RAPPORT DE STAGE

Université de Lorraine - LORIA

Stage de Licence Informatique effectué du 08/04/2019 au 31/05/2019

Mise en place d'une interface pour un système de révision des croyances dans un formalisme attributs-contraintes

Auteur :
Matthias Bertrand

Encadrant entreprise :
Jean Lieber
Encadrant université :
Emmanuel Jeandel

Mai 2019

Table des matières

Remerciements

Je tiens à remercier mon camarade de promotion Julien Romary pour m'avoir dirigé vers Jean Lieber, Jean Lieber pour m'avoir proposé un stage que j'ai toruvé particulièrement intéréssant et instructif et pour son encadrement, en particulier pour la rédaction de ce rapport, et à mes camarades de stage pour m'avoir assisté pour la rédaction de ce document. Merci.

Introduction

Contexte

Le Loria [1] est un centre de recherche mixte se situant au sein du campus Aiguilettes, partageant son infrastructure avec le CNRS [2] et l'Inria [3], deux autres centres de recherche, ainsi qu'avec l'université de Lorraine. Il a pour mission la recherche fondamentale et appliquée en sciences informatiques.

Ce centre est divisé en cinq départements de recherche : «Algorithmique, calcul, image et géométrie», «Méthodes formelles», «Réseaux, systèmes et services», «Traitement automatique des langues et des connaissances», «Systèmes complexes et intelligence artificielle» et comporte 30 équipes réparties parmis ces 5 départements.

Une de ces équipes, l'équipe K, fait partie du département «Traitement automatique des langues et des connaissances». Celle-ci a pour but d'appliquer le processus KDD [4] afin de constituer des bases de connaissances et de concevoir par la suite, des systèmes de résolution des problèmes en raisonnant sur ces bases de connaissances.

Deux types de raisonnement auxquels cette équipe s'intéresse est le *raisonnement à partir de cas* (RÀPC) et la révision des croyances. Le principe du RÀPC consiste à résoudre un problème à partir de la solution d'un autre problème se trouvant dans une base de cas, un cas étant constitué d'un problème, de sa solution et de l'explication de la solution. Cela s'effectue en deux étapes : la remémoration et l'adaptation. La phase de remémoration consiste à rechercher et trouver un problème similaire dans la base de cas. La phase d'adaptation consiste à prendre la solution du problème similaire trouvé dans cette base et l'adapter pour résoudre le problème actuel et ce, à partir de l'explication du cas remémoré.

La révision des croyances est un prncipe d'éliminiation des incohérences dans des formalismes et l'adaptation de la révision des croyances est un principe d'application de la RÀPC utilisant la révision des croyances. Afin de comprendre ce qu'est et en quoi consiste la révision des croyances, voici un exemple.
Soit un agent a et ψ l'ensemble des croyances de a, avec ψ contenant l'affirmation suivante : «Toutes les pizzas contiennent du fromage». Ainsi, cela s'interprète de la façon suivante : a croit que toutes les pizzas contiennent du fromage. Ensuite, lorsque a est confronté à un nouvel ensemble de croyances μ, il est possible que μ entre en contradiction avec ψ. Par exemple, si μ contient l'affirmation suivante : «Les pizzas végétaliennes ne contiennent pas de fromage», μ est en conflit avec ψ. Une révision est ainsi effectuée afin que

1. *Loria* : laboratoire lorrain de recherche en informatique et ses applications - http ://www.loria.fr
2. *CNRS* : Centre national de la recherche scientifique - http ://www.cnrs.fr/
3. *Inria* : Institut national de recherche en informatique et en automatique - https ://www.inria.fr/
4. *KDD* : Knowledge Discovery in Databases - en français : extraction de connaissances dans les bases de données

$\psi \wedge \mu$ soit cohérent. Ainsi, suite à la révision, les croyances ψ de l'agent a pourront, par exemple, devenir les suivantes : «Les pizzas végétaliennes ne contiennent pas de fromage. Toutes les autres pizzas contiennent du fromage». Cette révision est effectuée avec l'opérateur $\dot{+}$ et la formule $\psi \dot{+} \mu$ permet d'exprimer la révision des croyances ψ avec les croyances de μ.

L'adaptation de la révision des croyances permet d'utiliser en plus une base de connaissances lors de la révision. Celle-ci permet d'affecter la révision en adaptant les formules contenues dand cette base.

Ces principes de révision des croyances et d'adaptation peut s'appliquer à beaucoup de formalismes différents, dont PCLC [1].

PCLC est une logique où toutes les formules atomiques [2] sont des contraintes linéaire de la forme $a_1x_1+a_2x_2+\ldots+a_nx_n \leq b$ avec $x_1, x_2, \ldots x_n$ les variables de la contrainte qui sont entiers ou réels, $a_1, a_2, \ldots a_n$ également entiers ou réels et b entier ou réel. Ces formules utilisent les connecteurs de la logique propositionnelle, à savoir les connecteurs $\wedge, \vee, \neg, \Rightarrow, \Leftrightarrow, \oplus$. Ainsi, la formule $\varphi = (3x_1+5x_2+x_3 \geq 0) \vee (2x_1+4x_3 \leq 7) \wedge \neg(x_2+x_3 < 11)$ est une formule syntaxiquement et sémantiquement correcte de PCLC.

Le principe de révision des croyances appliqué à PCLC est le même que celui utilisé dans l'exemple suivant : soient φ et ψ deux formules de PCLC avec $\varphi = (x_1 \leq 10)$ et $\psi = (x_1 \geq 12)$, x_1 étant une variable entière. La formule $\varphi \wedge \psi$ est incohérente car x_1 ne peut pas être inférieur et supérieur à 11 à la fois. La révision $\varphi \dot{+} \psi$ est ainsi effectuée afin de trouver une approximation de ψ notée ψ' pour au final obtenir la formule cohérente $\varphi \wedge \psi'$.

Revisor est une bibliothèque dévelopée au sein du Loria permettant la révision des croyances dans différents formalismes par une machine. Quelques formalismes sont déjà acceptés par Revisor, mais PCLC ne l'est pas encore.

Sujet du stage

Afin de permettre à Revisor d'effectuer la révision en PCLC, deux réalisations différentes ont été effectuées. La première est l'implémentation de l'algorithme de révision PCLC qui a fait l'objet d'un autre stage, stage s'étant déroulé en paralèlle avec le mien. La deuxième a été la mission de mon stage qui a consisté à mettre à jour la plateforme Revisor en y ajoutant un nouveau moteur de révision utilisant cet algorithme afin de permettre l'adaptation de la révision des croyances dans un formalisme proche de PCLC que l'on a appelé PCSFC [3], sans devoir écrire du code compliqué. Ce moteur permet, à partir des instructions saisies, de basculer du formalisme PCSFC au formalisme PCLC et d'effectuer la révision des croyances dans PCLC.

Plan du rapport

Avant de pouvoir mettre en place la nouvelle interface de révision, il a été nécessaire de reprendre l'interface de révision existante ainsi que toute sa documentation et d'étudier le tout afin de comprendre comment fonctionne cette interface et comment y greffer la nouvelle fonctionnalitée. Dans le premier chapitre, je vais

1. *PCLC* : Propositional Closure of Linear Constraints
2. *Formule atomique* : formules qu'on ne peut pas diviser en plusieurs formules
3. *PCSFC* : Propositional Closure of Simple Features-Constraints

donc présenter l'application telle qu'elle était avant le début de ce stage. Ensuite, il a fallu étudier et comprendre le formalisme PCLC étant donné que la syntaxe et sémantique de PCSFC s'appuie sur celle-ci. Je présenterai donc ce formalisme dans le deucième chapitre. Etant donné que le formalisme implémenté permet plus de possibilités d'un point de vue utilisateur, il est nécessaire d'également présenter les éléments supplémentaires ayant été ajoutés. C'est le sujet du chapitre 3. C'est après avoir présenté tout cela que l'on aura tous les outils nécessaires à la réalisation de l'interface de Revisor/PCSFC. Je décrirai donc mes réalisations dans le chapitre 4.

Informations utiles

Il est important de noter que mon travail s'appuie sur celui de William Philbert [1] et que toutes les informations pouvant être trouvées ici à propos du formalisme PCLC ont été prélevés à partir du document de Jean Lieber [2].

Mon stage s'étant déroulé en paralèlle avec un autre, il a fallu trouver des solutions de coordination. Un serveur *Discord*[1] a donc été mis en place afin de pouvoir communiquer de manière instantanée. Un répertoire *GitHub*[2] a également été créée afin de pouvoir effectuer des échanges de code avec mon binôme[3].

1. *Discord* : https ://discordapp.com/
2. *GitHub* : https ://github.com
3. Lien du répertoire : https ://github.com/Dylanlicho/Revisor

Chapitre 1

REVISOR avant le stage

Dans ce chapitre, je vais présenter très brièvement les moteurs de REVISOR déjà existant avant ce stage. Je vais également présenter plus en détails les interfaces de révision mises en place. En annexe se trouvent des descriptions plus détaillées de chacun des moteurs de révision.

1.1 But de REVISOR

REVISOR a pour objectif de fournir aux utilisateurs une bibliothèque permettant la réalisation facile et aisée de la révision des croyances et ce, dans plusieurs formalismes différents. Jusqu'ici, REVISOR propose cinq moteurs de révision qui sont les suivants :

- REVISOR/PL (*Propositionnal Logic*)
- REVISOR/PLAK (*Propositionnal Logic with Adaptation Knowledge*)
- REVISOR/CLC (*Conjunctions of Linear Constraints*)
- REVISOR/QA (*Qualitative Algebra*)
- REVISOR/PCQA (*Propositionnal Closure of Qualitative Algebra*)

Chacun de ces moteurs possède une implémentation *Java* avec des méthodes *statiques* permettant de *parser* des expressions saisies par l'utilisateur dans les formalismes correspondants et d'effectuer ensuite les adaptations par révision des croyances. Tout cela sans devoir à écrire une quantité conséquente de code.

Dans le cas où l'utilisateur souhaite une visualisation rapide de l'adaptation par révision de chaque formalisme, une classe de test possédant des exemples pré-construits est à disposition pour chacun d'entres eux.

Tout cela est accessible facilement par deux types d'interface : une interface console propre à chaque moteur mais possédant toutes la même structure ainsi qu'une interface graphique offrant une application permettant le passage d'un formalisme à un autre en quelque secondes et également la configuration de l'apparence et du fonctionnement de l'application.

1.2 Les interfaces

A l'origine, les moteurs de révision implémentés étaient tous disjoints les uns des autres, c'est-à-dire que chaque moteur possédait sa propre interface, sa propre structure et sa propre exécution. Cela est dû au fait

que chaque moteur a été réalisé par une différent personne et que, bien que toutes les anciennes interfaces étaient écrites en Java, certains moteurs étaient complètement écrits dans différents langages. Ainsi, passer d'un moteur à un autre était long et fastidieux, que ce soit au déploiement ou à l'utilisation. C'est pourquoi William Philbert, ancien stagiaire, a travaillé sur un système permettant d'uniformiser la structure et le fonctionnement de tous les moteurs, rendant plus simple leur utilisation et leur déploiement, tout en conçevant ce système de façon flexible rendant possible l'intégration de nouveaux moteurs de façon aisée à ce dernier.

Tout d'abord, les différents moteurs de Revisor ont été regroupés et organisés en un arbre de projets *Maven*[1]. *Maven* est un outil de construction et gestion de projet automatique développé par *Apache*[2]. L'annexe B illustre l'ancienne arborescence et la nouvelle. Ensuite, tous les sous-projets ont été conditionnés en archive JAR afin de grandement faciliter et accélérer le déploiement et l'utilisation des différents moteurs. Une archive JAR est une archive similaire aux archives ZIP, RAR et autres, mais dédiée au stockage des classes compilées ainsi que les méta-données leur étant associé, le tout formant une librairie. Une archive JAR peut être également exécutable, dans ce cas, l'archive devient une application, exécutable sur toutes les plateformes ayant la *machive virtuelle Java* disponible et installée[3]. Ainsi, utiliser Revisor est devenu simple et rapide.

1.2.1 Les interfaces RevisorConsole

Tous les moteurs de révision disponibles possèdent une interface en mode console, toutes accessible via des archives JAR exécutable, une par interface. C'est d'ailleurs depuis ces interfaces que sont réalisées les implémentations nécessaires pour chaque moteur permettant de garantir leur fonctionnement.

Chaque implémentation de tous les moteurs en mode console doit permettre les fonctionnalités suivantes :

- entrer une expression correspondant au formalisme choisi, évaluer cette expression et afficher son résultat ;
- stocker le résultat de cette évaluation dans une variable n'appartenant pas au formalisme et ce, afin de permettre une saisie et exécution simple de l'expression ;
- obtenir de l'aide sur comment utiliser l'interpréteur du moteur choisi ;
- charger des commandes écrites dans un fichier ;
- réinitialiser l'environement, c'est-à-dire repartir de zéro en oubliant toutes les instructions saisies et résultats calculés ;
- exécuter des instructions spécifiques au formalisme chosi ;

Plus de détails techniques peuvent être trouvés en annexe C. Plus de fonctionnalités diverses peuvent être consultées en annexe E.

1.2.2 L'interface Revisor Platform

Revisor Platform est le nom donné à l'application graphique regroupant tous les moteurs de révision des croyances. Il existe en réalité deux variantes :

- Une application locale conçue avec la librairie *Java Swing*[4] pouvant être simplement lancée en exécutant le JAR contenant le programme. ;

1. *Maven* : https ://maven.apache.org/
2. *The Apache Software Foundation* : https ://www.apache.org/
3. *JAR* : plus d'informations sur les archives Java : https ://en.wikipedia.org/wiki/JAR_(file_format)
4. *Java Swing* : https ://en.wikipedia.org/wiki/Swing_(Java)

— Une application serveur conçue avec les *frameworks*[1] *PrimeFaces*[2] et *PrettyFaces*[3] en utilisant les spécifications de *Java EE*[4], qui bénéficie de la puissance de calcul d'un serveur. Cependant, cette version a été, jusqu'à présent, laissée en suspend, pour la simple raison que l'application locale est prioritaire sur celle-ci. ;

Dans la suite, nous ne considérerons que la version locale.

Etant donné que Revisor Platform est une synthèses de tout les différents moteurs de Revisor, elle doit pouvoir gérer toutes les fonctionnalités proposés par chacun des moteurs. De ce fait, tout ce qui a été expliqué dans la sous-section 1.2.1 est également valable ici. Pour ce faire, lors du lancement de l'application, celle-ci crée une instance en mode console de tous les moteurs disponibles et lie l'entrée standard de la console à la fenêtre de saisie de l'application. Pour que l'utilisateur bascule d'une console à une autre, un menu est à sa disposition. Il suffit d'y choisir l'élement souhaité pour effectuer le changement.

L'interface graphique possède également une visualisation plus claire et épurée, ce qui est rendu possible en ajoutant un code couleur à tous les types de message disponible. Ainsi, par défaut, les commandes exécutées sont en vert, les avertissements en jaune et les erreurs en rouge.

A proximité du menu des moteurs se trouve le menu de configuration. Celui-ci permet de paramétrer les différents moteurs depuis l'interface graphique. En plus de proposer les mêmes paramètres de fonctionnement que les moteurs en mode console, celui-ci permet également des paramètres pour configurer l'apparence de l'interface graphique elle-même. Ainsi, les couleurs des messages, la taille des messages, les bordures, etc. y sont configurables également.

Un *design pattern* de type *Observable/Observer* avec différents types de mise à jour (`ADD`, `VALIDATE`, `EXECUTE`, `CONFIG` et `VISIBILITY`) est présent, permettant une mise à jour de l'affichage de l'interface graphique rapide, efficace et automatisée.

1. *Framework* : ensemble de composants servant à créer des logiciels ou d'autres *frameworks*. - https ://en.wikipedia.org/wiki/Software_framework
2. *PrimeFaces* : https ://www.primefaces.org/
3. *PrettyFaces* : https ://www.ocpsoft.org/prettyfaces/
4. *Java EE* : https ://en.wikipedia.org/wiki/Java_Platform,_Enterprise_Edition

Chapitre 2

Le formalisme PCLC

Afin de pouvoir implémenter le moteur de révision Revisor/PCSFC, il est tout d'abord nécessaire d'étudier ce qu'est le formalisme PCLC, aussi dénommé logique PCLC (abrégé $(\mathcal{L}_{PCLC}, \models)$), étant donné que l'algorithme de révision de ce moteur s'effectue sous ce formalisme. Est expliqué en particulier dans ce chapitre ce qui est nécessaire à la saisie correcte des formules. Nous allons donc rapeler ce qu'est le formalisme PCLC, définir sa notion d'interprétation, sa syntaxe et sa sémantique. Précisons que le but ici n'étant pas d'implémenter l'algorithme de révision des croyances dans PCLC, nous finirons par expliquer que très brievement ce qu'est une formule sous DNF et en quoi consiste cet algorithme.

2.1 Introduction à PCLC

Le formalisme PCLC [2], extension du formalisme CLC, nous permet de travailler avec ce que l'on appelle des contraintes linéaires. Une contrainte linéaire peut être vue comme étant une inéquation avec n inconnues dans le membre de gauche de l'inégalité, chaque inconnue pouvant posséder un facteur (qu'il soit nul ou non), et un nombre constant dans le membre de droit que nous allons nommer seuil. Par exemple, l'inéquation

$$3x > 5$$

est une contrainte linéraire à 1 inconnue, inconnue possédant un facteur de 3 et possède comme seuil 5. Ainsi, cette contrainte s'interprète : «x multiplié par 3 doit être strictement supérieur à 5». Ici, pour que cette contrainte soit vérifiée, en supposant que x est une variable de type entier, il suffit de prendre n'importe quel $x \geq 2$.
Formellement, une contrainte linéaire est de la forme :

$$\sum_{k=1}^{n} a_k x_k \; \mathcal{R} \; b \qquad \text{où } n \in \mathbb{N}^* \,;\, a_k, x_k, b \in \mathbb{R} \,;\, \mathcal{R} \in \{<, \leq, =, \neq, \geq, >\}$$

En règle générale, nous ne possédons pas une seule contrainte mais un ensemble de contraintes. Chaque contrainte de l'ensemble est liée avec des connecteurs, ces connecteurs étant ceux de la logique propositionnelle. Ceci représente la particularité de PCLC vis-à-vis de CLC, car CLC ne possède qu'un seul connecteur (d'ailleurs, $\mathcal{L}_{CLC} \subset \mathcal{L}_{PCLC}$, ce qui peut se traduit par «CLC est un sous-ensemble de PCLC»). Le but d'un tel système est de satisfaire toutes les contraintes, chaque connecteur modifiant les conditions dans lesquel un ensemble de contraintes est satisfaite. Nous allons détailler ces différents connecteurs dans la prochaine

2. *PCLC* : Propositionnal Closure of Linear Constraints

section.

2.2 Syntaxe de $(\mathcal{L}_{PCLC}, \models)$

Les symboles dans $\mathcal{L}_{PCLC}$ qui nous intéressent sont appelés variables. Soit n un entier naturel non nul et constant, l'ensemble des symboles $\mathcal{V}$ est l'ensemble des variables des formules de PCLC avec $\mathcal{V} = \{x_1, x_2, \ldots, x_n\}$.
Une expression de $(\mathcal{L}_{PCLC}, \models)$ est appelée contrainte et correspond à une contrainte linéaire à m variables, $0 < m \leq n$ *(voir section 2.1 pour savoir ce qu'est une contrainte)*.
L'ensemble des formules de $(\mathcal{L}_{PCLC}, \models)$ est le plus petit ensemble tel que :

- Toute contrainte est une formule ;
- Soit φ une formula. Si $\varphi \in \mathcal{L}_{PCLC}$, alors $\neg\varphi \in \mathcal{L}_{PCLC}$;
- Soient φ_1, φ_2 deux formules. Si $\varphi_1, \varphi_2 \in \mathcal{L}_{PCLC}$, alors $\varphi_1 \wedge \varphi_2 \in \mathcal{L}_{PCLC}$, $\varphi_1 \vee \varphi_2 \in \mathcal{L}_{PCLC}$ et ainsi de suite pour les connecteurs $\Rightarrow, \Leftrightarrow, \oplus$.

Un littéral est soit une contrainte, soit une formule $\neg\varphi$ où φ est une contrainte avec $\neg\varphi$ s'écrivant : $\neg(a_1x_1 + a_2x_2 + \ldots + a_mx_m \leqslant b)$ équivalent à $(a_1x_1 + a_2x_2 + \ldots + a_mx_m > b)$. De la même façon, $\neg(a_1x_1 + a_2x_2 + \ldots + a_mx_m = b)$ est équivalent à $(a_1x_1 + a_2x_2 + \ldots + a_mx_m \neq b)$.
Enfin, un terme d'une somme avec facteur nul apparaissant dans une formule peut être omis. Ainsi, $3x_1 + 0x_2 + x_3 \leq 2$ s'écrira $3x_1 + x_3 \leq 2$

2.3 Sémantique de $(\mathcal{L}_{PCLC}, \models)$

Pour pouvoir définir la sémantique de $(\mathcal{L}_{PCLC}, \models)$, il nous faut définir ce qu'est un modèle de $(\mathcal{L}_{PCLC}, \models)$ et pour ce faire, nous allons d'abord définir ce qu'est une interprétation.

2.3.1 Interprétations

Soit φ une formule. Une interprétation $\mathcal{I}$ de φ correspond à un ensemble d'affectations de n valeurs numériques à n variables, avec chaque affectation étant soit une affectation d'une valeur entière pour une variable entière, soit une affectation d'une valeur réelle pour une variable réelle. Par exemple, prenons la formule $\psi = 3x_1 + 2x_2 \leq 10$, $x_1 \in \mathbb{N}$ et $x_1 \in \mathbb{N}$. Une interprétation possible de cette formule est le couple (10, 3), avec $x_1 = 10$ et $x_2 = 3$ (nous allons noter cette interprétation $\mathcal{I}_1$). De manière plus générale, comme la plupart du temps l'ensemble des valeurs que peuvent prendre les inconnus est infini, il existe par conséquent une infinité d'interprétations.

Soit $\mathcal{I}$ une interprétation de φ. On dit que $\mathcal{I}$ satisfait φ (qu'on notera par la suite $\mathcal{I} \models \varphi$) si $\mathcal{I}$ est une solution admissible de φ. Reprenons la formule ψ et son interprétation $\mathcal{I}_1$. Dans ce cas, $\mathcal{I}_1 \not\models \psi$ car le couple $(10, 3)$ n'est pas une solution admissible de ψ (car $3 \times 10 + 2 \times 3 \not\leq 10$). En revanche, soit le couple $\mathcal{I}_2 = (2, 2)$ une deuxième interprétation de ψ, $\mathcal{I}_2 \models \psi$ car $3 \times 2 + 2 \times 2 \leq 10$.

Comme indiqué dans la section 2.1, les formules sont rarement constitués que d'une seule contrainte. Elles en ont en général plusieurs et sont connectées entres-elles avec des connecteurs. Nous allons maintenant décrire ces connecteurs et donner sous quelles conditions les interprétations satisfassent les formules utilisant ces connecteurs, en utilisant une interprétation quelconques $\mathcal{I}$ et deux formules quelconque φ et ψ.

— $\varphi \wedge \psi : \mathcal{I} \models \varphi \wedge \psi$ si $\mathcal{I} \models \varphi$ et $\mathcal{I} \models \psi$;
— $\varphi \vee \psi : \mathcal{I} \models \varphi \vee \psi$ si $\mathcal{I} \models \varphi$ ou $\mathcal{I} \models \psi$ ou les deux ;
— $\varphi \oplus \psi : \mathcal{I} \models \varphi \oplus \psi$ si $\mathcal{I} \models \varphi$ ou $\mathcal{I} \models \psi$ mais pas les deux à la fois ;
— $\varphi \Rightarrow \psi : \mathcal{I} \models \varphi \Rightarrow \psi$ si $\mathcal{I} \models \neg\varphi \vee \psi$
— $\varphi \Leftrightarrow \psi : \mathcal{I} \models \varphi \Leftrightarrow \psi$ si $\mathcal{I} \models (\varphi \Rightarrow \psi) \wedge (\psi \Rightarrow \varphi)$

2.3.2 Modèles

Un modèle d'une formule φ est une interprétation qui satisfait φ. L'ensemble des modèles de φ est noté $\mathcal{M}(\varphi)$.
Soient φ_1, φ_2 deux formules de $\mathcal{L}_{PCLC}$ et $\mathcal{M}(\varphi_1), \mathcal{M}(\varphi_2)$ leur ensemble de modèles respectivements. On dit que $\varphi_1 \models \varphi_2$ si $\mathcal{M}(\varphi_1) \subseteq \mathcal{M}(\varphi_2)$ et $\varphi_1 \equiv \varphi_2$ si $\mathcal{M}(\varphi_1) = \mathcal{M}(\varphi_2)$

2.4 Disjonctive Normal Form

Soit φ une formule de $(\mathcal{L}_{PCLC}, \models)$. On dit que φ est sous *Disjonctive Normal Form* (abrégé DNF [1]) si φ peut s'écrire de la manière suivante :

$$\bigvee_{i=1}^{m} \bigwedge_{j=1}^{n_i} l_{ij} \qquad \text{où } l_{ij} \text{ est un littéral}$$

ce qui veut dire que φ est une disjonction de conjonctions de litéraux.

Toute formule peut-être mise sous DNF. Pour cela, deux étapes :

1. Mise sous *Negative Normal Form* (abrégé NNF [2]) – Une formule sous NNF est une formule dont le connecteur $\neg$ n'apparaît que devant des littéraux. Pour ce faire, on utilise des équivalences permettant de supprimer le connecteur de négation devant les formules, comme les *lois de De Morgan*[3] par exemple.
2. Mise de la formule NNF sous DNF – On transforme la formule mise sous NNF en une formule équivalente s'écrivant comme étant une disjonction de conjonctions de littéraux. Cela se fait en utilisant les mêmes propriétés de la multiplication, qui on rappel sont l'associativité, la distributivité et la commutativité. Des équivalences peuvent être également utilisés.

On remarquera que l'algorithme de mise sous DNF d'une formule de $(\mathcal{L}_{PCLC}, \models)$ est le même que celui pour $(\mathcal{L}_P, \models)$, en utilisant les contraintes linéaires à la place des variables propositionnelles.

L'annexe F montre un exemple de mise sous DNF d'une formule de PCLC quelconque.

2.5 Révision des croyances de $(\mathcal{L}_{PCLC}, \models)$ en bref

Comme indiqué dans l'introduction de ce chapitre, la révision des croyances de $(\mathcal{L}_{PCLC}, \models)$ n'étant pas inclue dans le sujet de l'implémentation de l'interface Revisor/PCSFC, nous nous contenterons de rapidement expliquer la notion de cohérence d'une formule ainsi que de mentionner sur quels principes s'appuie la

1. DNF en français : Forme Normale Disjonctive
2. NNF en français : Forme Normale Négative
3. *Lois de De Morgan* : https ://fr.wikipedia.org/wiki/Lois_de_De_Morgan

révision des croyances dans PCLC.

Une formule φ est dite cohérente si son modèle n'est pas vide, autrement dit si il existe des interprétations permettant de la satisfaire (formellement : $\mathcal{M}(\varphi) \neq \varnothing$).

L'algorithme de révision dans $(\mathcal{L}_{PCLC}, \models)$ possède trois cas différents, les deux premiers cas traitant uniquement une certaine forme de formule et le dernier étant le cas général. Nous allons donc grossièrement et rapidement expliquer chaque cas en nous appuyant de deux formules ψ et μ.

- Cas 1 : ψ et μ sont des conjonctions de littéraux – Dans ce cas, on cherche à calculer une valeur et une formule d^* et ψ'. d^* s'obtient grâce à l'algorithme d'optimisation linéaire du *Simplexe* et ψ' grâce à la *distance de Manhattan* en utilisant d^* et en partant de $\mathcal{M}(\psi)$. On obtient ainsi au final la formule révisée $\psi' \wedge \mu$. Ce bout de l'algorithme prend un paramètre constant nommé ε utilisé pour le calcul de distance. ;
- Cas 2 : ψ et μ sont sous DNF – Dans ce cas, on cherche à se ramener au cas 1 de la manière suivante : sachant que ψ et μ peuvent s'écrire $\psi = \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_m$ et $\mu = \mu_1 \wedge \mu_2 \wedge \ldots \wedge \mu_n$, on calcule d^* pour tous les couples $\psi_i \wedge \mu_j$ possibles et on applique le cas 1 au couple $\psi_i \wedge \mu_j$ avec la plus petite distance d^*. ;
- Cas 3 : cas général – Ce cas se ramène au cas 2, en passant ψ et μ sous DNF.

Chapitre 3

Formalisme de Revisor/PCSFC

Bien que le but de Revisor/PCSFC est de réaliser la révision des croyances dans PCLC, l'interface qui m'a été demandé de réaliser permet non seulement d'effectuer tout ce que PCLC permet de faire, mais également de pouvoir utiliser un langage plus large que PCLC qui, une fois *parsé*, permet de produire des formules de PCLC. Dans ce court chapitre, je vais présenter ce que ce langague, baptisé PCSFC[2], permet de faire. Précisons que la syntaxe utilisée ici est celle de l'interface qui a été développée et présentée dans le chapitre 4.

3.1 Déclarations

En ce qui concerne les variables quantitatives, Revisor/PCSFC permet de déclarer des entiers et des réels utilisés en tant qu'inconnus dans les contraintes linéaires.

```
a, b: integer;
x, y: real;
```

Un type de variable quantitative est le type const permettant de définir le paramètre ε.

```
const eps = 0.2;
```

On peut également déclarer des variables booléennes (ou variables propositionnelles).

```
p: boolean;
```

Nous avons également un type de variable qualitative à nombre de modalités fini à notre disposition qui est l'énumération.

```
enum e("red", "blue", "green");
```

Nous avons enfin le type des formules nous permettant d'y affecter les formules à traiter pour la révision.

```
phi, psi, mu: formula;
```

2. *PCSFC* : Propositional Closure of Simple Feature-Constraints

3.2 Formules

Une formule peut être :

— Soit une formule de PCLC.

— Soit un booléen. Lors de la révision, cette formule est transformée en une contrainte linéaire à une seule inconnue qui doit être supérieure ou égale à 1. Ainsi,

```
phi := p;
```

devient :

$$\varphi = n_p \geq 1$$

— Soit une énumération. Lors de la révision, on associe chaque modalité à un booléen avec la modalité souhaitée associée à un booléen vrai et toutes les autres à faux. De ce fait,

```
psi := e = blue;
```

devient :

$$\psi = \neg(red_e) \wedge blue_e \wedge \neg(green_e)$$

— Soit plusieurs formules reliées entres elles par des connecteurs.

Une affectation particulière de formule est l'utilisation de la pseudo-méthode `revise()` prenant 3 paramètres qui sont les deux formules à réviser et la constante ε. C'est cet appel en particulier qui va effectuer la traduction de PCSFC vers PCLC et appeler l'algorithme de révision qui à son terme, renvoie une formule.

3.3 Connecteurs

Il n'y a pas de connecteurs supplémentaire vis-à-vis de PCLC dans Revisor/PCSFC. Je vais tout de même les détailler ici afin de savoir quel connecteur correspond à quoi.

— `&` : Connecteur de conjonction *et* ($\alpha \wedge \beta$)

— `|` : Connecteur de disjonction *ou* ($\alpha \vee \beta$)

— `!` : Connecteur de négation *non* ($\neg\alpha$)

— `=>` : Connecteur d'implication ($\alpha \Rightarrow \beta$)

— `<=>` : Connecteur d'équivalence ($\alpha \Leftrightarrow \beta$)

— `^` : Connecteur de disjonction exclusive *xor* ($\alpha \oplus \beta$)

Chapitre 4

Réalisation de l'interface de Revisor/PCSFC

D'après les informations que j'ai pu trouver, le formalisme CLC étant un sous-ensemble du formalisme PCLC et vu le travail réalisé par William en ce concerne la généricité des classes constituant les moteurs de révision, j'avais prévu de conçevoir Revisor/PCSFC en tant qu'extension de Revisor/CLC. Cependant, par soucis d'indépendance lié à l'algorithme de révision, il a été convenu que l'implémentation se fasse de façon indépendant vis-à-vis de Revisor/CLC. Pour le moment, seul un fragment de Revisor/PCSFC a été implémenté, à savoir les conjonctions, les disjonctions et les négations. Les différents sections et sous-sections qui vont suivre détaillent chacune une étape de la réalisation dans l'ordre chronologique.
Les diagrammes de classes modélisant les réalisations techniques peuvent être trouvés en annexe H.

4.1 Récupération et installation du projet déjà existant

Avant de pouvoir développer quoi que ce soit, il a fallu récupérer et installer le projet comportant les sources afin de pouvoir travailler dessus. Etant donné que le projet est un projet *Maven* développé sous l'IDE *Eclipse*, j'ai dû installer cet IDE (je voulais utiliser à l'origine l'IDE *IntelliJ*). Avec ceci, il a fallu installer la librairie *lpsolve* afin de faire fonctionner Revisor/CLC. L'installation de cette librairie n'a pas été de tout repos. Il a été nécessaire de déplacer l'archive *JAR* dans le projet local de Revisor/CLC, installer la bibliothèque écrite en *C* de *lpsolve* dans l'environement de travail utilisé (*Ubuntu* dans mon cas) et effectuer la liaison entre la bibliothèque utilisé et le *JAR* afin que celui-ci appelle les bonnes ressources. A noter que les moteurs Revisor/QA et Revisor/PCQA n'ayant aucunes relations avec ma tâche, je n'ai pas pris la peine de les faire fonctionner, d'autant plus que ces moteurs sont partiellement écrits en *Perl*.

4.2 Bases de Revisor/PCSFC : conjonctions, disjonctions et négations

Ma première réalisation a été de construire la base de Revisor/PCSFC en n'implémentant uniquement que les conjonctions, disjonctions et négations. Cette section détaille cette première réalisation, toujours dans l'ordre chronologique.

4.2.1 Grammaire

Il a fallu premièrement concevoir la grammaire de base de Revisor/PCSFC. Etant donné que la syntaxe de PCLC est très proche de CLC et que PCSFC est également très proche de PCLC, j'avais eu comme première idée de reprendre la grammaire de Revisor/CLC. Mais comme celui-ci ne possède pas de *parseur*, il n'a donc pas de grammaire et par conséquent, il a fallu écrire une toute nouvelle grammaire.
Avant de conçevoir cette grammaire, j'ai réalisé en premier lieu des exemples d'utilisation afin de pouvoir obtenir une première approximation de cette grammaire. Voici un exemple pour cette première itération :

```
a : integer;
x : real;
phi, psi : formula;
const eps = 0.2;
phi := ((3a + 5x >= 7) | !(a + 2x > 10.5)) & (a == 2);
psi := (a == 3) & (x < 10);
rho : formula;
rho := revise(phi, psi, eps);
```

Une fois cela fait, pu construire la grammaire de la première itération de Revisor/PCSFC. Celle-ci peut être visualisée en annexe G. On va cependant noter les points importants et pertinents pour les futures itérations ici.

- Le non-terminal `RAW_INSTRUCTION` permet de différencier et d'ajouter les types d'instruction.
- Le non-terminal `DECLARATION` permet de gérer la déclaration de tous les types de variable indépendamment les unes des autres.
- Le non-terminal `UNARY_FORMULA` est utilisé pour gérer tous les différents types de formule atomique indépendamment du non-terminal `BINARY_FORMULA` qui lui, sert à détecter les formules binaire liées par des connecteurs, ainsi que les parenthèses et les négations.
- De la même manière, dans le membre gauche d'une contrainte linéaire, le non-terminal `CONSTRAINT_TERM` permet de définir ce qu'est un unique terme (dans notre cas, un inconnu avec éventuellement un coefficient), indépendamment du non-terminal `CONSTRAINT_TERM_LIST` qui est utilisé pour gérer tous les termes du membre gauche de la contrainte et de déterminer les opérateurs arithmétique possibles entre deux termes.
- Le non-terminal `BINARY_FORMULA_OPERATOR` permet de facilement ajouter de nouveaux connecteurs de formule. Celui-ci en particulier est précisé ici car il sera utile pour les prochaines itérations.
- `COMMENT`, `START_OF_COMMENT` et `END_OF_COMMENT` ont pour but de gérer les commentaires indépendament du reste de la grammaire.
- La totalité des opérateurs (arithmétique, affectation, comparaison, ...), caractères et chaînes de caractères ayant une signification particulière (début de commentaire, fin d'instruction, mot clé d'un type de variable, etc.) possèdent leur propre règle de grammaire afin de faciliter les modifications. Par exemple, si on souhaite changer le caractère de fin d'instruction par un dièse, il suffit de trouver le non-terminal `END_OF_INSTRUCTION` et de remplacer le caractère ou la chaine de caractère actuellement utilisé par un `#`. Toutes ces règles sont situées en fin de grammaire afin de les trouver rapidement.

Cette première gramaire a été implémentée avec les outils *JFlex* et *JavaCup*. *JFlex* permet de faire de l'analyse lexicale en construisant des *tokens* (ou symboles) à partir de chaînes de caractères et *JavaCup* permet de faire de l'analyse syntaxique en récupérant les *tokens* produits par *JFlex* pour en faire ensuite des

phrases (appelées «règles de grammaire»), permettant au final d'exécuter du code en fonction de ce qui a été analysé. Dans la suite, les mises à jour de grammaire seront toujours effectuées avec ces deux outils.

4.2.2 Structure et code

Dans cette première itération, le plus difficile a été d'adapter l'interface de William afin de permettre l'implémentation de Revisor/PCSLC. Dans certains cas, ceci s'est avéré très facile, comme la définition des formules unaires et binaires et dans d'autres cas, difficile, comme la gestion des variables ou la représentation des formules.

En premier lieu, il a fallu créer l'interface en mode console permettant la saisie des commandes. Pour ce faire, diverses classes héritant des classes de l'interface de William ont été créées. Les classes `AbstractRevisorConcolePCSFC` et `RevisorConsolePCSFC` permettent l'exécution des mécanismes de traitement des formules saisies. `RevisorEnginePCSFC` est la classe qui a le rôle d'effectuer les opérations de transformation des formules (la révision des croyances par exemple). La classe `RevisorPCSFCConsoleMode` permet de lancer l'interface en mode console. Enfin, une redéfinition de la classe `ConsoleMode` a été faite, la nouvelle classe étant `ConsoleModePCSFC`, afin de changer l'affichage des commandes disponibles.

Il a fallu ensuite implémenter les déclarations et les types de variables. C'est ici que j'ai été confronté au premier problème : l'interface de William ne permet pas la gestion de différents types de variable. De plus, lorsque d'une variable non déclarée est utilisée, elle est implicitement déclarée lors de l'execution de l'instruction. Or dans notre cas, toutes les variables doivent être déclarées par l'utilisateur. Le système de macro de l'interface ne peut donc pas être utilisé. Pour résoudre ce problème, deux mécanismes ont été implémentés :

— Toutes les classes dans le *package* `fr.loria.k.revisor.engine.revisorPCSFC.console.tos` ont pour rôle la gestion des variables déclarées/non déclarées ainsi que le typage des variables. Les valeur des constantes sont également toutes stockées ici.
— La classe `PCSFCFormulaVariableList` dans le *package* `fr.loria.k.revisor.engine.revisorPCSFC.pcsfc` est la classe stockant les formules associées aux identificateurs de variables de type `formula`.

Une fois cela implémenté, la seule chose à faire restante en ce qui concerne les déclaration a été de définir les instructions de déclaration. La classe abstraite `PCSFC_Declaration` permet la récupération des identifiants des variables ainsi que l'analyse sémantique d'une déclaration. Enfin, chaque type de variable possible possède sa propre classe instruction héritant de cette dernière. Chacune de ces classes permettent de traiter l'exécution et la sortie console indépendamment, rendant possible l'existence de plusieurs mode de déclarations en parallèle. Dans cette première itération, nous avons donc quatre classes : `PCSFC_DeclarationInteger`, `PCSFC_DeclarationReal`, `PCSFC_DeclarationFormula` et `PCSFC_DeclarationConstant`.
Ici, l'analyse sémantique est très simple, elle se contente d'empêcher les doubles déclarations de variable.

Après cela, les affectations de formules unaires sans connecteurs ont été implémentées. C'est également à ce moment précis que la représentation des formules a été décidée, ce qui a provoqué le deuxième problème majeur rencontré : l'interface a été pensée afin qu'elle adapte un système de représentation des formules déjà existant. Ici, étant donné que ce moteur de révision n'existe pas encore, il n'y a pas de système de représentation des formules existant. Or, mon but initial était de créer un système de formules effectuant à la fois le travail de l'interface et celui du système adapté. C'est une fois cela comprit que j'ai implémenté

un système de représentation des formules similaire aux autres moteurs qui est en fait composé de deux systèmes : un premier système qui est utilisé par l'interface qui s'occupe des analyses et affichages et un deuxième système indépendant du premier qui s'occupe des opérations. C'est d'ailleurs à partir de ce second système que la révision des croyances est effectuée. Les classes du premier système se trouvent dans le package `fr.loria.k.revisor.engine.revisorPCSFC.console.formula` et celles du deuxième dans le package `fr.loria.k.revisor.engine.revisorPCSFC.pcsfc`.
Les formules possédant maintenant une représentation, l'implémentation des instructions d'affectations de formules a été effectuée. Celle-ci crée des formules du premier système. A partir de celles-ci, les vérifications nécessaires sont effectuées et enfin, une formule du second système est créée et est immédiatement associée à l'identificateur de variable utilisée dans l'affectation. La classe instruction gérant les affectations est la classe `PCSFC_Assignment`.
L'analyse sémantique ici est assez conséquente. Elle vérifie tout d'abord que la variable qu'on tente d'affecter existe et est bien une variable de type `formula`. Ensuite, si le terme de droite de l'affectation est une contrainte linéaire, alors on vérifie que les variables utilisées dans la contrainte existent toutes et sont toutes bien des réels ou des entiers. Dans le cas où le terme de droite est un identificateur de variable, on vérifie que celle-ci existe et que l'identificateur est un identificateur de formule.

Les affectations étant prêtes, avant de continuer, un système d'affectation de formule par défault a été mis en place. Celui-ci permet d'affecter une formule par défault à toute variable de type `formula` venant juste d'être déclarée. Ce système affecte des tautologies à ces variables.

Après cela, l'instruction d'affectation a été mis à jour afin de permettre la révision. Le terme de droite d'une affectation peut maintenant être `revise()`, employant trois identificateurs de variable.
Le lien avec l'algorithme de révision est créé ici et est appelé à l'aide d'une simple méthode. Ainsi, l'algorithme de révision est entièrement indépendant du reste de l'interface.
L'analyse sémantique sur le terme de la révision vérifie que les deux premiers identificateurs sont des identificateurs de formules et que le troisième est un identificateur de constante.
L'interface de William a été simple à adapter ici. Cette partie s'est donc déroulée sans encombres.

La fin de cette première itération a été marquée par l'implémentation des connecteurs de base unaires et binaires des formules, à savoir les connecteurs `&`, `|` et `!`.
A ce niveau, l'analyse sémantique n'effectue rien de spécial.
De la même manière que pour la révision, l'interface déjà en place s'est très bien adaptée. Cette partie a donc été facile et rapide à implémenter.

4.3 Formules binaires avancées de Revisor/PCSFC

Cette deuxième itération a été très rapide par le fait que l'implémentation des connecteurs binaires a été très bien réalisée. C'est également durant cette itération que le système de traduction des formules de PCSFC vers PCLC a été implémenté.

4.3.1 Grammaire

Comme pour la première itération, des petits exemples se sont imposés afin de pouvoir obtenir un bref aperçu de ce qui est possible à l'issue de cette itération. En voici un :

```
a : integer;
x : real;
phi, psi : formula;
const eps = 0.2;
phi := ((3a + 5x >= 7) => !(a + 2x > 10.5)) & (a == 2);
psi := ((a == 3) <=> (x < 10)) ^(.4a != 6);
rho : formula;
rho := revise(phi, psi, eps);
```

La grammaire ici est quasiment la même. Une seule différence est présente : Le terminal `BINARY_FORMULA_OPERATOR` s'est vu rajouter les nouveaux opérateurs de formule binaire (donc les opérateurs `=>`, `<=>` et `^`).

4.3.2 Structure et code

Pour chaque nouvel opérateur, une nouvelle classe a été créée dans le *package* `fr.loria.k.revisor.engine.revisorPCSFC.console.formula`. Leur structure est exactement la même que pour les connecteurs binaires qui étaient déjà implémentés.
Un élément important est que bien que ces connecteurs supplémentaires fassent partie de la syntaxe de PCLC, l'algorithme de révision ne traite que les formules avec les connecteurs implémentés durant la première itération. De ce fait, dans Revisor/PCSFC, ces nouveaux connecteurs sont considérés comme étant non valides dans PCLC. Ainsi, il a fallu implémenter un système de traduction des formules de PCSFC vers des formules de PCLC.
La classe abstraite `PCSLCFormula` se voit ajouter deux nouvelles méthodes abstraites : `toPCLC()` qui est la méthode de traduction et `canRevise()` qui est une sécurité permettant de s'assurer que la formule peut être bien révisée. C'est la classe appellant l'algorithme de révision qui appelle ces deux méthodes durant la validation de l'instruction.

4.4 Booléens et énumérations

Cette troisième itération a été un peu plus longue de par le fait que l'introduction de nouveaux types de variables ayant tous leurs propres manière(s) d'utilisation(s) nécéssite de nouvelles analyses sémantique et que ces nouveaux types, ne faisant pas parti de PCLC, doivent être traduites avant la révision.

4.4.1 Grammaire

Toujours comme avant, des exemples ont été réalisés afin de visualiser les possibilités à l'issue de cette itération. En voici un :

```
psi, mu, rho: formula;
x, y: real;
const eps = 1.1111;
enum color("red", "blue", "green");
is_green: boolean;
```

```
psi := 3x >= 4.1 => (color = "red" & !is_green);
mu := .9x - y < 2 | color = "blue";
rho := revise(psi, mu, eps) ;
```

La grammaire a été modifiée à deux endroits :
— Le non-terminal `DECLARATION` possède deux nouvelles règles, une par nouveau type de variable. Ces nouvelles règles définissent ensuite la syntaxe requise pour chacunes d'entre elles.
— Le non-terminal `UNARY_FORMULA` possède une nouvelle règle, utilisée pour les énumérations, étant donné que celles-ci ne doivent pas être utilisées seules mais avec une de leurs modalités.

4.4.2 Structure et code

Deux nouvelles instructions de déclaration héritant de la classe abstraite `PCSFC_DECLARATION` ont été ajoutées, une pour les booléens et une pour les énumérations.
Etant donné que les énumérations ne s'utilisent pas seules, une nouvelle classe s'appellant `PCSFC_EnumIdentifierLitteral` a été crée. Celle-ci hérite de `PCSFC_IdentifierLitteral` et surcharge la méthode de vérification de sa super-classe afin de pouvoir effectuer les analyses sémantiques supplémentaires nécessaires et d'associer la modalité à l'identificateur de variable. Etant donné que les identificateurs de booléen s'utilisent exactement de la même façon que les identificateurs de formule, les booléens utilisent la classe déjà existante pour les identificateurs utilisés en tant que littéral.
La traduction des formules employant ces nouveaux types utilise le système implémenté dans la deuxième itération. Ainsi, il a juste fallu définir les modèles de traduction et préciser que ces formules ne peuvent pas être révisées. L'application gère le reste tout seul.

4.5 Réalisations mineures, améliorations et interface graphique

Au fur et à mesure de l'avancée de l'application, j'ai effectué des ajouts de ma propre initiative. Ces ajouts rendent l'interface plus agréable à utiliser et plus ergonomique. Certaines reprennent les fonctionnalités de base communs à tous les moteurs de révision et d'autres sont complètement nouvelles.
La commande `clear` permet de réinitialiser le système. Bien que cette commande soit commune, elle a été redéfinie afin de pouvoir en plus réinitialiser le système de variables de notre moteur de révision.
La commande `printvars` affiche toutes les variables déclarées, ainsi que leur contenu si elles en ont un. Saisir le nom d'une variable seule permet d'afficher uniquement le contenu de celle-ci.
La commande `load` a été très légèrement revue afin de modifier le formatage des erreurs que celle-ci peut produire. Autrement, elle fonctionne identiquement aux autres moteurs de révision.

La prochaine réalisation a été de mettre à jour l'application graphique *swing* afin de pouvoir utiliser Revisor/PCSFC avec celle-ci. Ceci s'est effectué en deux étape.
La première étape a consisté à rendre possible la saisie d'instructions dans l'application. Cette étape a été très courte, car il a juste fallu rajouter une ligne de code pour que tous le reste se mette à jour tous seul. Cette ligne de code correspond à l'instantiation de `RevisorConsolePCSFC` .
La deuxième étape, plus longue, a consisté à rendre Revisor/PCSFC compatible avec l'affichage en mode LaTeX. Pour ce faire, toutes les méthodes `createOutput()` de toutes les classes instruction ont été modifiées et toutes les méthodes `toString()` des classes formule ont été modifiées.
Enfin, j'ai pris un peu de temps à jongler avec les paramètres de l'application afin d'y ajouter de nouvelles

fonctionnalités utiles pour notre moteur de révision, comme par exemple accorder la possibilité d'afficher les formules sous le formalisme PCLC avant la révision.

Conclusion

Ce stage m'a permis de mettre en pratique les compétences de développement apprises tout au long de la licence, ainsi que les connaissances théoriques et pratiques acquises de la théorie des langages, de l'analyse syntaxique et de la théorie des modèles. Celui-ci m'a même permis de les approfondir (nouvelles notions en ce qui concerne les types génériques des classes et utilisation des mêmes chaînes de caractères pour la création de différents symboles dans l'outil d'analyse lexical *JFlex* en passant par des états différents).

Ce stage m'a apporté des notions très utiles qui me seront utiles pour la suite de mes études, que ce soit dans le domaine de la théorie des modèles, le développement ou la rédaction de documents.

J'ai mis en place une interface adaptant bien celle qui existe déjà et qui est complètement indépendante de l'algorithme de révision. Ainsi, même si celui-ci n'est pas implémenté et/ou ne fonctionne pas, l'interface est intégralement opérationnelle.

De ce fait, le moteur de révision Revisor/PCSFC peut être utilisé en toute simplicité, que ce soit en mode console ou dans une interface graphique.

Il reste tout de même du travail. En ce qui concerne Revisor/PCSFC, ce moteur de révision peut être amélioré et dispose de nombreuses perspectives d'évolution. La table des symboles et la table des formules peuvent être intégrées au sein du moteur au lieu qu'elles soient toutes les deux à part en tant que Singleton. A quelques endroits demeure également de la duplication de code. On peut ajouter de nouveaux types de variables et de nouvelles formules pouvant être affectées, tant que celles-ci peuvent être mises sous formule de PCSFC et donc, sous formule de PCLC. Si un jour un algorithme de révision propre à PCSFC voit le jour, celui-ci pourrait être implémenté en parallèle avec l'algorithme de révision de PCLC. En parlant de la révision, le système de convertion des formules de PCLC vers PCSFC n'est pas entièrement terminé. Toujours en ce qui concerne la révision, l'adaptation de la révision des croyances reste à faire ici. Enfin, de nouvelles instructions et de nouveaux paramètres peuvent être ajoutés afin de rendre l'interface plus conviviale et/ou plus ergonomique à utiliser.
Avec ceci, l'interface web demeure toujours incomplète. Avec trois moteurs utilisables avec un interpréteur, cette interface serait très utiles car elle permettrait d'éviter à devoir tout télécharger et permettrait à tous d'accéder à Revisor Platform[1] facilement. De plus, le fait que Revisor/CLC, Revisor/QA et Revisor/PCQA ne possèdent toujours pas d'interface utilisateur sous la forme de commances rendra cette interface web encore plus avantageuse une fois que tous ces moteurs en auront un.

1. *Revisor Platform* : nom de l'interface graphique commune (*Swing* et web)

Annexes

Annexe A

Descriptions des moteurs de révision

Dans la suite, pour la présentation des moteurs de révision, seul un exemple sera donné afin de comprendre essentielement comment ces moteurs fonctionnent.

A.1 Revisor/PL et Revisor/PLAK

Revisor/PL permet d'effectuer de la révision des croyances en logique propositionnelle. L'opérateur de révision de Revisor/PL utilise les *distances de Hamming*[2].

Une révision se déroule de la manière suivante :

1. On commence par réinitialiser le poids de toutes les variables à 1 ;
2. On donne une formule d'origine, ici, une recette de pizza au fromage et à la tomate : `"cheese & tomato & pizza"` ;
3. On donne une formule cible à partir de la formule d'origine. Dans l'exemple, on souhaite obtenir une recette de pizza sans à la fois de la tomate et du fromage : `"!(tomato & cheese)"` ;
4. On définit ensuite la base de connaissance. Ici, on n'en utilise pas, donc aucune formule n'est nécessaire ;
5. Nous avons ensuite la possibilité d'ajouter un ou plusieurs poids aux variables. Nous choisissons ici d'ajouter un poids de 2 à la tomate afin de d'exprimer le souhait de vouloir se débarasser de la tomate plus difficilement ;
6. Nous terminons par lançer la révision ;

Avec notre exemple, la révision produit le résultat suivant : `(pizza & tomato & !cheese)`, se qui se traduit par une recette de pizza avec de la tomate et sans fromage.

Revisor/PLAK est une extension de Revisor/PL, permettant d'ajouter des règles d'adaptations. Une règle d'adaptation permet, grossièrement, de donner explicitement la possibilité de remplacer une variable par une autre. Par exemple, la règle :

$$c \wedge a \rightsquigarrow c \wedge b$$

2. *Distance de Hamming* : https ://fr.wikipedia.org/wiki/Distance_de_Hamming

peut se traduire de la manière suivante : «c et a peut être remplacé par c et b». De la même manière que les variables, les règles peuvent se voir attribuer un poids (par défaut à 0.3 pour toutes les nouvelles règles), leur permettant d'ajouter de la priorité lors de la révision (en d'autres termes : définit si il est "facile" d'effectuer la subtitution ou non).

En reprenant l'exemple utilisé pour REVISOR/PL, nous ajoutons la règle d'adaptation suivante :

$$pizza \wedge cheese \rightsquigarrow pizza \wedge meat$$

et lui attribuons un poids de 2. Cela se traduit de la manière suivante : «une pizza avec du fromage peut être remplacée par une pizza avec de la viande et cette substitution est très facile à effectuer». Ainsi, REVISOR/PLAK nous produit le résultat suivant : `(pizza & tomato & meat & !cheese)`. De façon informelle, cela donne une recette de pizza avec de la tomate et de la viande et sans fromage.

A.2 REVISOR/QA et REVISOR/PCQA

Etant donné la complexité de ce genre de formalisme combiné à mon manque de connaissance dans le domaine, les explication et démonstrations seront très informels et brefs et ce, afin de rendre cette partie la plus compréhensible possible.

REVISOR/QA permet de travailler dans ce que l'on appelle des algèbres qualitatives. Un algèbre qualitatif permet, grossièrement, de travailler avec des ensemble (au sens mathématique du terme). Il en existe plusieurs et ceux pouvant être utilisé dans REVISOR/QA sont Allen [1] et RCC8 [2]. Nous allons ici présenter rapidement RCC8 et donner un exemple l'utilisant.

L'algèbre RCC8 définit un ensemble d'opérations sur les ensemble. En considérant deux ensembles X et Y, voici les opérations pouvant être effectués :

- X DC Y (DisConnected) : X et Y sont complètement disjoints ;
- X PO Y (Partially Overlapping) : X et Y ont une partie de leurs éléments en commun ($X \cup Y$) ;
- X EC Y (Externally Connected) : Les bords des ensembles X et Y sont en contact mais sont autrement disjoints ;
- X EQ Y (EQual) : X et Y sont égaux ($X \subseteq Y$), c'est-à-dire qu'ils possèdent exactement les mêmes éléments ;
- X TPP Y (Tangential Proper Part) : X est inclus dans Y et touche son bord ;
- X TPPI Y (Tangential Proper Part Inverse) : Y est inclus dans X et touche son bord ;
- X NTTP Y (Non-Tangential Proper Part) : X est inclus dans Y et ne touche pas son bord ;
- X NTTPI Y (Non-Tangential Proper Part Inverse) : Y est inclus dans X et ne touche pas son bord ;

La figure suivante [3] illustre les différentes opérations expliquées :

1. *Allen* : https ://en.wikipedia.org/wiki/Allen%27s_interval_algebra
2. *RCC8* : https ://en.wikipedia.org/wiki/Region_connection_calculus
3. Source de l'image : Wikipédia - https ://upload.wikimedia.org/wikipedia/en/1/1c/RCC8.jpg

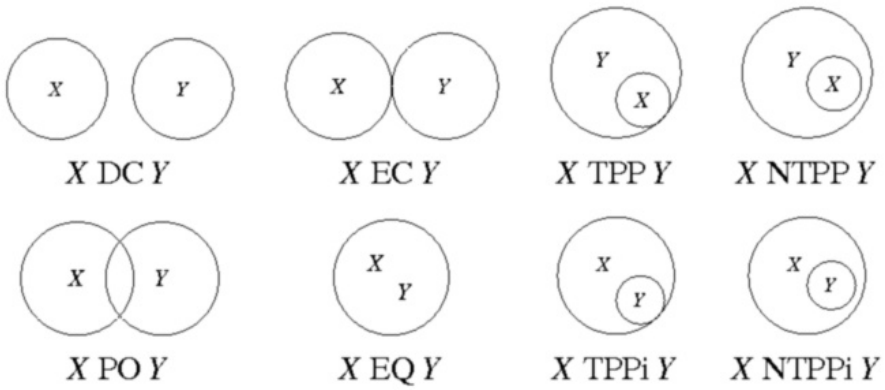


FIGURE A.1 – Les huits opérations de RCC8

Voici le déroulement d'une révision avec REVISOR/QA utilisant l'algèbre RCC8 :

1. On commence par donner une formule source. Nous allons prendre ici celle de la classe de test de REVISOR/QA pour plus de simplicité. Une exploitation agricole souhaite changer sa production de maïs en miscanthus[1]. Cette exploitation se trouve a proximité d'une rivière (notée `lit_mineur`) entourée (relation NTPP) d'une plaine inondable (notée `lit_majeur`). La parcelle de maïs se trouve en partie dans la plaine inondable (relation PO). Sous formule, cela donne :
   `(lit_mineur {ntpp} lit_majeur) & (parcelle(maïs) {po} lit_majeur)`

2. On définit ensuite une cible. Comme indiqué dans le premier point, l'exemple consiste à transformer la parcelle de maïs en parcelle de miscanthus. Ainsi, il nous reste uniquement : `(lit_mineur {ntpp} lit_majeur)`

3. Après cela, on remplit ensuite notre base de connaissances avec des contraintes. Ici, le miscanthus ne peut pas être planté dans des zones inondés. De ce fait, la future plantation de miscanthus et la plaine inondable doivent au mieux, uniquement avoir leurs bords en contact (relations DC et EC). Cela donne :
   `(parcelle(miscanthus) {dc,ec} lit_majeur)`
   On sait aussi qu'on veut maintenant planter du miscanthus dans la parcelle. Cette substitution est ainsi également ajoutée.

4. On termine par lancer la révision.

Le résultat de cette révision est le suivant :

```
(lit_majeur {nttpi} lit_mineur);
(lit_majeur {ec} parcelle(X0);
(lit_majeur {ec} parcelle(miscanthus));
(lit_mineur {dc} parcelle(X0));
(lit_mineur {dc} parcelle(miscanthus));
```

1. *Miscanthus* : plante originaire d'Afrique et d'Asie du sud très utilisée dans les secteurs agricoles, industriels et de l'énergie

```
(parcelle(X0) {eq} parcelle(miscanthus);
```

Ce qui se traduit de la manière suivante :

- La rivière est toujours entourée d'une plaine inondable (conformément au but).
- On a traité ensuite la parcelle de miscanthus ainsi qu'une parcelle de X0 (quelconque) en paralèlle afin de réduire la zone d'exploitation pour que la parcelle soit au pire en contact avec la plaine inondable (relation EC) et au mieux disjoints (relation DC).
- Enfin, on précise que la parcelle de X0 est en réalité la parcelle de miscanthus qui permet de vérifier qu'on trouve bien les même résultats.

Ainsi, l'exploitation agricole a été adaptée en respectant les contraintes en réduisant la zone d'exploitation.

Jusqu'à présent, seul le connecteur $\wedge$ a été utilisé (symbole `&` dans les exemples). Il est cependant possible d'utiliser tous les connecteurs de la logique propositionnelle avec ces algèbres qualitatives. On appelle cela le formalisme PCQA. Revisor/PCQA permet d'effectuer de l'adaptation de la révision des croyances avec ce formalisme. Le principe de révision avec l'algèbre RCC8 demeure principalement le même, mais en y ajoutant tous les sens sémantiques classiques de la logique propositionnelle. Ainsi, en reprenant un fragment de l'exemple de l'exploitation agricole, la formule de PCQA suivante :

$$(lit\_majeur\{DC\}parcelle(miscanthus)) \oplus \neg(lit\_majeur\{NTTP\}parcelle(miscanthus))$$

permet d'exprimer le fait que la parcelle de miscanthus est soit totalement disjoint de la zone inondable, soit n'est pas incluse et en bordure de la zone inondable, donc est disjointe et au bord de celle-ci, mais pas les deux en même temps.

A.3 Revisor/CLC

Revisor/CLC permet de travailler dans un formalisme de conjonctions de contraintes linéaire. Son opérateur de révision utilise la technique d'optimisation linéaire du simplexe [1].

/** Dans l'exemple qui va suivre, j'aurais bien aimé avoir eu la possibilité d'utiliser Revisor/CLC afin de pouvoir construire mon propre exemple avec des pizzas, mais n'ayant pas ce moteur à disposition pour le moment, je suis contraint d'utiliser l'exemple du rapport de stage de William. **/
Voici le déroulement d'une révision dans le formalisme CLC :

1. On donne une formule source avec le nombre et/ou la quantité de chaque élément. Comme exemple, on va prendre une recette de tarte utilisant 100 grammes de pâte, 40 grammes de saccharose et 3 pommes. On a une formule de la forme suivante :

```
"(1.0*dough mass = 100.0) & (1.0*saccharose mass = 40) &
(1.0*apple nb = 3) & (1.0*pear nb = 0)"
```

2. On précise ensuite la cible à obtenir à partir de la source. Dans notre exemple, on ne veut pas de pommes dans notre recette cible :

```
"(1.0*apple nb = 0)"
```

3. Une fois cela fait, on spécifie la base de connaissances. Dans notre cas, elle est la suivante :
   - Le nombre de pommes est supérieur ou égal à 0 ;

1. *Algorithme du Simplexe* : https ://fr.wikipedia.org/wiki/Algorithme_du_simplexe

— Le nombre de poires est supérieur ou égal à 0 ;
— La quantité de pâte est supérieure ou égale à 0 ;
— La quantité de saccharose est supérieure ou égale à 0 ;
— Une pomme est équivalent à 100 grammes de pomme ;
— Une poire est équivalent à 80 grammes de poire ;
— 1 gramme de fruit est équivalent à 1 gramme de pomme + 1 gramme de poire ;
— 1 gramme de sucre est équivalent à 1 gramme de saccharose + 0.1 gramme de pomme + 0.15 gramme de poire ;

4. On termine ensuite par lancer la révision.

Dans notre exemple, cela nous donne le résultat suivant :

```
1.0*fruit mass = 320.0
1.0*pear nb = 4.0
1.0*pear mass = 320.0
1.0*apple nb = 0.0
1.0*apple mass = 0.0
1.0*sugar mass = 70.0
1.0*saccharose mass = 22.0
1.0*dough mas = 100.0
```

Ce qui s'interprète de la façon suivante : ne voulant pas de pommes dans notre recette, il a fallu les remplacer par autre chose. Ayant des poires à disposition, les pommes ont été remplacées par des poires de manière à obtenir une masse de fruit la plus proche possible de celle d'origine. Ainsi, les 3 pommes ont été remplacées par 4 poires. Par notre base de connaissance, une poire est plus sucrée qu'une pomme. De ce fait, pour compenser le sucre supplémentaire apporté par le remplacement des pommes par des poires, la masse de saccharose a été réduite à 22 grammes.
En conclusion, cette révision a permit d'adapter un recette de tarte au pommes en recette de tarte au poires.

Annexe B

Arborescence des projets *Maven*

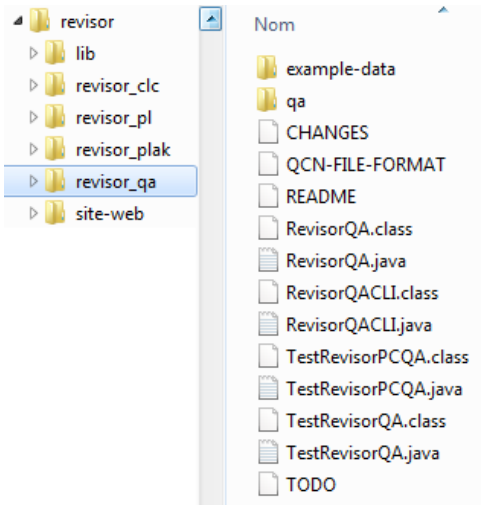


FIGURE B.1 – Ancienne structure du projet.

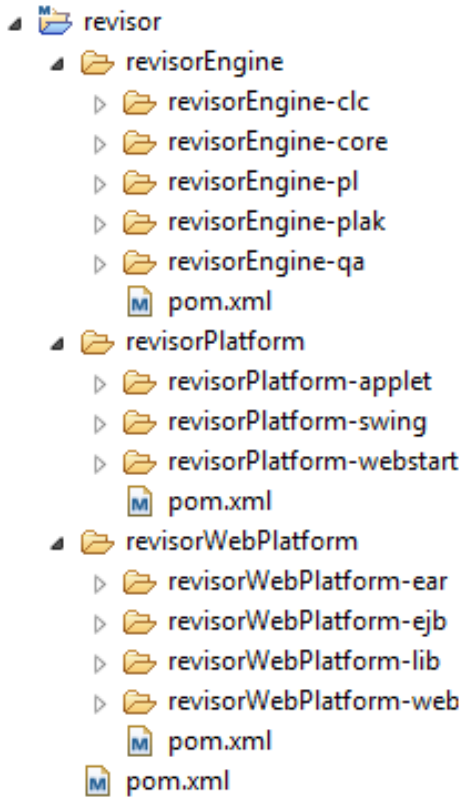


FIGURE B.2 – Nouvelle structure du projet effectuée par William.

Annexe C

Détails techniques des moteurs de révision en mode console

Dans cet annexe, afin de décrire le fonctionnement technique des différents moteurs, nous prendrons le moteur Revisor/PL comme exemple..

Dans un premier temps, nous avons deux éléments génériques permettant de fournir l'interface qui elle-même permet de *parser* les expressions saisies par l'utilisateur afin de les transformer en instructions, vérifier la validité syntaxique et sémantique de ces instructions, les exécuter et les stocker. Le premier élément s'agit de la classe abstraite `AbstractRevisorEngine` et le deuxième est la classe abstraite `AbstractRevisorConsole`. De ce fait, tous les moteurs de révision doivent hériter de ces deux classes. Dans l'exemple, nous avons la classe `RevisorEnginePL` héritant d'`AbstractRevisorEngine` et la classe `RevisorConsolePL` héritant d'`AbstractRevisorConsolePL`, elle-même héritant de la classe `AbstractRevisorConsolePL`. Dans le deuxième cas, nous avons une classe abstraite intermédiaire afin de pouvoir conserver la généricité et ce, nous permettant de greffer un autre moteur similaire sur celui-ci. Dans notre cas, PLAK est similaire à PL. Ainsi, il existe une classe `AbstractRevisorConsolePLAK` héritant de `AbstractRevisorConsolePL` et il existe une classe `RevisorConsolePLAK` héritant de `AbstractRevisorConsolePLAK`. De même, si par la suite, un formalisme proche de PLAK est implémenté, celui-ci poura hériter de `AbstractRevisorConsolePLAK`.
Ensuite, chaque instance de `RevisorConsole` contient une liste d'instances d'`Instruction`, également générique. Cette généricité s'explique de la même manière de pourquoi `RevisorConsole` et `RevisoreEngine` sont génériques : différents types d'instructions sont associés à différentes consoles et on veut pouvoir ajouter de nouveaux types d'instructions sans générer un quelconque couplage. Cette classe permet de faire la correspondance entre ce qui a été saisit par l'utilisateur et ce qui est possible d'entrer : c'est donc ici que la grammaire propre à chaque moteur est utilisé. De plus, c'est ce type générique qui réalise l'analyse sémantique et l'execution des instructions saisies par l'utilisateur. Plus précisément, l'analyse sémantique est lancée sur chacunes des instructions, et une fois cela fait, même principe pour l'exécution. L'analyse sémantique vérifie que l'utilisateur ne tente pas de créer des variables à partir d'identificateurs déjà utilisés pour créer d'autres variables.

Une instruction saisie peut bien évidemment contenir une expression. Ces instructions sont représentées par des instances de la classe `Expression` L'interface `Formule` existe également. Elle permet, à partir d'une expression qu'elle contient, de déterminer si celle-ci est unaire, ce qui sert à ajouter un parenthèsage si ce n'est pas le cas. Dans ce cas précis, les autres expressions, appelées expressions filles, sont contenues au sein

de l'expression mère (ou racine), ce qui forme au final un arbre d'expressions. Puisque ce sont les instructions qui contiennent les expressions, comme indiqué au-dessus, lorsque l'analyse sémantique est lancée sur une instruction contenant une ou plusieurs expression(s), cette analyse doit également être effectuée sur cette/ces expression(s). Ainsi, les instances de la classe `Expression` ont également à disposition des méthodes de vérification sémantique. L'instruction contenant l'expression racine appelle la méthode de vérification sémantique de cette expression, puis celle-ci appelle la même méthode mais celle de ses expressions filles à la place. Les expressions filles répètent la même opération et ainsi de suite, jusqu'à ce que tout l'arbre d'expressions ait été vérifiée et validée.

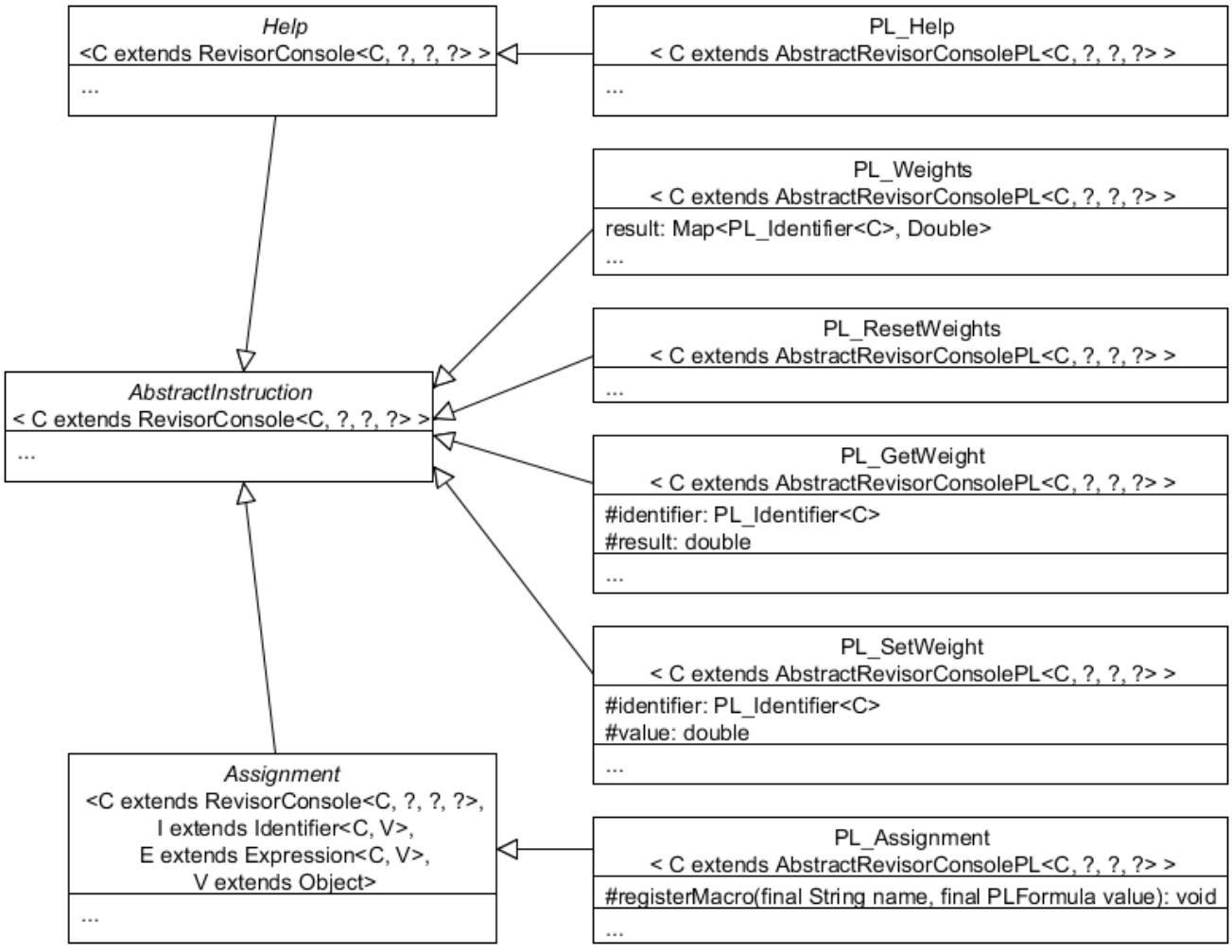
En ce qui concerne notre exemple, les diagrammes de classes des instructions et expressions peuvent être visionnés, ainsi que la grammaire correspondante par les annexes D.1, D.2 et D.3 respectivements.

Des outils de configuration, certains communs à toutes les consoles et d'autres propres, sont disponibles Les paramètres disponibles permettent de modifier le fonctionnement de la/des console(s). Ceux-ci sont stockés durablement dans un fichier.

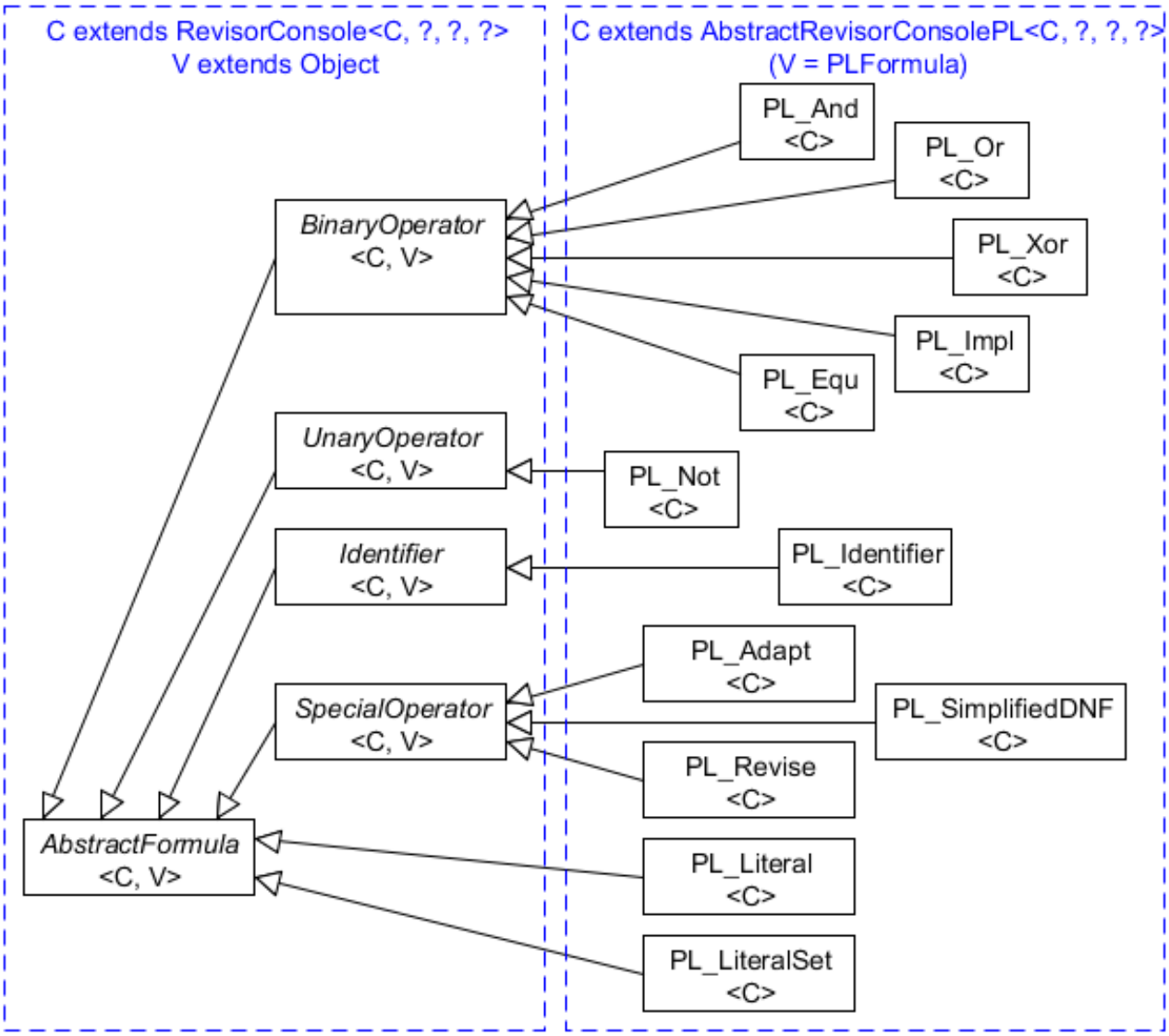
Annexe D

Instructions et expressions de Revisor/PL

D.1 Classes Instruction de Revisor/PL



D.2 Classes Expression et Formule de Revisor/PL



D.3 Grammaire de Revisor/PL

```
<Instruction>                    ::= help
                                   | load <File>
                                   | clear
                                   | reset weights
                                   | weights
                                   | <PropositionalExpression>
                                   | <Identifier> := <PropositionalExpression>
                                   | <VariableWeight>
                                   | <VariableWeight> := <PositiveReal>

<VariableWeight>                 ::= weight "(" <Identifier> ")"
                                   | <Identifier>.weight

<PropositionalExpression>        ::= <UnaryPropositionalExpression>
                                   | <NaryPropositionalExpression>

<UnaryPropositionalExpression>   ::= <Identifier>
                                   | <Boolean>
                                   | <UnaryPropositionalExpressionNotId>

<UnaryPropositionalExpressionNotId> ::= "(" <PropositionalExpression> ")"
                                   | "!" <UnaryPropositionalExpression>
                                   | dnf "(" <PropositionalExpression> ")"
                                   | adapt "(" <PropositionalExpression:dk>
                                       "," <PropositionalExpression:source>
                                       "," <PropositionalExpression:target> ")"
                                   | revise "(" <PropositionalExpression:psi>
                                       "," <PropositionalExpression:mu> ")"

<BinaryPropositionalExpression>  ::= <PropositionalExpression>
                                       "&" <PropositionalExpression>
                                   | <PropositionalExpression>
                                       "|" <PropositionalExpression>
                                   | <PropositionalExpression>
                                       "=>" <PropositionalExpression>
                                   | <PropositionalExpression>
                                       "<=>" <PropositionalExpression>
                                   | <PropositionalExpression>
                                       "^" <PropositionalExpression>

<Boolean>                        ::= true
                                   | false

<Digit>                          ::= [0-9]

<PositiveInteger>                ::= 0
```

```
                                  | [1-9] <Digit>*

<PositiveReal>                    ::= <PositiveInteger> "."?
                                  | <PositiveInteger>? "." <Digit>+

<IdentifierStartChar>             ::= [a-zA-Z_]

<IdentifierChar>                  ::= <IdentifierStartChar>
                                  | <Digit>
                                  | "_"

<Identifier>                      ::= <IdentifierStartChar> <IdentifierChar>*

<File>                            ::= .*
```

Annexe E

Fonctionnalités supplémentaires de RevisorPlatform

Des outils de *logging* sont à disposition. Ceux-ci permettent de suivre et d'enregistrer les exécutions des différentes commandes soit dans la console, soit dans un fichier, ce qui nous autorise à facilement lire et comprendre les erreurs produites.

En parlant de l'affichage, Revisor est aussi capable de produire du code proche de celui de LaTeX. Il suffit pour cela d'implémenter l'interface `LatexFormatable` afin de rendre possible la génération de code, en appelant ensuite la méthode `toLatex()`. Au niveau de l'interface graphique, l'affichage des formules est effectuée à l'aide de la bibliothèque *JLaTeXMath*[2].

Enfin, une gestion en arrière-plan de la durée d'exécution est présent, afin d'interrompre l'exécution de manière propre lorsque sa durée devient trop longue.

2. *JLaTeXMath* : https ://wiki.scilab.org/JLaTeXMath

Annexe F

Exemple de mise sous DNF d'une formule de PCLC

Soit $\varphi = \neg(3x_1+5x_2 < 0) \wedge \neg((x_1+2x_2+x_3 \geq 3) \vee (6x_2+2x_3 \neq 10) \vee \neg(4x_3 > 6)) \Rightarrow (3x_1+x_2+4x_3 \geq 0)$.
Soit ψ la formule φ mise sous DNF tel que $\varphi \equiv \psi$.
Pour se faire, on commence par calculer φ', la formule φ mise sous NNF.
On commence tout d'abord par supprimer le connecteur $\Rightarrow$ afin de faire apparaitre le connecteur $\neg$ en utilisant l'équivalence $\alpha \Rightarrow \beta \equiv \neg\alpha \vee \beta$.
On obtient alors $\varphi \equiv \neg(\neg(3x_1 + 5x_2 < 0) \wedge \neg((x_1 + 2x_2 + x_3 \geq 3) \vee (6x_2 + 2x_3 \neq 10) \vee \neg(4x_3 > 6))) \vee (3x_1 + x_2 + 4x_3 \geq 0)$.
On utilise ensuite les *lois de De Morgan* afin d'obtenir l'équivalence $\neg(\alpha \wedge \beta) \equiv \neg\alpha \vee \neg\beta$.
On obtient dans ce cas $\varphi \equiv \neg(\neg(3x_1 + 5x_2 < 0)) \vee \neg(\neg((x_1 + 2x_2 + x_3 \geq 3) \vee (6x_2 + 2x_3 \neq 10) \vee \neg(4x_3 > 6))) \vee (3x_1 + x_2 + 4x_3 \geq 0)$.
On applique après cela l'équivalence $\neg(\neg(\alpha)) \equiv \alpha$ à deux reprises dans la formule.
Ce qui donne $\varphi \equiv (3x_1+5x_2 < 0) \vee ((x_1+2x_2+x_3 \geq 3) \vee (6x_2+2x_3 \neq 10) \vee \neg(4x_3 > 6)) \vee (3x_1+x_2+4x_3 \geq 0)$.
Dans cette formule, le connecteur $\neg$ n'apparaît que devant des littéraux. Donc $\varphi \equiv (3x_1 + 5x_2 < 0) \vee ((x_1 + 2x_2 + x_3 \geq 3) \vee (6x_2 + 2x_3 \neq 10) \vee \neg(4x_3 > 6)) \vee (3x_1 + x_2 + 4x_3 \geq 0) \equiv \varphi'$.
Nous continuons ensuite par chercher ψ en commençant par utiliser l'associativité $\alpha \vee (\beta \vee \gamma) \equiv \alpha \vee \beta \vee \gamma$.
On obtient ainsi $\varphi' \equiv (3x_1+5x_2 < 0) \vee (x_1+2x_2+x_3 \geq 3) \vee (6x_2+2x_3 \neq 10) \vee \neg(4x_3 > 6) \vee (3x_1+x_2+4x_3 \geq 0)$.
On remarque que cette formule est une disjonction de littéraux. Or une disjonction peut être vue comme étant une disjonction de conjonctions de littéraux en utilisant l'équivalence $\varphi_1 \vee \varphi_2 \vee \ldots \vee \varphi_n \equiv (\varphi_1 \wedge \top) \vee (\varphi_2 \wedge \top) \vee \ldots \vee (\varphi_n \wedge \top)$ avec $\top$ une formule telle que $\forall \mathcal{I} \in \Omega, \mathcal{I} \models \top$, ce qui s'interprète par «$\top$ est une formule tout le temps satisfaite peu importe l'interprétation $\mathcal{I}$». Par conséquent, la formule $(3x_1 + 5x_2 < 0) \vee (x_1+2x_2+x_3 \geq 3) \vee (6x_2+2x_3 \neq 10) \vee \neg(4x_3 > 6) \vee (3x_1+x_2+4x_3 \geq 0)$ est une formule sous DNF. En conclusion, $\varphi \equiv (3x_1+5x_2 < 0) \vee (x_1+2x_2+x_3 \geq 3) \vee (6x_2+2x_3 \neq 10) \vee \neg(4x_3 > 6) \vee (3x_1+x_2+4x_3 \geq 0) = \psi$

Annexe G

Grammaire complète de Revisor/PCSFC

```
starts with INSTRUCTION

INSTRUCTION ::= RAW_INSTRUCTION END_OF_INSTRUCTION
|
RAW_INSTRUCTION END_OF_INSTRUCTION COMMENT
|
COMMENT
;

RAW_INSTRUCTION ::= DECLARATION
|
ASSIGNMENT
;

COMMENT ::= START_OF_COMMENT .* END_OF_COMMENT
;

DECLARATION ::= DECLARE_INTEGER
|
DECLARE_REAL
|
DECLARE_FORMULA
|
DECLARE_CONSTANT
|
DECLARE_BOOLEAN
|
DECLARE_ENUM
;

DECLARE_INTEGER ::= IDENTIFIER_LIST : INTEGER_DECLARATION_KEYWORD
;
```

```
DECLARE_REAL ::= IDENTIFIER_LIST : REAL_DECLARATION_KEYWORD
;

DECLARE_FORMULA ::= IDENTIFIER_LIST : FORMULA_DECLARATION_KEYWORD
;

DECLARE_CONSTANT ::= CONST_DECLARATION_KEYWORD IDENTIFIER SIMPLE_EQUAL REAL
;

DECLARE_BOOLEAN ::= IDENTIFIER_LIST : BOOLEAN_DECLARATION_KEYWORD
;

DECLARE_ENUM ::= ENUM_DECLARATION_KEYWORD IDENTIFIER ( MODALITY_LIST )
;

MODALITY_LIST ::= MODALITY_LIST , MODALITY
|
MODALITY
;

ASSIGNMENT ::= IDENTIFIER ASSIGNMENT_OPERATOR FORMULA
;

FORMULA ::= REVISE_FORMULA
|
BINARY_FORMULA
;

REVISE_FORMULA ::= REVISE_KEYWORD ( IDENTIFIER , IDENTIFIER , IDENTIFIER )
|
REVISE_KEYWORD ( IDENTIFIER , IDENTIFIER , REAL )
;

BINARY_FORMULA ::= ( BINARY_FORMULA )
|
NEGATIVE_FORMULA_SYMBOL BINARY_FORMULA
|
BINARY_FORMULA BINARY_FORMULA_OPERATOR UNARY_FORMULA
|
BINARY_FORMULA BINARY_FORMULA_OPERATOR ( BINARY_FORMULA )
|
BINARY_FORMULA BINARY_FORMULA_OPERATOR NEGATIVE_FORMULA_SYMBOL BINARY_FORMULA
|
UNARY_FORMULA
;
```

```
UNARY_FORMULA ::= CONSTRAINT_FORMULA
|
IDENTIFIER
|
IDENTIFIER SIMPLE_EQUAL MODALITY
|
TAUTOLOGY_FORMULA
;

CONSTRAINT_FORMULA ::= CONSTRAINT_LEFT_MEMBER CONSTRAINT_OPERATOR CONSTRAINT_RIGHT_MEMBER
;

CONSTRAINT_LEFT_MEMBER ::= CONSTRAINT_TERM_LIST
;

CONSTRAINT_TERM_LIST ::= CONSTRAINT_TERM_LIST CONSTRAINT_TERM_OPERATOR CONSTRAINT_TERM
|
CONSTRAINT_TERM
;

CONSTRAINT_TERM ::= REAL IDENTIFIER
|
IDENTIFIER
;

CONSTRAINT_RIGHT_MEMBER ::= REAL
;

IDENTIFIER ::= [a-zA-Z_][a-zA-Z0-9_]*
;

INTEGER ::= \-?[0-9]+
;

REAL ::= INTEGER
|
\-?[0-9]+\.[0-9]*
|
\-?[0-9]*\.[0-9]+
;

INTEGER_DECLARATION_KEYWORD ::= integer
;

REAL_DECLARATION_KEYWORD ::= real
;
```

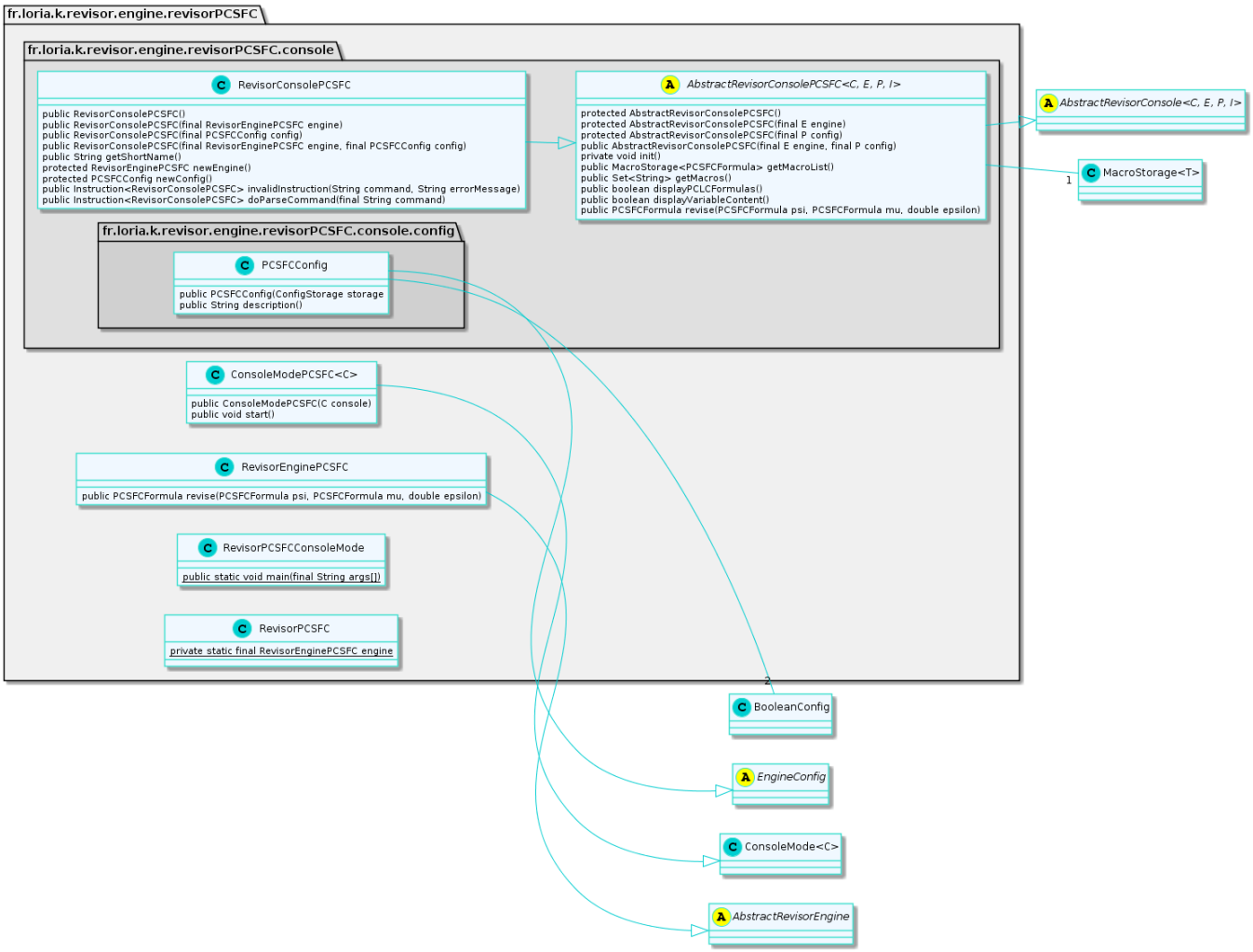
```
FORMULA_DECLARATION_KEYWORD ::= formula
;

CONST_DECLARATION_KEYWORD ::= const
;

BOOLEAN_DECLARATION_KEYWORD ::= bool
;

ENUM_DECLARATION_KEYWORD ::= enum
;

SIMPLE_EQUAL  ::= =
;

ASSIGNMENT_OPERATOR ::= :=
;

REVISE_KEYWORD ::= revise
;

END_OF_INSTRUCTION ::= ‘;‘
;

START_OF_COMMENT ::= #
;

END_OF_COMMENT ::= \n
;

NEGATIVE_FORMULA_SYMBOL ::= !
;

TAUTOLOGY_FORMULA ::= [tT]{1}[rR]{1}[uU]{1}[eE]{1}
;

BINARY_FORMULA_OPERATOR ::= &
|
‘|‘
|
=>
|
<=>
|
^
```

```
;

CONSTRAINT_TERM_OPERATOR ::= +
|
-
;

CONSTRAINT_OPERATOR ::= <
|
<=
|
=
|
!=
|
>=
|
>
;

MODALITY ::= \"[a-zA-Z0-9_]+\"
;
```

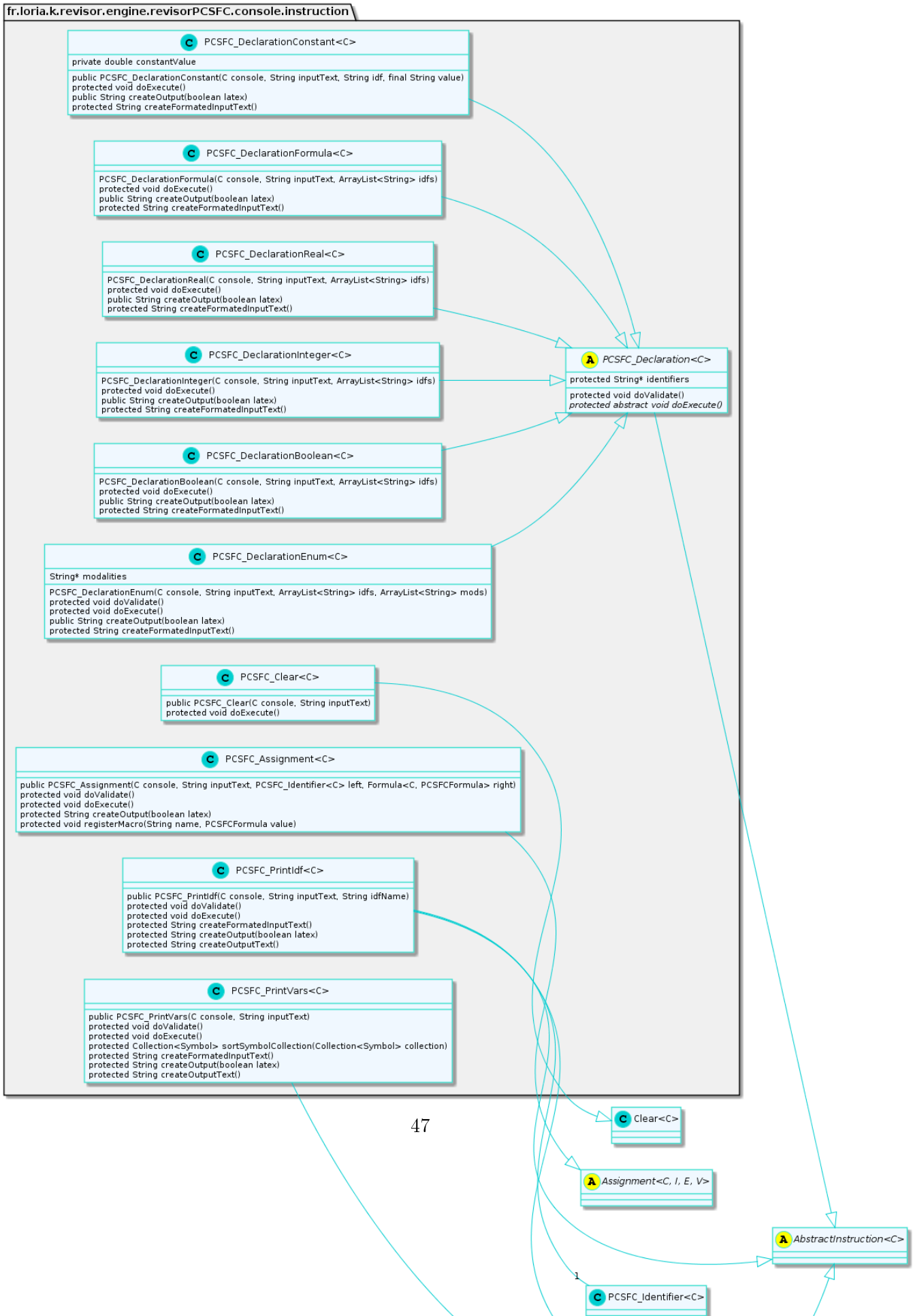
Annexe H

Diagrammes de classes utiles de Revisor/PCSFC

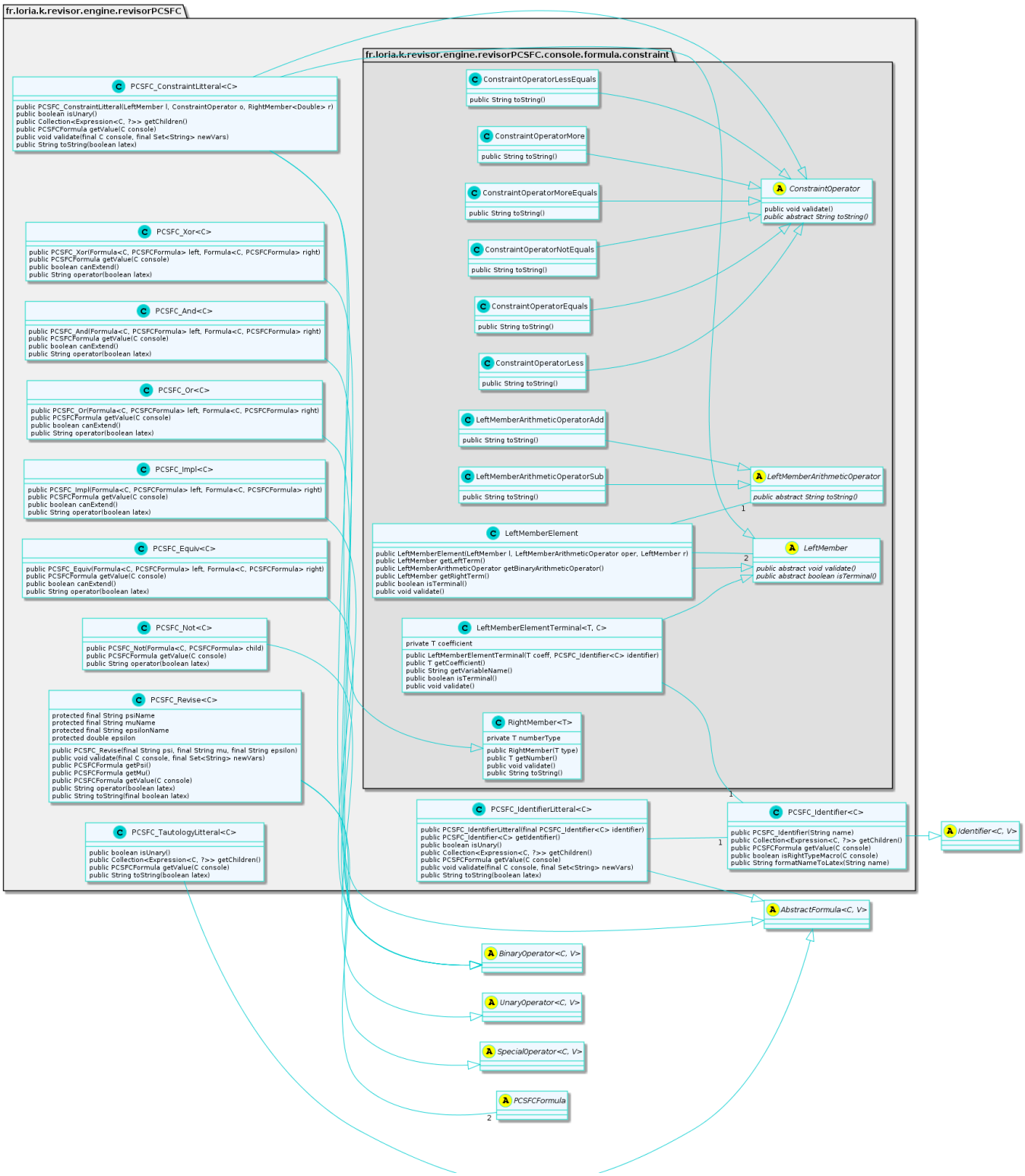
H.1 Classes console



H.2 Classes instruction



H.3 Classes formule



Bibliographie

[1] William Philbert : *Mise en place d'une plate-forme pour l'utilisation de la bibliothèque Revisor* Septembre 2014

[2] Jean Lieber : *Révision des croyances dans une clôture propositionnelle de contraintes linéaires* Nicolas Maudet ; Bruno Zanuttini. *Journée d'intelligence artificielle fondamentale, plate-forme intelligence artificielle*, Juin 2015, Rennes, France. pp.10.