'appartenance *est $C' = C \bigcup \{x \in D_x\}_{x \in^? \mathcal{X}}$, où C est un $\langle\Sigma,\mathcal{X},\mathcal{D}\rangle$-CSP.*

Nous utiliserons ces contraintes d'appartenance pour stocker dynamiquement l'ensemble des solutions d'un CSP et pour expliciter la réduction de l'ensemble des valeurs possibles pour les variables réalisée pendant le processus de résolution. En pratique, les ensembles $D_{x_{x \in \mathcal{X}}}$ seront initialisés à D au début du processus de résolution du problème et ils seront éventuellement réduits durant ce processus.

En ajoutant ces contraintes d'appartenance, on rend explicite, d'un point de vue conceptuel, la vérification de consistance. Dans la littérature standard on utilise le terme *réduction du domaine* pour désigner l'élimination de valeurs pour les variables qui sont inconsistantes par rapport à une certaine contrainte. Puisque le domaine est fixé lors de l'interprétation, le terme réduction du domaine n'est pas approprié. En utilisant les contraintes d'appartenance le domaine reste inchangé et ce sont les ensembles associés à chaque variable à travers les contraintes d'appartenance qui changent durant le processus de résolution des contraintes.

Comme l'ensemble D_x, associé par la contrainte d'appartenance à la variable x du problème, stocke toutes les valeurs potentielles prises par la variable, on peut définir l'espace de recherche d'un CSP à partir de ces contraintes d'appartenance.

Définition 4.6 *Etant donné $C \bigcup \{x \in D_x\}_{x \in^? \mathcal{X}}$ un CSP avec des contraintes d'appartenance, l'espace de recherche est défini comme le produit cartésien $D_{x_1} \times \ldots \times D_{x_n}$ de tous les ensembles D_{x_i}.*

L'espace de recherche d'un CSP peut être représenté par un arbre.

4.2.2 Une base de contraintes

Dans le langage de contraintes considéré dans le formalisme de règles avec objets et contraintes, on se restreint aux contraintes sur des domaines finis. Une contrainte c est alors composée de :

- $VarList$, une liste de contraintes d'appartenance de la forme $X \in^? D$ où X est une variable dite contrainte et D le domaine fini associé.
- $ConsSet$, un ensemble de conjonctions et de disjonctions de contraintes atomiques. Ces contraintes atomiques sont soit les constantes booléennes *true* ou $false$, soit des contraintes d'égalité ($t_1 =^? t_2$) et d'inégalité($t_1 ! =^? t_2$), soit des contraintes d'ordre ($t_1 >^? t_2$, $t_1 \geq^? t_2$, $t_1 <^? t_2$, $t_1 \leq^? t_2$) ou soit les négations des contraintes d'égalité, d'inégalité et d'ordre. Les termes t_1 et t_2 impliqués dans les contraintes atomiques sont construits à partir des variables contraintes et peuvent aussi inclure les opérateurs arithmétiques $+$, $-$, $*$ ou $/$.

La paire ($VarList$, $ConsSet$) est un CSP.

Exemple 4.4 *Considérons trois variables V_1, V_2 et V_3 prenant leurs valeurs dans le domaine fini $[0,\ldots,10]$ et la contrainte suivante $V_1 + V_2 =^? V_3 \wedge V_1 \geq^? 4 \wedge V_2 \leq^? 8 \wedge V_3 >^? 2$. Le CSP correspondant est composé de deux parties :*

$$VarList = V_1 \in^? [0,..,10], V_2 \in^? [0,..,10], V_3 \in^? [0,..,10]$$

$$ConsSet = V_1 + V_2 =^? V_3 \wedge V_1 \geq^? 4 \wedge V_2 \leq^? 8 \wedge V_3 >^? 2$$

Dans le formalisme développé et utilisé ici, une base de contraintes est simplement un ensemble de Problèmes de Satisfaction de Contraintes correspondant à la conjonction des contraintes et à l'ensemble des contraintes d'appartenance. Des opérations usuelles sont supposées être disponibles sur la base de contraintes, telles que l'obtention de la liste des variables, l'obtention du domaine courant d'une variable, le test d'égalité d'un domaine de variables avec l'ensemble vide (dans ce cas, le Problème de Satisfaction de Contraintes est insatisfaisable), l'ajout d'une nouvelle contrainte, le test de satisfaisabilité du Problème de Satisfaction de Contraintes, la demande de résolution du Problème de Satisfaction de Contraintes de même que l'obtention d'une ou de toutes les solutions au problème.

Il est maintenant courant de décrire les processus de résolution de contraintes par des règles de résolution qui expriment comment transformer une contrainte en une forme résolue ou comment simplifier une contrainte. C'est cette approche qui est suivie pour construire des résolveurs de contraintes dans les *Constraint Handling Rules* [Frü99] ou par des règles de réécriture dans [KR98].

En effet, T. Frühwirth a défini les *Constraint Handling Rules* (CHR) comme un langage spécialement destiné au développement de solveurs de contraintes. Les CHR sont des règles qui permettent de réécrire les contraintes en de nouvelles contraintes plus simples jusqu'à ce qu'elles soient résolues. Les techniques de simplification et de propagation sont ainsi implantées : les premières permettant de remplacer les contraintes par d'autres plus simples tout en conservant l'équivalence logique et les secondes permettent d'ajouter de nouvelles contraintes qui sont logiquement redondantes mais peuvent conduire par la suite à des simplifications intéressantes. Les règles des CHR sont présentées comme une généralisation des différentes constructions présentes dans CHIP [Sim95] pour des contraintes définies par l'utilisateur. Une règle de simplification ou de propagation des CHR s'applique à une contrainte lorsque son membre gauche filtre avec la contrainte et lorsque les préconditions sont vérifiées. Il existe différentes implantations des CHR dans des langages comme Prolog, Lisp ou encore Java.

Résoudre des Problèmes de Satisfaction de Contraintes comme défini ci-dessus peut aussi être formalisé en utilisant des règles de réécritures avec des stratégies comme cela a été montré par C. Kirchner et C. Ringeissen dans [KR98]. L'avantage de cette approche réside dans le fait d'avoir une représentation uniforme du processus de résolution de contraintes ainsi que du méta-langage utilisé pour manipuler les contraintes.

C'est une approche à base de règles et de stratégies qui a été utilisée dans les travaux de C. Castro [Cas98a] que nous détaillons plus particulièrement dans la partie 4.2.6.

Nous supposons ici que la sémantique opérationnelle du résolveur de contraintes et du vérificateur de satisfaisabilité sont décrites par des règles simplifiant des contraintes. Dans le formalisme de règles avec objets et contraintes, on considère un résolveur de contraintes avec les propriétés suivantes :

- lorsque l'on teste la satisfaisabilité d'une contrainte, on suppose que le résolveur retourne que la contrainte est satisfaisable (`Satisfiable`) ou insatisfaisable (`Unsatisfiable`). Si le résolveur répond "je ne sais pas", on retourne `Unsatisfiable`.
- lorsque des solutions sont demandées au solveur, on ne fait pas d'hypothèse quant à la complétude du solveur.

4.2.3 Des règles contraintes

A la règle incorporant condition et objet de la section 4.1.2 sont apportées certaines modifications qui permettent d'inclure les contraintes au formalisme :

$$[lab]\ \ O_1 \ldots O_k \ \Rightarrow\ O'_1 \ldots O'_m \parallel c\ \ [\textbf{if}\ \ t \mid \textbf{where}\ \ l]^*$$

où c est une contrainte locale.

En ce qui concerne les objets, le traitement est le même que dans le cas des règles manipulant uniquement une base d'objets présentée dans la section 4.1.2 : les conditions d'application de la règle sont les mêmes, les traitements des objets modifiés, contextuels, nouveaux et supprimés aussi.

La contrainte locale est quant à elle composée de contraintes d'appartenance et de conjonctions et de disjonctions de contraintes atomiques. Syntaxiquement, les contraintes d'appartenance $X \in^? D$ sont notées `X in? D`; la conjonction $\wedge$ est notée `&` tandis que la disjonction $\vee$ est notée `V`; les contraintes

d'ordre $t >^? s$, $t \geq^? s$, $t <^? s$ et $t \leq^? s$ sont quant à elles respectivement notées `t>?s`, `t>=?s`, `t<?s` et `t<=?s`. Les symboles arithmétiques sont quant à eux notés avec des parenthèses autour du symbole: $4 + 2$ est noté `4(+)2`.

Lorsqu'une règle est appliquée, la contrainte locale est ajoutée à la base de contraintes. Un test de satisfaisabilité n'est pas automatiquement fait lors de l'ajout. Le test est demandé via le mécanisme des stratégies ; cela répond donc à une demande de l'utilisateur. Quand une nouvelle contrainte est ajoutée à la base, l'ensemble des contraintes d'appartenance est mis à jour par l'ajout des contraintes d'appartenance concernant les nouvelles variables et les contraintes sont ajoutées à l'ensemble de contraintes.

Comme des variables contraintes sont associées à des objets, nous décrivons ici le comportement des règles combinant objets et contraintes. On appelle attribut contraint d'un objet tout attribut dont la valeur est susceptible d'apparaître dans une contrainte. La valeur d'un attribut contraint peut être un élément appartenant au domaine de l'attribut. A la place d'un élément de ce domaine, on peut aussi associer à un attribut une variable, dite variable contrainte. Cette variable contrainte prendra elle aussi ses valeurs dans le domaine autorisé tout en respectant la contrainte d'appartenance qui lui est liée. Ainsi, pour un attribut contraint `X` prenant ses valeurs entières dans le domaine `[2,6]`, une valeur peut être l'élément `5` mais la valeur peut aussi être une variable contrainte `V` prenant ses valeurs dans le même domaine `[2,6]`.

Considérons un objet O_1 d'une classe C avec un attribut contraint AT apparaissant dans le membre gauche de la règle. Etant donné un objet $O_1 : C :: [AT(X)]$ dans le membre gauche de la règle, la variable X peut seulement être instanciée par une variable contrainte. Après la phase de filtrage, on peut effectuer deux types de tests sur la variable contrainte X :

- $CVEqual(X,v)$ appelle le résolveur de contraintes et teste si la variable contrainte X peut prendre la valeur v tout en considérant le domaine de la variable X dans les contraintes d'appartenance.
- $CVDef(X)$ appelle le résolveur de contraintes et vérifie que la variable contrainte X n'a pas un domaine vide.

Ces tests doivent s'évaluer en *true* afin que l'application de la règle puisse être déclenchée. Si ces tests s'évaluent en *false*, soit la variable contrainte ne peut pas prendre la valeur testée pour le premier type de test, soit la variable contrainte a un domaine de définition vide pour le second. Dans les deux cas, la règle pour laquelle le test est effectué ne s'applique pas.

En prenant en compte les tests décrits ci-dessus, on peut donner une forme plus précise des règles de réécriture conditionnelles avec objets et contraintes telles qu'elles ont été présentées au début de cette section. On obtient alors la forme générale suivante :

$$
\begin{aligned}
[lab] \quad & O_1 \ldots O_k \Rightarrow O'_1 \ldots O'_m \parallel c \\
& [\textbf{if } t \mid \\
& \textbf{if } CVEqual(V_1,v) \mid \\
& \textbf{if } CVDef(V_2) \mid \\
& \quad \textbf{where } l]^*
\end{aligned}
$$

Exemple 4.5 *Considérons un déplacement d'une ville à une autre que l'on souhaite réaliser en un temps donné Time. On définit l'intervalle de temps D comme étant l'intervalle de 0 à Time. On dispose pour ce problème d'une carte avec les différentes villes et les estimations des trajets entre deux villes voisines ; ces trajets sont dits élémentaires. Le problème consiste à décomposer un problème éventuellement non élémentaire en trajets élémentaires afin d'avoir une estimation de la durée totale du trajet qui doit pouvoir être inférieure, voire égale, au temps imparti Time.*

La classe `Travel` est définie avec quatre attributs : la ville de départ (attribut `From`), la ville d'arrivée (attribut `To`), la date de départ (attribut `Dep`) et la date d'arrivée (attribut `Arr`). Les deux derniers attributs sont des attributs contraints. Cette classe définit aussi des méthodes qui seront détaillées plus loin.

La règle ci-dessous décrit la décomposition d'un trajet non élémentaire `X1` en deux autres trajets `X2` et `X3` :

```
[] X1:Travel::[Dep(T)] => X2(From<-X1.From)(To<-Town)(Dep<-V(1))(Arr<-V(2))
                          X3(From<-Town)(To<-X1.To)(Dep<-V(2))(Arr<-V(3))
        || (V(1) in? D, V(2) in? D, V(3) in? D)
           V(1)>=?X1.Dep & V(3)<=?X1.Arr & V(2)>=?V(1) & V(2)<=?V(3)
              if CVEqual(T,Time)
              if X1.NonElementaryTravel
              where Town := () X1.SelectTown
              where X2 := () TravelClass.new
              where X3 := () TravelClass.new
end
```

Lorsque cette règle est appliquée, l'objet $X1$ est remplacé par les nouveaux objets `X2` et `X3`.

Cette règle est sélectionnée pour n'importe quel objet O de la classe `Travel` appartenant à la base d'objets et qui filtre le membre gauche de règle. La variable `T` est instanciée avec la variable contrainte associée à l'attribut contraint `Dep` de O. La première condition de la règle consiste à vérifier que la variable contrainte peut prendre la valeur `Time` et qu'ainsi, on respecte bien les contraintes de durée du trajet. La seconde condition consiste à appeler la méthode `NonElementaryTravel` de la classe `Travel` qui permet de tester si le trajet désigné par l'objet `X1` est bien non élémentaire et décomposable. Si l'une de ces deux conditions n'est pas respectée, la règle ne s'applique pas. Si ce n'est pas le cas, la variable locale `Town` est alors instanciée par une nouvelle ville obtenue à partir de la méthode `SelectTown` appliquée à l'objet `X1` et qui sélectionne sur la carte une ville entre les deux villes de départ et d'arrivée de `X1`.

Les deux nouveaux objets `X2` et `X3` sont alors créés. Leurs attributs `From` et `To` peuvent être instanciés ; par contre, on ne connaît pas les valeurs des attributs `Dep` et `Arr`. On associe à ces attributs contraints des variables contraintes `V(1)`, `V(2)` et `V(3)` qui apparaissent dans la contrainte locale de la règle.

La contrainte locale est composée de deux parties. La première définit les domaines associés à chaque nouvelle variable contrainte ; ici, le domaine est `D`. La deuxième partie donne les contraintes assurant le fait que les dates d'arrivées et de départs correspondent bien aux dates requises par `X1` et qui permettent aussi de minimiser la durée du trajet. La contrainte locale est ajoutée à la base de contraintes si la règle est appliquée.

4.2.4 Les stratégies

Généralement, l'application des règles sur les objets peut retourner plusieurs résultats lorsque, par exemple, plusieurs objets, ou un multi-ensemble d'objets, filtrent le membre gauche de règle. Cela induit du non-déterminisme et le besoin de contrôler l'application des règles. C'est la raison pour laquelle la notion de stratégie est utilisée ici.

Les stratégies sont utilisées dans différents cas :

- pour contrôler l'application des règles sur la mémoire de travail : les stratégies permettent de définir des applications séquentielles de stratégies ou de règles, de faire des choix entre différentes stratégies ou règles pouvant s'appliquer, d'itérer, etc... Dans la section 4.2.7, nous donnons un exemple complet de règles avec objets et contraintes ainsi que les stratégies associées aux règles.
- pour contrôler le processus de résolution de contraintes ; en particulier, des stratégies spécifiques permettant de résoudre les contraintes peuvent être définies. Par exemple, le solveur de contraintes sur les domaines finis utilisé dans l'application décrite dans la section 4.2.6 est basé sur des techniques de propagation locale et d'énumération avec des retours arrière (*backtracking*). Le langage de stratégies offre des constructeurs tels qu'ils permettent d'avoir une grande flexibilité pour définir des stratégies appropriées à la résolution, pour tester la consistance ou pour calculer toutes ou partie des solutions. Cet aspect est particulièrement développé dans la section 4.2.6.
- pour contrôler l'interaction entre les règles et le résolveur de contraintes ou le testeur de satisfaisabilité : elle permettent de choisir entre un test de satisfaisabilité de la contrainte, la génération d'une seule ou de toutes les solutions ou encore de spécifier exactement lorsque les solutions sont nécessaires. Par la suite, on suppose l'existence d'au moins trois stratégies : `Sat` qui permet de tester la satisfaisabilité d'un Problème de Satisfaction de Contraintes ; `OneSolution` qui permet de

calculer la première solution d'un Problème de Satisfaction de Contraintes et `AllSolutions` qui permet d'énumérer toutes les solutions calculées par la stratégie.

4.2.5 L'architecture du système

L'architecture globale du système avec les relations entre mémoire d'objets, base de contraintes et l'utilisation des stratégies est résumée dans la Figure 4.1.

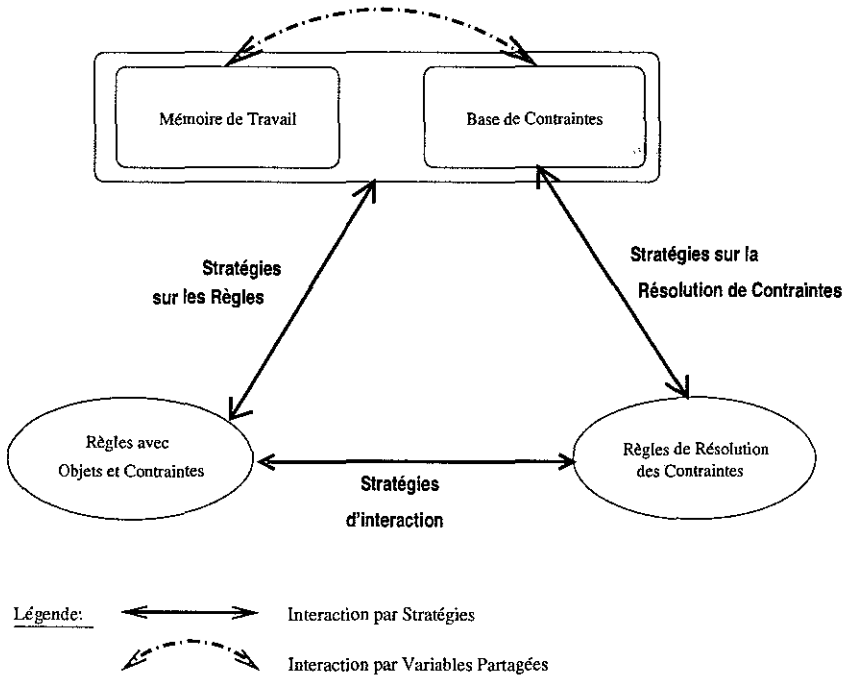


FIG. 4.1 – *Architecture du système*

Dans cette figure, les flèches en pointillés représentent les liens entre la mémoire d'objets et la base de contraintes à travers les attributs contraints et les variables contraintes associées : une variable contrainte peut apparaître à la fois en tant que valeur d'attribut et en même temps en tant que variable dans un Problème de Satisfaction de Contraintes et ainsi apparaître dans une contrainte locale aux règles.

4.2.6 Un résolveur de contraintes en ELAN : COLETTE

Afin de résoudre les Problèmes de Satisfaction de Contraintes, on utilise le système COLETTE réalisé par C. Castro [Cas98a, Cas98b] et écrit en ELAN. Le solveur COLETTE est implanté par des règles de réécriture et utilise le langage de stratégies d'ELAN. En raison des grandes similitudes entre notre système de règles avec objets et contraintes et ce système, nous avons réutilisé COLETTE comme une boîte noire sans modifier son code.

La base de contraintes et les contraintes locales aux règles sont traduites dans le langage de contraintes accepté par le solveur qui les transforme et donne soit des solutions soit une réponse sur la satisfaisabililité des contraintes. Les solutions doivent ensuite être remises dans le formalisme que l'on utilise.

Nous présentons dans un premier temps les primitives offertes par COLETTE ainsi que son fonctionnement global. Dans un deuxième temps, nous donnons une formalisation des mécanismes de communication qui nous permettent de dialoguer entre notre application écrite en ELAN et le solveur COLETTE vu comme une entité indépendante.

Une librairie pour résoudre des CSP

Le solveur COLETTE peut être vu comme une librairie à base de règles et de stratégies définie pour traiter et résoudre des Problèmes de Satisfaction de Contraintes.

Plusieurs techniques et algorithmes utilisés dans la communauté des Problèmes de Satisfaction de Contraintes ont été implantés dans le solveur. Parmi les algorithmes que le système fournit, nous pouvons en citer deux plus particulièrement : celui de *Full Look Ahead* et celui de *Forward Checking*.

Dans l'algorithme de *Full Look Ahead*, une fois qu'une variable est instanciée, le système teste la consistance globale du Problèmes de Satisfaction de Contraintes avant de chercher à en instancier une nouvelle variable. On donne ici le code de la stratégie d'une version de *Full Look Ahead* appelée `FLAFirstToLastAll` qui énumère toutes les solutions de la première à la dernière :

```
strategies for csp
implicit

/* Strategy : Full Lookahead */
/* Value selection : Value enumeration first to last */
/* Number of solutions : All */

[] FLAFirstToLastAll =>
LocalConsistencyForEC ;
repeat* (
dk (iterate* (EliminateFirstValueOfDomain)) ;
first one (InstantiateFirstValueOfDomain) ;
first one (ExtractConstraintsOnEqualityVar , id) ;
first one (Elimination , id) ;
LocalConsistencyForEC
) ;
first one (GetSolutionCSP)
end
end
```

La stratégie globale enchaîne différentes sous-stratégies : après un test de consistance locale, on répète le traitement suivant : recherche d'une variable à instancier, instanciation de cette variable, extraction des contraintes d'égalité portant sur cette variable, élimination et test de consistance locale pour contraintes élémentaires sur le nouveau Problème de Satisfaction de Contraintes engendré. Ce traitement est répété jusqu'à obtention d'une solution globale en posant les points de choix nécessaires pour obtenir toutes les solutions au problème.

L'algorithme de *Forward Checking* est une version restreinte de l'algorithme précédent : à chaque fois qu'une variable est instanciée, au lieu de tester la consistance globale de la contrainte, on ne teste la consistance que sur les contraintes qui ont été modifiées par le remplacement d'une variable par sa valeur lors de l'application de la règle `Elimination`. Le code de la stratégie est le suivant et diffère finalement très peu de la stratégie précédente si ce n'est l'appel de la stratégie `first one (LocalConsistencyInEC , id)` à la place de `LocalConsistencyForEC`, ce qui correspond à la différence de traitement entre ces deux algorithmes :

```
strategies for csp
implicit

/* Strategy : Forward Checking   */
/* Value selection : Value enumeration first to last */
/* Number of solutions : All */
```

```
[] FCFirstToLastAll =>
LocalConsistencyForEC ;
repeat* (
dk (iterate* (EliminateFirstValueOfDomain)) ;
first one (InstantiateFirstValueOfDomain) ;
first one (ExtractConstraintsOnEqualityVar, id) ;
first one (Elimination , id) ;
first one (LocalConsistencyInEC , id)
) ;
first one (GetSolutionCSP)
end
end
```

Différentes versions de ces algorithmes implantent des heuristiques comme par exemple la stratégie `FLAMinimumDomainFirstToLastAll` qui cherche à chaque fois que l'on doit choisir une nouvelle variable à instancier celle qui a le plus petit domaine. D'autres stratégies implantées sont par exemple des algorithmes de type *Generate and Test* ou de type *Backtracking*.

COLETTE offre aussi tout un ensemble de stratégies pour tester les consistances locales ou globales d'un Problème de Satisfaction de Contraintes. On a par exemple la stratégie `LocalConsistencyForEC` qui vérifie la consistance locale pour un ensemble de contraintes élémentaires :

```
strategies for csp
implicit

[] LocalConsistencyForEC =>
repeat* (
first one (ArcConsistency) ;
repeat*   (first one (
    first one (Instantiation) ;
    first one (ExtractConstraintsOnEqualityVar , id) ;
            first one (Elimination , id)
            ,
    first one (Falsity)
          )
  )
)
end
end
```

Cette stratégie procède comme suit : après chaque application de la règle `ArcConsistency` qui simplifie les domaines des variables, nous pouvons obtenir deux résultats possibles. Soit l'ensemble de valeurs prises par une variable a été réduit mais contient encore des éléments, soit l'ensemble est vide. Afin de détecter ces deux situations, il suffit de tester l'insatisfaisablité du problème maintenant considéré en essayant d'appliquer la règle `Falsity`. Cet algorithme peut être amélioré en considérant le cas où l'ensemble des valeurs prises par une variable a été réduit à un seul élément : on peut alors appliquer la règle `Instantiation` puis `Elimination` afin de simplifier l'ensemble des contraintes où cette variable instanciée peut encore apparaître.

Une autre technique fréquemment utilisée pour tester la consistance d'une contrainte consiste à engendrer la première solution et, dès qu'on l'a obtenue, à répondre que le Problème de Satisfaction de Contraintes est satisfaisable. On peut ainsi utiliser des stratégies comme `FCFirstToLastOne` ou encore `FLAFirstToLastOne` qui sont des versions des stratégies présentées ci-dessus où l'on ne rend que la première solution lorsqu'elle existe.

Le formalisme de CSP dans COLETTE

Dans cette section, on détaille la représentation des Problèmes de Satisfaction de Contraintes dans COLETTE.

Afin de traiter des problèmes de contraintes sur des domaines finis d'entiers, les contraintes d'appartenance, les contraintes d'égalité et l'opérateur de conjonction sont définis par les symboles de fonctions suivants :

```
operators global

   @ in? @    : (var domain) constraint;
   @ = @      : (var term) constraint;
   @ & @      : (constraint constraint) constraint (AC) pri 20;
end
```

La structure `domain` permet de définir les domaines bornés d'entiers. Le Problème de Satisfaction de Contraintes est quant à lui formalisé par la structure suivante :

$$CSP[lmc,lec,EC,DC,store]$$

où :

- *lmc* est une liste qui contient les contraintes d'appartenance où des domaines d'entiers sont associés à des variables.
- *lec* est une liste qui contient les contraintes d'égalité.
- *EC* est une liste qui contient les contraintes élémentaires à résoudre.
- *DC* est une liste qui contient les contraintes disjonctives à résoudre.
- *store* est une liste qui contient les contraintes élémentaires déjà vérifiées.

Des mécanismes de communication

Les communications entre le solveur COLETTE et les règles de notre formalisme se font à deux niveaux : on doit tout d'abord faire une correspondance entre les stratégies utilisées dans les systèmes de règles avec contraintes. On doit aussi faire une correspondance entre le formalisme de contraintes utilisée dans les règles et celui utilisé dans COLETTE.

Dans un premier temps, afin de définir le contrôle par stratégies du solveur de contraintes sur les contraintes que nous manipulons, nous devons faire le lien entre les stratégies `Sat`, `OneSolution` et `AllSolutions` déjà mentionnées dans la partie 4.2.4 et les stratégies de COLETTE définies ci-dessus.

Ce lien peut être défini via les stratégies `Sat`, `OneSolution` et `AllSolutions` :

```
[] Sat          => LocalConsistencyForEC
end
[] OneSolution  => first one(FCFirstToLastAll)
end
[] AllSolutions => FCFirstToLastAll
end
```

qui utilisent un appel aux stratégies implantées dans COLETTE et présentées dans la section précédente. Néanmoins, ces définitions ne sont pas figées et tout utilisateur peut à son gré utiliser n'importe quelle autre stratégie implantée en COLETTE s'il la juge plus adaptée au problème considéré.

Ensuite, pour passer du formalisme de contraintes de la forme (*VarList* , *ConsSet*) au formalisme utilisé dans COLETTE pour manipuler des Problèmes de Satisfaction de Contraintes qui est quant à lui de la forme $CSP[lmc,lec,EC,DC,store]$, on définit la primitive $CSP2csp(\_)$ qui, à partir du formalisme (*VarList* , *ConsSet*), rend une contrainte manipulable par COLETTE.

4.2.7 Exemple : un gestionnaire d'impressions

Afin d'illustrer cette partie sur les règles impliquant contraintes et objets, nous allons étudier le principe d'un gestionnaire d'impressions. Le gestionnaire d'impressions tel qu'il est présenté dans cet exemple s'inspire de travaux de recherche menés chez Xerox [ABC+98].

Le principe général du contrôleur d'impressions est d'aider l'utilisateur dans la décomposition de tâches complexes en une séquence de tâches dites primitives afin de déterminer quelle suite d'actions primitives est compatible avec un ensemble de contraintes. Les tâches complexes doivent être décomposées en tâches primitives dans la mesure du possible.

A chaque pas de la décomposition, on doit être en mesure de vérifier que la décomposition satisfait l'ensemble de contraintes courant. On doit alors donner l'ensemble de toutes les décompositions possibles à ce stade. On doit pouvoir aussi explorer l'ensemble des décompositions possibles afin de faire le meilleur choix possible.

Les contraintes utilisées dans cette étude sont des contraintes temporelles : les tâches d'impression doivent être réalisées avant une date limite. Elles doivent aussi respecter un certain ordonnancement suivant les types d'impressions réalisés : en effet, si on lance deux impressions du même document sur une imprimante, on doit d'abord assurer un chargement en mémoire de ce document avant d'effectuer la première puis la seconde impression. Le déchargement du document de la mémoire n'intervient qu'à la fin de l'impression de l'ensemble.

Les classes d'objets

Les objets utilisés dans cet exemple sont de deux types : certains objets représentent les tâches complexes tandis que d'autres objets représentent les tâches primitives. On appelle *action* toute tâche primitive et *tâche* toute tâche complexe.

La classe `Action` spécifie les actions et cette classe est définie par le module objet suivant :

```
class Action

imports TName ConstraintAttribute

attributes Name:TName = InitName
           B:ConstraintAttribute = InitB
           E:ConstraintAttribute = InitE
End
```

Les trois attributs composant la classe `Action` sont :

- `Name` qui prend ses valeurs dans l'ensemble `Tname` qui est importé. Cet ensemble est composé des différents noms d'actions possibles ainsi que du nom `InitName` qui sert à la création d'un nouvel objet de cette classe.
- `B` qui prend ses valeurs dans l'ensemble `ConstraintAttribute` importé. Cet attribut représente le temps de début d'une action. Les temps de début prennent leurs valeurs dans l'ensemble des entiers mais aussi dans un ensemble de variables contraintes. Le temps de début d'une action peut ne pas être connu dans l'absolu, mais dépendre des temps d'exécution d'autres actions ou tâches ; c'est la contrainte qui détient ces informations et la contrainte peut lier l'action via l'attribut contraint associé à `B`. L'attribut est initialisé avec une valeur donnée par `InitB`.
- `E` représente quant à lui le temps de fin de l'action. Il prend lui aussi ses valeurs dans l'ensemble importé `ConstraintAttribute` et son initialisation est faite avec une valeur `InitE`.

La durée de l'action pourrait être mise en attribut des objets de la classe `Action`, mais ce n'est pas ce choix d'implantation qui a été fait. Le choix qui a été fait consiste à prendre en compte les durées des actions dans les règles de décomposition. Ces deux choix sont totalement équivalents du point de vue de la spécification du problème.

La classe Task donne quant à elle la définition des tâches :

```
class Task

from Action

imports TStatus

attributes NumI:int = InitNumI
           Status:TStatus = InitStatus
End
```

Comme les objets de la classe Task ont les mêmes attributs Name, B et E que ceux définis dans la classe Action, la classe Task hérite de la classe Action. Néanmoins, à ces attributs s'en ajoutent deux autres qui sont NumI et Status :

- l'attribut NumI prend ses valeurs dans les entiers et sert à indiquer le nombre de jobs d'impressions du même document que l'on doit gérer. Cet attribut est initialisé avec une valeur initiale appelée InitNumI qui est donnée par les décompositions antérieures ou à l'initialisation du problème.
- l'attribut Status prend ses valeurs dans l'ensemble TStatus et est initialisé avec la valeur InitStatus. Cet attribut sert à définir un statut pour les tâches : soit on a une tâche d'impression qui est décomposable en plusieurs et dans ce cas, le statut de la tâche est split ; soit la tâche d'impression doit être gérée sans décomposition en d'autres tâches d'impressions et dans ce cas le statut est all.

Le problème consiste donc à trouver les différents ordonnancements possibles pour exécuter N travaux d'impression donnés au début. Dans cette décomposition, on a le choix entre différentes possibilités qui sont : décomposer la tâche en plusieurs autres tâches ou bien gérer toutes les impressions en une même tâche.

On doit ici noter que l'on ne cherche pas à minimiser le temps d'exécution de l'ensemble des impressions mais bien à générer l'ensemble des ordonnancements possibles respectant l'ensemble des contraintes afin d'aider un administrateur ou un utilisateur dans les choix qu'il peut être amené à faire.

Une vue générale du problème est donnée dans le schéma de la Figure 4.2 où une tâche consistant à gérer N impressions est décomposée suivant les deux possibilités déjà présentées.

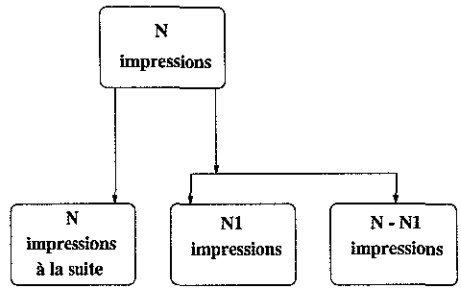


FIG. 4.2 – *Principe de décomposition des tâches du gestionnaire d'impressions.*

Les règles de décomposition

Afin de définir de façon plus précise le mécanisme de décomposition, des règles manipulant objets et contraintes ont été définies. Elles sont au nombre de six et permettent de décomposer des tâches en d'autres tâches ou en actions : deux règles permettent de définir la décomposition principale, on les appelle Main1 et Main2 ; deux règles gèrent la gestion de N travaux d'impressions comme indiqué dans la Figure 4.2, ce sont les règles appelées FP1 et FP2 ; deux dernières règles appelées MP1 et MP2 gèrent l'impression de N travaux à la suite.

Les règles Main1 et Main2 sont données ci-dessous :

```
[Main1] O1:Task::[Name(Main)]
        =>
         O2(Name<-FormPrint)(Status<-all)(NumI<-O1.NumI-1)(B<-V_1)(E<-V_2)
         O3(Name<-SimplePrint)(B<-V_3)(E<-V_4)
          || (V_1 in? D, V_2 in? D, V_3 in? D, V_4 in? D)
              V_2<=?O1.E & V_4<=?O1.E & V_4=?V_3(+)2 &
              ((V_1>=?O1.B & V_3=?O1.B & V_4=?V_1) V
               (V_1=?O1.B & V_3>?O1.B & V_2=?V_3))
         where O2 := () TaskClass.new
         where O3 := () ActionClass.new

[Main2] O1:Task::[Name(Main)]
        =>
         O2(Name<-FormPrint)(Status<-split)(NumI<-O1.NumI-1)(B<-V_1)(E<-V_2)
         O3(Name<-SimplePrint)(B<-V_3)(E<-V_4)
          || (V_1 in? D, V_2 in? D, V_3 in? D, V_4 in? D)
              V_1>=?O1.B & V_3>=?O1.B & V_2<=?O1.E & V_4<=?O1.E &
              V_4=?V_3(+)2 & V_3>=?V_1 & V_4<=?V_2
         where O2 := () TaskClass.new
         where O3 := () ActionClass.new
```

Dans chacune de ces deux règles, un objet de la classe Task dont l'attribut Name vaut Main est supprimé de la base d'objets et remplacé par deux nouveaux objets O2 et O3 : la tâche Main est décomposée en une tâche et en une action représentées par les deux nouveaux objets O2 et O3 qui sont respectivement des nouveaux objets des classes Task et Action.

A partir d'une tâche initiale Name traitant N impressions, on crée une tâche FormPrint qui consiste à imprimer ces N impressions moins une NumI<-O1.NumI-1 entre un temps de début V_1 et un temps de fin V_2. Le travail d'impression que n'a pas à gérer cette tâche est pris en compte par l'action SimplePrint qui s'exécute entre un temps de début V_3 et un temps de fin V_4. En ce qui concerne les déclarations de domaines des variables contraintes V_1 à V_4, elles sont toutes initialisées avec un domaine D de la forme $[0,\ldots,T]$ spécifique au problème où T est le temps final au delà duquel tout doit être terminé. Ce qui différencie ces deux règles, ce sont les contraintes et le statut affecté à la tâche FormPrint : soit on fait toutes les $N-1$ impressions à la suite (statut all), soit on peut les découper (statut split) :

- dans le cas du statut all, tâche et action doivent se terminer avant le temps de fin autorisé (V_2<=?O1.E & V_4<=?O1.E) et la durée de l'action est donnée par la contrainte V_4=?V_3(+)2. Par contre, la tâche peut s'exécuter soit avant l'action (V_1=?O1.B & V_3>?O1.B & V_2=?V_3), soit après l'action (V_1>=?O1.B & V_3=?O1.B & V_4=?V_1).

- dans le cas du statut split, tâche et action doivent commencer après le temps de début autorisé (V_1>=?O1.B & V_3>=?O1.B), elles doivent se terminer avant le temps de fin autorisé (V_2<=?O1.E & V_4<=?O1.E), la durée de l'action est donnée par la contrainte V_4=?V_3(+)2 et l'action peut s'intercaller entre deux découpages de la tâche (V_3>=?V_1 & V_4<=?V_2).

Les règles FP1 et FP2 gèrent la décomposition des tâches FormPrint et sont définies par :

S.C.D. - U.H.P. NANCY 1
BIBLIOTHÈQUE DES SCIENCES
Rue du Jardin Botanique
54600 VILLERS-LES-NANCY


```
[FP1] O1:Task::[Name(FormPrint) , Status(all)]
      O2:Action::[Name(SimplePrint)]
      =>
       O2
       O3(Name<-Load)(B<-V_1)(E<-V_2)
       O4(Name<-MultiPrint)(NumI<-O1.NumI)(B<-V_3)(E<-V_4)
       O5(Name<-FormKeep)(B<-V_5)(E<-V_6)
        || (V_1 in? D,V_2 in? D,V_3 in? D,V_4 in? D,V_5 in? D,V_6 in? D)
            V_1=?O1.B & V_2=?V_1(+)1 & V_3=?V_2 & V_4=?O1.E & V_5=?V_1 &
            V_6=?V_4 & (O2.B>=?V_4 V O2.E<=?V_1)
       where O3 := () ActionClass.new
       where O4 := () TaskClass.new
       where O5 := () ActionClass.new

[FP2] O1:Task::[Name(FormPrint) , Status(split)]
      =>
       O2(Name<-FormPrint)(NumI<-N1)(Status<-st1)(B<-V_1)(E<-V_2)
       O3(Name<-FormPrint)(NumI<-N2)(Status<-st2)(B<-V_3)(E<-V_4)
        || (V_1 in? D , V_2 in? D , V_3 in? D , V_4 in? D)
            V_1=?O1.B & V_2<=?V_3 & V_3>?V_1 & V_4=?O1.E
       where O2 := () TaskClass.new
       where O3 := () TaskClass.new
       where [N1,N2]   := () Split(O1.NumI)
       where [st1,st2] := () ChooseStatus
```

La première règle, FP1, décompose une tâche en deux nouvelles actions (les objets O3 et O5) et une nouvelle tâche (objet O4). Imprimer un document N fois (information contenue dans l'attribut NumI de O1) de telle sorte que les impressions se succèdent, consiste tout d'abord à charger le document en mémoire (action Load), puis à procéder aux N impressions (tâche MultiPrint) ; le tout s'assurant que l'on conserve en mémoire le document tout au long des travaux (action FormKeep). La contrainte locale de la règle prend en compte ces relations : on commence par charger en mémoire le document (V_1=?O1.B) ; la durée de l'action de chargement est une unité de temps (V_2=?V_1(+)1) ; on procède ensuite aux impressions (V_3=?V_2 & V_4=?O1.E) en s'assurant que le document reste en mémoire du début (V_5=?V_1) à la fin (V_6=?V_4). La tâche FormPrint dans sa globalité peut se positionner avant l'action SimplePrint (O2.B>=?V_4) ou bien après (O2.E<=?V_1).

La seconde règle décompose la tâche en deux sous-tâches FormPrint. Le nombre d'impressions géré par chaque sous-tâche est défini par la fonction Split qui, à partir d'un nombre N donné en paramètre, donne tous ses découpages possibles en deux nombres non nuls $N1$ et $N2$ et tels que $N = N1 + N2$. Chaque application de la règle utilise un résultat de cette fonction non-déterministe. Le statut de chaque sous-tâche est défini à l'aide de la fonction ChooseStatus qui génère tous les couples possibles de statut : [all,all], [all,split], [split,all] et [split,split]. Chaque application de la règle utilise là aussi un seul résultat de cette fonction non-déterministe. La contrainte locale associée à la règle consiste à faire démarrer la première sous-tâche au début de la tâche initiale (V_1=?O1.B) et à faire arrêter la deuxième sous-tâche à la fin (V_4=?O1.E). Les deux autres contraintes atomiques consistent à ordonnancer les deux sous-tâches l'une par rapport à l'autre (V_2<=?V_3 & V_3>?V_1).

Les deux dernières règles concernent l'impression des documents les uns à la suite des autres une fois qu'ils sont déjà chargés en mémoire. Le principe est ici très simple : pour imprimer N fois un même document, on l'imprime déjà une fois puis on l'imprime $N - 1$ fois. Ceci est exprimé par les deux règles MP1 et MP2 :

```
[MP1] O1:Task::[Name(MultiPrint) , NumI(1)]
      =>
       O2(Name<-Print)(B<-V_1)(E<-V_2)
        || (V_1 in? D , V_2 in? D)
            V_1=?O1.B & V_2=?O1.E & V_2=?V_1(+)1
       where O2 := () ActionClass.new

[MP2] O1:Task::[Name(MultiPrint) , NumI(N)]
      =>
       O2(Name<-Print)(B<-V_1)(E<-V_2)
       O3(Name<-MultiPrint)(NumI<-N-1)(B<-V_3)(E<-V_4)
        || (V_1 in? D , V_2 in? D , V_3 in? D , V_4 in? D)
            V_1=?O1.B & V_2=?V_1(+)1 & V_3=?V_2 & V_4=?O1.E
       if N>1
       where O2 := () ActionClass.new
       where O3 := () TaskClass.new
```

Les deux règles s'expliquent aisément : la première règle `MP1` consiste à engendrer une seule action d'impression lorsqu'on n'a plus qu'une seule impression à réaliser. La contrainte locale consiste à définir le début d'action comme étant égal au début de la tâche (`V_1=?O1.B`) et la fin de l'action comme étant égale à la fin de la tâche (`V_2=?O1.E`). La durée de l'action est égale à une unité de temps (`V_2=?V_1(+)1`). La seconde règle `MP2` n'est exécutée que si le nombre d'impressions est supérieur strictement à 1. Dans ce cas, on génère une sous-tâche et une action, l'action se déroulant avant la sous-tâche (`V_3=?V_2`). Le temps de début de la tâche coïncide avec le temps de début de l'action (`V_1=?O1.B`) et de fin de la tâche avec le temps de fin de la sous-tâche (`V_4=?O1.E`). La durée de l'action est définie de la même manière que dans la première règle.

Tout ceci conduit à un schéma général des transformations de tâches en sous-tâches et actions que l'on trouvera en Figure 4.3.

Les stratégies

Différentes stratégies sont définies afin de contrôler l'application de ces règles sur la base d'objets.

Les premières stratégies contrôlent directement l'application des six règles précédentes, ce sont les stratégies `Main`, `FP` et `MP`. `Main` est une stratégie permettant d'explorer toutes les possibilités dans l'application des règles `Main1` et `Main2`. La stratégie `FP` permet quant à elle d'exploiter tous les choix possibles des règles `FP1` et `FP2`. La dernière stratégie, `MP`, tente tout d'abord d'appliquer la règle `MP1` avant d'essayer ensuite la règle `MP2` en cas d'échec de `MP1`.

Les définitions de ces stratégies sont :

```
[] Main  => dk (Main1 , Main2)
end
[] FP    => dk (F1 , F2)
end
[] MP    => first (MP1 , MP2)
end
```

Ensuite, on peut décider d'explorer l'arbre de recherche sans faire de test de satisfaisabilité sur les contraintes et d'énumérer les solutions seulement lorsque l'on est arrivé à une feuille de l'arbre. Ce type d'exploration est réalisé par la stratégie `EnumerateWithoutCheck` qui utilise aussi la stratégie `AllSolutions` présentée dans la Section 4.2.6 :

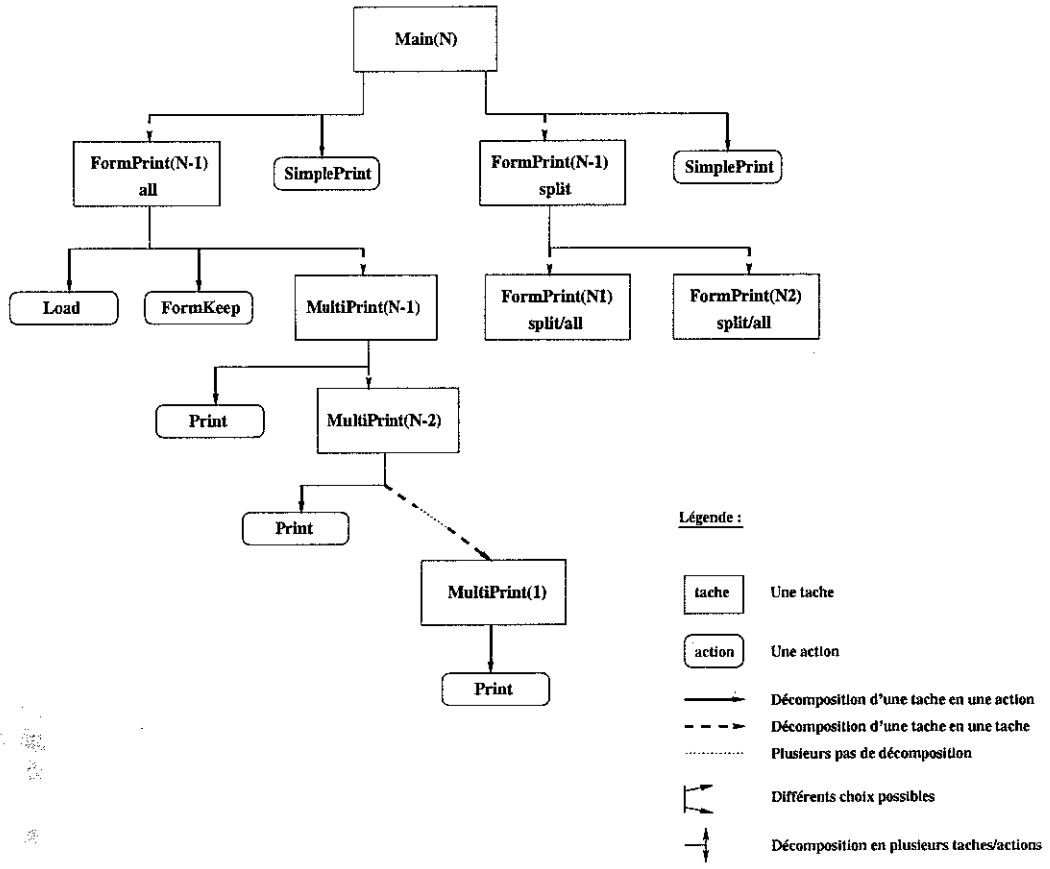


FIG. 4.3 – *Le principe général du contrôleur d'impressions.*

```
[] EnumerateWithoutCheck => Main;repeat*(FP;repeat*(MP));AllSolutions
end
```

On peut par contre décider de réaliser un test de satisfaisabilité sur l'ensemble de contraintes après chaque décomposition, ceci est réalisé par la stratégie `EnumerateAndTest` :

```
[] EnumerateAndTest => Main;Sat;repeat*(FP;Sat);repeat*(MP;Sat)
end
```

La flexibilité offerte par le langage de stratégie permet de tester facilement un nouvel enchaînement des règles. On peut par exemple considérer tout type d'alternance entre des stratégies traitant de décomposition dans la base d'objets et des stratégies opérant sur l'ensemble de contraintes.

Exécution du programme

On souhaite maintenant exécuter l'ensembles de règles et de stratégies sur un problème donné. Différentes stratégies sont testées dans cette partie.

La première stratégie consiste à énumérer toutes les décompositions, c'est la stratégie `Enumerate` :

```
[] Enumerate => Main;repeat*(FP;repeat*(MP));
end
```

Cette stratégie ne fait aucun appel au résolveur de contraintes, elle ne fait appel qu'à la partie manipulant la base d'objets.

La deuxième stratégie est la stratégie `EnumerateAndCheck` définie par :

```
[] EnumerateAndCheck => Main;repeat*(FP;repeat*(MP));Sat
end
```

Cette stratégie consiste à engendrer l'ensemble des décompositions possibles sans tenir compte de la satisfaisabilité des contraintes à chaque nœud de l'arbre. La satisfaisabilité n'est testée qu'à chaque feuille de l'arbre. Aucun ordonnancement n'est alors retourné : on utilise la stratégie `Sat` en effet et non une stratégie énumérant les solutions comme `AllSolutions` par exemple. On est par contre assuré qu'il existe au moins un ordonnancement possible à chaque feuille grâce au test de satisfaisabilité.

La dernière stratégie est `EnumerateWithoutCheck` déjà présentée dans la section précédente. On rappelle que cette stratégie procède à toutes les décompositions possibles et qu'elle engendre à chaque feuille tous les ordonnancements possibles répondant au problème.

On donne dans le tableau suivant différents résultats. On donne tout d'abord le type d'exemples avec le nombre de tâches d'impressions demandées et l'intervalle de temps considéré. On donne ensuite les temps d'exécution et le nombre de pas de réécriture effectués en considérant les stratégies `Enumerate`, `EnumerateAndCheck` et enfin `EnumerateWithoutCheck`. La première ne fait pas appel au résolveur de contraintes, la seconde uniquement pour le test de satisfaisabilité (obtention d'une première solution) et la dernière pour calculer l'ensemble des solutions. On donne aussi le nombre de décompositions et d'ordonnancements différents pour chaque cas considéré.

Exemple		Décompositions (pas d'appel au solveur)			Décompositions valides (test de satisfaisabilité)			Ordonnancements (toutes les solutions)		
Nombre d'impressions	Intervalle de temps	Nbre	Temps (en sec.)	Nbre d'étapes de réécriture	Nbre	Temps (en sec.)	Nbre d'étapes de réécriture	Nbre	Temps (en sec.)	Nbre d'étapes de réécriture
3	[0,...,6]	6	0.060	2 043	6	3.510	420 909	6	5.350	682 841
4	[0,...,6]	38	0.540	17 442	2	97.600	9 427 726	2	94.970	9 442 233
4	[0,...,8]	38	0.540	17 442	38	96.520	9 765 407	38	192.170	20 716 180

Tests réalisés sur un Pentium III, 500MHz avec 128Mo de RAM

Vers plus de flexibilité

L'utilisation de la méthode précédente présuppose que nous ayons déjà une stratégie bien déterminée à l'esprit pour pouvoir lancer le programme avec cette stratégie préalablement implantée. Il peut par contre s'avérer utile de pouvoir dérouler l'arbre pas à pas afin de décider à chaque nœud de la règle ou de la stratégie que l'on veut alors suivre. On peut alors voir l'outil ELAN comme un outil d'aide à la décision.

Une telle utilisation du système ELAN suppose la prise en compte d'entrées/sorties gérées par l'interpréteur. Les performances sont alors moins bonnes, notamment dans la gestion des contraintes, comparées à l'utilisation du compilateur comme dans la section précédente.

Dans l'implantation réalisée, on considère qu'à chaque pas, l'utilisateur se voit proposer le menu suivant :

```
Could you give us the number associated to the strategy
you want now to execute (terminated by 'end')?:
 1- Main
 2- FormPrint
 3- MultiPrint
 4- Satisfiability Test
 5- All Results
 6- One Result
 7- Cut this branch
```

Les trois premiers cas permettent respectivement d'appeler une des trois stratégies `Main`, `FP` ou `MP`. Ces trois premiers cas permettent de développer l'arbre de recherche en décomposant une tâche à chaque fois que cela est faisable. Le quatrième cas permet d'appeler le test de satisfaisabilité sur l'état courant de la base de contraintes. Les cinquième et sixième cas demandent la satisfaction de la contrainte en vue d'obtenir soit un résultat, soit l'ensemble des résultats. Le septième et dernier cas permet à l'utilisateur de couper la branche de l'arbre en cours de développement et de revenir au dernier point de choix posé avant

de continuer l'exploration. On peut tout aussi bien augmenter le nombre de choix offerts à l'utilisateur en configurant le menu différemment.

Ainsi, si l'on considère un problème avec quatre tâches d'impression à réaliser entre les temps 0 et 8, la mémoire d'objets n'est initialement constituée que de l'objet :

```
O(0):Task::[Name(Main) , NumI(4) , Status(InitStatus) , B(0) , E(8)]
```

Quant à la base de contraintes, elle est initialement vide et initialisée à `0||[0,..,8]||nil||true` où `0` désigne le nombre de variables contraintes déjà utilisées, `[0,...,8]` désigne le domaine de valeurs autorisées pour les variables contraintes, `nil` un ensemble intermédiaire utilisé lors des calculs sur la contrainte qui est quant à elle initialisée à `true` puisqu'aucune contrainte n'est encore définie.

Si l'on note par *base d'objets* `with` *base de contraintes* les deux ensembles manipulés par les règles, on a une situation initiale de la forme :

```
O(0):Task::[Name(Main) , NumI(4) , Status(InitStatus) , B(0) , E(8)]
with  0||[0,..,8]||nil||true
```

Avec les choix qui s'offrent à nous, on peut à ce moment, soit décomposer l'unique tâche par la stratégie `Main` (choix 1), soit tester la satisfaisabilité de la contrainte (choix 4), soit en donner une ou plusieurs solutions (choix 5 et 6), soit couper la branche (choix 7) mais ainsi arrêter l'exécution puisqu'aucun point de choix n'a encore été posé. On choisit donc la première possibilité qui nous mène à la situation suivante :

```
O(0):Action::[Name(SimplePrint) , B(V(3)) , E(V(4))]
O(1):Task::[Name(FormPrint) , NumI(3) , Status(all) , B(V(1)) , E(V(2))]
with  4||[0,..,8]||
      V(1)in?[0,..,8].V(2)in?[0,..,8].V(3)in?[0,..,8].V(4)in?[0,..,8].nil||
      V(1)>=?0&V(3)=?0&V(2)<=?8&V(4)<=?8&V(4)=?V(1)&V(4)=?V(3)(+)2
```

Dans cette situation, la tâche `Main` a été supprimée de la base d'objets et remplacée par une autre tâche (objet `O(1)`) et une action (objet `O(0)`) comme cela est exprimé dans la règle `Main1`. La contrainte est aussi mise à jour.

A ce moment là, le menu s'affiche à nouveau et le système nous demande de faire un nouveau choix parmi les sept. On choisit alors le second choix pour appeler la stratégie `FP` afin de décomposer la tâche présente. On obtient alors :

```
O(0):Action::[Name(FormKeep) , B(V(9)) , E(V(10))]
O(1):Action::[Name(Load) , B(V(5)) , E(V(6))]
O(2):Action::[Name(SimplePrint) , B(V(3)) , E(V(4))]
O(3):Task::[Name(MultiPrint) , NumI(3) , Status(InitStatus) , B(V(7)) , E(V(8))]
with  10||[0,..,8]||
      V(5)in?[0,..,8].V(6)in?[0,..,8].V(7)in?[0,..,8].V(8)in?[0,..,8].
      V(9)in?[0,..,8].V(10)in?[0,..,8].V(1)in?[0,..,8].V(2)in?[0,..,8].
      V(3)in?[0,..,8].V(4)in?[0,..,8].nil||
      V(1)>=?0&V(3)=?0&V(2)<=?8&V(4)<=?8&V(4)=?V(1)&V(4)=?V(3)(+)2&V(5)=?V(1)&
      V(6)=?V(5)(+)1&V(7)=?V(6)&V(8)=?V(2)&V(9)=?V(5)&V(10)=?V(8)&
      V(3)>=?V(8)V V(4)<=?V(5)
```

La décomposition suit celle définie par la règle `F1` : deux nouvelles actions sont engendrées (`FormKeep` et `Load`) ; la tâche `FormPrint` est remplacée par une autre tâche, à savoir `MultiPrint`. La contrainte est elle aussi mise à jour.

Si à ce moment, on veut savoir si la contrainte est satisfaisable, on entre le quatrième choix et on obtient :

```
This constraint is Satisfiable:
CSP[V(10)in?[0,..,8].V(2)in?[0,..,8].V(8)in?[0,..,8].nil,
    V(9)=2.V(7)=3.V(3)=0.V(4)=2.V(1)=2.V(5)=2.V(6)=3.nil,nil,nil,
    1(*)V(8)(+)0(+)0=?1(*)V(2)(+)0(+)0.1(*)V(10)(+)0(+)0=?1(*)V(8)(+)0(+)0.nil]
```

La contrainte est donc satisfaisable et l'ensemble des solutions est donné, exprimé dans la syntaxe CSP. Les variables contraintes dont les valeurs peuvent être d'ores et déjà fixées sont dans la seconde partie ; dans l'exemple, V(9)=2, V(7)=3, V(3)=0, V(4)=2, V(1)=2, V(5)=2 et V(6)=3. Les variables contraintes dont la valeur ne peut pas encore être fixée sont mises dans la première partie où leurs domaines sont exprimés ; ainsi, V(2), V(8) et V(10) prennent leurs valeurs dans [0,..,8]. Les contraintes relatives à ces variables contraintes non encore déterminées se trouvent dans la dernière partie.

On peut ainsi continuer à explorer l'ensemble de l'arbre jusqu'à tomber sur une feuille où plus aucune tâche n'est présente comme celle-ci :

```
O(0):Action::[Name(Print) , B(V(35)) , E(V(36))]
O(1):Action::[Name(Print) , B(V(33)) , E(V(34))]
O(2):Action::[Name(Print) , B(V(31)) , E(V(32))]
O(3):Action::[Name(FormKeep) , B(V(29)) , E(V(30))]
O(4):Action::[Name(Load) , B(V(25)) , E(V(26))]
O(5):Action::[Name(FormKeep) , B(V(23)) , E(V(24))]
O(6):Action::[Name(Load) , B(V(19)) , E(V(20))]
O(7):Action::[Name(FormKeep) , B(V(13)) , E(V(14))]
O(8):Action::[Name(Load) , B(V(9)) , E(V(10))]
O(9):Action::[Name(SimplePrint) , B(V(3)) , E(V(4))]
with  36||[0,..,8]||
      V(35)in?[0,..,8].V(36)in?[0,..,8].V(33)in?[0,..,8].V(34)in?[0,..,8].
      V(31)in?[0,..,8].V(32)in?[0,..,8].V(25)in?[0,..,8].V(26)in?[0,..,8].
      V(27)in?[0,..,8].V(28)in?[0,..,8].V(29)in?[0,..,8].V(30)in?[0,..,8].
      V(19)in?[0,..,8].V(20)in?[0,..,8].V(21)in?[0,..,8].V(22)in?[0,..,8].
      V(23)in?[0,..,8].V(24)in?[0,..,8].V(15)in?[0,..,8].V(16)in?[0,..,8].
      V(17)in?[0,..,8].V(18)in?[0,..,8].V(9)in?[0,..,8].V(10)in?[0,..,8].
      V(11)in?[0,..,8].V(12)in?[0,..,8].V(13)in?[0,..,8].V(14)in?[0,..,8].
      V(5)in?[0,..,8].V(6)in?[0,..,8].V(7)in?[0,..,8].V(8)in?[0,..,8].
      V(1)in?[0,..,8].V(2)in?[0,..,8].V(3)in?[0,..,8].V(4)in?[0,..,8].nil||
      V(1)>=?0&V(3)>=?0&V(2)<=?8&V(4)<=?8&V(4)=?V(3)(+)2&V(3)>?V(1)&V(4)<?V(2)&
      V(5)=?V(1)&V(7)>?V(1)&V(6)<=?V(7)&V(2)=?V(8)&V(9)=?V(7)&V(10)=?V(9)(+)1&
      V(11)=?V(10)&V(12)=?V(8)&V(13)=?V(9)&V(14)=?V(12)&V(3)>=?V(12)V V(4)<=?V(9)&
      V(15)=?V(5)&V(17)>?V(5)&V(16)<=?V(17)&V(6)=?V(18)&V(19)=?V(17)&
      V(20)=?V(19)(+)1&V(21)=?V(20)&V(22)=?V(18)&V(23)=?V(19)&V(24)=?V(22)&
      V(3)>=?V(22)V V(4)<=?V(19)&V(25)=?V(15)&V(26)=?V(25)(+)1&V(27)=?V(26)&
      V(28)=?V(16)&V(29)=?V(25)&V(30)=?V(28)&V(3)>=?V(28)V V(4)<=?V(25)&
      V(31)=?V(27)&V(32)=?V(31)(+)1&V(28)=?V(32)&V(33)=?V(21)&V(34)=?V(33)(+)1&
      V(22)=?V(34)&V(35)=?V(11)&V(36)=?V(35)(+)1&V(12)=?V(36)
```

Une demande de test de satisfaisabilité nous donne le résultat suivant où l'on voit que la contrainte est satisfaisable et où aucune variable contrainte ne peut être fixée même si les domaines ont été fortement réduits pour bon nombre des trente-six variables contraintes. Cette fois-ci, les contraintes associées n'ont pas été reproduites et elles ont été remplacées par "...":

```
This constraint is Satisfiable:
CSP[V(3)in?[1,..,5].V(9)in?[4,..,6].V(24)in?[4,..,6].V(18)in?[4,..,6].V(13)in?[4,..,6].
V(19)in?[2,..,4].V(31)in?[1,..,3].V(23)in?[2,..,4].V(7)in?[4,..,6].V(15)in?[0,..,2].
V(11)in?[5,..,7].V(20)in?[3,..,5].V(5)in?[0,..,2].V(6)in?[4,..,6].V(2)in?[6,..,8].
V(10)in?[5,..,7].V(33)in?[3,..,5].V(1)in?[0,..,2].V(27)in?[1,..,3].V(29)in?[0,..,2].
V(14)in?[6,..,8].V(28)in?[2,..,4].V(21)in?[3,..,5].V(34)in?[4,..,6].V(17)in?[2,..,4].
V(30)in?[2,..,4].V(26)in?[1,..,3].V(32)in?[2,..,4].V(25)in?[0,..,2].V(8)in?[6,..,8].
V(35)in?[5,..,7].V(36)in?[6,..,8].V(12)in?[6,..,8].V(22)in?[4,..,6].V(16)in?[2,..,4].
V(4)in?[3,..,7].nil,nil,nil,...]
```

On peut alors demander le calcul des solutions et le système affiche alors la solution en remplaçant les variables contraintes dans les attributs correspondant pour une meilleure lisibilité. Voici une solution possible au problème précédent :

```
0(0):Action::[Name(Print) , B(7) , E(8)]
0(1):Action::[Name(FormKeep) , B(6) , E(8)]
0(2):Action::[Name(Load) , B(6) , E(7)]
0(3):Action::[Name(Print) , B(5) , E(6)]
0(4):Action::[Name(FormKeep) , B(4) , E(6)]
0(5):Action::[Name(Load) , B(4) , E(5)]
0(6):Action::[Name(SimplePrint) , B(2) , E(4)]
0(7):Action::[Name(Print) , B(1) , E(2)]
0(8):Action::[Name(FormKeep) , B(0) , E(2)]
0(9):Action::[Name(Load) , B(0) , E(1)]
```

La solution présentée ci-dessus est affichée de façon très lisible : on a pour chaque action ses temps de début et de fin par simple lecture des valeurs des attributs `B` et `E`. Néanmoins, on donne dans la Figure 4.4 un diagramme temporel correspondant à cette solution, de façon à rendre la solution plus visuelle.

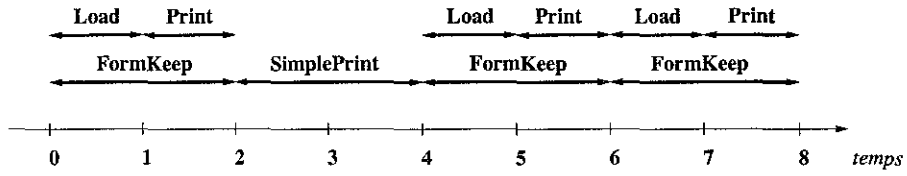


FIG. 4.4 – *Un ordonnancement pour quatre tâches d'impressions.*

Conclusion

Dans ce chapitre, on a tout d'abord montré comment utiliser les concepts objets présentés dans le chapitre précédent peuvent être utilisés dans des règles. On a aussi montré comment des contraintes de type CSP peuvent être utilisées conjointement dans des règles.

L'ensemble représente donc un nouveau formalisme permettant d'utiliser dans des règles à la fois des objets et des contraintes. Ce formalisme peut quant à lui être utilisé dans le système ELAN et, ainsi, on peut utiliser toute la puissance des stratégies offertes par ELAN afin de contrôler l'application des règles.

Le formalisme de règles avec objets uniquement ou de règles avec objets et contraintes est adapté à des problèmes de décomposition de tâches et d'ordonnancement. Il permet facilement, avec l'aide des stratégies, de parcourir l'ensemble de l'arbre de recherche. De plus, avec le mécanisme des entrées/sorties, on peut offrir à un utilisateur des outils l'aidant dans les phases de décomposition ou de parcours d'arbre : l'utilisateur peut ainsi, à chaque moment de la phase de décomposition des tâches, faire des vérifications sur la base de contraintes et choisir parmi tout un ensemble de stratégies qu'il a lui-même défini, la stratégie qui lui paraît être la plus opportune. La fait de pouvoir couper des branches de l'arbre de recherche en cours d'exécution permet à l'utilisateur d'arrêter une branche en cours de développement qui ne lui paraît plus judicieuse et ainsi de revenir sur des points de choix déjà posés antérieurement.

Une première version du problème de l'ascenseur a été présentée dans [DK98]. Le formalisme de règles avec objets et contraintes, après avoir été introduit dans [DK99], a été présenté dans [DK00c].

Chapitre 5

Transformation de programme en ELAN

Un langage permettant de définir des modules objets, de définir et de manipuler des règles avec objets et contraintes a été décrit dans les chapitres précédents. Afin d'exécuter un programme dans ce langage, nous avons deux possibilités : soit le système ELAN est modifié pour prendre en compte les nouveaux modules objets et les nouvelles règles ; soit ces nouveaux éléments sont traduits dans le langage ELAN. C'est la seconde solution qui a été choisie et quelques éléments ont déjà été présentés dans les chapitres précédents comme la traduction des méthodes en règles ELAN par exemple.

Afin de réaliser cette transformation des OModules en modules ELAN et des modules dans lesquels sont définis les règles avec objets et contraintes là aussi en modules ELAN, un outil faisant de la transformation purement syntaxique n'est pas suffisant : on doit en effet au cours de la transformation faire du calcul, notamment dans la transformation des modules objets en modules ELAN. Nous avons très vite choisi d'utiliser ELAN comme outil de transformation de programme : ELAN étant un système dans lequel on peut faire à la fois de la déduction et du calcul, il nous permet d'allier les transformations purement syntaxiques et les transformations après calcul au moyen des règles et des stratégies. Le fait de ne pas avoir des transformations uniquement syntaxiques ne nous permet par contre pas d'utiliser les constructions de pré-processeur.

Néanmoins, nous avons voulu favoriser la réutilisation des programmes dans ces processus de transformation de programme : certains programmes utilisés pour transformer les modules objets seront aussi utilisés pour transformer les règles.

Nous présentons dans cette partie tout d'abord le mécanisme général que nous utilisons pour le processus de transformation ainsi que le détail de certaines étapes qui peuvent être globalisées car non liées aux applications. Ensuite, nous présentons les étapes de transformation fortement liées aux applications, et notamment celles utilisées pour les modules objets et celles utilisées pour les règles avec objets et contraintes.

5.1 Un schéma de traduction

Nous présentons dans cette première partie le principe général de la transformation de programme qui utilise le système ELAN comme langage de définition de la transformation de programmes. Nous présentons aussi toutes les parties dites "générales" qui sont en fait les sous-programmes, ou les modules, qui sont indépendants de ce que l'on cherche à transformer.

En effet, on doit bien garder en mémoire que l'on cherche ici à transformer deux choses : les OModules en modules ELAN standard et les modules dans lesquels sont définis les règles avec objets et contraintes en modules standard aussi. C'est le même schéma de traduction qui est utilisé dans les deux cas. Mais, sur l'ensemble du programme définissant la transformation, seule une partie de la traduction est adaptée au problème traité. Le reste est réutilisable quel que soit le type de module que l'on cherche à transformer.

C'est donc le schéma général de transformation qui est présenté ici, ainsi que les parties du programme du transformation qui sont communes à la fois à la transformation des OModules et à la transformation

des modules de règles avec objets et contraintes. Ces parties communes, ou "générales" comme on l'a dit précédemment, sont appelés les éléments persistants du programme.

5.1.1 Une traduction en cinq étapes

Le schéma général de traduction est présenté dans la Figure 5.1.

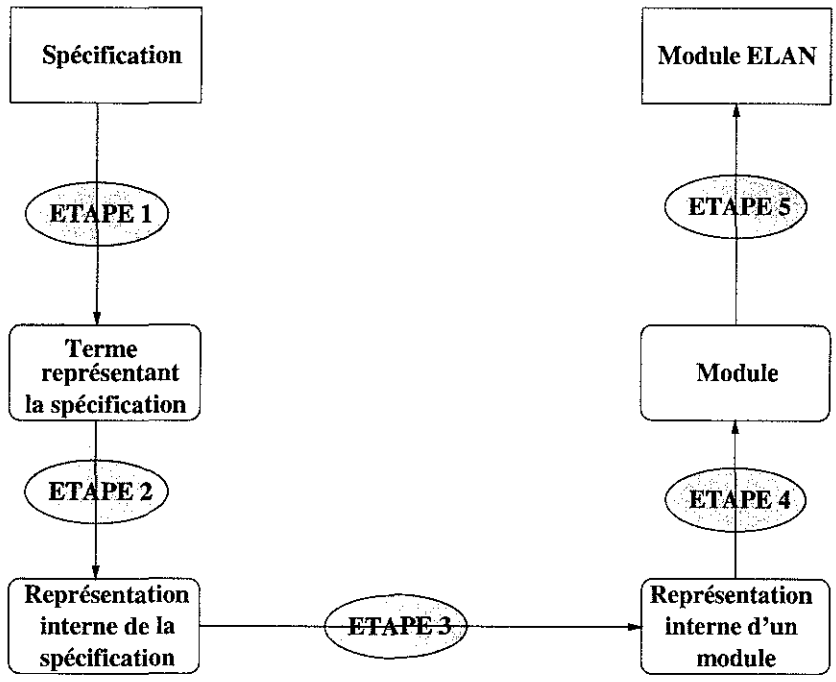


FIG. 5.1 – *Principe général de la transformation de programme en* ELAN.

On peut voir que la transformation se fait en cinq étapes :

1. La première étape consiste, à partir de la spécification donnée en entrée, à construire un terme sur lequel on travaillera pendant tout le processus de transformation. La spécification d'entrée doit vérifier une certaine syntaxe qui lui est spécifique : syntaxe des modules objets, syntaxe des règles avec contraintes et objets, etc. Le résultat est donc le terme correspondant à la spécification donnée. Cette étape inclut aussi une vérification syntaxique.
2. La deuxième étape consiste à construire, à partir du terme précédent, une représentation interne de la spécification qui doit être appropriée à la transformation.
3. La troisième étape consiste à faire la correspondance entre la représentation interne de la spécification et la représentation interne du module ELAN correspondant. Le schéma de représentation interne d'un module ELAN est quant à lui fixé.
4. Une fois obtenue la représentation interne du module ELAN correspondant à la spécification de départ, on peut construire le module ELAN correspondant. En fait, on constuit un module dans une syntaxe très proche de celle exécutable en ELAN. On appelle cette syntaxe la pseudo-syntaxe ELAN par la suite ; les différences entre la pseudo-syntaxe et la syntaxe ELAN étant liées à certains mots clés difficilement productibles par le système.
5. La cinquième et dernière phase consiste à prendre le module précédemment produit et à effectuer des modifications sur les mots clés principalement afin de produire un module ELAN exécutable. Cette phase ne peut pas se faire en utilisant le système ELAN, sinon, quatrième et cinquième phases

auraient pu être fusionnées en une seule et même étape. On utilise pour cette dernière phase un *shell-script* basé sur des commandes `sed`.

Il est intéressant de noter que les quatre premières phases sont réalisées par le système ELAN lui-même.

5.1.2 Les éléments persistants

Le programme ELAN permettant d'enchaîner les quatre premières étapes est décrit par la règle suivante :

```
[] Transformation(file) => true
where Term1  := () GetTermFromSpecif(file)
        where Term2  := () InternalRepOfTerm(Term1)
        where Term3  := () FromInternalRepToInternalRepModules(Term2)
        where Module := () FromInternalRepModuleToModule(Term3)
end
```

Dans cette règle ELAN, la primitive `GetTermFromSpecif` permet, à partir d'un nom de fichier donné en entrée, de construire le terme correspondant, c'est la première phase. La primitive `InternalRepOfTerm` concerne la deuxième phase du processus de transformation. L'opérateur `FromInternalRepToInternalRepModules`, associé à la troisième phase, permet la transformation à proprement parler en faisant passer d'une représentation interne de la spécification d'entrée à celle d'un module. Le dernier opérateur `FromInternalRepModuleToModule` correspond à la quatrième phase et génère le module en pseudo-syntaxe ELAN.

Les éléments persistants du programme de transformation en ELAN sont dans la première et dans la troisième phase ; on les présente dans la suite. La cinquième et dernière phase, la phase des *shell-script*, est aussi décrite dans cette partie.

Phase 1 : Lecture de la spécification

Afin de lire la spécification d'entrée, on doit définir en ELAN la syntaxe associée à la spécification d'entrée. Ainsi, si on souhaite transformer un OModule, on doit tout d'abord lire ce OModule ; on doit donc définir en ELAN la syntaxe des termes qui représentent les OModules donnés en entrée.

Le module gérant cette phase est valable pour n'importe quelle application. Par contre, le module dans lequel est définie la syntaxe doit être importé. Dans ce module, on trouvera une déclaration formelle de cette syntaxe en se basant sur les définitions de constantes et d'opérateurs.

Phase 3 : Production de la représentation interne du module à générer

La représentation interne d'un module ELAN est définie de façon globale, quelle que soit l'application étudiée. En effet, on peut définir une représentation interne de tout module ELAN ; si jamais une partie seulement de ce module est utilisée (par exemple, on produit un module avec seulement un nom, des importations et des déclarations d'opérateurs), alors, seule la partie correspondante de la représentation interne du module sera produite.

La représentation interne d'un module ELAN peut être définie en ELAN par un ensemble d'opérateurs. On ne va pas donner ici l'ensemble de ces opérateurs, mais seulement donner une idée générale de cette représentation :

```
@ : (tuple[ModuleName,
           GlobalImports,
           LocalImports,
           Sorts,
           Operators,
           StrategyDecl,
           FamilyOfRules,
           Strats]) IntRepModule;
```

```
@ : (ident) ModuleName;
@ : (list[SortName]) GlobalImports;
@ : (LocalImportsOpt) LocalImports;
@ : (SortDefinitionOpt) Sorts;
@ : (OperatorDefinitionOpt) Operators;
@ : (StrategyDefinitionOpt) StrategyDecl;
@ : (list[Rules]) FamilyOfRules;
@ : (ListOfFamilyOfStrategies) Strats;
```

Un module ELAN est donc représenté par un 8-uplet composé :

1. du nom du module, de sorte `ModuleName`, qui est un identificateur ;
2. de la liste des importations globales du module, de sorte `GlobalImports`. Ces importations sont définies par une liste de noms de modules ;
3. de la liste des importations locales du module, de sorte `LocalImports`. Cette liste optionnelle est définie par la définition de la sorte `LocalImportsOpt` qui n'est pas détaillée ici mais ressemble fortement à la représentation interne des importations globales ;
4. de la liste des sortes définies dans le module, des sorte `Sorts`. Cette liste optionnelle est définie par la définition de `SortDefinitionOpt` et consiste en une liste de termes représentant les sortes ;
5. de la liste des définitions d'opérateurs, de sorte `Operators`. Cette liste est définie par la définition de la sorte `OperatorDefinitionOpt` où l'on retrouve toutes les possibilités de définition d'opérateurs possibles ;
6. de la liste des définitions d'opérateurs de stratégies, de sorte `StrategyDecl`. Cette liste est définie par la définition de la sorte `StrategyDefinitionOpt` ;
7. de la liste des définitions des règles, de sorte `FamilyOfRules`. Elle est définie par une liste de termes de sorte `Rules` où chaque terme représente une famille de règle, c'est-à-dire un ensemble de règles travaillant sur la même sorte ;
8. et finalement de la liste des définitions des stratégies, de sorte `Strats`. Elle est définie par un terme de sorte `ListOfFamilyOfStrategies` dans lequel est donné l'ensemble des stratégies.

Phase 4 : Production du module en pseudo-syntaxe

Cette phase est bien un élément persistant puisque l'on part d'une syntaxe abstraite d'un module ELAN afin de produire le module correspondant.

On peut donner le module de base dans lequel une partie de la pseudo-syntaxe ELAN est définie :

```
module elanModule

import global elanImports elanSortDefinition elanOperatorDefinition
       elanStrategyDefinition elanFamilyOfRules elanFamilyOfStrategies;
end

sort Module FormalModuleName IdentList;
end

operators global
  module @ @ @ @ @ @ @ End : ( FormalModuleName ImportsOpt SortDefinitionOpt
                               OperatorDefinitionOpt StrategyDefinitionOpt
                               ListOfFamilyOfRules
                               ListOfFamilyOfStrategies ) Module;
```

```
  @           : ( ident ) FormalModuleName;
  @ '[' @ ']' : ( ident IdentList ) FormalModuleName;

  @       : ( ident ) IdentList;
  @ ',' @ : ( ident IdentList ) IdentList;

end

end
```

Ce module représente le fichier racine décrivant la syntaxe d'un module à la ELAN : un module est décrit par le premier opérateur `module _ _ _ _ _ _ _ End` prenant différents paramètres et retournant un terme représentant un module.

Phase 5 : les *shell-scripts*

La dernière phase de notre schéma de traduction est réalisée par des scripts qui permettent de passer du module en pseudo-syntaxe au module ELAN correspondant. Ces scripts permettent de corriger les "erreurs" syntaxiques que l'on produit automatiquement.

Mais pourquoi est-on amené à corriger des "erreurs" syntaxiques produites automatiquement alors qu'il aurait été plus simple de produire directement quelque chose de correct initialement?

Ceci est dû au fait qu'ELAN est le langage permettant de faire la transformation de programme dans notre étude. En effet, certaines limitations d'ELAN font que l'on ne peut pas, par exemple, générer le mot clé `end` de façon automatique. Ainsi, dans tout le schéma de traduction, à la place de `end`, on génère le mot-clé `End`. Néanmoins, ce mot-clé ne sera jamais accepté par l'interpréteur ELAN, ni même par le compilateur d'ailleurs.

C'est ce genre d'"erreur" que corrige cette phase. C'est donc un script enchaînant différentes commandes, principalement des commandes utilisant l'éditeur `sed`.

5.2 Applications

Maintenant que les étapes de la traduction sont présentées, nous nous intéressons aux parties plus spécifiques de la transformation de programme, c'est-à-dire à la partie intimement liée à l'application traitée. Nous montrons ici comment ce schéma de transformation est applicable aux modules objet (OModules) ainsi qu'aux modules dans lesquels nous définissons des règles avec objets et contraintes.

Le processus de transformation étant facilement adaptable, nous concluons cette partie par la présentation d'une autre application qui n'est pas liée au formalisme de règles avec objets et contraintes que nous étudions dans cette thèse. Cette dernière application, liée à Prolog, est par contre intéressante pour les fonctionnalités d'ELAN qu'elle utilise.

5.2.1 Modules objets

L'adaptation du processus de transformation à la gestion des objets en ELAN est présentée en deux points.

On donne tout d'abord l'ensemble des modules ELAN permettant d'implanter les concepts objets en ELAN. On trouve ainsi dans cette partie l'ensemble des opérateurs et règles que l'on a introduit dans le chapitre 3, mais de façon plus modulaire. En effet, dans le chapitre 3, nous avons défini un ensemble de déclarations et de définitions propres au codage des objets en ELAN. Dans ces déclarations, on a pu remarquer que les types de déclarations étant les mêmes quelque soit les classes définies. La seule chose qui change est le nom d'un attribut, son type, le nom de la classe, le nom d'une méthode, etc... On peut donc définir un ensemble de modules génériques pour l'ensemble du codage des objets ; modules génériques que l'on doit instancier par l'information adéquate correspondant au OModule en cours de transformation.

Ensuite, on définit de façon plus détaillée le mécanisme de traduction pour cette application.

Les modules ELAN implantant les objets

L'implantation des objets en ELAN est faite par un ensemble de modules qui peuvent être paramétrés. On présente dans cette partie les modules les plus représentatifs.

Tout d'abord, le module dans lequel est définie la structure générale d'un objet est le module `Object` :

```
module Object

sort MName MBody method Methods; end

operators global

  @,@  : (Methods Methods) Methods (AC);
  @    : (method) Methods;

  @(@) : (MName MBody) method;
  @    : (MName) method;

end
end
```

Dans ce module, on trouve les définitions des sortes servant à construire les objets : `MName`, `MBody`, `method` et `Methods`. Les déclarations associées permettant de définir ces sortes suivent.

Une fois les objets définis, lorsque l'on définit une classe C, on importe alors le module `DefClass` paramétré avec le nom de la classe créée. Ce module utilise le module général de définition des objets, le module `Object`, présenté avant :

```
module DefClass[Class]

import global Object; end

sort Class; end

operators global
 [@]   : (Methods) Class;
end
end
```

Ce module définit la sorte *Class* à partir du paramètre du module. Les objets de cette classe sont ensuite définis par la déclaration de l'opérateur `[_]`.

Toute classe doit pouvoir permettre à un objet d'être créé à partir d'une méthode `new` envoyée à la classe. C'est la définition de cette méthode ainsi que l'appel aux modules présentés avant que gère le module `ClassInit` présenté ci-dessous. Il est paramétré par le nom de la classe `Class` et par le nom donnée à l'objet implantant la classe, c'est-à-dire `ClassName`.

```
module ClassInit[Class,ClassName]

import  global  DefClass[Class]; end

operators       global

    new     : MName;
```

```
    ClassName.new   : Class;
    send(@,new)     : (Class) Class;
    new(@)          : (Class) Class;

    ClassName       : Class;
end


rules for Class
   LM       : Methods;
   o        : Class;
global
[] ClassName.new        => send(o,new)
        where o := () ClassName
end
[] send([new,LM],new)   => new([new,LM])
end
end
end
```

Toute classe est définie par un ensemble d'attributs qui la caractérisent. Afin de définir un attribut, on utilise le module `AttributeDecl` paramétré avec le nom de l'attribut que l'on crée. Ce module est donné par :

```
module AttributeDecl[Att]

import global Object; end

operators global
  Att : MName;
end
end
```

La définition de la sorte de l'attribut comme sorte possible dans les objets se fait par l'importation du module `SortAttDecl` suivant, paramétré avec le nom de la sorte manipulée :

```
module SortAttDecl[SortAtt]

import global Object SortAtt; end

operators global

  @ : (SortAtt) MBody;
end
end
```

Ces modules ne font que déclarer l'attribut et son type comme étant un attribut et un type possibles. Ce sont ces modules qui sont importés lorsqu'on cherche à manipuler un attribut, notamment dans la définition des méthodes associées à chaque attribut.

Ainsi, pour tout attribut créé, les méthodes associées sont celles permettant d'obtenir la valeur associée à chaque attribut et de modifier sa valeur. Les définitions de ces méthodes sont données dans le module `AttributeGet` ci-dessous. Il est paramétré avec le nom de la classe `Class`, le nom de l'attribut `Att`, le nom de la méthode associée `GetAtt` ainsi que de la sorte du l'attribut `SortAtt` :

```
module AttributeGet[Class,Att,GetAtt,SortAtt]

import global DefClass[Class] AttributeDecl[Att] SortAttDecl[SortAtt];
end

operators global

  GetAtt          : MName;

  @.Att           : (Class) SortAtt;
  send(@,GetAtt)  : (Class) SortAtt;
  GetAtt(@)       : (Class) SortAtt;
end


rules for SortAtt
  LM  : Methods;
  V   : SortAtt;
  o   : Class;
global

[] o.Att                    => send(o,GetAtt)
end
[] send([GetAtt,LM],GetAtt) => GetAtt([GetAtt,LM])
end
[] GetAtt([Att(V),LM])      => V
end
end end
```

On importe dans ce module une partie des modules présentés avant. On définit ensuite les opérateurs et les règles correspondant aux méthodes d'accès à un attribut donné.

Ce module illustre bien la définition générique que l'on effectue dans ce chapitre : on voit la correspondance entre les méthodes génériques et les méthodes présentées précédemment dans le chapitre 3 dans la section 3.4..

A ces modules génériques s'ajoute un module crée automatiquement par le processus de transformation et dans lequel on trouve toutes les importations nécessaires ainsi que toutes les déclarations qui ne peuvent être génériques, comme la définition des méthodes utilisateurs par exemple.

On a donc présenté dans cette section les modules permettant de définir les objets en ELAN. Voyons maintenant comment la transformation est automatisée.

De la spécification objet aux modules ELAN

Pour passer de la spécification objet, c'est-à-dire d'un ensemble de OModules aux modules ELAN correspondants, on suit le processus de transformation décrit en Figure 5.1.

La première phase utilise la formalisation en ELAN de la syntaxe générale des modules objets. Cette syntaxe, donnée en Annexe A est en partie décrite par les déclarations suivantes :

```
...
operators

   @    : (NSModule) Specif;
   @    : (Class) NSModule;
   @ @  : (Class NSModule) NSModule;
...
```

```
class @ @ @ End       : (ident Attributes Methods) Class;
class @ @ @ @ End     : (ident Imports Attributes Methods) Class;
class @ @ @ @ End     : (ident Inherits Attributes Methods) Class;
class @ @ @ @ @ End   : (ident Inherits Imports Attributes Methods) Class;

from @ ';'    : (ident) Inherits;

imports @ ';': (SortNameList) Imports;

attributes @ : (AttributeDeclareList) Attributes;
@            : (AttributeDeclare) AttributeDeclareList;
@ @          : (AttributeDeclare AttributeDeclareList) AttributeDeclareList;
@ ':' @ = @  : (AttributeName SortName InitValue) AttributeDeclare;
@            : (ident) AttributeName;
@            : (ident) InitValue;
@            : (int) InitValue;

@    : (Method) Methods;
@ @  : (Method Methods) Methods;

method @ for @ <@>       : (MethodName SortMethodRes MethodBody) Method;
method @ (@) for @ <@>   : (MethodName MethodArgs SortMethodRes MethodBody) Method;
method @ for @ @ <@>     : (MethodName SortMethodRes MethodVars MethodBody) Method;
method @ (@) for @ @ <@>: (MethodName MethodArgs SortMethodRes MethodVars
                                                          MethodBody) Method;
...
end
```

Dans cet ensemble de déclarations, certaines permettant :

- de définir comme spécification d'entrée (sorte `Specif`), les définitions des classes qui peuvent se succéder au sein de la même spécification ;
- de définir comment est déclarée une classe : des importations, des déclarations d'attributs, de méthodes, etc... . On donne ici une partie des déclarations permettant de définir des classes avec :
  - attributs et méthodes (cela correspond à la définition de `method @ for @ <@>`),
  - importations, attributs et méthodes,
  - héritage, attributs et méthodes,
  - héritage, importations, attributs et méthodes.

  Toutes les autres combinaisons ne sont pas mises ici.
- de définir les différentes parties composants la définition d'une classe : l'héritage, les importations de modules, les déclarations d'attributs et la syntaxe générale des méthodes.

Exemple 5.1 *Considérons la définition d'une classe* `PointTranslation` *avec un attribut* `X` *et une méthode* `GetXTranslated(_)` *prenant un entier et rendant comme entier la valeur de l'attribut* `X` *incrémentée de la valeur de l'argument. Cette classe est définie par :*

```
class PointTranslation
attributes X:int = 0
method GetXTranslated(N:int) for int <self.GetX + N>
End
```

Après la première phase du processus de transformation, le terme représentant cette classe est donnée par :

```
class PointTranslation attributes X:int=0
method GetXTranslated(N:int)for int<self.GetX+N>End
```

Dans la seconde phase, on transforme le terme issu de la première phase en une représentation interne des définitions de classes. Toute classe est représentée en interne par un quintuple composé du nom de la classe, d'une représentation interne de l'information d'héritage, d'une représentation interne des importations de modules, d'une représentation interne des définitions d'attributs et d'une représentation interne des définitions de méthodes. Ce quintuplet est de sorte `IntRepClass`. Le terme résultat de la seconde phase, de type `IntRepClasses`, est une liste de ces quintuplets car plusieurs classes peuvent être définies. Les déclarations correspondantes sont données dans :

```
operators global
   @ : (tuple[ClassName,ClassInherits,ClassImports,ClassAttributes,ClassMethods])
         IntRepClass;
   @ : (list[IntRepClass]) IntRepClasses;
end
```

Afin de construire la représentation interne des définitions de classes, on définit un opérateur `InternalRepOfClass(_)` qui prend une définition de classe et rend le quintuplet associé. On donne ici les règles de cet opérateur concernant les types de définitions de classe donnés précédemment : on définit ainsi tout d'abord le quintuplet à partir d'une définition de classe avec attributs et méthodes. On utilise pour cela des opérateurs secondaires `InternalAttributes(_,_)` et `InternalMethods(_,_)` qui sont spécialisés dans la construction des représentations internes pour les attributs et les méthodes. Les traitements sont similaires pour les trois autres cas montrés ici :

```
[] InternalRepOfClass(class I1 attributes A M End)                => [$I1,nil,nil,IA,IM]
     where IA := () InternalAttributes(A,nil)
     where IM := () InternalMethods(M,nil)
end
[] InternalRepOfClass(class I1 imports SNL ; attributes A M End) => [$I1,nil,IS,IA,IM]
     where IS := () InternalImports($SNL|nil)
     where IA := () InternalAttributes(A,nil)
     where IM := () InternalMethods(M,nil)
end
[] InternalRepOfClass(class I1 from I2 ; attributes A M End) => [$I1,$I2.nil,nil,IA,IM]
     where IA := () InternalAttributes(A,nil)
     where IM := () InternalMethods(M,nil)
end
[] InternalRepOfClass(class I1 from I2;imports SNL;attributes A M End)
                                                          => [$I1,$I2.nil,IS,IA,IM]
     where IS := () InternalImports($SNL|nil)
     where IA := () InternalAttributes(A,nil)
     where IM := () InternalMethods(M,nil)
end
end
```

Exemple 5.2 *Prenons le terme résultat de la première phase donné dans l'exemple 5.1. Suite à la seconde phase, la représentation interne de la définition de la classe* `PointTranslation` *est donnée par le terme :*

```
[PointTranslation,nil,nil,[X,int,0].nil,
 [GetXTranslated,[N,int].nil,int,nil,(self.GetX+N).nil].nil].nil
```

La troisième phase consiste à effectuer la correspondance entre modules objets et modules ELAN. Elle prend en entrée le terme donnant la représentation interne des modules objets et donne comme résultat un terme donnant la représentation interne d'un module ELAN. La représentation interne d'un module a été donnée dans la section 5.1.2. On définit l'implémentation de la traduction par l'opérateur `IRModule(_)` qui prend un quintuplet, élément de la liste obtenue après la seconde phase, et donne l'octuplet définissant la représentation interne du module ELAN correspondant :

S.C.D. - U.H.P. NANCY 1
BIBLIOTHÈQUE DES SCIENCES
Rue du Jardin Botanique
54600 VILLERS-LÈS-NANCY


```
rules for IntRepModule
  Id   : ident;
  CIn  : ClassInherits;
  CIm  : ClassImports;
  CA   : ClassAttributes;
  CM   : ClassMethods;
  GI   : GlobalImports;
  FR   : FamilyOfRules;
  O    : Operators;
global
[] IRModule([Id,CIn,CIm,CA,CM]) => [$Id,GI,,,O,,FR,]
       where GI := () GlobalImports($Id,CIn,CIm,CA)
       where O  := () DefOperators($Id,CM)
       where FR := () FamilyOfRules($Id,CIn,CA,CM)
end
end
```

Pour toute définition de module objet, on génère un module ELAN dont les composantes suivantes sont à définir : le nom, les importations globales, les définitions d'opérateurs et les définitions de familles de règles. Les autres composants, à savoir les importations locales, les définitions de sortes, d'opérateurs de stratégies et les règles associées, ne sont pas utiles dans le cas de la transformation de module objets car le codage des objets en ELAN ne fait pas intervenir ces composants.

L'opérateur `GlobalImports(_,_,_,_)` définit les importations à réaliser. Ces importations sont composées des importations déjà mentionnées dans le module objet mais aussi des importations de modules génériques décrits dans la section 5.2.1 correctement paramétrés.

L'opérateur `DefOperators(_,_)` définit les déclarations d'opérateurs nécessaires. Cela concerne principalement les déclarations d'opérateurs associées aux définitions de méthodes utilisateur.

Le dernier opérateur, `FamilyOfRules(_,_,_,_)`, définit quand à lui les règles associées aux définitions des méthodes utilisateurs et à la méthode de création `new`. Cet opérateur implémente le schéma de traduction donné dans la section 3.4.4. Il fait correspondre à chaque définition de méthode les règles de réécriture associées.

Exemple 5.3 *Reprenons pour la classe `PointTranslation` le terme obtenu après la seconde phase et présenté dans l'Exemple 5.2. On donne pour cette classe le terme définissant la représentation interne du module* ELAN *qui est associé au module objet de départ. Le terme obtenu après la troisième phase est :*

```
[PointTranslation,
 DefClass[PointTranslation].ClassInit[PointTranslation,PointTranslation Class].
 AttributeGet[PointTranslation,X,Get X,int].
 AttributeModif[PointTranslation,X,Set X,int].nil,
 ,
 ,
 [[@.GetXTranslated(@),PointTranslation.int.nil,int].
  [GetXTranslated(@,@),PointTranslation.int.nil,int].nil,nil],
 ,
 [PointTranslation,[o.nil,PointTranslation].nil,
  [[PointTranslation Class,[new,X(0),Get X,Set X,GetXTranslated],nil].
   [o.new,[X(0),Get X,Set X,GetXTranslated],nil].nil,nil],].
 [int,[o.nil,PointTranslation].[self.nil,PointTranslation].[N.nil,int].nil,
  [[o.GetXTranslated(N),GetXTranslated(o,N),nil].
   [GetXTranslated(self,N),GetX(self)+N,nil].nil,nil],].
 nil
 ,
].nil
```

On trouve dans ce terme les huit composants de l'octuplet définissant la représentation interne d'un module.

- *Le premier élément est le nom du module.*
- *Vient ensuite une liste de noms de modules qui sont les modules devant être importés : on retrouve les modules présentés dans la section 5.2.1 paramétrés comme le module* `AttributeGet` *paramétré avec le nom de la classe, le nom de l'attribut, le nom de la méthode permettant de récupérer la valeur de cet attribut et le type de l'attribut.*
- *Ici, les troisièmes et quatrièmes champs de la structure sont vides. Ils correspondent aux importations locales et aux déclarations de nouvelles sortes, inexistantes dans la transformation des OModules.*
- *Le cinquième champ est composé de la représentation interne des déclarations d'opérateurs associés aux méthodes utilisateurs ; ici, les déclarations associées à la méthode* `GetXTranslated`*.*
- *Ici, le sixième champ est vide : on n'a pas de déclaration d'opérateur de stratégie dans la transformation des OModules.*
- *Le septième champ est composé de la représentation interne des règles du module. On a ici deux groupes de règles : un premier groupe de deux règles définies pour la sorte* `PointTranslation` *et un second groupe de deux règles pour la sorte* `int`*. Ces groupes sont des quatruplets composés du type de la règle, d'une liste de variables locales et d'une liste de règles globales puis locales (champ vide ici car ce cas n'est pas utilisé dans le contexte des modules objets).*
- *Ici, le huitième champ est vide : il n'y a pas de stratégies définies.*

La quatrième phase consiste à générer un terme dans la pseudo-syntaxe de module à partir du terme généré par la troisième phase. La pseudo-syntaxe a été donnée dans la section 5.1.2. La règle permettant de générer un module dans la pseudo-syntaxe à partir de la représentation interne est donnée ci-dessous :

```
rules for Module

  MN   : ModuleName;
  GI   : list[SortName];
  FR   : list[Rules];
  MNL  : ModuleNameList;
  R    : ListOfFamilyOfRules;
  M    : Module;
  IRO  : Operators;
  O    : SymbolDeclarationList;

global

[] ModuleFromIRM([MN,GI,,,O,,FR,]) => M
      if $GI != nil
      if $FR != nil
       where MNL :=() ModuleNameListFromGlobalImports(GI)
       where O :=() OperatorsFromIntRepOps(IRO)
       where R :=() RulesFromIntRepRules(FR)
       where M :=() module $MN import global $MNL ; End operators global $O End $R End
end
end
```

Dans cette règle, on retrouve les quatre champs définis dans l'octuplet résultat de la troisième phase. A partir de ces quatre champs est généré le terme définissant le nouveau module. Les trois opérateurs `ModuleNameListFromGlobalImports(_)`, `OperatorsFromIntRepOps(_)` et `RulesFromIntRepRules(_)` permettent respectivement de créer les importations, les déclarations d'opérateurs et les règles à partir

des représentations internes de ces trois composants. L'ensemble est finalement assemblé en un seul terme M qui est le résultat escompté.

Exemple 5.4 *Reprenons l'exemple 5.3 précédent et montrons le terme obtenu après la quatrième phase. Par souci de lisibilité, la présentation du terme a été clarifiée par des retours à la ligne :*

```
module PointTranslation

import global DefClass[PointTranslation]
              ClassInit[PointTranslation,PointTranslation Class]
              AttributeGet[PointTranslation,X,Get X,int]
              AttributeModif[PointTranslation,X,Set X,int];\End

operators global @.GetXTranslated(@):(PointTranslation int) int;
                 GetXTranslated(@,@):(PointTranslation int) int;\End

Rules for PointTranslation
 o:PointTranslation;
global
[]PointTranslation Class=>[new,X(0),Get X,Set X,GetXTranslated]\End
[]o.new=>[X(0),Get X,Set X,GetXTranslated]\End
\End

Rules for int
 o:PointTranslation;
 self:PointTranslation;
 N:int;
global
[]o.GetXTranslated(N)=>GetXTranslated(o,N)\End
[]GetXTranslated(self,N)=>GetX(self)+N\End
\End
End.nil
```

On retrouve l'ensemble des composants d'un module : nom du module, importations, opérateurs et règles. Tous ces composants sont obtenus directement à partir du terme issu de la troisième phase.

La cinquième et dernière phase est quant à elle une phase purement syntaxique et est détaillée dans la section 5.1.2. Notons toutefois qu'alors que le terme obtenu après la quatrième phase est une liste d'autant de sous-termes qu'il y a de classes définies, on obtient après la cinquième phase, autant de nouveaux modules que de classes définies : un module par sous-terme composant la liste.

Exemple 5.5 *Le résultat est similaire à celui de l'Exemple 5.4, excepté pour les mots-clé la structure de liste (le ".nil" à la fin) qui n'a plus lieu d'être.*

On a ainsi détaillé dans cette partie le processus de transformation des modules objets en modules ELAN correspondants.

5.2.2 Règles avec objets et contraintes

Nous présentons maintenant le processus de transformation des règles avec objets et contraintes. De façon similaire à la transformation des OModules, on part des règles avec objets et contraintes décrites dans le formalisme que nous proposons dans le chapitre 4 et on les transforme en règles ELANstandard avec le même schéma de transformation en cinq phases.

Dans cette section, au lieu de détailler l'ensemble des cinq phases pour cette application, on va plutôt s'intéresser à la formalisation du processus de transformation, c'est-à-dire s'intéresser plus particulièrement à la transformation des règles avec objets et contraintes en des règles ELAN.

On rappelle ici le formalisme de règles avec objets et contraintes défini dans la section 4.2.3 :

$$[lab] \quad \{O_i : ClassName_i :: [Att_i^1(Value_1) \ , \ \ldots \ , \ Att_i^n(Value_n)]\}_{i=[1,\ldots,k]} \{O_j : ClassName_j\}_{j=[1,\ldots,l]}$$
$$\Rightarrow O'_1 \ldots O'_m \parallel c$$
$$[\textbf{if } t \mid$$
$$\textbf{if } CVEqual(V_1,v) \mid$$
$$\textbf{if } CVDef(V_2) \mid$$
$$\textbf{where } l]^*$$

Ces règles, suivant le même formalisme de transformation que celui défini dans toute cette section, sont transformées en règles ELAN de la forme :

$$\begin{array}{l}
Variables \\
\quad \{O_i \ : ClassName_i;\}_{i=[1,\ldots,k]} \\
\quad \{O_j \ : ClassName_j;\}_{j=[1,\ldots,l]} \\
\quad Z \ : configuration; \\
\quad b_1,b_2 \ : bool; \\
\quad C1 \ : CSP; \\
global \\
\quad [lab] \ \{O_i\}_{i=[1,\ldots,k]} \ \{O_j\}_{j=[1,\ldots,l]} \ Z \ with \ C \\
\qquad \Rightarrow \ Translate(O'_1,C.Max) \ldots Translate(O'_m,C.Max) \ Z \ with \ C1 \\
\qquad \{\textbf{if } O_i.GetAtt_i^j == Value_j\}_{i=[1,\ldots,k],j=[1,\ldots,n]} \\
\qquad \textbf{where } b_1 := () \ CVEqual(V_1,v) \\
\qquad \textbf{if } b_1 == true \\
\qquad \textbf{where } b_2 := () \ CVDef(V_2) \\
\qquad \textbf{if } b_2 == true \\
\qquad [\textbf{if } t \mid \textbf{where } l]^* \\
\qquad \textbf{where } C1 := () \ C \ U \ Translate([C.VarList,C.ConsSet],C.Max) \\
end
\end{array}$$

On voit dans cette transformation que le contexte de la mémoire de travail, noté `Z`, apparaît dans les membres gauche et droit de la nouvelle règle. La contrainte est aussi explicitement mentionnée avec la construction `_ with _`. Les tests de valeurs d'attributs sont reportés dans les tests de la règle ; les tests portant sur les variables contraintes sont eux-aussi modifiés en affectations de variables locales à la règle dont on vérifie ensuite la valeur (sont-elles réductibles en *vrai*?). L'opérateur `Translate(_,_)` permet quant à lui de prendre en compte le nombre de variables contraintes déjà utilisées et décale les indices des nouvelles variables contraintes de ce nombre.

Exemple 5.6 *Prenons comme exemple une des règles définies dans l'exemple présenté en section 4.2.7 :*

```
[Main1] O1:Task::[Name(Main)]
        =>
         O2(Name<-FormPrint)(Status<-all)(NumI<-O1.NumI-1)(B<-V_1)(E<-V_2)
         O3(Name<-SimplePrint)(B<-V_3)(E<-V_4)
          || (V_1 in? D, V_2 in? D, V_3 in? D, V_4 in? D)
              V_2<=?O1.E & V_4<=?O1.E & V_4=?V_3(+)2 &
              ((V_1>=?O1.B & V_3=?O1.B & V_4=?V_1) V
               (V_1=?O1.B & V_3>?O1.B & V_2=?V_3))
         where O2 := () TaskClass.new
         where O3 := () ActionClass.new
```

L'application de la transformation présentée dans cette section conduit ainsi à la règle ELAN:

```
[Main1] O1 Z with C =>
          O2(Name<-FormPrint)(Status<-all)(NumI<-O1.NumI-1)(B<-V(C.Max+1))(E<-V(C.Max+2))
          O3(Name<-SimplePrint)(B<-V(C.Max+3))(E<-V(C.Max+4)) Z with C1
            if O1.Name == Main
            where O2 := () TaskClass.new
            where O3 := () ActionClass.new
            where C1 := () C U [V(C.Max+1) in? C.DD,V(C.Max+2) in? C.DD,
                                V(C.Max+3) in? C.DD,V(C.Max+4) in? C.DD |
                                V(C.Max+1)>=?O1.B & V(C.Max+3)=?O1.B &
                                V(C.Max+2)<=?O1.E & V(C.Max+4)<=?O1.E &
                                V(C.Max+4)=?V(C.Max+1) &
                                V(C.Max+4)=?V(C.Max+3)(+)2]
end
```

De la même façon que pour la transformation des OModules, afin de réaliser la transformation des règles avec objets et contraintes, la syntaxe de ces règles est donnée dans un module pour la phase 1, un format interne est défini pour la phase 2, la transformation utilisée dans la phase 3 est implantée par des opérateurs et des règles ELAN et la phase 4 produit un terme qui est ensuite transformé en nouveau module ELAN dans lequel les règles avec objets et contraintes apparaissent.

Pour les règles ne traitant que d'objets et sans contraintes, la transformation est la même sauf que seule la configuration d'objets apparaît et ni le constructeur `_ with _`, ni les variables `C`, `C1` ni les opérations de translation avec `Translate(_,_)` ne figurent dans les nouvelles règles générées.

5.2.3 Autres applications

Nous avons traité dans [DK00b] différents exemples autres que ceux liés directement au codage des objets en ELAN ou encore à la définition du formalisme de règles avec objets et contraintes. Par exemple, nous avons traité la transformation d'un programme permettant de définir en ELAN un programme Prolog.

L'intérêt de cette application réside dans le fait que le programme ELAN permettant de définir un programme Prolog tel qu'il est actuellement défini en ELAN utilise le mécanisme du pré-processeur d'ELAN tel que cela est présenté dans la section 2.4.4. La gestion des commandes du pré-processeur est un mécanisme assez complexe en ELAN et une simplification de ce mécanisme est à l'étude actuellement. Le schéma de transformation tel que nous le définissons ici peut s'adapter à des constructions du type de celles gérées par le pré-processeur et notre schéma de transformation peut alors s'avérer intéressant.

Une spécification Prolog est donnée en ELAN par un fichier de la forme :

```
Specification Family

Vars       X Y Z
Ops
Predicates father:2 male:1 female:1 grandfather:2
           brother:2 sister:2
Clauses father(John,Ann) :- .
        father(Bill,John) :- .
        father(John,Al) :- .
        male(Al) :- .
        female(Ann) :- .
        grandfather(X,Z) :- father(X,Y),father(Y,Z) .
        brother(Y,Z):-father(X,Y),father(X,Z),male(Y) .
        sister(Y,Z):-father(X,Y),father(X,Z),female(Y).
End of Specification
```

Une spécification Prolog est définie par une liste de variables, une liste d'opérateurs avec leur arité, une liste de prédicats avec leur arité et une liste de clauses. On donne ici une exemple classique définissant les liens de parenté entre différentes personnes.

Le but de la transformation est ici de produire les modules ELAN correspondant à la spécification Prolog. Ensuite, on donne un but Prolog au système qui est évalué par un programme ELAN implantant la SLD-résolution utilisée en Prolog.

Le transformation utilise les mêmes étapes que celles présentées dans la Figure 5.1.

Pour la première étape, on donne la syntaxe d'une spécification dans un fichier ELAN :

```
operators       global

  Specification @ @ End of Specification :
         (identifier Spec:SpecifParts) Specif;

  Vars @ Ops @ Predicates @ Clauses @     :
         (V:SpecifVars
          O:SpecifOps
          P:SpecifPreds
          C:SpecifClauses) SpecifParts;

  @          : (identifier) SpecifVars;
  @ @        : (identifier SpecifVars) SpecifVars;

  @ ':' @    : (identifier int) SpecifOps;
  @ ':' @ @  : (identifier int SpecifOps) SpecifOps;

  @ ':' @    : (identifier int) SpecifPreds;
  @ ':' @ @  : (identifier int SpecifPreds)SpecifPreds;

  @ .        : (Clause) SpecifClauses;
  @ . @      : (Clause SpecifClauses) SpecifClauses;

  @          : (Atom) Clause;
  @ ':'- @   : (Atom ListAtoms) Clause;

  @          : (Atom) ListAtoms;
  @ , @      : (Atom ListAtoms) ListAtoms;

  @          : (term) Atom;
  @          : (equation) Atom;

  @          : (identifier) term;
  @(@)       : (identifier Subterms) term;

  @          : (Subterm) Subterms;
  @ , @      : (Subterm Subterms) Subterms;

  @          : (term) Subterm;
  @          : (identifier) Subterm;

  @ = @     : (term term) equation;
end
```

On retrouve dans ce fichier toutes les définitions d'opérateurs nécessaires pour la définition d'un module dans la syntaxe particulière à l'application Prolog.

La première phase de la transformation produit un terme dont l'existence nous assure que la syntaxe donnée en entrée correspond à la syntaxe de la spécification (la première phase est en effet similaire à une phase de *parsing* puisque l'on doit vérifier que le terme d'entrée doit respecter une syntaxe donnée). Cette correspondance est bien entendu syntaxique : les mots-clés sont respectés, les nombres d'arguments, etc. Mais le module d'entrée respecte aussi les types imposés dans la spécification : par exemple, lorsqu'on définit les opérateurs avec leur arité, on doit avoir un identificateur et un entier pour chaque déclaration. Une fois cette phase réalisée, nous pouvons garantir la cohérence à la fois en syntaxe mais aussi en typage du module d'entrée par rapport à la spécification donnée. Nous donnons ici le terme résultat de la première phase pour la spécification Prolog `Family`, c'est un terme de sorte `Specif` :

```
Specification Family Vars X Y Z Predicates
father:2male:1female:1grandfather:2brother:2
sister:2Clauses father(John,Ann).father(Bill,John).
father(John,Al).male(Al).female(Ann).
grandfather(X,Z):-father(X,Y),father(Y,Z).
brother(Y,Z):-father(X,Y),father(X,Z),male(Y).
sister(Y,Z):-father(X,Y),father(X,Z),female(Y).
End of Specification
```

La seconde étape consiste à produire une représentation interne structurée correspondant au terme précédent généré par la première phase. Dans le cas de la spécification Prolog, la représentation interne est définie par un quadruplet correspondant aux déclarations de variables, d'opérateurs, de prédicats et de clauses.

Un terme de sorte `Specif` est ainsi transformé en un terme de sorte `InternalRep` par l'opérateur de la seconde phase `InternalRepOfTerm`.

```
operators global

  InternalRepOfTerm(@)  : (Specif) InternalRep;

  @ : (tuple[IntRepVars,IntRepOps,IntRepPreds,IntRepClauses]) InternalRep;

  @ : (list[identifier]) IntRepVars;
  @ : (list[pair[identifier,int]]) IntRepOps;
  @ : (list[pair[identifier,int]]) IntRepPreds;
  @ : (list[pair[Atom,list[Atom]]]) IntRepClauses;
end
```

On donne ici la définition du quadruplet utilisé : la représentation interne des variables est une liste d'identificateur (terme de sorte `IntRepVars`) ; la représentation interne des opérateurs est une liste de paire, chacune étant composée de l'identificateur de l'opérateur et de son arité (terme de sorte `IntRepOps`) ; la représentation interne des prédicats est une liste de paire, chacune étant composée de l'identificateur du prédicat et de son arité (terme de sorte `IntRepPreds`) ; finalement, la représentation interne des clauses est une liste de paires où le premier élément est un atome et le second, la liste d'atomes permettant de la déduire (terme de sorte `IntRepClauses`).

Ainsi, pour le terme donné en exemple, le terme résultat de la seconde phase est :

```
[X.Y.Z.nil,
nil,
[father,2].[male,1].[female,1].[grandfather,2].
  [brother,2].[sister,2].nil,
[father(John,Ann),nil].
  [father(Bill,John),nil].
  [father(John,Al),nil].
  [male(Al),nil].
```

```
  [female(Ann),nil].
  [grandfather(X,Z),father(X,Y).father(Y,Z).nil].
  [brother(Y,Z),father(X,Y).father(X,Z).male(Y).nil].
  [sister(Y,Z),father(X,Y).father(X,Z).female(Y).nil].
  nil]
```

Vient ensuite la troisième phase qui permet de produire le terme donnant la représentation interne du module que l'on souhaite produire. La syntaxe suit la représentation interne d'un module et est déjà donnée dans les deux exemples précédents. On donne ici une partie du résultat pour le terme produit par la troisième phase pour l'application Prolog :

```
[Prolog,
nil,
nil,
 [[X,0],[nil,variable]].
 [[Y,0],[nil,variable]].
...
 [[female,0],[nil,Psymbol]].
 [[female,1],[atom.nil,term]].
 [[grandfather,0],[nil,Psymbol]].
 [[grandfather,2],[atom.atom.nil,term]].nil,
[term,[t_ 2,term].[t_1,term].nil,
 [NOIDENT,2-th subterm(brother(t_ 1,t_ 2)),t_ 2].
 [NOIDENT,1-th subterm(brother(t_ 1,t_ 2)),t_ 1].nil].
...
[Psymbol,[t_ 2,term].[t_1,term].nil,
 [NOIDENT,head(brother(t_ 1,t_ 2)),brother].nil].
...
[list[pair[atom,list[atom]]],nil,
 [NOIDENT,clpProg,[father(John,Ann),nil].
 [father(Bill,John),nil].
 [father(John,Al),nil].[male(Al),nil].
 [female(Ann),nil].
 [grandfather(X,Z),father(X,Y).father(Y,Z).nil].
 [brother(Y,Z),father(X,Y).father(X,Z).male(Y).nil].
 [sister(Y,Z),father(X,Y).father(X,Z).female(Y).nil].nil].
 nil].
nil]
```

La quatrième phase permet quant à elle de produire un dernier terme correspondant au module que l'on souhaite produire. La syntaxe du module produit est là aussi la même que dans les deux exemples précédents. On donne ici la partie du terme correspondant au terme résultat de la troisième phase :

```
module Prolog

operators global
   X:variable;
   Y:variable;
...
   female:Psymbol;
   female(@):(atom)term;
   grandfather:Psymbol;
   grandfather(@,@):(atom atom)term;
End
...
```

```
Rules for term
   t_ 2:term;
   t_1:term;
   NOVAR
global
[]2-th subterm(brother(t_ 1,t_ 2))=>t_ 2 End
[]1-th subterm(brother(t_ 1,t_ 2))=>t_ 1 End
NORULE End

Rules for Psymbol
   t_ 2:term;
   t_1:term;
   NOVAR
global
[]head(brother(t_ 1,t_ 2))=>brother End
NORULE End
...
Rules for list[pair[atom,list[atom]]]
   NOVAR
global
[]clpProg=>[father(John,Ann),nil].
           [father(Bill,John),nil].
           [father(John,Al),nil].
           [male(Al),nil].
           [female(Ann),nil].
           [grandfather(X,Z),father(X,Y).
                             father(Y,Z).nil].
           [brother(Y,Z),father(X,Y).
                         father(X,Z).
                         male(Y).nil].
           [sister(Y,Z),father(X,Y).
                        father(X,Z).
                        female(Y).nil] End
NORULE End
End
```

La dernière phase, la cinquième, est comme précédemment juste une phase syntaxique qui travaille sur certains mots-clé d'ELAN (End, Rules, etc.) et supprime des éléments décoratifs tels que les mots NORULE, NOVAR, etc. Ces éléments que l'on qualifie de "décoratifs" sont en fait des constantes qui ont été introduites dans les phases de transformation précédentes. On ne donne pas le résultat de cette ultime phase de traduction car le résultat est très proche de celui de la quatrième phase.

On peut ainsi utiliser le même schéma de traduction pour les modules objets, pour les règles avec objets et contraintes ou pour d'autres application comme l'application Prolog développée ici.

Conclusion

Dans ce chapitre, on s'est intéressé à la traduction de programmes afin de pouvoir travailler directement avec les OModules et les règles définis dans les chapitres précédents tout en gardant en vue que le système dans lequel les applications sont exécutées est le système ELAN.

Afin de permettre au formalisme présenté dans cette thèse d'être un outil de développement utilisable dans le système ELAN, nous avons mis en place le processus de transformation présenté dans ce chapitre. Il permet de définir des OModules et des règles avec objets et contraintes dans des modules, qui, une fois transformés, deviennent des modules ELAN standard.

De plus, le processus de transformation présenté dans ce chapitre peut être adapté à d'autres types d'applications que le formalisme objet. Nous en avons donné un exemple avec Prolog. Il peut aussi s'avérer intéressant pour le développement futur du système ELAN notamment, en remplaçant une phase de transformation qui est elle réalisée par le pré-processeur.

Conclusion et Perspectives

Conclusion

Dans cette thèse, nous nous sommes attachés à définir un formalisme en logique de réécriture qui soit adapté au développement de problèmes traités communément par des systèmes de règles de production. En effet, dans des systèmes basés sur la logique de réécriture, un contrôle sur l'application des règles de réécriture est possible grâce aux stratégies. Notre formalisme permet donc d'avoir la possibilité d'utiliser des stratégies dans le développement de systèmes de règles de production.

Après avoir étudié les formalismes utilisés dans les systèmes de règles de production, nous avons mis en évidence la structure suivante : un mécanisme d'inférence qui fait évoluer une base de faits par applications successives de règles. Cette mémoire de travail doit permettre de représenter des faits manipulables par le système de règles. Parmi les différentes formes de représentations possibles de la mémoire de travail, nous avons décidé de représenter chacun des faits la composant par des objets et la base de faits est ainsi construite comme un multi-ensemble d'objets. De plus, dans les problèmes de décomposition de tâches avec ordonnancement, on utilise en général des contraintes pour emmagasiner des informations non utilisables directement au moment de leur génération et de pouvoir les utiliser plus tard dans le processus de décomposition de tâches.

Partant du système ELAN basé sur le concept de règles et de stratégies, nous avons donc étudié comment parvenir à un nouveau formalisme de règles intégrant objets, mémoire de travail et contraintes et comment exécuter ce nouveau formalisme dans l'environnement standard d'ELAN.

Nous avons tout d'abord étudié comment les principes objets pouvaient être codés en logique de réécriture. Des classes sont définies dans des modules objets appelés OModules et pour chaque classe, on peut définir un ensemble d'attributs typés et initialisés. On peut aussi définir un ensemble de méthodes que l'on peut décomposer en deux catégories : les méthodes définies implicitement par le système et qui consistent en la manipulation des attributs (accès, modification) et en la création d'un nouvel objet de chaque classe ; et les méthodes définies par l'utilisateur. Ces dernières sont paramétrables, avec variables locales, et leur corps consiste en une succession d'instructions qui peuvent être des appels fonctionnels, des tests ou des définitions de variables intermédiaires. Chaque classe peut importer des modules ELAN standard utilisables pour sa définition. On peut aussi définir une hiérarchie de classe basée sur le concept d'héritage simple qui permet à toute classe héritant d'une autre classe de pouvoir utiliser les attributs et méthodes de la classe héritée.

Afin d'implanter des classes et objets en ELAN, nous avons choisi le concept objet des langages dits à prototype. Classes et objets y sont représentés de façon similaire par la même structure d'objet. La structure que nous définissons ici est basée sur les concepts suivants : chaque objet, représentant soit une classe, soit un objet, est composé d'une liste d'attributs valués et d'une liste de noms de méthodes permettant de définir les méthodes accessibles par cet objet. Cette structure est propre à chaque objet et chaque objet et chaque classe ont ainsi leurs propres structures. Les méthodes sont par contre implantées par des règles de réécriture non nommées.

Le codage que nous avons défini nous permet, après transformation, de définir un ρ-terme associé à chaque objet avec ses méthodes. Notre implantation est ainsi basée sur la sémantique définie par le ρ-calcul. De plus, nous avons montré que notre implantation permettait d'avoir une préservation du type lors de la transformation d'un objet.

Fort de ce codage des objets en ELAN, nous avons ensuite défini un formalisme de règles permettant de faire évoluer une base d'objets en liaison avec une base de contraintes.

Tout d'abord, nous avons présenté un formalisme de règle de réécriture permettant de manipuler une mémoire de travail composée d'objets. Nous avons ainsi défini un ensemble de primitives permettant de définir des nouveaux objets ou de pouvoir envoyer un message à un objet afin de le modifier. Ces règles peuvent être nommées et on peut alors utiliser le mécanisme des stratégies d'ELAN afin de contrôler leur application.

Ensuite, nous avons enrichi ce formalisme en y ajoutant une gestion des contraintes de type CSP. Les objets sont liés à la contrainte par ce que l'on appelle des attributs contraints qui sont en fait des variables apparaissant à la fois dans les objets de la base et dans la base de contraintes. La base de contraintes, représentée à l'aide de CSP, évolue par ajouts successifs de nouvelles contraintes au fur et à mesure des applications des règles sur la base d'objets. Néanmoins, la base de contraintes peut être manipulée indépendemment et nous avons défini un ensemble de primitives permettant notamment d'obtenir des informations sur les variables contraintes. De plus, à l'aide des stratégies, on peut à tout moment faire un test de satisfaisabilité de la contrainte ainsi qu'obtenir tout ou partie de ses solutions par le solveur.

Précisément, dans notre étude, c'est au solveur COLETTE que nous faisons appel pour tout travail sur les contraintes. Ce solveur est réalisé lui aussi en ELAN et est défini comme une librairie permettant de gérer des contraintes entières sur des domaines finis à l'aide de règles et de stratégies.

Pour la programmation à base de règles avec objets, nous avons illustré notre propos avec l'étude d'un gestionnaire d'ascenseurs. Pour la programmation à base de règles avec objets et contraintes, nous avons donné l'exemple d'un gestionnaire d'impressions.

Après avoir défini à la fois ce formalisme de règles et cette possibilité de définir des OModules, nous avons fourni à l'utilisateur un moyen pour pouvoir utiliser ceci dans le système ELAN. Nous avons présenté le mécanisme de transformation de programmes qui est utilisé pour pouvoir transformer ces OModules en modules ELAN afin de les exécuter.

Cette transformation se fait en cinq phases : les quatre premières sont réalisées en ELAN et la dernière utilise des *shell-scripts* à des fins purement syntaxiques. Ce schéma de transformation en cinq phases est défini de telle sorte que sa structure peut être reprise et appliquée de façon générique pour tout type de transformation.

Perspectives

Les perspectives de nos travaux sont diverses et touchent à différents sujets. Elles portent tout d'abord sur différents aspects de codage des objets qui pourraient évoluer ; ensuite, sur l'étude de différentes améliorations concernant l'intégration des contraintes ; finalement, sur les travaux possibles sur la transformation de programme en ELAN.

Le codage des objets

Une première évolution sur les objets concerne des extensions possibles sur le formalisme que l'on a proposé. Ainsi, on peut assez facilement imaginer que les objets implantés soient constitués d'attributs locaux et globaux, de méthodes globales et locales. Un des avantages de cette évolution serait de pouvoir isoler certaines parties des définitions des classes lors de leur importation par d'autres classes. Mais les moyens utilisés pour cet apport ne seraient pas fondés sur les notions d'importations globale ou locale de modules comme cela est fait en ELAN. En effet, cette solution a déjà été étudiée et la notion de globalité/localité utilisée dans l'importation des modules en ELAN ne correspond pas exactement à la visibilité des attributs ou méthodes telle qu'on l'entend. Une autre solution à envisager serait d'enrichir la structure des objets en ELAN avec une liste d'attributs scindée en deux et une liste d'accès à des méthodes, elle aussi scindée en deux parties. Un tel découpage permettrait une granularité plus fine dans l'héritage et un contrôle plus facile des accès aux attributs et aux méthodes.

On peut se demander aussi ce qu'on pourrait ajouter au codage des objets en ELAN. Ainsi, dans l'implantation actuelle, les stratégies ne sont pas utilisées lors de l'utilisation des méthodes définies dans

les classes. L'utilisation de stratégies permet de contrôler l'application des règles de réécriture du système ; doit-on offrir cette même possibilité aux méthodes, sachant que les méthodes sont implantées dans le système par des règles de réécriture ? Il serait intéressant d'étudier le parallèle pouvant exister entre l'utilisation actuelle des stratégies permettant de contrôler l'application des règles et le fait de pouvoir définir des stratégies au sein d'une définition de classe. D'un point de vue pratique, cela permettrait, soit de définir différentes règles pour une même méthode et d'utiliser les stratégies pour gérer leur application, soit de définir des stratégies pour l'application de différentes méthodes.

A propos des méthodes, dans l'implantation réalisée, nous sommes restrictifs par rapport à ce que la sémantique du ρ-calcul objet permet pour définir des méthodes dynamiquement. En effet, dans le ρ-calcul, on pourrait ajouter à la structure de ρ-termes un nouveau ρ-terme lors des transformations. Cela se traduirait en ELAN par la possibilité d'ajouter des nouveaux opérateurs avec les règles associées en cours d'évaluation par le système. Ce n'est pas possible dans l'implantation actuelle. Mais, si jamais on devait respecter parfaitement la sémantique du ρ-calcul objet et ne pas se contenter de la restriction implantée actuellement, on devrait définir ELAN comme un système réflexif.

Une approche utilisant la réflection en ELAN a déjà été présentée dans la thèse de P. Borovanský [Bor98]. Elle est surtout basée sur le fait de pouvoir interpréter le système par le système lui-même et son analyse est illustrée par un exemple consistant à faire une complétion de programmes ELAN afin de produire un programme en un format REF. Le format REF est un format de représentation interne des données directement interprétable et compilable par le système ELAN. Les aspects réflexifs ont été abordés dans le domaine de la réécriture dans la thèse de M. Clavel [Cla98] où plusieurs applications basées sur la réflexivité en logique de réécriture sont présentées, notamment la construction du langage interne de stratégies du langage Maude [CDE+00]. Par contre, les aspects réflexifs n'ont guère été étudiés de façon plus approfondie en ELAN depuis ces travaux et un vaste champ de recherches reste encore ouvert sur ce sujet.

Une étude de l'héritage simple a été faite dans nos travaux mais une étude plus approfondie serait nécessaire pour permettre une meilleure gestion de l'héritage. Actuellement, seul un héritage simple est permis dans l'extension objet. La hiérarchie de classes s'en trouve déjà très largement simplifiée. Le fait d'autoriser un héritage multiple est usuel dans les langages objet et permettre cette fonctionnalité serait intéressant. La gestion de l'héritage définie dans l'implantation actuelle est basée sur la surcharge des opérateurs : une méthode implantée dans deux classes différentes sera surchargée au niveau de sa déclaration et une nouvelle règle créée pour la méthode héritée, faisait un lien vers la méthode originale. Cette gestion se justifie par le fait qu'ELAN est un système fortement typé et qu'une double définition des méthodes est nécessaire si l'on veut qu'une méthode puisse s'appliquer à la fois à des objets d'une classe mère et d'un classe fille. A priori, les techniques permettant d'éviter la redéfinition des méthodes héritées seraient soit basées sur une approche réflexive du système, soit basées sur la possibilité d'avoir un ordre sur les sortes, c'est-à-dire un système ordo-sorté, comme c'est le cas dans le système OBJ [GKK+87] par exemple.

La gestion des contraintes

Pour la gestion des contraintes dans les règles étendues présentées dans nos travaux, le fait de travailler avec le système COLETTE a permis de définir un ensemble cohérent puisque basé comme ELAN sur les concepts de règles et stratégies. Néanmoins, l'approche utilisée dans la gestion des contraintes a été de désolidariser le plus possible les contraintes utilisées dans les règles du formalisme présenté ici des contraintes utilisées par le solveur. Ainsi, on a bien conçu un codage différent des CSP pour les règles avec objets et contraintes et une passerelle a été définie permettant de passer aisément d'un formalisme à l'autre. De cette façon, on pourrait facilement permettre la connexion à un autre solveur de contraintes plus adapté au problème considéré dans l'application. De même, on pourrait facilement imaginer l'utilisation de différents solveurs [Mon96] coopérants. Les entrées/sorties utilisées afin de collaborer avec un autre système sont détaillées dans la thèse de M. Vittek [Vit94].

Le fait de définir les contraintes par des classes et donc par des objets pourrait être extrapolé en la définition des contraintes comme objets de première classe du langage ELAN. Ce genre de traitement des

contraintes aurait impliqué d'avoir des classes spécifiques aux types de contraintes traitées et d'associer les méthodes adéquates pour leur résolution. Cela permettrait de travailler sur des contraintes et sur la base associée de façon explicite dans les règles. Cela reviendrait à définir des primitives spécifiques de communication encore plus générales que celles présentées dans le formalisme de règles avec objets et contraintes. On devrait alors définir un système où, dans une règle, on peut ajouter une contrainte donnée à la base; on devrait aussi pouvoir tester la satisfaisabilité de la base, récupérer l'ensemble des solutions, etc... Dans les travaux de Y. Caseau [Cas94] ou plus récemment dans des systèmes comme Mozart [VRH99] par exemple, l'intégration des contraintes comme objets de première classe a été étudiée. Le fait de pouvoir intégrer des contraintes à un système comme ELAN permettrait par contre de pouvoir utiliser et promouvoir le langage de stratégies et d'avoir alors un cadre uniforme de développement d'applications.

Formalisme de règles avec objets et contraintes

Le formalisme de règles avec objets et contraintes peut quant à lui se prêter à diverses autres applications que celles présentées dans cette thèse. Il serait notamment intéressant d'utiliser ce formalisme pour d'autres exemples dans lesquels la décomposition des tâches peut s'accompagner de gestion de contraintes.

On pourrait aussi envisager de mettre en place des techniques de vérification adaptées à notre formalisme. En effet, les techniques disponibles sur la terminaison ou sur la confluence par exemple, seraient intéressantes à adapter au formalisme de règles avec objets et contraintes. Tel que nous l'avons défini, les règles de notre formalisme sont traduites en règles standard, et ce serait alors naturel d'utiliser les techniques de vérification sur ces dernières. Néanmoins, il peut être utile, en cours de développement, de procéder à une forme de vérification du système que l'on est en train de décrire. Dans ce cas, un utilisateur aimerait avoir des techniques directement utilisables et adaptées au formalisme qu'il utilise.

Dans certains systèmes à base de règles, on peut permettre à l'utilisateur de donner comme condition d'application d'une règle l'absence d'un objet dans la base de faits. Ainsi, on peut avoir dans le membre gauche des règles des conditions de la forme "*not*(*Object*)" qui désigne le fait qu'un objet de la forme *Object* ne doit pas figurer dans la mémoire de travail pour permettre à la règle de s'exécuter. Ce type de condition n'est pas possible au même niveau que les conditions d'appartenance autorisées dans notre formalisme. En effet, lorsque l'on a une condition d'appartenance d'un objet dans la base de faits, on vérifie la condition par filtrage ou unification. Si jamais on devait vérifier qu'un objet est absent de la mémoire de travail, ce genre de mécanisme n'est pas utilisable; il faudrait alors faire un calcul sur la base de faits. Le type de calcul à effectuer est un parcours exhaustif de la base d'objets afin de pouvoir déterminer si l'objet désigné par la condition négative est bien absent de la base. Ce type de condition pourrait alors être permis dans notre formalisme et on pourrait alors proposer une extension de la règle proposée dans la section 4.2.3 en des règles de la forme:

$$[lab]\ \ O_1 \dots O_k \neg O''_1 \dots \neg O''_l \ \Rightarrow\ O'_1 \dots O'_m \parallel c\ \ [\textbf{if}\ \ t \mid \textbf{where}\ \ l]^*$$

avec les mêmes notations. La notation $\neg O''_j$ désignerait le fait que que l'objet O''_j ne doit pas faire partie de la mémoire de travail. Dans la transformation de la règle du formalisme avec objets et contraintes en règle ELAN standard, ce type de condition n'est pas du tout vérifiable par simple filtrage. On devrait alors transformer ce type de condition du membre gauche en une condition des règles de réécriture de la forme:

$$\textbf{if}\ not(in(O''_j, B))$$

où B désignerait l'ensemble de la base de faits, l'opérateur $in(\_,\_)$ permettrait de tester qu'un objet donné comme premier argument appartient à une base d'objets donnée en second argument. L'opérateur $not(\_)$ permettrait quant à lui d'obtenir la négation recherchée. Ce genre d'extension est tout à fait envisageable dans notre formalisme.

Transformation de programmes

Le mécanisme utilisé pour la transformation de programme est améliorable sur plusieurs aspects.

Un premier point serait de définir une transformation qui aide l'utilisateur lors du développement de son programme. Lors de la transformation des programmes, l'ensemble de la déclaration de la classe est analysé par le système afin de produire le module ELAN correspondant. En cas d'erreur syntaxique dans la déclaration, l'ensemble de l'analyse du OModule échoue et le système retourne un message d'erreur peu explicite car il ne situe pas l'endroit de l'erreur. Certaines améliorations sont réalisables à ce sujet qui permettraient à un utilisateur de pouvoir situer exactement où l'erreur a été faite. De même, en cas d'erreur non plus syntaxique, mais logique cette fois-ci, le système ne détecte l'erreur que lors de la phase d'exécution de l'ensemble du programme. Ce genre d'erreur concerne par exemple un appel de méthode incorrect où on appelle une méthode `m1` au lieu de l'appeler `m2`. Dans ce cas, l'erreur est détectée après la phase de transformation lors du chargement par le système de tous les modules utilisés par l'application. Les erreurs retournées à ce moment sont bien ciblées : on sait où est l'erreur. Par contre, les fichiers mis en cause sont ceux issus de la phase de transformation et non plus les OModules initiaux. L'utilisateur se trouve donc dans la situation de débogguer des modules qu'il n'a pas créés lui-même et qui lui sont donc inconnus. La solution est donc soit de relier l'erreur pointée par le système aux modules de base et de permettre de la corriger avant de relancer la phase de traduction suivie de celle d'exécution, soit de corriger directement les modules issus de la phase de traduction mais dans ce cas, les modules originaux ne correspondent d'ores et déjà plus au problème. Aucune de ces solutions n'apparaît comme satisfaisante.

Une solution alors serait de revoir le mécanisme de transformation, ce qui aurait aussi un autre intérêt : celui d'éviter d'avoir deux phases successives ; une phase de transformation des modules et une phase d'exécution de l'ensemble du programme. Différentes options s'offrent à nous :

- Une première solution consisterait à modifier l'évaluateur du système ELAN afin de détecter, lors du chargement des modules, si le module chargé est un module standard ELAN ou si c'est un module suffixé par `class` dans lequel on définit une classe. Dans ce dernier cas, il déclencherait automatiquement le processus de transformation et poursuivrait ensuite le processus de chargement des modules avec les nouveaux modules créés. Un autre avantage de ce procédé serait qu'il facilite le déboggage de programme car la phase de transformation serait complètement intégrée au système et plus ou moins transparente. Le même genre de procédé serait utilisé pour les modules dans lesquels les règles avec objets et contraintes seraient définies. Le système serait alors composé de plusieurs syntaxes possibles de modules suivant les applications développées.
- On peut aussi imaginer utiliser une autre technique de transformation qui permettraient de générer un autre format à l'issue de la phase de traduction des OModules en modules ELAN. Au lieu de produire du langage ELAN, ce qui nécessite de relancer l'ensemble du système par la suite, on pourrait produire un module directement formulé dans le format d'échange utilisé par le système, à savoir le format EFIX. Différents outils basés sur des techniques développées dans le projet ASF+SDF [VdBHdJ+01] ont été développés et pourraient être réutilisés. D'une manière générale, on peut citer différents travaux présentés sur un site Internet dédié à la transformation de programme [Web].

Le processus de transformation de programme développé ici et l'ensemble des améliorations proposées peuvent être un point de départ à une utilisation potentielle de ces techniques dans le système ELAN : la définition du mécanisme de pré-processing utilisé dans le système. Les techniques générales de transformations de programmes peuvent être appliquées au mécanisme de pré-processing utilisé actuellement dans le système. Le but serait ici de pouvoir éviter cette phase en utilisant d'autres outils. En effet, la phase de pré-processing actuellement implantée génère des appels au parser à différents niveaux du chargement des modules. Ce qui ne favorise pas la modularité des outils utilisés dans le système ELAN et ne permet pas une grande souplesse dans l'architecture du système. Une nouvelle architecture du système est actuellement en cours d'étude et on peut citer les travaux sur MTOM [MRV01], un module permettant d'isoler la phase de filtrage et de la rendre disponible et exportable vers de nombreuses architectures. Ces travaux permettent une architecture plus modulaire d'ELAN pour les années à venir.

Annexe A
Syntaxe des modules objets

<*module objet*>::= **class** <*nom de classe*>
<*héritage*>
<*importations*>
<*attributs*>
<*méthodes*>
End

<*héritage*>::= **from** <*nom de classe*>

<*importations*>::= **imports** <*liste noms de module*>

<*liste noms de module*>::= <*nom de module*> |
<*nom de module*> <*liste noms de module*>

<*attributs*>::= **attributes** <*liste attributs*>

<*liste attributs*>::= <*attribut*> |
<*attribut*> <*liste attributs*>

<*attribut*>::= <*nom de l'attribut*>:<*type*> = <*valeur initiale*>

<*méthodes*>::= <*méthode*> |
<*méthode*> <*méthodes*>

<*méthode*>::= **method** <*nom de méthode*>[(<*arguments*>)] **for** <*type du résultat*>
[<*déclarations de variable*>] < <*corps de méthode*>>

<*arguments*>::= <*argument*>: <*type argument*> |
<*argument*>: <*type argument*>, <*arguments*>

<*déclarations de variable*>::= <*noms de variable*>: <*type*> |
<*noms de variable*>: <*type*> <*déclarations de variable*>

<*noms de variable*>::= <*variable*> |
<*variable*> , <*noms de variable*>

<*corps de méthode*>::= <*appel de méthode*> |
<*test*> ; <*appel de méthode*> |
<*affectation locale*> ; <*corps de méthode*> |
<*corps de méthode*> ; <*corps de méthode*>

<*appel de méthode*>::= <*identificateur*> . <*nom de méthode*>[(<*paramètres*>)] |
<*identificateur*>[(<*valeurs de paramètres*>)]

<*valeurs de paramètres*>::= <*valeur*> |
<*valeur*> <*valeurs de paramètres*>

<*test*>::= **if** <*opération booléenne*>

<*affectation locale*>::= <*variable*>:= <*appel de méthode*>

Bibliographie

[ABC+98] J.M. Andreoli, U.M. Borghoff, S. Castellani, R. Pareschi, and G. Teege. Agent Based Decision Support for managing Print Tasks. In *Proceedings of the PAAM'98, London, UK*, 1998.

[Abr96] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.

[AC96] M. Abadi and L. Cardelli. *A Theory of Objects.* Springer-Verlag, 1996.

[AF92] P. Albert and F. Fages. XRETE: un outil pour les systèmes experts temps réel. *Génie Logiciel & Systèmes Experts*, 28:22–34, 1992.

[AG96] K. Arnold and J. Gosling. *The Java programming language.* Addison Wesley, 1996.

[AL91] M. Ayel and J.P. Laurent. Validation, Verification and Test of Knowledge-Based Systems. Chichester, UK. Wiley, 1991.

[AN00] C. Alvarado and Q-H. Nguyen. ELAN for equational reasoning in Coq. In J. Despeyroux, editor, *Proceeding of 2nd Workshop on Logical Frameworks and Metalanguages*, June 2000. Santa Barbara (California), USA.

[Bar85] H. Barringer. Up and down the temporal way. Technical Report UMCS–85–9–3, Department of Computer Science, Manchester University, Oxford Rd., Manchester M13 9PL, UK, 1985.

[BBC+97] B. Barras, S. Boutin, C. Cornes, J. Courant, J.-C. Filliatre, E. Gimenez, H. Herbelin, G. Huet, C. Munoz, C. Murthy, C. Parent, C. Paulin-Mohring, A. Saibi, and B. Werner. The Coq proof assistant reference manual: Version 6.1. Technical Report RT-0203, Inria (Institut National de Recherche en Informatique et en Automatique), France, 1997.

[BBKK01] E. Beffara, O. Bournez, H. Kacem, and C. Kirchner. Verification of timed automata using rewrite rules and strategies. technical report, LORIA, February 2001.

[BCD+00] P. Borovanský, H. Cirstea, H. Dubois, C. Kirchner, H. Kirchner, P.-E. Moreau, C. Ringeissen, and M. Vittek. ELAN *V 3.4 User Manual.* LORIA, Nancy (France), fourth edition, January 2000.

[BCL87] A. Ben Cherifa and P. Lescanne. Termination of rewriting systems by polynomial interpretations and its implementation. *Science of Computer Programming*, 9(2):137–160, October 1987.

[BDMN79] G.M. Birtwistle, O.-J. Dahl, B. Myhrhaug, and K. Nygaard. *Simula Begin.* Studenlitteratur, 1979.

[BF78] B.G. Buchanan and E.A. Feigenbaum. DENDRAL and META-DENDRAL: Their applications dimension. *Artificial Intelligence*, 11(1–2):5–24, 1978.

[BFKM85] L. Brownston, R. Farrell, E. Kant, and N. Martin. *Programming Expert Systems in OPS5: An Introduction to Rule-based Programming.* Addison-Wesley, 1985.

[BH98] E. Börger and J. Huggins. Abstract State Machines 1988-1998: Commented ASM Bibliography. *Bulletin of EATCS*, 64:105–127, February 1998.

[Bir35] G. Birkhoff. On the structure of abstract algebras. *Proceedings Cambridge Phil. Soc.*, 31:433–454, 1935.

[BKK98] P. Borovanský, C. Kirchner, and H. Kirchner. A functional view of rewriting and strategies for a semantics of ELAN. In M. Sato and Y. Toyama, editors, *The Third Fuji International*

Symposium on Functional and Logic Programming, pages 143–167, Kyoto, April 1998. World Scientific.

[BN98] F. Baader and T. Nipkow. *Term Rewriting and all that.* Cambridge University Press, Cambridge, 1998.

[Bor95] P. Borovanský. Implementation of higher-order unification based on calculus of explicit substitutions. In M. Bartošek, J. Staudek, and J. Wiedermann, editors, *Proceedings of the SOFSEM'95: Theory and Practice of Informatics*, volume 1012 of *Lecture Notes in Computer Science*, pages 363–368. Springer-Verlag, 1995.

[Bor98] P. Borovanský. *Le contrôle de la réécriture: étude et implantation d'un formalisme de stratégies.* PhD thesis, Université Henri Poincaré - Nancy I, 1998.

[BP85] L. Bachmair and D. Plaisted. Associative path orderings. In *Proceedings 1st Conference on Rewriting Techniques and Applications, Dijon (France)*, volume 202 of *Lecture Notes in Computer Science*. Springer-Verlag, 1985.

[Bré98] P. Brézillon. Successes and failures of KBSs in real-world aplications: Report on the International Conference. *Knowledge-Based Systems*, 10:253–257, 1998.

[Cas91] Y. Caseau. Rule-aided constraint resolution in LAURE. In H. Boley and M.M. Richter, editors, *Proceedings of the International Workshop on Processing Declarative Knowledge (PDK'91)*, volume 567 of *LNAI*, pages 237–256, Kaiserslautern, FRG, July 1991. Springer Verlag.

[Cas94] Y. Caseau. Constraint satisfaction with an object-oriented knowledge representation language. *Journal of Applied Intelligence*, 4(2):157–184, 1994.

[Cas98a] C. Castro. Building Constraint Satisfaction Problem Solvers Using Rewrite Rules and Strategies. *Fundamenta Informaticae*, 34:263–293, September 1998.

[Cas98b] C. Castro. *Une approche déductive de la résolution de problèmes de satisfaction de contraintes.* PhD thesis, Université Henri Poincaré - Nancy I, 1998.

[CDE+00] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martì-Oliet, J. Meseguer, and J.F. Quesada. Towards Maude 2.0. In K. Futatsugi, editor, *WRLA2000, the 3rd International Workshop on Rewriting Logic and its Applications, September 2000, Kanazawa, Japon.* Electronic Notes in Theoretical Computer Science, 2000.

[Cir00] H. Cirstea. *Le Rho Calcul: Fondements et Applications.* PhD thesis, Université Henri Poincaré - Nancy I, 2000.

[CK97] H. Cirstea and C. Kirchner. Theorem Proving Using Computational Systems: The Case of the B Predicate Prover. In *Workshop CCL'97*, Schloß Dagstuhl, Germany, September 1997.

[CK99a] H. Cirstea and C. Kirchner. Combining higher-order and first-order computation using ρ-calculus: Towards a semantics of ELAN. In D. Gabbay and M. de Rijke, editors, *Frontiers of Combining Systems 2*, Research Studies, ISBN 0863802524, pages 95–120. Wiley, 1999.

[CK99b] H. Cirstea and C. Kirchner. An introduction to the rewriting calculus. Research Report RR-3818, INRIA, 1999.

[CK01] H. Cirstea and C. Kirchner. Specifying Authentication Protocols Using Rewriting and Strategies. In *Third International Workshop on Practical Aspects of Declarative Languages*, Lecture Notes in Computer Science, Las Vegas, USA, March 2001.

[CKL00] H. Cirstea, C. Kirchner, and L. Liquori. Matching power. Technical Report, LORIA, 2000.

[CKL01a] H. Cirstea, C. Kirchner, and L. Liquori. Matching power. In *Proceedings 12th Conference on Rewriting Techniques and Applications, Utrecht, The Netherlands*, Lecture Notes in Computer Science. Springer-Verlag, 2001. To Appear.

[CKL01b] H. Cirstea, C. Kirchner, and L. Liquori. The rho cube. In F. Honsell, editor, *Proceedings of FOSSACS'2001*, April 2001.

[CL96] Y. Caseau and F. Laburthe. CLAIRE: Combining objects and rules for problem solving. In *Proceedings of the JICSLP'96 workshop on multi-paradigm logic programming,TU Berlin, Germany*, 1996.

S.C.D. - U.H.P. NANCY 1
BIBLIOTHÈQUE DES SCIENCES
Rue du Jardin Botanique
54600 VILLERS-LES-NANCY

[Cla98] M. Clavel. *Reflection in general logics, rewriting logic, and Maude.* PhD thesis, University of Navarre, Spain, 1998.

[Dau89] M. Dauchet. Simulation of Turing machines by a left-linear rewrite rule. In N. Dershowitz, editor, *Proceedings 3rd Conference on Rewriting Techniques and Applications, Chapel Hill (N.C., USA)*, volume 355 of *Lecture Notes in Computer Science*, pages 109–120. Springer-Verlag, April 1989.

[DCKH95] M. Duribreux-Cocquebert, C. Kolski, and B. Houriez. Towards an integration of cognitive ergonomics concepts in knowledge based system development methodologies. In *Proceedings 6th IFAC/IFIP/IFORS/IEA Symposium on Analysis, Design and Evaluation of Man-Machine Systems, M.I.T., Cambridge, USA, June 27-29*, 1995.

[DELM00] F. Durán, S. Eker, P. Lincoln, and J. Meseguer. Principles of mobile maude. In D. Kotz and F. Mattern, editors, *Procs. ASA/MA 2000*, volume 1882 of *Lecture Notes in Computer Science*, pages 73–85. Springer-Verlag, 2000.

[Der82] N. Dershowitz. Orderings for term-rewriting systems. *Theoretical Computer Science*, 17:279–301, 1982.

[Der87] N. Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3(1 & 2):69–116, 1987.

[DHK00] G. Dowek, T. Hardin, and C. Kirchner. Higher-order unification via explicit substitutions. *Information and Computation*, 157(1/2):183–235, 2000.

[DJ90] N. Dershowitz and J.-P. Jouannaud. Rewriting systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 243–320. Elsevier Publishers, Amsterdam, 1990.

[DK77] R. Davis and J. King. An overview of production systems. *Machine Intelligence*, 10:300–332, 1977.

[DK98] H. Dubois and H. Kirchner. Actions and Plans in ELAN. In *Proceedings of the Workshop on Strategies in Automated Deduction - CADE-15, Lindau, Germany*, pages 35–45. B. Gramlich and F. Pfenning, 1998. Technical Report LORIA 98-R-275.

[DK99] H. Dubois and H. Kirchner. Modelling Planning Problems with Rules and Strategies. Technical Report 99-R-029, LORIA, mars 1999. Poster à JFPLC'99, juin 1999.

[DK00a] H. Dubois and H. Kirchner. Objects, rules and strategies in ELAN. Technical Report A00-R-245, LORIA, 2000.

[DK00b] H. Dubois and H. Kirchner. Objects, rules and strategies in ELAN. In *Proceedings of the second AMAST workshop on Algebraic Methods in Language Processing, Iowa City, Iowa, USA*, May 2000.

[DK00c] H. Dubois and H. Kirchner. Rule Based Programming with Constraints and Strategies. In K.R. Apt, A.C. A. C. Kakas, E. Monfroy, and F. Rossi, editors, *New Trends in Constraints, Papers from the Joint ERCIM/Compulog-Net Workshop, Cyprus, October 25-27, 1999*, volume 1865 of *Lecture Notes in Artificial Intelligence*, pages 274–297. Springer-Verlag, 2000.

[DMT00] G. Denker, J. Meseguer, and C. Talcott. Formal Specification and Analysis of Active Networks and Communication Protocols: the Maude Experience. In *Proceedings DARPA Information Survivability Conference and Exposition DICEX 2000, Vol. 1, Hilton Head, South Carolina, January 2000*, IEEE, pages 251–265, 2000.

[DO90] N. Dershowitz and M. Okada. A rationale for conditional equational programming. *Theoretical Computer Science*, 75:111–138, 1990.

[Dur93] J. Durkin. *Expert Systems, Catalog of Applications.* Intelligent Computer Systems Inc., PO Box 4117, Akron, OH 444321-117, USA, 1993.

[EHR+95] K.D. Eason, S. Harker, R.F. Raven, J.R. Brailsford, and A.D. Cross. Expert or Assistant: Supporting Power Engineers in the Management of Electricity Distribution. *AI & Society*, 9(1):91–104, 1995.

[FH86] F. Fages and G. Huet. Complete sets of unifiers and matchers in equational theories. *Theoretical Computer Science*, 43(1):189–200, 1986.

[FHM94] K. Fisher, F. Honsell, and J. C. Mitchell. A Lambda Calculus of Objects and Method Specializatio n. *Nordic Journal of Computing*, 1(1):3–37, 1994.

[Fil78] R. Filman. Personal communication in [Der87], 1978.

[Fis00] O. Fissore. Terminaison par induction. Mémoire de DEA, Université Henri Poincaré – Nancy 1, June 2000.

[FN97] K. Futatsugi and A. Nakagawa. An overview of CAFE specification environment – an algebraic approach for creating, verifying, and maintaining formal specifications over networks. In *Proceedings of the 1st IEEE Int. Conference on Formal Engineering Methods*, 1997.

[For81] C.L. Forgy. OPS5 User's Manual. Technical Report CMU-CS-81-135, Department of Computer Science, Carnegie-Mellon University, 1981.

[Frü99] T. Frühwirth. Constraint Solving with Constraint Handling Rules. In *Proceedings of 12th International Symposium on Languages for Intensional Programming (ISLIP99), Athens, Greece*, June 1999.

[FS94] K. Futatsugi and T. Sawada. Cafe as an extensible specification environment. In *Proceedings of the Kunming International CASE Symposium*, 1994.

[GG97] T. Genet and I. Gnaedig. Termination proofs using GPO ordering constraints. In M. Dauchet, editor, *Proceedings 22nd International Colloquium on Trees in Algebra and Programming, Lille (France)*, volume 1214 of *Lecture Notes in Computer Science*, pages 249–260. Springer-Verlag, 1997.

[GKF01] I. Gnaedig, H. Kirchner, and O. Fissore. Termination of Rewriting with Local Strategies. In *Proceedings of 4th International Workshop on Strategies in Automated Deduction, IJCAR'01, Siena, Italy*, 2001.

[GKK+87] J.A. Goguen, C. Kirchner, H. Kirchner, A. Mégrelis, J. Meseguer, and T. Winkler. An introduction to OBJ-3. In J.-P. Jouannaud and S. Kaplan, editors, *Proceedings 1st International Workshop on Conditional Term Rewriting Systems, Orsay (France)*, volume 308 of *Lecture Notes in Computer Science*, pages 258–263. Springer-Verlag, July 1987. Also as internal report CRIN: 88-R-001.

[Gog78] J.A. Goguen. Some design principles and theory for OBJ-0, a language for expressing and executing algebraic specifications of programs. In *Proceedings of International Conference on Mathematical Studies of Information Processing*, pages 429–475. IFIP Working Group 2.2, Kyoto, Japan, 1978.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, 1983.

[GR89] J. Giarratano and G. Riley. *Expert Systems: Principles & Programming.* PWS-KENT, 1989.

[GRBC94] R.F. Gamble, G.-C. Roman, W. Ball, and H.C. Cunningham. Applying Formal Verification Methods to Rule-Based Programs. *International Journal of Expert Systems*, 7(3):303–339, 1994.

[Gro95] H. Groiss. A Formal Semantic for mOPS5. In *Proceedings of the Seventh IEEE International Conference on Tools with Artificial Intelligence, Washington, DC*, November 1995.

[GT77] J.A. Goguen and J. Tardo. OBJ-0 preliminary users manual. Semantics and Theory of Computation Report 10, UCLA, 1977.

[Gur88a] Y. Gurevich. Algorithms in the World of Bounded Resources. In R. Herken, editor, *The Universal Turing Machine – A Half-Century Story*, pages 407–416. Oxford University Press, 1988.

[Gur88b] Y. Gurevich. Logic and the Challenge of Computer Science. In E. Börger, editor, *Current Trends in Theoretical Computer Science*, pages 1–57. Computer Science Press, 1988.

[GW93] A. Ginsberg and K. Williamson. Inconsistency and Redundancy Checking for Quasi-First-Order-Logic Knowledge Bases. *International Journal of Expert Systems*, 6(3):321–340, 1993.

[HD77] P.E. Hart and R.O. Duda. PROSPECTOR – A computer based consultation system for mineral exploration. (SRI) Technical Note 155, sri, October 1977.

[HL78] G. Huet and D. S. Lankford. On the uniform halting problem for term rewriting systems. Technical Report 283, Laboria, France, 1978.

[HR85] F. Hayes-Roth. Rule-based systems. *Communications of the ACM*, 28(9):921–932, September 1985.

[Hue80] G. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *Journal of the ACM*, 27(4):797–821, October 1980. Preliminary version in 18th Symposium on Foundations of Computer Science, IEEE, 1977.

[Jac99] P. Jackson. *Introduction to Expert Systems. Third edition.* Addison-Wesley, 1999.

[JK86] J.-P. Jouannaud and H. Kirchner. Completion of a set of rules modulo a set of equations. *SIAM Journal of Computing*, 15(4):1155–1194, 1986. Preliminary version in Proceedings 11th ACM Symposium on Principles of Programming Languages, Salt Lake City (USA), 1984.

[JLR82] J.-P. Jouannaud, P. Lescanne, and F. Reinig. Recursive decomposition ordering. In D. Bjørner, editor, *Formal Description of Programming Concepts 2*, pages 331–348, Garmisch-Partenkirchen, Germany, 1982. Elsevier Science Publishers B. V. (North-Holland).

[KB70] D.E. Knuth and P.B. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, Oxford, 1970.

[KK99] C. Kirchner and H. Kirchner. Rewriting, solving, proving. A preliminary version of a book available at `www.loria.fr/~ckirchne/rsp.ps.gz`, 1999.

[KKM88] C. Kirchner, H. Kirchner, and J. Meseguer. Operational semantics of OBJ-3. In *Proceedings of 15th International Colloquium on Automata, Languages and Programming*, volume 317 of *Lecture Notes in Computer Science*, pages 287–301. Springer-Verlag, 1988.

[KKV93] C. Kirchner, H. Kirchner, and M. Vittek. Implementing computational systems with constraints. In P. Kanellakis, J.-L. Lassez, and V. Saraswat, editors, *Proceedings of the first Workshop on Principles and Practice of Constraint Programming, Providence (R.I., USA)*, pages 166–175. Brown University, 1993.

[KKV95a] C. Kirchner, H. Kirchner, and M. Vittek. Designing constraint logic programming languages using computational systems. In P. Van Hentenryck and V. Saraswat, editors, *Principles and Practice of Constraint Programming. The Newport Papers.*, chapter 8, pages 131–158. The MIT press, 1995.

[KKV95b] C. Kirchner, H. Kirchner, and M. Vittek. ELAN *V 1.17 User Manual.* Inria Lorraine & Crin, Nancy (France), first edition, November 1995.

[KL80] S. Kamin and J.-J. Lévy. Attempts for generalizing the recursive path ordering. Unpublished manuscript, 1980.

[Klo90] J.W. Klop. Term Rewriting Systems. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 1, chapter 6. Oxford University Press, 1990.

[KLS96] C. Kirchner, C. Lynch, and C. Scharff. A fine-grained concurrent completion procedure. In Harald Ganzinger, editor, *Proceedings of RTA'96*, volume 1103 of *Lecture Notes in Computer Science*, pages 3–17. Springer-Verlag, September 1996.

[KM95] H. Kirchner and P.-E. Moreau. Prototyping completion with constraints using computational systems. In J. Hsiang, editor, *Proceedings 6th Conference on Rewriting Techniques and Applications, Kaiserslautern (Germany)*, volume 914 of *Lecture Notes in Computer Science*, pages 438–443. Springer-Verlag, 1995.

[KM96] H. Kirchner and P.-E. Moreau. A reflective extension of Elan. In José Meseguer, editor, *Proceedings of the first international workshop on rewriting logic*, volume 4, Asilomar (California), September 1996. Electronic Notes in Theoretical Computer Science.

[Koc91] S. Kocabas. A review of learning. *The Knowledge Engineering Review*, 6(3):195–222, 1991.

[KR98] C. Kirchner and C. Ringeissen. Rule-Based Constraint Programming. *Fundamenta Informaticae*, 34(3):225–262, September 1998.

[KSZ90] D. Kapur, G. Sivakumar, and H. Zhang. A new method for proving termination of AC-rewrite systems. In *Proceedings 10th Conf. on Foundations of Software Technology and Theoretical Computer Science*, volume 472 of *Lecture Notes in Computer Science*, pages 133–148. Springer-Verlag, 1990.

[LA91] CA Los Alamitos. *Validating and Verifying Knowledge-Based Systems.* IEEE Computer Society Press, 1991.

[Lan75] D.S. Lankford. Canonical algebraic simplifications. Technical report, Louisiana Tech. University, 1975.

[Lan77] D.S. Lankford. Some approaches to equality for computational logic: A survey and assessment. Memo ATP-36, Automatic Theorem Proving Project, University of Texas, Austin (Texas, USA), 1977.

[LBFL80] R.K. Lindsay, B.G. Buchanan, E.A. Feigenbaum, and J. Lederberg. *Applications of Artificial Intelligence for Organic Chemistry: The DENDRAL Project.* McGraw-Hill, New York, 1980.

[LG99] M. Le Goc. Ontological models as shared model to validate a knowledge-based system. In *Proceedings of the Twelth Workshop on Knowledge Acquisition, Modeling and Management*, October 1999.

[LJY99] B. Liu, J. Jaffar, and H.C. Yap. Constraint rule-based programming, 1999. http://www.iscs.nus.edu.sg/ joxan/papers/crp.ps.

[LO93] S. Lee and R.M. O'Keefe. Subsumption Anomalies in Hybrid Knowledge Bases. *International Journal of Expert Systems*, 6(3):299–320, 1993.

[LPN86] T.J. Laffey, W.A. Perkins, and T.A. Nguyen. Reasoning about Fault Diagnosis with LES. *IEEE Expert, Intelligent Systems and Their Applications*, 1(1):13–20, 1986.

[Mes92] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[Mey92] B. Meyer. *Eiffel: The Language.* Prentice Hall, 1992.

[MG91] A. Majchrzak and L. Gasser. On Using Artificial Intelligence to Integrate the Design of Organizational and Process Change in US Manufacturing. *AI & Society*, 5(4):321–338, October-December 1991.

[MK97] P.-E. Moreau and H. Kirchner. Compilation Techniques for Associative-Commutative Normalisation. In Alex Sellink, editor, *Second International Workshop on the Theory and Practice of Algebraic Specifications*, Electronic Workshops in Computing, eWiC web site: http://ewic.springer.co.uk/, Amsterdam, September 1997. Springer-Verlag.

[MK98] P.-E. Moreau and H. Kirchner. A compiler for rewrite programs in associative-commutative theories. In *"Principles of Declarative Programming"*, number 1490 in Lecture Notes in Computer Science, pages 230–249. Springer-Verlag, September 1998.

[MOM93] N. Martì-Oliet and J. Meseguer. Rewriting logic as a logical and semantical framework. Technical report, SRI International, May 1993.

[Mon96] E. Monfroy. *Collaboration de solveurs pour la programmation logique à contraintes.* PhD thesis, Université Henri Poincaré - Nancy 1, November 1996. Also available in english.

[Mon99] E. Monfroy. The Constraint Solver Collaboration Language of **BALI**. In Dov Gabbay and Maarten de Rijke, editors, *Frontiers of Combining Systems 2*, Research Studies, pages 211–230. Wiley, 1999.

[MRV01] P.-E. Moreau, C. Ringeissen, and M. Vittek. A pattern-matching compiler. In Didier Parigot and Mark van den Brand, editors, *Proceedings of the 1st International Workshop on Language Descriptions, Tools and Applications*, volume 44 of *Electronic Notes in Theoretical Computer Science*, Genova, April 2001.

[New42] M.H.A. Newman. On theories with a combinatorial definition of equivalence. In *Annals of Math*, volume 43, pages 223–243, 1942.

[Ngu87] T.A. Nguyen. Verifying Consistency of Production Systems. In *Proceedings of the third Conference on Artificial Intelligence Applications Washington D.C.* IEEE Computer Society Press, 1987.

[NO79] G. Nelson and D.C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, October 1979.

[NPLP87] T.A. Nguyen, W.A. Perkins, T.J. Laffey, and D. Pecora. Knowledge Base Verification. *AI Magazine*, 8(2):69–75, Summer 1987.

[Pla78] D. Plaisted. A recursively defined ordering for proving termination of term rewriting sytems. *Dept. of Computer Science Report 78-943*, 1978.

[Rin96] C. Ringeissen. Cooperation of decision procedures for the satisfiability problem. In *Proc. of FroCoS'96*, pages 121–140, Munich, March 1996.

[Rin97] C. Ringeissen. Prototyping Combination of Unification Algorithms with the ELAN Rule-Based Programming Language. In *Proceedings 8th Conference on Rewriting Techniques and Applications, Sitges (Spain)*, volume 1232 of *Lecture Notes in Computer Science*, pages 323–326. Springer-Verlag, 1997.

[Ros73] B.K. Rosen. Tree-manipulating systems and Church-Rosser theorems. *Journal of the ACM*, 20(1):160–187, 1973.

[Rou88] M.-C. Rousset. On the Consistency of Knowledge Bases: The COVADIS System. In *Proceedings of the 8th European Conference on Artificial Intelligence*, pages 79–84, 1988.

[SA97] ILOG SA. ILOG Rules - White Paper, March 1997.

[SC87] R.A. Stachowitz and J.B. Combs. Validation of Expert Systems. In *The Proceedings of the 20th Hawaii International Conference on System Sciences (HICSS)*, volume 1, pages 689–695, 1987.

[Sho76] E.H. Shortliffe. *Computer-Based Medical Consultations: MYCIN.* Elsevier/North-Holland, Amsterdam, London, New York, 1976.

[Sim95] H. Simonis. Application development with the CHIP system. In Gabriel M. Kuper and Mark Wallace, editors, *Constraint Databases and Applications*, volume 1034 of *Lecture Notes in Computer Science*, pages 1–21, Friedrichshafen, Germany, 8–9 September 1995. Springer, 1996.

[SLU89] L.A. Stein, H. Lieberman, and D. Ungar. *Object-oriented concepts, applications, and databases*, chapter A shared view of sharings: The treaty of Orlando, pages 31–48. Addison-Wesley, 1989.

[Smo95] G. Smolka. The Oz Programming Model. volume 1000 of *Lecture Notes in Computer Science*, pages 324–343, 1995.

[SS93] J.G. Schmolze and W. Snyder. Using Confluence to Control Parallel Production Systems. In *Second International Workshop on Parallel Processing for Artificial Intelligence (PPAI-93), Chambéry, France*, August 1993.

[SS94] J.G. Schmolze and W. Snyder. Confluence and Verification for Production Rule Systems. In R. Plant, editor, *Validation and Verification of Knowledge-Based Systems. Proceedings of AAAI-94 workshop, Seattle, Washington*, 1994.

[SS95] J.G. Schmolze and W. Snyder. A Tool for Testing Confluence of Production Rule Systems. In M. Ayel and M.-C. Rousset, editors, *Proceedings of the European Symposium on the Validation and Verification of Knowledge-Based Systems. Université de Savoie, Chambéry, France*, 1995.

[SS96] W. Snyder and J.G. Schmolze. Rewrite Semantics for Production Rule Systems: Theory and Applications. In Michael McRobbie and John Slaney, editors, *Proceedings 13th International Conference on Automated Deduction, New Brunswick NY (USA)*, volume 1104 of *Lecture Notes in Artificial Intelligence*, pages 508–522. Springer-Verlag, July 1996.

[SS97] W. Snyder and J.G. Schmolze. Detecting Redundant Production Rules. In *AAAI-97 Workshop on Verification & Validation of Knowledge-Based Systems, Providence, Rhode Island (USA)*, July 1997.

Résumé

Dans cette thèse nous formalisons des systèmes de règles de production dans le système ELAN basé sur la logique de réécriture où l'application des règles est contrôlée par des stratégies. Le calcul de réécriture fournit une sémantique opérationnelle à ELAN.

Nous avons ainsi été amenés à étendre ELAN tout en respectant sa sémantique. Cette extension comporte tout d'abord la possibilité de définir des classes et des objets en ELAN. Ce langage s'implante en ELAN comme un langage objet à prototype. Nous avons également défini un formalisme de règles travaillant à la fois avec une base d'objets et avec une base de contraintes. Ces deux bases coopérent par l'intermédiaire de variables partagées.

Ce nouveau paradigme de programmation avec règles, objets, contraintes et stratégies nous permet de modéliser des problèmes de planification ou d'ordonnancement.

Mots clés : règles de réécriture, langage objet, contraintes, stratégies, calcul de réécriture, systèmes de règles de production.

Abstract

In this thesis, we design production rule systems in the ELAN system which is based on the rewriting logic and where strategies control the application of the rules. The rewriting calculus gives an operational semantics of ELAN.

Thus, we developed an extension of ELAN that respects its semantics. Firstly, in this extension, we give the possibility to define classes and objects in ELAN. This language is implemented in ELAN as a prototype object-language. Then, we define a new formalism of rules working together with an object and a constraint store such that objects and constraints share variables. The application of the set of rules is controlled by strategies.

This new programming paradigm with rules, objects, constraints and strategies is here used to model problems such that planification or scheduling.

Keywords: rewrite rules, object language, constraints, strategies, rewriting calculus, production rule systems.