Contributions à l'apprentissage automatique de réseau de contraintes et à la constitution automatique de comportements sensorimoteurs en robotique

Mathias Paulin

▶ **To cite this version:**

Mathias Paulin. Contributions à l'apprentissage automatique de réseau de contraintes et à la constitution automatique de comportements sensorimoteurs en robotique. Informatique [cs]. Université Montpellier II - Sciences et Techniques du Languedoc, 2008. Français. NNT : . tel-00340438

HAL Id: tel-00340438

https://theses.hal.science/tel-00340438

Submitted on 20 Nov 2008

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.

Académie de Montpellier
Université Montpellier II
- Sciences et Techniques du Languedoc -

Thèse

présentée au Laboratoire d'Informatique, de Robotique
et de Microélectronique de Montpellier

Spécialité	:	**Informatique**
Formation Doctorale	:	**Informatique**
École Doctorale	:	**Information, Structures, Systèmes**

Contributions à l'apprentissage automatique de réseau de contraintes et à la constitution automatique de comportements sensorimoteurs en robotique.

par

Mathias PAULIN

Soutenue le 3 juillet 2008, devant le jury composé de :

M. Jean-Pierre Merlet, Directeur de recherche INRIA, Sophia Antipolis ..... Rapporteur
M. Gérard Verfaillie, Maître de recherches ONERA, Toulouse ..... Rapporteur
M. Christian Bessiere, Directeur de recherche CNRS, LIRMM ..... Examinateur
M. Jacques Blanc-Talon, Responsable de Domaine Scientifique DGA, Bagneux ..... Examinateur
M. Jean Sallantin, Directeur de recherche CNRS, LIRMM ..... Directeur de thèse
M. Dominique Luzeaux, Directeur du CTSI DGA, Bagneux ..... Co-directeur de thèse
M. Éric Bourreau, Maître de conférences, LIRMM, Université Montpellier II ..... Invité
M. Sébastien Krut, Chargé de recherche CNRS, LIRMM ..... Invité

Remerciements

Comme le veut la tradition, je remercie mes directeurs de thèse Jean Sallantin et Dominique Luzeaux. J'adresse ensuite mes plus sincères remerciements à Jean-Pierre Merlet et Gérard Verfaillie, rapporteurs, qui ont accepté de donner de leur temps et de leur sueur pour évaluer mes travaux, et dont les nombreux commentaires et conseils ont indéniablement permis d'améliorer ce manuscrit. Je remercie aussi Christian Bessière, Jacques Blanc-Talon, Eric Bourreau et Sébastien Krut qui ont accepté de participer à mon jury de thèse.

Je tiens ensuite à adresser mes plus vifs remerciements à Philippe Andary pour m'avoir fait découvrir et apprécier l'Informatique alors que j'étais sur les bancs de l'Université de Rouen. Je garde un souvenir intact de ses extraordinaires qualités de pédagogue et de son souci permanent de placer l'étudiant au cœur de son enseignement. De la même manière, je tiens à remercier Rémi Coletta pour m'avoir tout d'abord guidé dans mes premiers pas d'apprenti chercheur durant mon stage de DEA, pour avoir ensuite continué à me faire profiter de ses travaux et de ses excellents conseils durant mes quatre années de doctorat, et je souhaite enfin le remercier pour son amitié.

Au moment de conclure cette thèse, je souhaite par ailleurs remercier très chaleureusement tous les collègues & amis qui m'ont soutenu durant cette thèse. Merci à Guillaume, indéniablement l'un des meilleurs co-bureau qui soit (Les crocros Haribo ont perdu de leur saveur depuis que nous ne les partageons plus ;o). Pour nos moments de détente, autour d'une tasse de café ou de thé, merci à Laurent, Arnaud, Jean-Luc, Nicolas, Martine & Pascal. Pour nos sorties grimpes et leur amitié, merci à mes compères Clément et Christophe : Revenez vite, nos journées à tâter du caillou tout en refaisant le monde me manquent terriblement ! Pour nos nombreuses conversations et échanges scientifiques, en plus de Rémi, Eric & Guillaume précédemment cités, je souhaite remercier Sébastien et Robin. Enfin, merci beaucoup à l'ensemble des copains du groupe Bufalo pour leur soutien et leur amitié : Carol, Géraldine, Hélène, Johann, Julie, Rémi, Steve, Thomas et toute la petite marmaille née en cours de route.

Ces remerciements seraient incomplets sans une pensée pour ma famille : Michel & Martine,

mes parents, mon frère Mikaël, mon "grand" petit cousin Laurent, mon oncle Denis et ma grand-mère Fernande. Merci pour votre soutien.

Enfin, merci Marianne de m'avoir supporté (dans tous les sens du terme) durant cette thèse. Tu as su me soutenir, m'encourager, me changer les idées... Ta présence et ton soutien sans faille ont indéniablement contribué à l'aboutissement de ce long travail.

Table des matières

Introduction **9**

1 Programmation par contraintes : Présentation générale **13**

1.1 Définitions et concepts fondamentaux 14

1.1.1 Définitions de base 14

1.1.2 Recherche, filtrage et propagation 15

1.2 L'importance de la modélisation 18

1.3 Programmation par contraintes : extensions 20

1.3.1 Extensions pour les réseaux de contraintes discrets 20

1.3.2 Les CSP continus 21

1.4 Conclusion 22

I Apprentissage supervisé d'un réseau de contraintes **25**

2 Apprentissage automatique de réseau de contraintes : État de l'art **27**

2.1 Apprentissage automatique 28

2.1.1 Présentation générale 28

2.1.2 Apprentissage numérique et apprentissage symbolique 30

2.2 Le problème d'acquisition de contraintes 31

2.2.1 Formalisation du problème d'acquisition de contraintes 31

2.2.2 Approches existantes 33

2.3 La plate-forme d'apprentissage Conacq 33

2.3.1 Processus d'apprentissage de la plate-forme Conacq 34

2.3.2 Complexité des opérations courantes 37

2.3.3 Contraintes redondantes et détection du Backbone 37

2.4 Conclusion 39

3 Apprentissage interactif de réseaux de contraintes **41**

3.1 Le problème d'acquisition interactive de contraintes 41

3.2 Une première approche de questionnement 43

3.2.1 Principe général 43

3.2.2 Formalisation 44

3.3 Vers une génération de questions optimales 44

3.3.1 Stratégies proposées 45

3.3.2 Implémentation 47

3.4 Expérimentations 51

3.5 Conclusion 53

4 Conclusion **55**

II Une approche programmation par contraintes pour la constitution automatique de comportements sensorimoteurs en robotique **57**

5 Introduction **59**

6 Architectures logicielles de contrôle : État de l'art **63**

6.1 Les architectures délibératives 63

6.2 Les architectures comportementales 65

6.3 Les architectures hybrides 67

6.4 Problématique de travail 68

6.5 Conclusion 69

7 Architecture proposée **71**

7.1 Architecture générale du système de supervision 71

7.2 Pourquoi une approche programmation par contraintes ? 73

7.3 Formalisation du processus de supervision 74

7.3.1 Modélisation CSP des actions élémentaires 74

7.3.2 Acquisition automatique des actions élémentaires 76

7.3.3 Planification d'un comportement sensorimoteur 78

7.3.4 Instanciation d'un comportement sensorimoteur 80

7.3.5 Supervision d'un comportement sensorimoteur 81

7.4 Conclusion 82

8 Expérimentations **85**

8.1 Expérimentations préliminaires . . . 85

8.1.1 Validation de la plate-forme CONACQ pour l'acquisition automatique d'actions élémentaires . . . 86

8.1.2 Validation du résolveur de contraintes CHOCO pour une utilisation en contexte réel . . . 88

8.2 Le robot TRIBOT . . . 90

8.2.1 Dispositif expérimental . . . 90

8.2.2 Programmation du TRIBOT via le logiciel LEGO MINDSTORMS NXT . . 91

8.2.3 Programmation du TRIBOT via le langage URBI . . . 92

8.2.4 Objectif de l'expérimentation TRIBOT . . . 94

8.3 Actions élémentaires étudiées dans le cadre de l'expérimentation TRIBOT . . . 94

8.4 Acquisition automatique des actions élémentaires du robot TRIBOT sous la forme de réseaux de contraintes . . . 97

8.4.1 Objectif de l'acquisition automatique . . . 97

8.4.2 Format des données d'entraînement fournies à CONACQ . . . 98

8.4.3 Acquisition automatique de l'action élémentaire *Careful move* . . . 98

8.4.4 Acquisition automatique des autres actions élémentaires de TRIBOT . . 100

8.5 Planification et supervision d'un comportement sensorimoteur : saisie d'un objet par TRIBOT . . . 100

8.5.1 Volet expérimental 1 : Saisie d'un objet *sans* perturbation extérieure volontaire . . . 101

8.5.2 Volet expérimental 2 : Saisie d'un objet *avec* perturbations extérieures volontaires . . . 103

8.6 Synthèse . . . 105

9 Conclusion et perspectives **107**

10 Conclusion générale **111**

Introduction

Depuis sa formalisation par Montanari en 1974 [Mon74] puis Mackworth en 1977 [Mac77], la programmation par contraintes est utilisée pour résoudre des problèmes d'affectation de valeurs sur des domaines où les variables sont liées entre elles par des contraintes. Forte de trente années de travaux académiques et industriels, la programmation par contraintes est désormais un paradigme efficace pour résoudre des problèmes combinatoires d'envergure. Son usage s'est ainsi répandu dans de nombreux domaines d'application, tels que la configuration de produits [AFM02], l'allocation de ressources ou l'ordonnancement [Wal96].

La programmation par contraintes se réclame d'être un paradigme déclaratif. Lorsque l'on souhaite résoudre un problème à l'aide de la programmation par contraintes, il faut dans un premier temps *modéliser* son problème, c'est-à-dire identifier les variables, les valeurs qu'elles peuvent prendre, ainsi que l'ensemble des contraintes. Dans un second temps, on fournit le résultat de cette modélisation (*i.e.* un réseau de contraintes) à un résolveur de contraintes qui détermine si le problème est soluble et, le cas échéant, calcule une solution.

Si la programmation par contraintes est un paradigme de choix pour résoudre des problèmes fortement combinatoires, il convient de noter que des problèmes réels d'envergure ne peuvent être abordés, en général, sans une très bonne connaissance de la programmation par contraintes. Une des difficultés inhérentes à la programmation par contraintes consiste en effet à savoir modéliser efficacement son problème et à savoir choisir la méthode de résolution la plus appropriée parmi toutes celles proposées par les résolveurs de contraintes actuels. Ainsi, malgré le succès qu'elle connaît, la programmation par contraintes est difficile d'accès à des non-spécialistes et sa diffusion n'est pas aussi soutenue qu'escomptée.

Dans le cadre de cette thèse, nous nous intéressons à l'acquisition automatique de réseau de contraintes, aussi appellée *apprentissage* automatique de réseau de contraintes, qui consiste à développer des solutions capables d'aider un utilisateur à modéliser son problème sous la forme d'un réseau de contraintes.

Dans la première partie de cette thèse, nous nous intéressons plus particulièrement à la

plate-forme d'acquisition automatique de réseau de contraintes CONACQ [BCO+03], développée et maintenue au LIRMM (Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier). La plate-forme CONACQ s'adresse à un expert "métier" souhaitant résoudre un problème combinatoire à l'aide de la programmation par contraintes, mais se trouvant incapable de modéliser ce problème en contraintes. Cet expert "métier" possède une bonne connaissance de son problème. Aussi, bien qu'il ne sache pas le modéliser en programmation par contraintes, il est cependant capable de se prononcer sur la validité d'une instance. En d'autres termes, il sait reconnaître les exemples dans lesquels les contraintes de son problème sont satisfaites ou violées. Il fournit à ce titre un ensemble de solutions et de non-solutions à son problème. L'objectif de la plate-forme CONACQ consiste alors à induire un réseau de contraintes qui soit *consistant* avec les solutions et les non-solutions fournies par l'utilisateur.

Dans son implémentation standard, la plate-forme CONACQ est passive vis-à-vis de l'utilisateur, c'est-à-dire basée sur la capacité de ce dernier à fournir des instances significatives de son problème. Dans la première partie de cette thèse, nous proposons une version interactive de CONACQ, capable de poser à l'utilisateur des questions dont le but est d'augmenter plus rapidement et de manière conséquente la connaissance acquise par la plate-forme. Afin de limiter le nombre d'interactions, nous proposons différentes stratégies de questionnement que nous validons ensuite empiriquement.

La première partie de cette thèse s'organise comme suit : Le chapitre 1 présente les principes fondamentaux de la programmation par contraintes, paradigme que nous manipulerons tout au long de cette thèse. Nous présentons ensuite, dans le chapitre 2, un état de l'art en acquisition automatique de réseau de contraintes ainsi que la plate-forme CONACQ. Le chapitre 3 présente les différentes stratégies de questionnement que nous proposons afin de rendre la plate-forme CONACQ interactive. Enfin, nous concluons cette première partie au travers du chapitre 4.

Dans la deuxième partie, nous nous intéressons à une utilisation pratique de l'acquisition automatique de réseau de contraintes dans le domaine de la robotique.

Depuis quelques décennies, une partie des recherches en robotique s'intéresse à la conception de robots *autonomes*, c'est-à-dire des machines polyvalentes capables de réaliser des tâches complexes dans des environnements très variés. Différentes architectures logicielles de contrôle, telles que l'architecture LAAS [ACF+98], CLARATY [VNE+01] et HARPIC [LD03] ont ainsi été proposées pour conférer de l'autonomie à des robots. Pour permettre à un robot d'accomplir différentes tâches, ces architectures de contrôle planifient des séquences de *comportements sensorimoteurs*. Bien qu'ils soient considérés par les architectures de contrôle comme les capacités de *base* du robot, les comportements sensorimoteurs ne sont cependant pas basiques, mais cor-

respondent à la combinaison d'actions élémentaires câblées exécutables par le robot. À l'heure actuelle, la définition des comportements sensorimoteurs est réalisée de manière *ad-hoc* par les automaticiens. Chaque action élémentaire exécutable par le robot est tout d'abord modélisée à l'aide de systèmes d'équations mathématiques mettant en jeu les lois physiques du monde (loi de conservation des énergies, moments cinétiques, *etc.*). Ensuite, il convient de combiner des actions élémentaires, et par conséquent les systèmes d'équations qui les modélisent, dans l'optique de définir un comportement sensorimoteur. Ainsi, la définition des comportements sensorimoteurs d'un robot constitue une difficulté de taille, qui nécessite un travail de modélisation ardu (parfois fastidieux), ainsi que de nombreux calculs, pouvant aller jusqu'à plusieurs heures.

Dans la deuxième partie de cette thèse, nous nous intéressons à une utilisation pratique de l'apprentissage automatique de réseau de contraintes, qui vise à automatiser le processus de définition de comportements sensorimoteurs. Nous proposons pour cela une architecture logicielle, complémentaire aux architectures de contrôle précédemment citées, qui utilise le paradigme de la programmation par contraintes pour modéliser, planifier et superviser l'exécution de comportements sensorimoteurs. Elle utilise la plate-forme d'acquisition automatique CONACQ étudiée dans la première partie de cette thèse pour modéliser *automatiquement* les actions élémentaires d'un robot sous la forme de réseaux de contraintes. Notre architecture utilise par ailleurs un planificateur de tâches inspiré de CSP-PLAN [LB03] pour combiner les réseaux de contraintes acquis et ainsi définir *automatiquement* des comportements sensorimoteurs. Différents résultats expérimentaux sont par ailleurs présentés afin de valider notre approche.

La deuxième partie de cette thèse s'organise de la manière suivante. Le chapitre 5 permet tout d'abord d'introduire nos travaux. Nous présentons ensuite, dans le chapitre 6, un rapide survol des architectures logicielles de contrôle proposées en robotique autonomes. Dans le chapitre 7, nous présentons formellement l'architecture que nous proposons. Nous validons ensuite, au travers d'une série d'expérimentations (chapitre 8), la capacité de notre architecture à planifier et à superviser automatiquement l'exécution de comportements sensorimoteurs. Le chapitre 9 conclue la deuxième partie de cette thèse et présente les limites actuelles ainsi que les perspectives de ce travail.

Enfin, le chapitre 10 récapitule les travaux présentés dans cette thèse avant de conclure.

Chapitre 1

Programmation par contraintes : Présentation générale

Forte de trente années de recherche et de progrès constants, la programmation par contraintes se revèle être un paradigme de choix, tant par son expressivité que par son efficacité à résoudre des problèmes combinatoires d'envergure issus de domaines variés, tels que la configuration de produits [AFM02], l'allocation de ressources ou l'ordonnancement [Wal96].

De manière informelle, l'idée principale sous-jacente à la programmation par contraintes consiste à modéliser un problème sous la forme d'un réseau de contraintes. Un réseau de contraintes est défini par un ensemble de variables, un ensemble de domaines et un ensemble de contraintes. On associe à chaque variable un domaine distinct. Ce domaine est constitué de l'ensemble des valeurs qui peuvent être affectées à cette variable. Lorsque le domaine d'une variable est réduit à une valeur, on dit que la variable est instanciée. Une contrainte est définie sur un sous-ensemble de variables et exprime une propriété que ces variables doivent satisfaire. Une solution d'un réseau de contraintes est une instanciation de variables qui satisfait la conjonction de toutes les contraintes. L'existence d'une solution à un réseau de contraintes est un problème NP-Complet. Le problème de satisfaction de contraintes (CSP : Constraint Satisfaction Problem), qui consiste à rechercher une telle solution, est un problème NP-Difficile.

Dans cette thèse, nous manipulons le paradigme de la programmation par contraintes comme un outil de résolution et utilisons les techniques et algorithmes de résolution en « boîtes noires ». Ce chapitre présente les définitions de base du problème de satisfaction de contraintes ainsi qu'un rapide survol des principes fondamentaux utilisés en programmation par contraintes. Pour plus de précisions sur les techniques employées en programmation par contraintes, le lecteur se reportera aux ouvrages [Dec03] et [RvBW06].

1.1 Définitions et concepts fondamentaux

1.1.1 Définitions de base

Depuis sa formalisation par Montanari en 1974 [Mon74] puis Mackworth en 1977 [Mac77], la programmation par contraintes est utilisée pour résoudre des problèmes d'affectation de valeurs sur des domaines où les variables sont liées entre elles par des contraintes.

Définition 1 (Réseau de contraintes) *Un réseau de contraintes $\mathcal{P}$ pour un CSP est un triplet $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ où :*

- *$\mathcal{X} = \{x_1, \ldots, x_n\}$ est l'ensemble fini des **variables** mises en jeu dans le problème,*
- *$\mathcal{D} = \{D(x_1), \ldots, D(x_n)\}$ est l'ensemble de leurs **domaines finis** respectifs, chaque variable x_i prenant ses valeurs dans $D(x_i)$,*
- *$\mathcal{C} = \{c_1, \ldots, c_m\}$ est une séquence de **contraintes** sur $\mathcal{X}$. Une contrainte c_i est définie par la séquence $var(c_i)$ des variables sur lesquelles elle porte et par la relation $rel(c_i)$ qui spécifie les m-uplets autorisés sur $var(c_i)$. L'affectation de valeurs aux variables de $var(c_i)$ satisfait c_i si et seulement si elle appartient à $rel(c_i)$.*

Une contrainte est dite *binaire* lorsqu'elle met en jeu exactement deux variables et un réseau de contraintes $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ est dit *binaire* lorsque $\mathcal{C}$ est un ensemble de contraintes binaires. Par un léger abus de notation, nous noterons c_{ij} la contrainte binaire placée entre les variables x_i et x_j. À titre d'exemple, $\leq_{12}$ signifie que la contrainte "inférieur ou égal à" est placée entre les variables x_1 et x_2 du problème.

Définition 2 (Instanciation) *Soit $\mathcal{Y} = \{y_1, \ldots, y_k\}$ un sous-ensemble de $\mathcal{X}$. Une* instanciation *$e_\mathcal{Y}$ sur $\mathcal{Y}$ est un k-uplet $(v_1, \ldots, v_k) \in D(y_1) \times \ldots \times D(y_k)$. Cette instanciation est partielle si $\mathcal{Y} \neq \mathcal{X}$, complète sinon (notée e). Une instanciation $e_\mathcal{Y}$ sur $\mathcal{Y}$* viole *la contrainte c_i si et seulement si $var(c_i) \subseteq \mathcal{Y}$ et $e_\mathcal{Y}[var(c_i)] \notin rel(c_i)$. Autrement dit, une instanciation $e_\mathcal{Y}$ viole une contrainte c_i si la projection de $e_\mathcal{Y}$ sur les variables de c_i est un tuple rejeté par c_i. Sinon, on dit que $e_\mathcal{Y}$* satisfait *la contrainte c_i.*

On définit alors une solution à un réseau de contraintes $\mathcal{P}$ à l'aide de la définition 3.

Définition 3 (Solution) *Soit $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ un réseau de contraintes. Une instanciation complète e sur $\mathcal{X}$ est une* solution *du réseau $\mathcal{P}$ si et seulement si elle satisfait toutes les contraintes du réseau. Sinon, c'est une non-solution. On note $Sol(\mathcal{X}, \mathcal{D}, \mathcal{C})$ l'ensemble des solutions de $(\mathcal{X}, \mathcal{D}, \mathcal{C})$.*

L'existence d'une solution à un réseau de contraintes est un problème NP-Complet. Le problème de satisfaction de contraintes, qui consiste à rechercher une telle solution, est un problème NP-Difficile.

L'exemple 1 présente un problème, volontairement simpliste, que l'on peut résoudre à l'aide de la programmation par contraintes.

FIG. 1.1 – Un exemple d'application de la PPC : la coloration de carte.

FIG. 1.2 – Une solution au problème de coloration de carte.

Exemple 1 *Comme le suggère la figure 1.1, il s'agit de colorier les pays dessinés à l'aide des différentes couleurs autorisées de façon à ce que deux pays limitrophes soient coloriés avec des couleurs distinctes.*

Pour résoudre ce problème, il est possible de définir le réseau de contraintes suivant. Les variables du réseau correspondent aux cinq pays à colorier, soit $\mathcal{X} = \{Italie, Suisse, France, Espagne, Portugal\}$*, auxquels on attribue les domaines suivants* $D(Italie) = D(Suisse) = D(Portugal) = \{vert, rouge\}$*,* $D(France) = \{vert, bleu, rouge\}$ *et* $D(Espagne) = \{bleu, rouge\}$*. On place enfin la contrainte de différence sur tous les couples de pays limitrophes. Le problème de coloration de carte se retrouve alors modélisé sous la forme du réseau de contraintes présenté par la figure 1.3. En utilisant cette modélisation, une solution possible (non-unique) est représentée sur la figure 1.2.* □

1.1.2 Recherche, filtrage et propagation

Lorsque l'on souhaite résoudre un problème combinatoire à l'aide de la programmation par contraintes, il faut dans un premier temps *modéliser* son problème sous la forme d'un

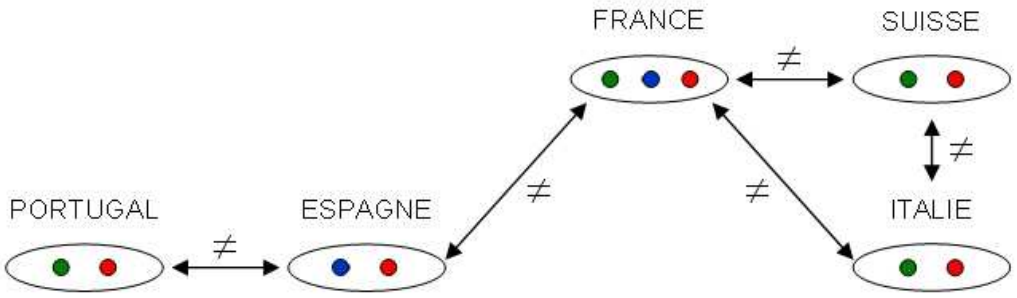

FIG. 1.3 – Le réseau de contraintes modélisant le problème de coloration de l'exemple 1.

réseau de contraintes (*c.f.* exemple 1). Dans un second temps, on fournit le triplet $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ à un résolveur de contraintes qui détermine si le problème est soluble et, le cas échéant, calcule une solution. La recherche d'une solution est alors réalisée par l'intermédiaire d'algorithmes dérivés de l'algorithme BACKTRACK [GB65].

L'algorithme BACKTRACK est une méthode de résolution complète énumérative qui effectue un parcours en profondeur de l'arbre de recherche. Dans sa version originelle, le principe consiste à instancier les variables du problème successivement selon un ordre prédéfini. À chaque fois qu'une valeur est affectée à une variable, on teste si l'instanciation partielle satisfait toutes les contraintes du réseau de contraintes. Si c'est le cas, une nouvelle variable est instanciée, et ce jusqu'à ce que toutes les variables le soient. Dans le cas contraire, on essaie une nouvelle valeur pour la variable courante. Si toutes les valeurs d'une variable sont testées sans succès, on revient à la variable précédente, afin d'essayer une nouvelle valeur (on réalise alors un backtrack). S'il n'en existe pas, on effectue à nouveau un backtrack. L'algorithme BACKTRACK s'arrête lorsqu'il a trouvé une instanciation complète sur $\mathcal{X}$ qui satisfasse toutes les contraintes. Si l'algorithme parcourt tout l'arbre de recherche sans trouver d'instanciation complète satisfaisant l'ensemble des contraintes alors le CSP étudié ne possède pas de solution. L'énumération réalisée par l'algorithme BACKTRACK est rédhibitoire dès lors que les problèmes abordés grossissent, en raison du trop grand nombre de combinaisons possibles. Afin d'élaguer l'espace de recherche, de nombreux travaux ont porté sur la conception de techniques de *filtrage* et d'algorithmes de *propagation de contraintes*.

Le *filtrage* vise à supprimer les valeurs de variables qui, compte tenu des autres domaines du problème, ne peuvent appartenir à une solution. On parle alors de valeurs *inconsistantes*. À titre d'exemple, si x_1 et x_2 sont deux variables de domaines $D(x_1) = D(x_2) = \{1, 2, 3\}$ et si l'on considère la contrainte $<_{12}$ alors la valeur 3 peut être supprimée de $D(x_1)$ car il n'existe

aucune valeur pour x_2 telle que la contrainte $(x_1 < x_2)$ soit satisfaite si x_1 est instanciée avec 3. De manière analogue, la valeur 1 sera filtrée pour la variable x_2. Il convient de noter que la suppression des valeurs inconsistantes d'un CSP évite de parcourir des branches de l'arbre de recherche qui ne peuvent aboutir à une solution.

Après chaque modification du domaine d'une variable, on étudie l'ensemble des contraintes impliquant cette variable afin d'identifier si cette modification peut conduire à de nouvelles déductions. On appelle cette méthode la *propagation* de contraintes. Lorsque le domaine d'une variable est réduit (les valeurs inconsistantes ayant été supprimées par filtrage), on « réveille » les contraintes qui impliquent cette variable. Pour chaque contrainte réveillée, on supprime les valeurs inconsistantes des autres variables de la contrainte. Au cours de la recherche, la propagation de contraintes et le filtrage permettent la suppression des valeurs inconsistantes du problème et réduisent ainsi l'espace de recherche. L'exemple 2 illustre le fonctionnement de la propagation de contraintes.

Exemple 2 *Dans cet exemple, nous cherchons à illustrer l'impact de la propagation de contraintes durant la résolution du problème de coloration de carte introduit dans l'exemple 1. Nous supposons pour cela qu'à la première étape de résolution du problème, la variable Italie est instanciée à la valeur* $Vert$.

Le domaine de la variable Italie ayant été modifié, les contraintes $\neq_{Italie,France}$ *et* $\neq_{Italie,Suisse}$ *sont réveillées. La contrainte* $\neq_{France,Italie}$ *(*resp.$\neq_{Italie,Suisse}$*) impose que la France et l'Italie (*resp. *l'Italie et la Suisse) doivent être coloriées différemment. En conséquence, la valeur* $Vert$ *est supprimée (*i.e. *étape de filtrage) des domaines* $D(France)$ *et* $D(Suisse)$. *Le domaine* $D(France)$ *se réduit alors à* $\{Bleu, Rouge\}$ *et la variable Suisse est instanciée à la valeur Rouge. Cette dernière instanciation permet d'écarter la valeur Rouge pour la France car elle ne peut appartenir à une solution du problème de coloration de carte. La France prend ainsi la valeur Bleu et, en propageant cette information, on fixe les valeurs pour l'Espagne et le Portugal, ce qui permet d'aboutir à la solution au problème de coloration de carte.* □

Il est admis qu'utiliser des algorithmes de filtrage et de propagation efficaces est une condition nécessaire à la résolution de problème d'envergure, dans la mesure où ils permettent de réduire significativement l'espace de recherche. Cependant, ces algorithmes ne doivent pas être trop coûteux en complexité afin d'offrir un bon compromis entre le temps nécessaire au filtrage et la qualité de celui-ci.

La propagation de contraintes est un élément central du succès de la programmation par contraintes. Elle a fait l'objet de nombreuses recherches qui ont permis la définition d'algo-

rithmes efficaces [SF94] [BFR95] [BR01]. Les dernières générations de résolveurs de contraintes, tels que ILOG Solver [CP07] ou CHOCO [Cho07], mettent en œuvre ces stratégies sophistiquées pour élaguer significativement l'espace de recherche, les temps de résolution étant alors drastiquement diminués. Pour plus de précisions sur les techniques de propagation, le lecteur se reportera au chapitre 3 de l'ouvrage [RvBW06].

1.2 L'importance de la modélisation

Lorsque l'on souhaite résoudre un problème à l'aide de la programmation par contraintes, il existe en général différentes modélisations possibles (*i.e.* différents réseaux de contraintes). Cette non unicité des modèles résulte principalement du fait qu'un problème peut être abordé de différentes manières, la définition des variables, des domaines et des contraintes étant alors en général très différente d'un modèle à l'autre. Par ailleurs, une modélisation naïve se révèle souvent rédhibitoire lorsque l'on s'attaque à des problèmes réels. Enfin, il est admis que modéliser directement un problème sous la forme d'un CSP sémantiquement correct et efficace en termes de temps de résolution est une tâche particulièrement ardue.

Après avoir concentré ses travaux sur les techniques de résolution, une partie de la communauté "contraintes" a ensuite étudié l'impact de la modélisation d'un problème sur les performances de résolution des résolveurs de contraintes. Ces recherches ont ainsi permis de nouveaux outils de modélisation parmi lesquels on peut citer les contraintes *globales*, les contraintes *implicites*, les *channeling-constraints* et la détection des *symétries*. Ces différentes techniques sont généralement utilisées pour améliorer la modélisation d'un CSP sémantiquement correct mais dont le temps de résolution est prohibitif.

Une contrainte est dite *globale* lorsqu'elle implique un nombre non fixé de variables et qui, si elle est décomposable en une conjonction de contraintes sémantiquement équivalente, permet d'améliorer les performances du modèle. Ainsi, les contraintes opérationnellement globales filtrent strictement plus de valeurs que leur décomposition, tandis que les contraintes algorithmiquement globales produisent le même filtrage mais avec un coût algorithmique plus faible [BH03]. La contrainte *alldifferent*[1] est un exemple de contrainte globale. Il existe un grand nombre de contraintes globales, répertoriées dans le catalogue [BD08]. Le chapitre 6 de [RvBW06] présente précisément le concept de contrainte globale.

Une contrainte *implicite* [SSW99] n'est pas requise dans la formulation du problème mais

[1] La contrainte $alldifferent(x_1, \ldots, x_n)$ impose que les valeurs prises par les variables $x_1, \ldots, x_n$ soient deux à deux distinctes.

permet en revanche de résoudre de le résoudre plus efficacement. La notion de contrainte implicite est une notion importante pour concevoir des modèles efficaces. En effet, la présence d'une contrainte implicite stratégiquement placée dans un CSP permet d'accélérer la résolution du problème, en effectuant une réduction rapide des domaines des variables sur lesquelles elle porte. L'utilisation des contraintes implicites a par exemple montré son intérêt pour résoudre efficacement des problèmes de championnats sportifs [Ró1]. Dans cette thèse, nous distinguons les contraintes implicites des contraintes *redondantes*, dont nous reparlerons plus en détails dans les chapitres 2 et 3. Par opposition aux contraintes implicites, on parle de contrainte *redondante* quand elle n'est pas requise pour la formulation du problème et qu'elle n'améliore pas le filtrage.

Pour les problèmes de permutation, dans lesquels il y a autant de variables que de valeurs, il est généralement impossible de dégager une modélisation optimale et on utilise alors deux modèles duaux pour modéliser le problème [Gee92]. Dans ce cas, les variables du modèle dual correspondent aux valeurs du modèle primal et réciproquement. On utilise en fait ces deux modèles pour exprimer au mieux les contraintes du problème à résoudre. Certaines contraintes étant plus simples à exprimer dans le modèle primal, alors que d'autres sont mieux exprimées dans le dual. Pour garantir la cohérence entre les deux modèles lors de la recherche de solution et pour améliorer l'efficacité de la résolution, il convient de rajouter des contraintes, appelées *channeling-constraints* entre ces deux modèles [CCLW99] [Wal01].

Enfin, la présence de *symétries* dans un réseau de contraintes est parfois la cause de l'explosion de la taille de l'arbre de recherche. À titre d'exemple, si plusieurs variables d'un problème jouent des rôles symétriques, le résolveur de contraintes explore plusieurs fois les mêmes branches de l'arbre de recherche à une permutation de valeurs près, ce qui pénalise grandement le temps de résolution. L'exploitation des propriétés de symétries (*c.f.* [RvBW06], chapitre 10), additionné au filtrage et à la propagation, permet de réduire drastiquement l'arbre de recherche de certains CSP en imposant par exemple au résolveur de n'explorer qu'une seule branche d'une permutation.

À l'image des techniques de propagation de contraintes évoquées en section 1.1.2, une modélisation efficace est une condition nécessaire à la résolution de problèmes d'envergure. La reformulation de CSP, qui vise à obtenir un second modèle qui peut être résolu efficacement par un résolveur de contraintes, fait ainsi l'objet de nombreuses recherches et d'un groupe de travail annuel intitulé *Workshop on Constraint Modelling and Reformulation* [MP06]. Par ailleurs, cette section ne prétend pas être une étude exhaustive de toutes les techniques de modélisation. Une telle étude peut être trouvée dans le chapitre 11 de l'ouvrage [RvBW06].

1.3 Programmation par contraintes : extensions

Afin d'étendre le pouvoir expressif de la programmation par contraintes dans l'optique de s'attaquer à la résolution de nouveaux problèmes, différentes extensions ont été proposées pour les réseaux de contraintes sur domaines discrets (section 1.3.1). Une partie de la communauté "contraintes" s'est par ailleurs intéressée à la modélisation et à la résolution de problèmes mettant en jeu des contraintes continues (section 1.3.2).

1.3.1 Extensions pour les réseaux de contraintes discrets

Parmi les nombreuses approches proposées pour étendre les CSP définis sur des domaines discrets [Mon74], nous présentons succinctement dans cette section trois extensions majeures : les *réseaux de contraintes pondérées*, les *explications* et les *réseaux de contraintes quantifiées*.

Les réseaux de contraintes pondérées

Les *réseaux de contraintes pondérées* [Lar02] [LS06] sont une extension des réseaux de contraintes dans laquelle on associe un coût de violation à chaque contrainte. Les réseaux de contraintes pondérées sont généralement utilisés pour modéliser et résoudre des problèmes sur-contraints[2]. Dans ce cas, on cherche à trouver une instanciation complète de coût minimal, c'est-à-dire violant "le moins" les contraintes du problème. Les réseaux de contraintes pondérées ont récemment été utilisés avec succès en bio-informatique pour la localisation d'ARN non-codants [Zyt07].

Les explications

Initialement apparu dans les approches proposées dans [McA80] et [Bes91], puis développé par N. Jussien [Jus97], le concept d'explication a pour objectif de tracer et de synthétiser l'activité d'un résolveur de contraintes . Une *explication* est ainsi un ensemble de contraintes permettant de justifier une action du résolveur de contraintes telle que le filtrage d'une valeur, l'instanciation d'une variable ou l'identification d'une contradiction.

Les explications peuvent être utilisées efficacement à différentes fins [Jus03]. Elles peuvent tout d'abord être proposées comme un outil d'analyse et de débogage servant à comprendre le déroulement ou le résultat d'une résolution en répondant à des questions telles que "*Pourquoi le problème n'a pas de solution ?*", "*Pourquoi la variable x prend la valeur a ?*", *etc.*.

[2]Un problème est dit *sur-contraint* lorsqu'il n'y a pas de solution si on impose que toutes les contraintes posées soient satisfaites.

Les explications ont ensuite montré leur intérêt pour aborder des problèmes dynamiques[3] ou sur-contraints [VJ03]. Pour les problèmes dynamiques, les explications sont utilisées pour proposer des systèmes de résolution incrémentaux pour l'ajout et le retrait de contraintes. Dans le cas des problèmes sur-contraints, l'idée est d'utiliser les explications pour identifier puis retirer automatiquement la/les contrainte(s) responsable(s) des échecs. Enfin, les explications peuvent être utilisées pour améliorer les algorithmes de résolution actuels, en développant de nouveaux algorithmes [Cam06] capables de combiner les méthodes prospectives, basées sur les mécanismes de filtrage et de propagation, et les approches rétrospectives, qui analysent les échecs pour élaguer l'espace de recherche.

Il est à noter que le résolveur de contraintes Choco propose un système de résolution basé sur les explications : le système Palm [PwM07].

Les réseaux de contraintes quantifiées

Récemment introduits [BM02] [BLV06] [VB06], les *problèmes de satisfaction de contraintes quantifiées* (QCSP : Quantified CSP) sont une généralisation des problèmes de satisfaction de contraintes dans lesquels chaque variable est quantifiée soit existentiellement soit universellement. L'utilisation des quantificateurs permet une plus grande expressivité de problèmes. Les CSP quantifiés sont ainsi notamment utilisés pour appréhender des problèmes de planification en présence d'incertain ou faire du model checking.

Il convient cependant de noter que les QCSP se situent dans la classe P-SPACE-Complet, les problèmes de P-SPACE étant considérés plus difficiles que ceux de NP[4]. P-SPACE est la classe de complexité des problèmes se résolvant avec une complexité spatiale polynomiale, sans borne sur la complexité temporelle.

1.3.2 Les CSP continus

Les précédentes sections portaient sur le cadre CSP *classique*, où les variables sont définies sur des domaines finis et discrets [Mon74]. Certains problèmes réels mettent cependant en jeu des variables continues sur lesquelles portent des contraintes correspondant à des fonctions mathématiques à valeurs réelles, que le cadre CSP classique ne peut modéliser finement. Pour pallier cet écueil, les réseaux de contraintes *continus* ou *numériques* ont été développés [Dav87] [Lho93]. Les CSP continus portent sur des variables à valeurs réelles sur lesquelles portent des équations non-linéaires de la forme $f(x_1, \ldots, x_n) = 0$ où est f est une fonction réelle.

[3] On parle de problème *dynamique* lorsque l'ensemble des contraintes du problème évolue au cours du temps.
[4] En acceptant la conjecture P $\neq$ NP.

En CSP continus, l'objectif consiste à trouver une approximation fine des solutions de systèmes de contraintes non-linéaires. Pour cela, les méthodes de résolution les plus utilisées sont basées sur les techniques de *bissection*, issues de l'arithmétique des intervalles, et de filtrage par consistance locale. La bissection consiste à choisir un intervalle selon différentes stratégies de sélection. Cet intervalle est alors coupé en plusieurs parties, puis les sous-problèmes correspondants sont résolus de manière indépendante. Les algorithmes de filtrage permettent quant à eux d'identifier les *trous* dans les domaines, c'est-à-dire des intervalles sur lesquels certaines contraintes du problème ne sont pas satisfaites. Ces trous sont alors utilisés pour déterminer le "plus petit" intervalle englobant des solutions et ainsi répondre au problème posé (*i.e.* trouver une approximation fine des solutions). Le lecteur pourra se reporter au chapitre 16 de [RvBW06] pour plus de détails.

Les réseaux de contraintes continus ont récemment été utilisés en robotique [Mer01] [Jau06] et en infographie [BGLC04].

1.4 Conclusion

Une des caractéristiques principales de la programmation par contraintes réside dans son aspect déclaratif et dans la dissociation inhérente entre un modèle décrivant le problème (le "Quoi") et les techniques de résolution utilisées pour la recherche de solutions valides (le "Comment"). La programmation par contraintes fournit ainsi un cadre unique (les réseaux de contraintes) pour modéliser des problèmes combinatoires, ainsi que des algorithmes généraux de résolution, notamment basés sur les techniques de filtrage et de propagation.

L'ensemble des recherches réalisées par la communauté contraintes et le développement de puissants résolveurs de contraintes open source (Choco [Cho07]) ou commerciaux (ILOG Solver [CP07], Chip [Chip08]) témoignent de l'intérêt succité par la programmation par contraintes. Forte de trente années de travaux académiques et industriels, la PPC propose un large panel de techniques de modélisation ainsi que des algorithmes de recherche de solution efficaces, qui permettent désormais d'appréhender des problèmes combinatoires d'envergure.

Il convient cependant de noter que, pour résoudre efficacement un problème réel à l'aide de la programmation par contraintes, il faut une **modélisation efficace** et des **algorithmes de recherche performants**. Une des difficultés inhérente à la programmation par contraintes est donc de savoir modéliser efficacement son problème et/ou de savoir choisir la méthode de

résolution la plus appropriée parmi la grande diversité de méthodes de résolution proposées en PPC.

Ainsi, si la programmation par contraintes est un paradigme de choix pour résoudre des problèmes fortement combinatoires, il convient de noter que des problèmes réels d'envergure ne peuvent être abordés sans une très bonne connaissance de la programmation par contraintes, ce qui rend cette dernière difficile d'accès à des non spécialistes. La première partie de cette thèse, dédiée à l'apprentissage automatique de réseaux de contraintes, proposera différentes techniques d'interaction permettant d'aider un expert "métier" totalement novice en PPC, à modéliser son problème sous la forme d'un réseau de contraintes. La seconde partie utilisera en revanche le paradigme de la programmation par contraintes comme un outil de résolution pour la planification d'actions en robotique.

Première partie

Apprentissage supervisé d'un réseau de contraintes

Chapitre 2

Apprentissage automatique de réseau de contraintes : État de l'art

Malgré le succès qu'elle connaît, la diffusion de la programmation par contraintes n'est pas aussi soutenue qu'escomptée. Une raison, non déterminante mais néanmoins influente, vient du fait que de nombreuses applications réelles (la mise en place des emplois du temps par exemple) qui pourraient être résolues à l'aide de CSP, ne font pas appel à la programmation par contraintes car les experts "métier" (la personne chargée du planning dans notre exemple) en charge de ces applications ne maîtrisent pas ce paradigme et sont donc incapables de modéliser leur problème sous la forme de réseaux de contraintes. Depuis près de cinq ans, une partie de la communauté "contraintes" travaille sur l'acquisition automatique de réseau de contraintes à partir d'instances que l'utilisateur accepte ou n'accepte pas comme solution à son problème [BCO+03] [LL05].

L'acquisition automatique de réseau de contraintes, vu comme un problème d'apprentissage automatique, consiste à développer des solutions capables d'aider un expert "métier", totalement novice en programmation par contraintes, à modéliser son problème sous la forme d'un réseau de contraintes sémantiquement correct,[1] c'est-à-dire représentant fidèlement le problème de l'utilisateur.

Ce chapitre commence par présenter l'apprentissage automatique (section 2.1), domaine dans lequel s'inscrit l'acquisition automatique de réseau de contraintes. Nous formalisons ensuite, dans la section 2.2, le problème d'acquisition de contraintes comme un problème d'apprentissage automatique. Enfin, la section 2.3 présente la plate-forme d'apprentissage de réseau

[1] À l'heure actuelle, la production de modèles efficaces relève quant à elle du problème de reformulation évoqué à la section 1.2 du chapitre 1.

de contraintes CONACQ pour laquelle nous proposerons une extension au chapitre 3.

2.1 Apprentissage automatique

2.1.1 Présentation générale

L'objectif de l'*apprentissage automatique* (*Machine Learning* en anglais) est de concevoir des machines capables d'évoluer automatiquement grâce à l'expérience [Mit97]. L'apprentissage automatique regroupe pour cela l'ensemble des méthodes permettant à une machine de construire un *modèle* de la réalité à partir de données, soit en améliorant un modèle partiel ou moins général, soit en créant complètement ce modèle [CMK02].

L'apprentissage automatique est à l'heure actuelle une discipline en plein essor et ses domaines d'application sont très nombreux. Les techniques d'apprentissage automatique sont ainsi utilisées par exemple pour la reconnaissance de forme (écriture, parole, vision), la fouille de données (extraction de connaissance), la mise en place d'outils d'aide à la décision, mais aussi pour le filtrage de courriels indésirables (anti-spam) ou bien encore par les moteurs de recherche pour l'indexation de pages web. Par ailleurs, il fait l'objet de nombreux travaux académiques publiés dans des revues spécialisées sélectives telles que le *Machine Learning Journal* ou le *Journal of Artificial Intelligence Research*[2], ainsi que dans le cadre de conférences parmi lesquelles l'*International Conference on Machine Learning* (ICML), l'*European Conference on Machine Learning* (ECML) et la *Conférence francophone d'Apprentissage* (CAP).

L'objectif de cette section est de présenter succinctement l'apprentissage automatique afin de situer les travaux portant sur l'acquisition automatique de CSP. Cette section ne se prétend pas une étude exhaustive des techniques d'apprentissage. Aussi, le lecteur est invité à se reporter aux ouvrages [Mit97] et [CMK02], et dans une moindre mesure [RN03] (Partie VI), pour approfondir les notions abordées dans cette section.

Apprentissage supervisé, non-supervisé ou par renforcement

On distingue trois types d'apprentissage automatique : l'apprentissage *supervisé*, l'apprentissage *non-supervisé* et l'apprentissage par *renforcement*.

En apprentissage *supervisé*, la machine apprenante, aussi appelée *apprenant*, analyse des *données d'entraînement* (ou *exemples*), chacune pourvue d'une *étiquette* fournie par un *oracle* (expert). La machine apprenante doit alors trouver ou approximer le *concept cible*, c'est-à-dire

[2] *http ://www.jmlr.org*

le modèle (la fonction) permettant d'affecter la bonne étiquette à chacun des exemples. Dans le cas de l'apprentissage *non-supervisé*, aucun expert n'est disponible. L'apprentissage se fait de manière totalement autonome sans l'aide d'un superviseur. L'apprenant doit être capable de déterminer, uniquement à partir des données d'entraînement, la structure de ces données. Enfin, en apprentissage par *renforcement* (*Q-learning*), la machine apprenante interagit avec son environnement en exécutant des actions, et chaque action produit une valeur de retour (sous la forme d'une récompense ou d'une sanction). L'objectif de l'apprentissage par renforcement consiste alors à analyser ces valeurs de retour afin de déterminer quelle(s) est (sont) la (les) meilleure(s) action(s) à exécuter.

Dans le cas de l'apprentissage automatique de réseau de contraintes, l'expert "métier" possède une bonne connaissance du problème qu'il cherche modéliser sous la forme d'un réseau de contraintes. Ainsi, bien qu'il ne sache pas exprimer son problème en termes de contraintes, il est supposé savoir reconnaître les exemples dans lesquels les contraintes de son problème sont satisfaites ou violées. Il est donc capable d'étiqueter les données d'entraînement. L'apprentissage automatique de réseau de contraintes se place par conséquent dans le cadre de l'apprentissage supervisé, dans lequel l'expert est sollicité pour étiqueter correctement les exemples fournis à l'apprenant.

Biais d'apprentissage

Le *biais d'apprentissage*, aussi appellé *biais inductif*, d'un algorithme d'apprentissage automatique est l'ensemble des hypothèses à partir desquelles l'apprenant va chercher à exprimer le concept cible. Le biais d'apprentissage réduit et structure l'espace des hypothèses, tout en "indiquant" la nature de la fonction cible, et est indispensable à l'apprentissage automatique [Mit80]. Cependant, comme l'expliquent Cornuejols & Miclet dans [CMK02], "*si les « œillères » dont on a muni l'apprenant ne correspondent pas avec la fontion cible [...], on ne peut pas l'apprendre correctement*". Il convient donc de noter que le choix du biais d'apprentissage est capital en vue d'un apprentissage efficace. Ce choix correspond par ailleurs à un pré-requis très fort, incontournable en apprentissage automatique.

L'acquisition automatique de réseau de contraintes a pour but de fournir un réseau de contraintes modélisant de manière adéquate le problème de l'utilisateur ; ce réseau de contraintes étant ensuite fourni (éventuellement après reformulation) à un résolveur de contraintes. Comme nous le verrons plus précisément dans la section 2.2, le biais d'apprentissage utilisé pour l'acquisition automatique de réseau de contraintes sera une bibliothèque de contraintes issue de ce résolveur. Le processus d'acquisition consistera alors à modéliser le

problème de l'utilisateur sous la forme d'un CSP à l'aide de ces contraintes.

2.1.2 Apprentissage numérique et apprentissage symbolique

Il existe deux tendances principales en apprentissage, celle issue des statistiques, qualifée de *numérique*, et celle issue de l'intelligence artificielle, qualifiée de *symbolique*.

Apprentissage numérique

Les techniques d'apprentissage *numérique* telles que les réseaux de neurones, les machines à vecteur de support ou les algorithmes génétiques sont particulièrement efficaces pour appréhender des problèmes de *prédiction*, c'est-à-dire des problèmes dans lesquels l'apprenant doit « seulement » chercher à établir (prédire) correctement l'étiquettage d'un nouvel exemple. Le principal défaut de l'apprentissage numérique est l'absence de sémantique. Ces techniques calculent en effet des coefficients numériques afin d'optimiser une fonction de coût, qui sert alors de fonction de prédiction. La seule explication possible de l'acceptation ou du rejet d'un exemple est que la fonction de prédiction retourne une valeur au-dessus (ou en dessous) d'un seuil arbitraire. L'apprentissage numérique ne semble par conséquent pas adapté pour l'apprentissage de réseau de contraintes pour lequel on cherche une modélisation **explicite** (*i.e.* le réseau de contraintes) du problème étudié.

Les techniques d'apprentissage numérique sont généralement utilisées en apprentissage non-supervisé pour appréhender des problèmes d'apprentissage où le volume de données est très grand, comme en bio-informatique par exemple. Il convient cependant de noter que l'apprentissage numérique peut être utilisé pour apprendre les préférences d'un problème, une fois celui-ci modélisé sous la forme d'un réseau de contraintes [RS98] [RS04].

Apprentissage symbolique

L'objectif de l'apprentissage *symbolique* est plus large que celui de l'apprentissage numérique. En effet, en plus de la tâche de prédiction (*i.e.* l'étiquetage correct des exemples), l'apprentissage symbolique cherche à identifier une explication **globale explicite** qui rende correctement compte des données d'entraînement. Pour cela, les techniques d'apprentissage symbolique cherchent à déterminer, parmi l'ensemble des hypothèses du biais d'apprentissage, un sous-ensemble d'hypothèses modélisant correctement le concept cible. L'apprentissage symbolique, vu comme une tâche d'identification de concepts, semble ainsi plus adapté que l'approche numérique pour appréhender l'apprentissage automatique de réseau de contraintes.

L'apprentissage symbolique fait l'objet de nombreuses recherches. Parmi les approches développées en apprentissage symbolique, nous pouvons citer, sans être exhaustifs, l'apprentissage avec borne du nombre d'erreurs et l'algorithme associé WINNOW [Lit88], la programmation logique inductive [MR94] [Rae97] et l'Espace des Versions [Mit82] dont nous reparlerons en section 2.3.1.

2.2 Le problème d'acquisition de contraintes

2.2.1 Formalisation du problème d'acquisition de contraintes

Comme l'illustre la figure 2.1, l'apprentissage de réseau de contraintes a pour but de soulager l'utilisateur en lui fournissant des méthodes semi-automatiques pour acquérir automatiquement un modèle du problème qu'il cherche à modéliser sous la forme d'un réseau de contraintes. Comme point de départ, nous supposons que l'utilisateur connaît l'ensemble $\mathcal{X}$ des variables du problème ainsi que leur domaine $\mathcal{D}$ de valeurs possibles. Il est aussi supposé capable de se prononcer sur la validité d'une instance et fournit à ce titre E^+ un sous-ensemble des solutions de son problème et E^- un ensemble de non-solutions.[3]

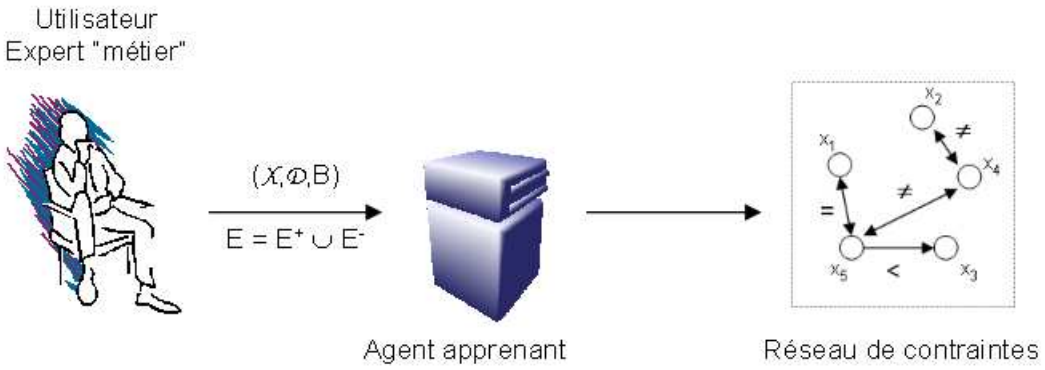

FIG. 2.1 – Schéma de principe de l'apprentissage automatique réseau de contraintes.

L'objectif de l'apprentissage de contraintes consiste à modéliser le problème de l'utilisateur dans un résolveur de contraintes. Le *biais d'apprentissage* noté B est une librairie de contraintes

[3]La connaissance des variables et de leur domaine est un pré-requis fort. Il est cependant envisageable de relâcher cette hypothèse. Lorsque l'utilisateur n'est pas capable d'extraire les variables de son problème, l'approche présentée dans [RBQ06] propose de les extraire directement de l'historique des solutions.

issue de ce résolveur. Dans un souci de clarté, nous restreignons notre étude au cas binaire dans la suite de ce chapitre. Le biais d'apprentissage est dans ce cas défini comme suit :

Définition 4 (Biais) *Un* biais local B_{ij} *est un ensemble de contraintes binaires* c_{ij} *portant sur les contraintes* x_i *et* x_j *d'un problème. Un* biais B *est un ensemble de contraintes formé par l'union de biais locaux.*

Un biais est *uniforme* lorsque la même librairie de contraintes est utilisée sur tous les biais locaux. Il est *complet* lorsque l'union des scopes de ses biais locaux correspond au graphe complet.

Définition 5 (Admissibilité à un biais) *Soient* B *un biais d'apprentissage et* $\mathcal{C}$ *un ensemble de contraintes,* $\mathcal{C}$ *est* admissible *pour* B *si pour toute contrainte* c_{ij} *de* $\mathcal{C}$*, il existe un ensemble de contraintes* $\{\mathsf{b}_{ij}^1, \cdots, \mathsf{b}_{ij}^k\}$ *de* B *tel que* $c_{ij} = \mathsf{b}_{ij}^1 \cap \cdots \cap \mathsf{b}_{ij}^k$*.*

Apprendre un réseau de contraintes consiste à rechercher une séquence de contraintes $\mathcal{C}$ admissible pour un biais d'apprentissage B donné et dont l'ensemble des solutions est un sur-ensemble de E^+ ne contenant aucun élément de E^-. Le problème d'acquisition de contraintes se définit alors formellement à l'aide de la définition 6.

Définition 6 (Pb. d'Acquisition de Contraintes) *Étant donnés un ensemble de variables* $\mathcal{X}$*, leur domaine* $\mathcal{D}$*, deux ensembles* E^+ *et* E^- *d'instances sur* $\mathcal{X}$ *et un biais d'apprentissage* B*, le problème d'acquisition de contraintes consiste à trouver une séquence de contraintes* $\mathcal{C}$ *telle que :*

$$\begin{cases} \mathcal{C} \text{ est admissible pour } \mathsf{B}, \\ \forall e^- \in E^-, e^- \text{ n'est pas solution de } (\mathcal{X}, \mathcal{D}, \mathcal{C}), \text{ et,} \\ \forall e^+ \in E^+, e^+ \text{ est solution de } (\mathcal{X}, \mathcal{D}, \mathcal{C}). \end{cases}$$

Exemple 3 *La figure 2.2 permet d'illustrer le principe d'apprentissage automatique de contraintes. Dans cet exemple, les variables du problème sont* $\mathcal{X} = \{x_1, x_2, x_3\}$*,* $D(x_i) = [1, 2, 3, 4]\, \forall x_i \in \mathcal{X}$ *et on utilise le biais d'apprentissage* $\mathsf{B} = ((\{x_1, x_2\}, L), (\{x_2, x_3\}, L))$ *où* $L = \{<, \leq, =, \geq, >, \neq\}$ *pour acquérir un CSP à partir d'une instance positive et de deux instances négatives. Le réseau de contraintes représenté sur la droite de la figure 2.2 répond au problème d'acquisition de contraintes puisqu'il accepte pour solution l'instance* $e_1^+ = (1, 3, 2)$ *et rejette les non-solutions* $e_1^- = (3, 2, 1)$ *et* $e_2^- = (3, 3, 2)$*.* □

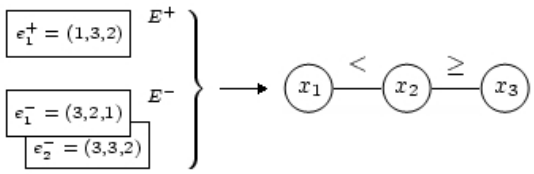

FIG. 2.2 – Le principe d'apprentissage automatique de contraintes, illustré par un exemple.

2.2.2 Approches existantes

L'acquisition automatique de contraintes est un domaine de recherche récent. Différents travaux, basés sur des techniques d'apprentissage automatique, ont néanmoins été proposés pour aider un utilisateur à modéliser un problème sous la forme d'un réseau de contraintes.

Dans [RS98] et plus récemment [RS04], le but ne consiste pas exactement à aider l'utilisateur à acquérir un réseau de contraintes, mais à l'aider à modéliser les préférences d'un problème déjà modélisé sous la forme d'un CSP. L'approche décrite dans [FW02] propose quant à elle des stratégies de suggestion pour des modélisations incomplètes. Cette approche est proposée pour des applications où les utilisateurs n'ont pas prévu toutes les contraintes à l'avance, mais sont néanmoins capables d'en rajouter lorsqu'ils sont confrontés à de mauvaises solutions.

L'approche décrite dans [LL05] permet l'apprentissage de contraintes en extension (aussi appellées *ad-hoc*). L'idée consiste à apprendre les propagateurs de la contrainte : pour chaque valeur v de chaque variable x de la contrainte, un classifieur détermine, en fonction des domaines des autres variables de la contrainte, si v peut être filtrée. Cette approche se limite néanmoins à l'apprentissage d'une seule contrainte. Elle ne répond donc qu'en partie au problème d'acquisition de contraintes, tel que nous l'avons défini dans la section 2.2.1.

Enfin, la plate-forme CONACQ, développée et maintenue au LIRMM, vise à aider un utilisateur novice à modéliser son problème sous la forme d'un réseau de contraintes, à partir d'instances que l'utilisateur accepte ou n'accepte pas comme solution à son problème.

2.3 La plate-forme d'apprentissage CONACQ

Présentée en 2003 [BCO+03], CONACQ est une plate-forme d'apprentissage de réseau de contraintes basée sur l'espace des versions [Mit82]. Dans cette section, nous nous intéressons à la formulation SAT de CONACQ [BCKO05] pour laquelle nous proposerons une extension dans le chapitre 3.

2.3.1 Processus d'apprentissage de la plate-forme CONACQ

L'ensemble des données d'entraînement fournies à CONACQ par l'utilisateur est constitué d'exemples positifs et négatifs.

Définition 7 (Données d'entraînement) *L'ensemble E^f des données d'entraînement fournies à* CONACQ *est constitué d'un ensemble E d'instances et d'une fonction de classification $f : E \rightarrow \{0, 1\}$. Une instance e de E est telle que $f(e) = 1$ si elle correspond à une solution du problème que l'utilisateur cherche à modéliser en contraintes. On parle alors d'exemple* positif*, que l'on note e^+. Si e est une non-solution au problème, $f(e) = 0$ et on parle d'exemple* négatif*, noté e^-.*

On dit qu'un réseau de contraintes $\mathcal{C}$ est consistant avec les données d'entraînement E^f si chaque exemple positif e^+ de E^f appartient à l'ensemble des solutions de $\mathcal{C}$ et si chaque exemple négatif e^- de E^f est une non-solution pour $\mathcal{C}$. Étant donnés un biais d'apprentissage B et un ensemble E^f de données d'entraînement, le problème d'acquisition de contraintes revient donc à trouver un réseau de contraintes $\mathcal{C}$ admissible pour B et consistant avec les données d'entraînement E^f.

CONACQ est une plate-forme d'apprentissage automatique de réseau de contraintes basée sur l'espace des versions. Introduit par Mitchell [Mit82], l'espace des versions est une technique d'apprentissage symbolique supervisé à partir d'exemples et de contre-exemples d'un concept cible. De manière informelle,[4] étant donnés $\mathcal{H}$ un ensemble d'hypothèses et E un ensemble de données d'entraînement constitué d'exemples et de contre-exemples du concept cible que l'on cherche à apprendre, l'espace des versions est l'ensemble des hypothèses de $\mathcal{H}$ qui sont cohérentes avec les données d'entraînement. Dans le cadre d'un problème d'acquisition de contraintes, l'espace des versions $V_{\mathsf{B}}(E^f)$ d'un problème d'acquisition de contraintes est l'ensemble de tous les réseaux admissibles pour un biais B donné et consistants avec les données d'entraînement E^f. Dans sa formulation SAT de CONACQ, l'espace des versions est codé par une théorie clausale K où chaque modèle de cette théorie est un réseau de contraintes de $V_{\mathsf{B}}(E^f)$.

Si B est un biais d'apprentissage, un littéral est soit un atome b_{ij} de B, soit sa négation $\neg\mathsf{b}_{ij}$. Il convient de noter que $\neg\mathsf{b}_{ij}$ n'est pas une contrainte mais indique que b_{ij} n'appartient pas au réseau appris. Une clause est une disjonction de littéraux et la théorie clausale K est une conjonction de clauses.

[4] Pour plus de précisions sur l'espace des versions, le lecteur se rapportera à [CMK02], chapitre 4.

Définition 8 (Interprétation) *Soit* B *un biais d'apprentissage, une* interprétation I *sur* B *est une fonction qui assigne à chaque atome* b_{ij} *de* B *une valeur* $I(\mathsf{b}_{ij})$ *dans* $\{0, 1\}$.

Définition 9 (Transformation) *Soit* B *un biais d'apprentissage, une* transformation *est une fonction* ϕ *qui assigne à toute interprétation* I *de* B *le réseau de contraintes* $\phi(I)$ *admissible pour* B *et tel que :*

$$\mathcal{C}_{ij} \in \phi(I) \text{ ssi } \mathcal{C}_{ij} = \bigcap \{\mathsf{b}^p_{ij} \in \mathsf{B} : I(\mathsf{b}^p_{ij}) = 1\}.$$

Une interprétation I est un *modèle* de la théorie clausale K si K est vrai dans I selon la sémantique propositionnelle standard. L'ensemble de tous les modèles de K est noté $Modeles(\mathsf{K})$.

La construction de K suit le mode opératoire suivant. Pour chaque instance d'entraînement e fournie par l'utilisateur, CONACQ construit l'ensemble $\kappa(e)$ des littéraux correspondant aux contraintes $\mathsf{b}_{ij} \in \mathsf{B}$ qui permettent de rejeter e. CONACQ ajoute alors itérativement un ensemble de clauses de manière à ce que, pour chaque interprétation I de $Modeles(\mathsf{K})$, le réseau $\phi(I)$ classifie correctement les intances d'entraînement précédemment analysées, ainsi que l'instance e. La mise à jour de la théorie clausale K diffère selon la classe d'appartenance de e :

- Si e est un exemple négatif alors une seule des contraintes identifiées dans $\kappa(e)$ suffit à expliquer le rejet de e. On ajoute donc à K la disjonction $\{\bigvee_{\mathsf{b}_{ij} \in \kappa(e)} \mathsf{b}_{ij}\}$ des éléments de $\kappa(e)$.
- Si e est un exemple positif, alors CONACQ est assurée qu'aucune des contraintes identifiées dans $\kappa(e)$ n'appartient au réseau cible (sinon, e serait une non-solution). En conséquence, K est mise à jour en ajoutant la conjonction des clauses unitaires $\{\neg \mathsf{b}_{ij}\}$ de $\kappa(e)$.

L'analyse des instances positives de E^f permet de simplifier par propagation unitaire les clauses de rejet des instances négatives. La correction de l'algorithme utilisé par CONACQ est par ailleurs formalisée par le théorème 1.

Théorème 1 (Correction de CONACQ) *Soient* E^f *des données d'entraînement et* B *un biais d'apprentissage, la théorie clausale* K *construite par* CONACQ *pour* E^f *et* B *est telle que :*

$$V_{\mathsf{B}}(E^f) = \{\phi(m) \mid m \in Modeles(\mathsf{K})\}$$

Preuve. La preuve du théorème 1 est donnée dans [BCKO05].

L'exemple 4 permet d'illustrer simplement le mode de fonctionnement de la plate-forme CONACQ.

Exemple 4 (CONACQ) *Dans cet exemple, nous cherchons un réseau de contraintes mettant en jeu 4 variables* $x_1, \ldots, x_4$ *dont les domaines sont* $D(x_1) = \ldots = D(x_4) = \{1, 2, 3, 4\}$. *Notre biais d'apprentissage* B *est complet et uniforme, et utilise la librairie* $L = \{\leq, \neq, \geq\}$. *Ainsi, pour tout* $1 \leq i < j \leq 4$, B *contient* $\leq_{ij}, \neq_{ij}$ *et* $\geq_{ij}$. *Dans cet exemple, le problème que nous souhaitons acquérir met uniquement en jeu la contrainte :* $x_1 \neq x_4$. *Pour chaque instance d'entraînement* e, *le tableau 2.1 présente l'évolution de la théorie clausale* K *par construction de l'ensemble* $\kappa(e)$ *des contraintes rejetant* e. □

E^f	exemple	clauses ajoutées à K
$\{e_1^+\}$	(1,2,3,4)	$\neg \geq_{12} \wedge \neg \geq_{13} \wedge \neg \geq_{14} \wedge \neg \geq_{23} \wedge \neg \geq_{24} \wedge \neg \geq_{34}$
$\{e_2^+\}$	(4,3,2,1)	$\neg \leq_{12} \wedge \neg \leq_{13} \wedge \neg \leq_{14} \wedge \neg \leq_{23} \wedge \neg \leq_{24} \wedge \neg \leq_{34}$
$\{e_3^-\}$	(1,1,1,1)	$(\neq_{12} \vee \neq_{13} \vee \neq_{14} \vee \neq_{23} \vee \neq_{24} \vee \neq_{34})$

TAB. 2.1 – Évolution de la base de connaissance K de CONACQ pour l'exemple 4.

L'objectif du processus d'apprentissage de CONACQ consiste à modéliser le problème de l'utilisateur sous la forme d'un réseau de contraintes, destiné à être résolu à l'aide du résolveur de contraintes CHOCO. Comme nous l'avons déjà évoqué dans le chapitre 1, une condition nécessaire à une résolution efficace est l'utilisation de contraintes possèdant des algorithmes de propagation performants. Aussi, le biais d'apprentissage de CONACQ est uniquement constitué de contraintes ayant des caractéristiques de propagation efficaces. Cette précaution d'utilisation ne garantit cependant pas la production de modèles efficaces. La modélisation fournie par CONACQ à l'issue du processus d'apprentissage pourra donc faire l'objet d'une reformulation (à l'aide des techniques évoquées dans la section 1.2 du chapitre 1) afin d'améliorer l'efficacité de sa résolution. Par ailleurs, lorsque certaines contraintes du problème de l'utilisateur ne peuvent être exprimées à l'aide des contraintes du biais d'apprentissage (parce que celui-ci est trop restreint), il est possible d'ajouter des contraintes *ad-hoc*[5] au modèle établi par CONACQ afin de modéliser correctement le problème.

Pour conclure cette section, il convient de noter que CONACQ présente deux très bonnes propriétés : la *commutativité* et l'*incrémentalité*. En apprentissage, la commutativité impose d'arriver au même résultat (*i.e.* à la même approximation du concept cible) quel que soit l'ordre des données d'entraînement. L'incrémentalité impose quant à elle de modifier de manière minimale l'ensemble des règles apprises lorsqu'un nouvel exemple est ajouté aux données

[5] Ces contraintes *ad-hoc* peuvent elles-mêmes être acquises automatiquement au moyen de l'approche décrite dans [LL05].

d'entraînement. Enfin, CONACQ peut par ailleurs être utilisée de deux manières différentes. La plate-forme est en effet destinée à aider un utilisateur novice à modéliser son problème sous la forme d'un réseau de contraintes, mais peut aussi être utilisée pour améliorer un réseau existant, via l'acquisition automatique de contraintes globales implicites [BCP07].

2.3.2 Complexité des opérations courantes

Dans cette section, nous présentons la complexité des principales opérations réalisées par la plate-forme CONACQ, telles qu'elles ont été établies dans [BCKO05]. Pour cela, nous considérons que le biais B contient b contraintes et que les données d'entraînement (E^+, E^-) contiennent m exemples.

On dit que l'espace des versions s'est *effondré* lorsqu'il est vide, ce qui caractérise le fait qu'il n'existe pas de réseau de contraintes $\mathcal{C}$ admissible pour B tel que $\mathcal{C}$ soit consistant avec les données d'entraînement (E^+, E^-). Lors du processus d'apprentissage réalisé par CONACQ, la complexité temporelle du test d'effondrement est $\mathcal{O}(bm)$.

Le test d'*appartenance*, qui consiste à déterminer si un réseau de contraintes appartient ou non à l'espace des versions, est de complexité temporelle $\mathcal{O}(bm)$.

L'opération de *mise à jour* consiste à calculer le nouvel espace des versions lorsqu'un nouvel exemple e a été ajouté aux données d'entraînement. La complexité temporelle de cette opération de mise à jour est $\mathcal{O}(b)$.

Enfin, le test de *classification* consiste à déterminer si un exemple e est rejeté par tous les réseaux de contraintes de l'espace des versions (classifié '0'), s'il est accepté par tous ces réseaux (classifié '1') ou bien encore seulement accepté par un sous-ensemble de ces réseaux (classifié '?'). La complexité temporelle de classification réalisée par CONACQ est $\mathcal{O}(b^2m)$.

2.3.3 Contraintes redondantes et détection du Backbone

Dans le domaine de la programmation par contraintes, les contraintes peuvent être interdépendantes. On parle alors de *redondance* entre contraintes. Dans cette section, nous nous intéressons aux deux approches employées par la plate-forme CONACQ pour exploiter la redondance : les *règles de redondance* et la détection du *backbone*.

Contraintes redondantes et acquisition automatique de réseau

En programmation par contraintes, les contraintes peuvent être interdépendantes. Par exemple, deux contraintes telles que $\geq_{12}$ et $\geq_{23}$ imposent une restriction sur la relation de toute contrainte définie entre x_1 et x_3. La conséquence de cette interdépendance est que certaines contraintes peuvent être *redondantes*. Une contrainte c_{ij} est redondante[6] pour un réseau $\mathcal{C}$ si l'ensemble des solutions de $\mathcal{C}$ est identique à l'ensemble des solutions du réseau $\mathcal{C}$ privé de c_{ij}. La contrainte $\geq_{13}$ est ainsi redondante pour les contraintes $\geq_{12}$ et $\geq_{23}$.

En acquisition de contraintes, la connaissance des contraintes redondantes est cruciale. En effet, cette connaissance spécifique au domaine permet à l'espace des versions de converger ou permet tout du moins d'identifier les parties du réseau de contraintes qui ne sont pas encore apprises précisément. L'une des méthodes proposées dans [BCFO04] consiste à ajouter à K l'ensemble des *règles de redondance* relatives à la librairie de contraintes L utilisée pour construire le biais d'apprentissage B. À titre d'exemple, si la librairie contient la contrainte $\leq$ pour laquelle nous savons que $\forall x, y, z, (x \leq y) \wedge (y \leq z) \rightarrow (x \leq z)$, alors, pour chaque paire de contraintes $\leq_{ij}, \leq_{jk}$ de B, nous ajoutons la clause de Horn $\leq_{ij} \wedge \leq_{jk} \rightarrow \leq_{ik}$ dans K, ce qui permettra au processus d'apprentissage de converger.

La détection du Backbone

Les règles de redondance présentées ci-dessus permettent de découvrir des implications de "conjonctions" de contraintes. [BCKO05] montre qu'il existe cependant des formes plus complexes de redondance, correspondant à des combinaisons de "conjonctions" et de "disjonctions" de contraintes, que les règles de redondance ne peuvent capturer.

La raison de cette incapacité vient du fait que les règles de redondance sont des clauses de Horn, qui sont appliquées seulement quand *tous* les littéraux de la partie gauche de la règle sont vrais. Pour capturer la redondance issue d'une combinaison de "conjonctions" et de "disjonctions" de contraintes, la version SAT de Conacq proposée dans [BCKO05] utilise le puissant concept de *backbone* d'une formule propositionnelle. Informellement, un littéral appartient au backbone si et seulement si il appartient à tous les modèles d'une formule [MZK+99]. Durant le processus d'apprentissage, les littéraux du backbone sont exploités dès qu'ils sont détectés, pour mettre à jour l'espace des versions contruit par Conacq.

La formulation SAT de Conacq que nous utiliserons dans la suite de cette thèse utilise pleinement les règles de redondance et la détection du backbone, qui améliorent significativement les performances de la plate-forme. De manière générale, le traitement de la redondance

[6] Nous nous intéressons ici à la notion de contraintes de redondance relativement à la tâche d'apprentissage et laissons de côté l'impact des redondances dans un réseau de contraintes évoqué dans le chapitre 1.

permet de ne pas surestimer la taille de l'espace des versions et d'identifier avec précision les sous-parties du problème pour lesquelles peu de connaissance a été acquise.

Pour plus de précisions sur les notions de règles de redondance et de détection du backbone, ainsi que sur l'impact de leur utilisation, nous recommandons au lecteur de se reporter à [BCKO05].

2.4 Conclusion

La programmation par contraintes est un paradigme désormais largement utilisé pour résoudre des problèmes combinatoires issus d'applications réelles. Sa diffusion n'est cependant pas aussi soutenue qu'escomptée. Aussi, différentes approches, basées sur des techniques d'apprentissage automatique, ont été développées dans l'optique d'aider un utilisateur à modéliser son problème sous la forme d'un réseau de contraintes. Dans la première partie de cette thèse, nous nous intéressons à la plate-forme CONACQ, basée sur l'espace des versions.

CONACQ se révèle être un outil efficace d'apprentissage automatique de réseau de contraintes, possédant de très bonnes propriétés telles que la commutativité et l'incrémentalité. Comme tout système d'apprentissage automatique, l'efficacité de CONACQ est liée au biais d'apprentissage, qui doit être choisi avec soin.

Il convient cependant de noter que l'implémentation actuelle de CONACQ est passive vis-à-vis des données d'entraînement. Cette approche est perfectible dans la mesure où elle repose sur la capacité de l'utilisateur à fournir des instances significatives de son problème. Dans le chapitre 3, nous proposerons une version interactive de CONACQ, capable de poser des questions à l'utilisateur afin de limiter de manière conséquente le nombre d'instances nécessaires à la convergence de la plate-forme.

Chapitre 3

Apprentissage interactif de réseaux de contraintes

La plate-forme d'apprentissage automatique CONACQ présentée dans le chapitre 2 est passive vis-à-vis de l'utilisateur, c'est-à-dire basée sur la capacité de ce dernier à fournir des instances significatives de son problème. Dans ce chapitre, nous proposons une approche théorique permettant de dévolopper un CONACQ interactif, capable de générer des questions que l'utilisateur doit classifier comme solution ou non-solution à son problème. Afin de limiter le nombre d'interactions avec l'utilisateur, nous proposons différentes stratégies de questionnement que nous validons ensuite empiriquement.

Le chapitre est organisé comme suit. La section 3.1 définit formellement le problème d'acquisition interactive de réseau de contraintes. Dans les sections 3.2 et 3.3, nous présentons différentes stratégies de génération de questions que nous validons à l'aide d'un panel de résultats expérimentaux présentés en section 3.4.

3.1 Le problème d'acquisition interactive de contraintes

Le problème d'acquisition de contraintes, tel que nous l'avons défini dans le chapitre 2, consiste à trouver un réseau de contraintes consistant avec les données d'entraînement fournies par l'utilisateur. Il s'agit cependant d'une vision très *statique* du problème où les données d'entraînement sont fournies à l'avance et en nombre suffisant. Dans la réalité, statuer sur la validité des données d'entraînement requiert l'analyse de l'expert "métier", à laquelle est associé un coût. Fournir des données d'entraînement en nombre suffisant n'est donc pas gratuit. Nous devons de ce fait essayer de minimiser le nombre d'instances nécessaires à l'acquisition du *réseau cible* C_T qui modélise en contraintes le problème que l'utilisateur a en tête. Étant

donnés l'ensemble des variables $\mathcal{X}$ et l'ensemble de leur domaine respectif $\mathcal{D}$, C_T doit être tel que toute instance sur $\mathcal{X}$ est un exemple positif si et seulement si c'est une solution de C_T.

Durant le processus d'apprentissage, CONACQ possède l'information nécessaire à caractériser quelle serait la prochaine donnée d'entraînement idéale. Comme nous le verrons dans les sections 3.2 et 3.3, le système d'acquisition est capable d'identifier les "bons" exemples d'entraînement qui, en fonction de la classification de l'utilisateur, sont susceptibles de réduire autant que possible la taille de l'espace des versions. Nous définissons les notions de *question* et de *classification* de la manière suivante :

Définition 10 (Question et Classification) *Une question est une instance sur $\mathcal{X}$ construite par le système d'acquisition. L'utilisateur classifie une question q en utilisant une fonction f telle que $f(q) = 1$ si q est une solution à C_T et $f(q) = 0$ sinon.*

Dans [Ang04], Angluin définit la notion de *question d'appartenance* qui correspond exactement au type de questions que nous proposons. Dans le cadre de l'apprentissage de réseau de contraintes, poser une question d'appartenance revient à présenter une instance non-classifiée à l'utilisateur afin que celui-ci la classifie comme solution ou non-solution à son problème.

Nous définissons désormais le problème d'acquisition interactive de réseau de contraintes à l'aide de la définition 11.

Définition 11 (Problème d'acquisition interactive de réseau de contraintes) *Étant donnés un biais d'apprentissage B et une fonction de classification f, le* problème d'acquisition interactive de réseau de contraintes *consiste à trouver une séquence de questions $\mathcal{Q} = q_1, \ldots, q_m$ telle que q_{i+1} est une question sur $V_{\mathsf{B}}(E_i^f)$ où $E_i = \{q_1, \ldots, q_i\}$ et f_i est la restriction de f aux instances de E_i, et telle que $|V_{\mathsf{B}}(E_m^f)| = 1$.*

Il convient de noter que, dans la définition précédente, la séquence de questions est construite de manière incrémentale, c'est-à-dire que chaque question q_{i+1} est générée en fonction de la classification des questions précédentes. Minimiser la longueur de $\mathcal{Q}$ est par conséquent impossible car nous ne connaissons pas à l'avance les réponses que donnera l'utilisateur. Néanmoins, nous proposons dans la suite de ce chapitre un ensemble de techniques qui répondent au problème d'acquisition interactive de contraintes.

3.2 Une première approche de questionnement

Dans cette section, nous présentons une technique simple (polynomiale en temps), basée sur une sélection judicieuse d'instances, qui assure que quelque chose de nouveau est appris par la classification de l'utilisateur.

3.2.1 Principe général

L'attente passive de CONACQ suppose de la part de l'utilisateur la capacité à produire des instances significatives de son problème. En pratique, le manque d'efficacité du processus d'apprentissage résulte de deux causes principales :

1. L'utilisateur est limité par ses propres connaissances du problème qu'il souhaite modéliser en contraintes. Le problème étant difficile à résoudre, les exemples qu'il fournit sont en général relativement similaires (peu de valeurs diffèrent d'une instance à l'autre). Dans ce cas, leur analyse augmente peu, voire pas, la connaissance acquise.
2. Durant le processus d'apprentissage, deux instances apparemment très différentes peuvent être sémantiquement identiques pour un biais d'apprentissage donné. Par exemple, $e_1 = \langle(x_1, 1), (x_2, 2), (x_3, 3), (x_3, 4)\rangle$ et $e_2 = \langle(x_1, 2), (x_2, 3), (x_3, 4), (x_3, 5)\rangle$ produisent la même connaissance si on utilise la librairie arithmétique binaire $L = \{<, \leq, =, \geq, >, \neq\}$. On parle alors d'instances *redondantes*.

Pour éviter ces problèmes, nous proposons un premier générateur de questions. Notre approche consiste à doter la plate-forme d'un générateur d'instances aléatoires que CONACQ utilisera pour présenter à l'utilisateur des instances afin que ce dernier statue sur leur validité. La réponse obtenue servira alors à augmenter la connaissance acquise. Doté d'un tel générateur, la plate-forme pourra alors combattre les problèmes liés au biais de l'utilisateur (point 1).

Pour combattre les problèmes de similarité sémantique (point 2) et ainsi éviter de demander à l'utilisateur de classifier un nombre excessif d'instances, nous ajoutons la fonctionnalité suivante : avant de demander à l'utilisateur de statuer sur la validité des instances générées aléatoirement, nous les présentons d'abord à CONACQ afin de réaliser une sélection parmi ces instances et écarter ainsi les instances redondantes. Pour chaque instance e générée, CONACQ essaie de déduire la classification de e à partir de la connaissance actuelle K. Si CONACQ est capable de statuer sur la validité de e, alors e est une instance redondante qu'il est inutile de présenter à l'utilisateur, car sa classification n'apporterait aucune information supplémentaire. Les seules instances présentées sont celles pour lesquelles la réponse de l'utilisateur apporte un gain de connaissance. Notre démarche s'inscrit donc dans le cadre de l'*Active Learning* proposé dans [CGJ95] pour les techniques d'apprentissage numériques.

3.2.2 Formalisation

Dans cette section, nous présentons une formalisation du processus permettant d'empêcher de présenter des questions redondantes à l'utilisateur. Cette technique de questions *non redondantes* demande la classification par l'utilisateur d'un exemple e s'il ne peut pas être classifié par la représentation courante K de l'espace des versions. Un exemple e peut être classifié par $V_{\mathsf{B}}(E^f)$ s'il est, soit solution de tous les réseaux de $V_{\mathsf{B}}(E^f)$, soit non-solution de tous les réseaux de $V_{\mathsf{B}}(E^f)$. e est une solution de tous les réseaux de $V_{\mathsf{B}}(E^f)$ si et seulement si le sous-ensemble $\kappa(e)_{[\mathsf{K}]}$ de $\kappa(e)$, obtenu en retirant toutes les contraintes qui apparaissent déjà comme un littéral négatif de K, est vide. De manière symétrique, e est une non-solution de tous les réseaux de $V_{\mathsf{B}}(E^f)$ si et seulement si $\kappa(e)_{[\mathsf{K}]}$ est un sur-ensemble d'une clause existante dans K.

L'exemple 5 permet d'illustrer le principe de sélection des questions non redondantes.

Exemple 5 *Dans cet exemple, nous cherchons un réseau de contraintes mettant en jeu trois variables* $\mathcal{X} = \{x_1, x_2, x_3\}$ *où* $D(x_1) = D(x_2) = D(x_3) = \{1, 2, 3, 4, 5, 6\}$ *et pour lequel nous utilisons le biais d'apprentissage* $\mathsf{B} = ((\{x_1, x_2\}, L), (\{x_2, x_3\}, L))$, *où* L *est la librairie* $\{<, \leq, =, \geq, >, \neq\}$. *Soient* $e_1^+ = \langle (x_1, 2), (x_2, 2), (x_3, 5)\rangle$ *et* $e_2^- = \langle (x_1, 1), (x_2, 3), (x_3, 2)\rangle$ *les deux premières instances fournies à* CONACQ. *Après analyse de ces deux instances,* CONACQ *calcule la théorie clausale* $\mathsf{K} = \neg \neq_{12} \wedge \neg \geq_{23} \wedge (\geq_{12} \vee \leq_{23})$.

Soient maintenant $e_3 = \langle (x_1, 4), (x_2, 4), (x_3, 6)\rangle$, $e_4 = \langle (x_1, 5), (x_2, 6), (x_3, 1)\rangle$ *et* $e_5 = \langle (x_1, 5), (x_2, 5), (x_3, 5)\rangle$ *trois instances produites par notre générateur aléatoire. Comme présenté dans le tableau 3.1, toutes les contraintes de* $\kappa(e_3)$ *apparaissent dans* K *comme un littéral négatif.* $\kappa(e_3)_{[\mathsf{K}]}$ *est donc vidé par propagation unitaire, ce qui signifie que* e_3 *est une question redondante qui ne doit pas être présentée à l'utilisateur. L'analyse de* e_4 *montre quant à elle que* $\kappa(e_4)_{[\mathsf{K}]}$ *est un sur-ensemble de la clause ayant codé le rejet de l'instance d'entraînement* e_2^-. e_4 *est donc aussi écartée pour cause de redondance. L'instance* e_5, *pour laquelle* $\kappa(e_5)_{[\mathsf{K}]} = \{\neq_{23}\}$, *est en revanche une question non redondante qu'il est intéressant de présenter à l'utilisateur car la classification de ce dernier permettra d'augmenter la connaissance de la plate-forme.* □

3.3 Vers une génération de questions optimales

La technique de questions non redondantes présentée dans la section précédente garantit que la classification de chaque question posée à l'utilisateur ajoute de nouvelles informations à K. Cependant, le gain de connaissance apporté varie selon la question générée. En effet, le

E^f	x_1	x_2	x_3	$\kappa(e)$	$\kappa(e)_{[\mathsf{K}]}$
e_3	4	4	6	$\{\neq_{12}; \geq_{23}\}$	$\emptyset$
e_4	5	6	1	$\{\geq_{12}; \leq_{23}\}$	$\{\geq_{12}; \leq_{23}\}$
e_5	5	5	5	$\{\neq_{12}; \neq_{23}\}$	$\{\neq_{23}\}$

TAB. 3.1 – Analyse effectuée par CONACQ pour les instances de l'exemple 5.

gain d'une question est directement relié à la taille k de $\kappa(q)_{[\mathsf{K}]}$ et à sa classification $f(q)$. Si $f(q) = 1$, k clauses unaires sont ajoutées à K (*c.f.* chapitre 2, section 2.3), ce qui fixe k littéraux à 0. CONACQ ne permet pas d'accéder directement à la taille de l'espace des versions, à moins d'un calcul très coûteux sur K. En assumant que les modèles de K sont uniformément distribués, fixer k littéraux dans K permet de diviser le nombre de modèles par 2^k. Si $f(q) = 0$, une clause positive de taille k est ajoutée à K, ce qui permet de supprimer $1/2^k$ des modèles.

L'objectif de l'acquisition interactive de réseau de contraintes est de minimiser le nombre de questions nécessaires à la convergence de l'espace des versions à une hypothèse unique. Pour cela, nous distinguons les questions *optimistes* et les questions *optimales*.

3.3.1 Stratégies proposées

Une question *optimiste* est une question dont le gain de connaissance est maximum si elle est classée "en notre faveur" mais qui ne nous fournit que très peu d'information dans le cas contraire. Plus précisément, plus le $\kappa(q)_{[\mathsf{K}]}$ est grand, plus une question q est optimiste. Lorsque l'utilisateur classifie positivement une telle question, $|\kappa(q)_{[\mathsf{K}]}|$ littéraux sont fixés à 0 dans K, ce qui permet de diviser par $2^{|\kappa(q)_{[\mathsf{K}]}|}$ le nombre de modèles de K. Dans le cas contraire, CONACQ ajoute simplement une clause de taille $|\kappa(q)_{[\mathsf{K}]}|$. En conséquence, une question optimiste est maximalement informative lorsqu'elle est classifiée comme un exemple positif par l'utilisateur mais est en revanche peu informative si elle est classifiée négativement.

La stratégie optimale de questionnement consiste à proposer des questions qui réduisent de moitié la taille de l'espace versions quelle que soit leur classification [Mit97]. Nous qualifions ainsi d'*optimale* une question pour laquelle nous garantissons qu'un littéral sera fixé à 0 ou à 1 quelle que soit la réponse de l'utilisateur. Formellement, une question optimale est telle que $\kappa(q)_{[\mathsf{K}]}$ est de taille 1. Ainsi, le littéral de $\kappa(q)_{[\mathsf{K}]}$ sera fixé à 0 dans K en cas de classification positive et à 1 dans le cas contraire.

L'exemple 6 permet d'illustrer cette stratégie optimale.

Exemple 6 (Questions optimales) *Dans cet exemple, nous cherchons à modéliser le réseau cible de l'exemple 4 du chapitre 2. Ce réseau porte sur 4 variables* $x_1, \ldots, x_4$ *de domaine* $D(x_1) = \ldots = D(x_4) = \{1, 2, 3, 4, 5\}$ *et met uniquement en jeu la contrainte* $x_1 \neq x_4$. *Pour ce problème d'acquisition, nous utilisons le biais* B *complet et uniforme utilisant la librairie* $L = \{\leq, \neq, \geq\}$. *Ainsi, pour tout* $1 \leq i < j \leq 4$, B *contient* $\leq_{ij}, \neq_{ij}$ *et* $\geq_{ij}$. *Comme le présente le tableau 2.1 du chapitre 2, après analyse des données d'entraînement* $e_1^+ = \langle (x_1, 1), (x_2, 2), (x_3, 3), (x_3, 4) \rangle$, $e_2^+ = \langle (x_1, 4), (x_2, 3), (x_3, 2), (x_3, 1) \rangle$ *et* $e_3^- = \langle (x_1, 1), (x_2, 1), (x_3, 1), (x_3, 1) \rangle$, *l'unique clause positive de* K *est* $Cl = (\neq_{12} \vee \neq_{13} \vee \neq_{14} \vee \neq_{23} \vee \neq_{24} \vee \neq_{34})$. *Tous les autres littéraux de* K *sont fixés à 0 par analyse de* e_1^+ *et* e_2^+. *Dans la suite de cet exemple, nous notons* $\mathsf{K}_{\{e_1^+, e_2^+\}}$ *la conjonction* $(\neg \geq_{12}) \wedge \ldots (\neg \geq_{34}) \wedge (\neg \leq_{12}) \wedge \ldots \wedge (\neg \leq_{34})$. *Ainsi, après analyse des données d'entraînement,* CONACQ *a construit la théorie clausale* $\mathsf{K} = \mathsf{K}_{\{e_1^+, e_2^+\}} \wedge Cl$.

Le tableau 3.2 présente une séquence de questions optimales permettant de converger vers le réseau cible. La première colonne du tableau contient les questions e *générées dans le cadre de la stratégie optimale. La deuxième colonne renferme l'ensemble* $\kappa(e)_{[\mathsf{K}]}$ *des contraintes encore possibles dans un réseau de l'espace des versions qui pourraient rejeter* e. *La troisième colonne indique la classification fournie par l'utilisateur. Enfin, la dernière colonne nous indique l'évolution de la base de connaissance* K. *La question* e_4 *est telle que* $\neq_{12}$ *est la seule contrainte encore possible de l'espace des versions capable de rejeter* e_4. e_4 *étant classifiée positive par l'utilisateur, nous sommes sûrs que* $\neq_{12}$ *ne peut pas appartenir à un réseau de l'espace des versions. En conséquence,* CONACQ *ajoute la clause unaire* $(\neg \neq_{12})$ *dans* K *et la propagation unitaire supprime* $\neq_{12}$ *de la clause* Cl. *Le processus se repète pour les questions* e_5, e_6 *et* e_7. *À chaque étape, un littéral est supprimé de* Cl, *ce qui permet de diviser l'espace des versions de moitié après la classification de chaque question. Enfin, la dernière question* e_8 *permet de faire converger l'espace des versions vers le réseau cible, qui contient l'unique contrainte* $x_1 \neq x_4$.

Il convient de noter qu'après analyse des données d'entraînement e_1^+, e_2^+ *et* e_3^-, *l'espace des versions* $V_{\mathsf{B}}(E)$ *contenait* 2^6 *réseaux de contraintes admissibles. Il nous a fallu générer* $\mathcal{O}(log_2|V_{\mathsf{B}}(E)|)$ *questions pour converger vers le concept cible, ce qui correspond à la stratégie de questionnement optimale au sens de Mitchell ([Mit97], page 38).* □

Dans l'exemple 6, les questions générées sont toujours telles que $|\kappa(e)_{[\mathsf{K}]}| = 1$, comme l'impose la stratégie optimale. Cependant, la redondance (*c.f.* chapitre 2) peut nous empêcher de générer une question e pour une taille $\kappa(e)_{[\mathsf{K}]}$ donnée. Considérons en effet par exemple un problème d'acquition de contraintes, utilisant le biais complet et uniforme avec $L = \{\leq, \neq, \geq\}$, avec le réseau $x_1 = x_2 = x_3$ pour réseau cible. Après analyse d'un premier exemple positif (par exemple $e_1^+ = (2, 2, 2)$), les contraintes possibles de l'espace des versions sont $\leq_{12}, \leq_{13}, \leq_{23}, \geq_{12}, \geq_{13}$ et $\geq_{23}$. Tout nouvel exemple négatif aura, soit un $\kappa(e)_{[\mathsf{K}]}$ de taille

e	$\kappa(e)_{[\mathsf{K}]}$	$f(e)$	K
$e_4 = (1,1,2,3)$	$\{\neq_{12}\}$	$+$	$\mathsf{K}_{\{e_1^+,e_2^+\}} \wedge (\neg \neq_{12}) \quad \wedge \quad (\neq_{13} \vee \neq_{14} \vee \neq_{23} \vee \neq_{24} \vee \neq_{34})$
$e_5 = (2,1,1,3)$	$\{\neq_{23}\}$	$+$	$\mathsf{K}_{\{e_1^+,e_2^+\}} \wedge (\neg \neq_{12}) \wedge (\neg \neq_{23}) \quad \wedge \quad (\neq_{13} \vee \neq_{14} \vee \neq_{24} \vee \neq_{34})$
$e_6 = (2,3,1,1)$	$\{\neq_{34}\}$	$+$	$\mathsf{K}_{\{e_1^+,e_2^+\}} \wedge (\neg \neq_{12}) \wedge (\neg \neq_{23}) \wedge (\neg \neq_{34}) \quad \wedge \quad (\neq_{13} \vee \neq_{14} \vee \neq_{24})$
$e_7 = (1,3,1,2)$	$\{\neq_{13}\}$	$+$	$\mathsf{K}_{\{e_1^+,e_2^+\}} \wedge (\neg \neq_{12}) \wedge (\neg \neq_{23}) \wedge (\neg \neq_{34}) \wedge (\neg \neq_{13}) \quad \wedge \quad (\neq_{14} \vee \neq_{24})$
$e_8 = (2,1,3,1)$	$\{\neq_{24}\}$	$+$	$\mathsf{K}_{\{e_1^+,e_2^+\}} \wedge (\neg \neq_{12}) \wedge (\neg \neq_{23}) \wedge (\neg \neq_{34}) \wedge (\neg \neq_{13}) \wedge (\neg \neq_{24}) \wedge (\neq_{14})$

TAB. 3.2 – La stratégie de questions optimales pour l'exemple 6.

3 (*i.e.* aucune variable égale), soit un $\kappa(e)_{[\mathsf{K}]}$ de taille 2 (*i.e.* deux variables égales) mais ne pourra jamais avoir un $\kappa(e)_{[\mathsf{K}]}$ de taille 1 à cause de la redondance.

3.3.2 Implémentation

Les stratégies optimiste et optimale se caractérisent toutes deux par le nombre t de contraintes encore possibles dans l'espace des versions qui permettent de rejeter la question q que l'on cherche à générer. Comme nous l'avons illustré à la fin de la section 3.3.1, du fait de la redondance, il arrive qu'il n'existe pas de réseau dans l'espace des versions dont une solution s soit telle que $|\kappa(s)_{[\mathsf{K}]}| = t$. Nous devons donc introduire une tolérance sur le nombre de contraintes qui rejettent une instance.

Nous avons implémenté le problème de génération de questions selon un processus à deux étapes. L'algorithme 1 tente tout d'abord de trouver une interprétation I sur B telle que toute solution s de $\phi(I)$ respecte la condition $t - \epsilon \leq |\kappa(s)_{[\mathsf{K}]}| \leq t + \epsilon$, où ϵ est la tolérance acceptée pour la taille $\kappa(q)_{[\mathsf{K}]}$ de la question q à générer. Le second paramètre de l'algorithme correspond à l'ensemble L de contraintes dans lequel $\kappa(q)_{[\mathsf{K}]}$ doit être inclus. Nous expliciterons dans la suite cette fonctionnalité servant à "guider" notre stratégie de questionnement. Lorsque I a été trouvée, nous cherchons ensuite une solution s à $\phi(I)$, qui correspond alors à la question que nous posons à l'utilisateur.

Nous décrivons tout d'abord l'algorithme et son mode de fonctionnement puis nous présenterons sa complexité avant d'indiquer comment implémenter les stratégies optimiste et optimale en choisissant judicieusement t et ϵ.

Algorithme de génération de questions

L'algorithme 1 génère une question de la manière suivante. Il considère en entrée la taille t que nous souhaitons obtenir, la tolérance ϵ que nous nous accordons et l'ensemble L des contraintes sur lequel nous souhaitons concentrer nos efforts. Notre objectif est de construire

Algorithme 1 : Algorithme de génération de question

Entrées : B le biais d'apprentissage, K la théorie clausale de Conacq, L un ensemble de littéraux, t la taille souhaitée et ϵ la tolérance.

Sorties : Une interprétation I

1 $\mathsf{F} \leftarrow \mathsf{K}$;
2 **pour chaque** $\mathsf{b}_{ij} \in \mathsf{B} \setminus \{\mathsf{b}_{ij} \mid (\neg\mathsf{b}_{ij}) \in \mathsf{K}\}$ **faire**
3 **si** $\mathsf{b}_{ij} \notin \mathsf{L}$ **alors** $\mathsf{F} \leftarrow \mathsf{F} \wedge (\mathsf{b}_{ij})$;
4 **sinon** $\mathsf{F} \leftarrow \mathsf{F} \wedge (\mathsf{b}_{ij} \vee \overline{\mathsf{b}_{ij}})$;
fin
5 $lower \leftarrow max(|\mathsf{L}| - t - \epsilon, 1)$;
6 $upper \leftarrow min(|\mathsf{L}| - t + \epsilon, |\mathsf{L}|)$;
7 $\mathsf{F} \leftarrow \mathsf{F} \wedge atLeast(lower, \mathsf{L}) \wedge atMost(upper, \mathsf{L})$;
8 **si** $Modeles(\mathsf{F}) \neq \emptyset$ **alors** retourner un modèle de F ;
9 **sinon** retourner "inconsistance" ;

une formule F pour laquelle tout modèle I répondra aux critères fournis en entrée. F est ainsi initialisée par K afin de garantir que tout modèle corresponde à un réseau de contraintes appartenant à l'espace des versions (ligne 1). Pour tout littéral b_{ij} non encore nié dans B (ligne 2), si b_{ij} n'appartient pas à L, nous ajoutons la clause (b_{ij}) à F pour s'assurer que la contrainte b_{ij} appartiendra au réseau $\phi(I)$ pour tout modèle de F (ligne 3). De cette manière, toute solution s de $\phi(I)$ ne violera que des contraintes de L ou des contraintes n'appartenant plus à l'espace des versions. À ce stade de l'algorithme, nous sommes assurés d'avoir $\kappa(s)_{[\mathsf{K}]} \subseteq \mathsf{L}$. Il nous reste désormais à forcer la taille de $\kappa(s)_{[\mathsf{K}]}$ afin qu'il respecte l'intervalle voulu. Si b_{ij} appartient à L (ligne 4), la clause $(\mathsf{b}_{ij} \vee \overline{\mathsf{b}_{ij}})$ est ajoutée à F afin de garantir que, soit b_{ij}, soit sa contrainte opposée $\overline{\mathsf{b}_{ij}}$ appartiendra au réseau résultant.[1] $\overline{\mathsf{b}_{ij}}$ est nécessaire car $\neg\mathsf{b}_{ij}$ exprime simplement l'absence de la contrainte b_{ij}, ce qui n'est pas suffisant pour forcer sa violation. Nous ajoutons alors deux contraintes de cardinalité pour forcer le nombre de contraintes violées par toute solution $\phi(I)$ à appartenir à l'intervalle $[t - \epsilon \,..\, t + \epsilon]$. La contrainte *atLeast(lower,L)* impose que au moins *lower* littéraux de L prennent la valeur *VRAI* et la contrainte *atMost(upper,L)* impose que au plus *upper* littéraux de L prennent la valeur *VRAI*. Ainsi, nous forçons au

[1] Une librairie de contraintes ne contient pas forcément l'opposée de chaque contrainte. Il est cependant possible d'exprimer l'opposée d'une contrainte à l'aide d'une conjonction d'autres contraintes. Par exemple, dans la librairie $\{\leq, \neq, \geq\}$, $\overline{\leq}$ n'existe pas mais peut être formulée par la conjonction $(\geq \wedge \neq)$. Si aucune conjonction ne permet d'exprimer l'opposée d'une contrainte, nous pouvons poster une conjonction de contraintes (issues de la librairie) qui approxime la négation de la contrainte considérée, ou rien du tout. Dans ce cas, nous perdons simplement la garantie du nombre de contraintes de L qui rejetteront la question générée.

moins $|\mathsf{L}| - t - \epsilon$ contraintes et au plus $|\mathsf{L}| - t + \epsilon$ contraintes à être satisfaites (lignes 7 et 5). 'min' et 'max' permettent quant à elles d'éviter les cas triviaux (aucune contrainte de L violée) et de rester sous la taille de L. La ligne 8 recherche un modèle de F et le retourne le cas échéant. La redondance peut nous empêcher de générer une question pour une taille $\kappa(q)_{[\mathsf{K}]}$ donnée. Aussi, si la tolérance ϵ est trop petite, F peut être insatisfiable et une inconsistance est retournée (ligne 9).

Propriétés de l'algorithme de génération de questions

La propriété suivante indique lorsque l'algorithme 1 garantit la production d'une question.

Propriété 1 (Satisfiabilité) *Soient* B *un biais d'apprentissage,* K *une théorie clausale construite par* Conacq *et* I *un modèle de* K. *Si* K *contient toutes les règles de redondance sur* B *et si, pour toute contrainte* $\mathcal{C}_{ij} \in \mathsf{B}$, *le biais d'apprentissage* B *contient son opposée* $\overline{\mathcal{C}_{ij}}$ *alors* $\phi(I)$ *est satisfiable (*i.e. *possède des solutions).*

Preuve (par l'absurde). Soit B un biais d'apprentissage tel que pour toute contrainte $\mathcal{C}_{ij} \in \mathsf{B}$, B contienne la contrainte opposée $\overline{\mathcal{C}_{ij}}$, et soient K une théorie clausale construite par Conacq contenant toutes les règles de redondance et I un modèle de K tel que $\phi(I)$ ne soit pas satisfiable, c'est à dire n'admettant pas de solution. Soit alors S un ensemble de contraintes $\mathcal{C}_{ij}$ de $\phi(I)$ tel que S soit insatisfiable et tel que tout sous-ensemble strict de S soit satisfiable. S est un minimal conflict set [Jun01]. La clause $C = (\wedge_{(i,j)|\mathcal{C}_{ij} \in S}\ b_{ij})$ est fausse pour K et sa négation $\neg C = (\vee_{(i,j)|\mathcal{C}_{ij} \in S}\ \neg b_{ij})$ est vraie pour la théorie clausale K. L'ensemble S étant minimal, on en déduit que pour toute contrainte $\mathcal{C}_{ij} \in S$, $((\wedge_{(p,q)\neq(i,j)|\mathcal{C}_{pq} \in S}\ \mathcal{C}_{pq}) \rightarrow \neg\mathcal{C}_{ij})$ est vraie pour K. Pour chaque contrainte $\mathcal{C}_{ij} \in S$, la clause de Horn $((\wedge_{(p,q)\neq(i,j)|\mathcal{C}_{pq} \in S}\ \mathcal{C}_{pq}) \rightarrow \overline{\mathcal{C}_{ij}})$ est alors une règle de redondance sur B. L'une (au moins) de ces règles de redondance n'est pas présente dans K car, dans le cas contraire, I ne pourrait être un modèle pour K. En conséquence, K ne contient pas toutes les règles de redondance, ce qui achève la démonstration. △

Si K ne contient pas toutes les règles de redondance sur B, l'algorithme 1 est susceptible de produire une interprétation I telle que $\phi(I)$ soit inconsistant. Dans ce cas, nous identifions un conflict set S de $\phi(I)$,[2] c'est-à-dire un ensemble de contraintes produisant un réseau inconsistant et nous ajoutons la clause $\bigvee_{\mathsf{b}_{ij} \in S} \neg \mathsf{b}_{ij}$ à K. L'ajout de cette nouvelle règle de redondance nous permet d'éviter de générer à l'avenir de nouvelles interprétations I' dont l'inconsistance de $\phi(I)$ est causée par cette règle.

[2] En utilisant par exemple l'algorithme QuickXplain [Jun01].

Comme l'énonce la propriété 2, la génération de questions respectant les conditions sur le nombre de contraintes violées est NP-difficile.

Propriété 2 *Soient* B *un biais d'apprentissage,* K *une théorie clausale, un ensemble* L *de contraintes, une taille cible* t *et une tolérance* ϵ*, générer une question* q *telle que* $\kappa(q)_{[\mathsf{K}]} \subset \mathsf{L}$ *et* $t - \epsilon \leq |\kappa(q)_{[\mathsf{K}]}| \leq t + \epsilon$ *est un problème NP-difficile.*

Preuve (3-coloriage). Soit (N, E) un graphe tel que N est l'ensemble de ses sommets et E l'ensemble de ses arêtes. Dans cette preuve, nous noterons n_i le i-ième sommet du graphe et e_{ij} l'arête entre les sommets n_i et n_j. Le problème du 3-coloriage de (N, E), qui consiste à déterminer si il existe un coloriage en 3 couleurs du graphe tel que deux sommets adjacents soient coloriés différemment, est un problème *NP*-complet ([GJ79], page 191).

Transformons maintenant ce problème de coloration de graphe en un problème de génération de questions. Nous considérons pour cela un problème d'acquisition de réseau de contraintes tel que $\mathcal{X} = \bigcup_{i|n_i \in N}\{x_i\} \cup \{x_a, x_b\}$ avec $D(x_i) = \{1, 2, 3\}$, $D(x_a) = D(x_b) = \{1\}$ et $\mathsf{B} = \bigcup_{(i,j)|e_{ij}\in E}\{=_{ij}, \neq_{ij}\} \cup \{=_{ab}, \neq_{ab}\}$. Soit $e_1^+ = (1, \ldots, 1)$ la première instance positive fournie comme donnée d'entraînement. Après analyse de e_1^+, la théorie clausale construite par Conacq est $\mathsf{K} = \bigwedge_{(i,j)|e_{ij}\in E}\{\neg \neq_{ij}\} \wedge \{\neg \neq_{ab}\}$. Soient enfin $L = \mathsf{B} \setminus \{b_{ij}|(\neg b_{ij}) \in \mathsf{K}\}$, $t = 1$ et $\epsilon = 0$. Notre transformation est polynomiale en la taille de (N, E) et la résolution du problème de génération de question répond au problème initial de 3-coloriage de graphe.[3] Notre transformation est donc une réduction polynomiale entre le problème du 3-coloriage de graphe et la génération de questions. Nous pouvons ainsi en déduire que le problème de génération de questions est *NP*-difficile. △

La section *Expérimentations* nous permettra de montrer que malgré cette complexité, l'algorithme 1 permet de traiter très efficacement le problème de génération de questions.

L'algorithme permet de déterminer s'il existe une question rejetée par $t \pm \epsilon$ contraintes de l'ensemble L fourni en entrée. La stratégie optimale nécessite $t = 1$ alors que l'optimiste requiert un t plus grand. Pour notre étude, nous avons choisi d'être "à moitié" optimiste en fixant t à $|\mathsf{L}|/2$. Il reste désormais à déterminer quel ensemble L de contraintes va guider la génération et quelle sera la valeur de la tolérance ϵ. ϵ est toujours initialisée à 0. Pour L, nous prenons la plus petite clause positive de K. En effet, une clause positive code le rejet d'un exemple négatif analysé par Conacq. Nous sommes donc assurés qu'au moins une contrainte de L rejette cet exemple. Le fait de choisir la plus petite clause augmente quant à lui nos chances de converger rapidement vers une clause unaire. Si K ne contient toutefois aucune clause non-unaire, nous prenons pour L l'ensemble des littéraux non encore fixés dans K.

[3] Car les domaines de x_a et x_b sont réduits au singleton $\{1\}$.

Lorsque l'algorithme 1 retourne une inconsistance, nous devons trouver un autre ensemble de paramètres pour invoquer l'algorithme à nouveau. t étant fixé par la stratégie, seuls L et ϵ peuvent être modifiés. Si K contient plusieurs clauses non-unaires, nous remplaçons L par la prochaine clause non-unaire de K (ordonné par taille). Si nous avons essayé toutes les clauses non-unaires sans succès, il nous reste à augmenter la valeur de ϵ. Deux options sont possibles. La première, baptisée *plus-proche*, consiste à générer une question sur L avec la plus petite tolérance possible. La seconde, baptisée *approximée*, consiste à générer une question par paliers successifs. Elle tente tout d'abord de trouver un ensemble L pour lequel il existe une question avec $\epsilon = 0.25 \cdot |\mathsf{L}|$. S'il n'en existe pas, elle essaiera avec $0.50 \cdot |\mathsf{L}|$, $0.75 \cdot |\mathsf{L}|$ et enfin $|\mathsf{L}|$.

En procédant ainsi, nous avons donc deux stratégies de génération de questions : optimiste et optimale, combinées aux options plus-proche et approximée. Pour la stratégie optimiste, nous fixons $t = |\mathsf{L}|/2$ tandis que pour l'optimale, nous choisissons $t = 1$. Enfin, l'option plus-proche cherche le plus petit ϵ possible alors que l'option approximée augmente ϵ par paliers de 25%.

3.4 Expérimentations

Dans cette section, nous présentons un panel de résultats expérimentaux permettant de valider et comparer les différentes approches de génération de questions présentées dans ce chapitre. Pour implémenter ces dernières, notre choix s'est porté sur le langage Java, nous permettant ainsi d'utiliser Choco [Cho07] comme résolveur de contraintes et la librairie Sat4J [Sat07] comme résolveur SAT. Notre implémentation exploite par ailleurs autant que possible la redondance, en utilisant conjointement les règles de redondance et la détection du backbone (*c.f.* chapitre 2, section 2.3.3).

Classes de problèmes

Notre panel d'expérimentations combine des problèmes binaires et non-binaires. Pour le binaire, nous avons testé nos différentes stratégies sur des problèmes avec et sans structure. Pour ces expérimentations, nous avons utilisé la librairie de contraintes $L = \{\leq, \geq, \neq\}$, avons dans un premier temps généré aléatoirement des réseaux de contraintes portant sur 14 variables (ayant des domaines uniformes de taille 20), mettant en jeu un nombre spécifié de contraintes de $\{<, \leq, =, \geq, >, \neq\}$ et n'avons retenu que les réseaux solubles. Dans un deuxième temps, nous avons considéré des problèmes dans lesquels nous avons introduit un motif [BCKO05] de manière à estimer l'impact de la redondance dans la génération de questions. Nous créons pour cela dans le réseau cible un chemin constitué d'une même contrainte de $\{<, \leq, =, \geq, >, \neq\}$.

Target Network		*Random*		*Irredundant*		*Optimistic*				*Optimal*			
						approximate		*closest*		*approximate*		*closest*	
$\|X\|$	$\|C\|$	$\#q$	time	$\#q$	time	$\#q$	time	$\#q$	time	$\#q$	time	$\#q$	time
		Random Binary Problem											
14	1	48	<1	36	<1	24	19	**24**	46	106	12	99	57
14	2	118	<1	71	<1	55	87	**50**	204	102	13	97	58
14	4	> 1000	<1	729	<1	101	237	94	573	81	19	**75**	63
14	14	> 1000	<1	> 1000	<1	235	412	219	918	72	23	**58**	67
14	40	> 1000	<1	> 1000	<1	298	1314	273	3048	71	27	**44**	66
		Pattern Binary Problem											
14	14	> 1000	<1	> 1000	<1	220	17	197	34	42	45	**32**	76
		Sudoku 4×4											
16	72	> 1000	<1	> 1000	<1	178	154	168	186	69	31	**57**	82
		Schur's lemma											
6	6	88	<1	27	<1	21	167	**19**	382	24	198	23	432
8	12	298	<1	66	<1	56	274	51	772	46	218	**44**	563

TAB. 3.3 – Comparaison entre les stratégies de génération de questions, testées sur différentes classes de problèmes. Le temps d'exécution est mesuré en millisecondes pour des expérimentations réalisées sur un Pentium IV cadencé à 1.8 GHz. Pour chaque expérimentation, le résultat en gras indique le plus petit nombre nombre de questions nécessaire à la convergence.

Enfin, nous utilisons un Sudoku de taille 4×4 pour lequel le problème d'acquisition consiste à identifier les règles de ce jeu logique à partir d'exemples et de contre-exemples de grilles.

Pour le non-binaire, nous étudions le Schur's lemma, problème numéro 15 de la CSPLIB,[4] pour lequel nous utilisons la librairie de contraintes ternaires {ALLDIFF, ALLEQUAL, NOTALLDIFF, NOTALLEQUAL}.

Résultats expérimentaux

Le tableau 3.3 referme la moyenne des résultats de 100 expérimentations réalisées pour chaque stratégie de génération de questions et testées sur chacune des classes de problèmes considérées. Dans chacun des cas, l'ensemble d'entraînement fourni à CONACQ contient initialement un seul exemple positif. La première colonne du tableau décrit le réseau cible que nous cherchons à identifier et nous reportons dans les colonnes suivantes les résultats obtenus par nos différentes stratégies. La stratégie *Random* correspond à l'approche basique, complètement aléatoire qui peut proposer des questions redondantes. *Irredondant* est la stratégie consistant à produire des questions de manière aléatoire et à présenter à l'utilisateur uniquement celles qui apporteront une information nouvelle (*c.f.* section 3.2). Enfin, *Optimistic* et *Optimal* cor-

[4] Disponible sur `http ://www.csplib.org`.

respondent respectivement aux stratégies optimiste et optimale présentées en section 3.3, sur lesquelles ont été testées les variantes *approximée* et *plus-proche*. Chaque colonne du tableau est divisée en deux parties. La partie gauche indique le nombre de questions nécessaires pour converger sur le réseau cible ; une limite de 1000 questions a été imposée. La partie droite indique quant à elle le temps moyen nécessaire à la génération d'une question.

À l'exception des problèmes aléatoires très peu denses et du Schur's Lemma, l'approche *Random* ne permet pas de converger vers le réseau cible, même avec un grand nombre de questions. La stratégie *Irredundant* est strictement meilleure que *Random* et permet de converger dans la plupart des cas. Cependant, lorsque la densité du réseau cible augmente, *Irredundant* commence à peiner et devient rapidement incapable de converger.

Les résultats de ces expérimentations permettent de montrer que les stratégies *Optimiste* et *Optimale* sont les plus abouties puisqu'elles permettent toujours de converger quel que soit le réseau de contraintes cible. La variante *plus-proche* nécessite un temps moyen de génération de 2 à 5 fois plus important que la variante *approximée*, comme on pouvait s'y attendre. Les stratégies *plus-proche* ont cependant l'avantage de converger en posant jusqu'à 40% moins de questions que les variantes *approximée*. *Optimiste* est la stratégie la plus efficace pour les problèmes peu denses mais *Optimale* devient la meilleure stratégie lorsque le nombre de contraintes augmente dans le réseau cible. *Optimale* génère en effet moins de questions que *Optimiste* et requiert moins de temps de calcul pour générer ses questions. Par ailleurs, il convient de noter que le nombre de questions nécessaires à la stratégie *Optimale* diminue lorsque la densité du réseau cible augmente. Cette diminution s'explique par le fait que les règles de redondance s'appliquent plus fréquemment dans ce cas, ce qui permet d'identifier plus de contraintes. Les performances de la stratégie *Optimiste* sont en revanche moins bonnes lorsque la densité augmente. Ce phénomène s'explique de la manière suivante. Lorsque la densité du réseau cible augmente, le nombre de solutions de celui-ci diminue, la probabilité qu'une question soit classifiée négativement par l'utilisateur, c'est-à-dire *en notre défaveur* (*c.f.* section 3.3.1), augmente par conséquent d'autant.

Les résultats obtenus pour les problèmes aléatoires mettant en jeu des motifs, ainsi que ceux obtenus sur le Sudoku 4×4 montrent que les approches proposées dans ce chapitre nécessitent moins de questions lorsque le problème est structuré, ce qui est souvent le cas dans les problèmes réels.

3.5 Conclusion

Dans son implémentation standard [BCKO05], la plate-forme CONACQ attend passivement les données d'entraînement. La convergence du processus d'apprentissage repose alors sur la

capacité de l'utilisateur à fournir des exemples significatifs de son problème. Produire des données significatives s'avère cependant difficile en contexte réel. En effet, lorsqu'un utilisateur cherche à modéliser en contraintes un problème difficile, il fournit en général des exemples similaires ou sémantiquement identiques pour le biais d'apprentissage et l'analyse qu'en fait CONACQ augmente peu, voire pas, la connaissance de la plate-forme.

L'approche développée dans ce chapitre s'est attachée à développer une version interactive de CONACQ, capable de poser à l'utilisateur des questions dont le but est d'augmenter plus rapidement et de manière conséquente la connaissance de la plate-forme. L'évaluation théorique des différentes stratégies proposées ainsi que les expérimentations réalisées nous ont permis de valider l'intérêt de notre approche.

Une version préliminaire de ces travaux avait donné lieu à une première publication [Pau05]. Sa formalisation ainsi que l'approfondissement des techniques décrites dans ce chapitre sont le fruit d'une collaboration étroite avec Rémi Coletta (LIRMM), Christian Bessière (LIRMM) et Barry O'Sullivan (Université de Cork, Irlande) qui a été publiée à IJCAI'07 [BCOP07].

Chapitre 4

Conclusion

Comme nous l'avons évoqué dans le chapitre 1, modéliser et résoudre un problème réel à l'aide de la programmation par contraintes revet un double enjeu. Il convient d'une part de fournir un modèle sémantiquement adéquat du problème étudié, c'est-à-dire représentant de manière exacte ce dernier. D'autre part, ce modèle doit être défini dans l'optique d'une résolution *efficace*. L'apprentissage de réseau de contraintes, qui vise à **automatiser** le processus de modélisation, ou du moins à assister un utilisateur à modéliser son problème sous la forme d'un CSP, constitue par conséquent un objectif ambitieux. L'apprentissage automatique de réseau de contraintes est une discipline très jeune. Les approches actuelles telles que [FW02], [RS04], [LL05] & [BCKO05] ont néanmoins ouvert de nombreux champs de recherche dans l'apprentissage de contraintes.

Dans la première partie de cette thèse, nous nous sommes intéressés à la plate-forme d'apprentissage automatique de contraintes CONACQ [BCKO05]. Basée sur l'espace des versions, CONACQ est destinée à aider un utilisateur, novice en contraintes, à modéliser son problème sous la forme d'un réseau de contraintes. Pour cela, l'utilisateur fournit à la plate-forme des instances identifiées comme solutions ou non-solutions du problème que l'on cherche à modéliser en contraintes.

Dans son implémentation actuelle, la plate-forme attend passivement ces données d'entraînement. L'efficacité de l'apprentissage de CONACQ est alors basée sur la capacité de l'utilisateur à fournir des instances significatives de son problème. Pour pallier cet écueil, nous avons proposé dans le chapitre 3 une approche dans laquelle le système est actif vis-à-vis de la sélection des exemples utilisés pour l'apprentissage. Les techniques d'interaction que nous avons développées offrent à CONACQ la possibilité de poser des questions à l'utilisateur, plutôt que d'attendre passivement les données d'entraînement.

Nous avons défini trois stratégies de génération de questions. La première, baptisée *ir-*

redondante, est une technique simple (polynomiale en temps) qui empêche de proposer des questions redondantes à l'utilisateur. Cette stratégie garantit que la classification de chaque question posée à l'utilisateur ajoute de nouvelles informations à CONACQ. Cependant, nous avons montré que le gain de connaissance des questions non-redondantes varie selon la question générée. Afin de minimiser le nombre d'interactions avec l'utilisateur (*i.e.* le nombre de questions posées), nous avons proposé deux autres stratégies : La stratégie *optimiste* et la stratégie *optimale* (au sens de Mitchell [Mit97]). Une question optimale est une question qui fournit à CONACQ un large gain de connaissance si elle est classifiée positive par l'utilisateur, mais apporte peu d'informations dans le cas contraire. Une question est optimale lorsque sa classification, quelle qu'elle soit, permet de diviser par deux la taille de l'espace des versions de CONACQ. Empiriquement, la stratégie optimale s'est avérée supérieure à toutes les autres sur tous les réseaux de contraintes que nous avons étudiés, exceptés les plus petits réseaux aléatoires.

D'un point de vue général, les stratégies de génération de questions que nous avons développées dans la première partie de cette thèse ont permis d'améliorer sensiblement les performances de CONACQ.

Deuxième partie

Une approche programmation par contraintes pour la constitution automatique de comportements sensorimoteurs en robotique

Chapitre 5

Introduction

Dans la seconde partie de cette thèse, nous nous intéressons à l'utilisation de l'apprentissage automatique de réseau de contraintes dans le domaine de la robotique. L'approche que nous proposons consiste à utiliser la plate-forme CONACQ étudiée précédemment pour modéliser automatiquement, sous la forme de réseaux de contraintes, les actions élémentaires d'un robot, puis à combiner par planification les réseaux de contraintes acquis pour définir automatiquement des comportements plus évolués. L'approche que nous proposons dans cette seconde partie vise à développer, à terme, une architecture logicielle pour la constitution automatique de comportements sensorimoteurs en robotique.

Contexte

La robotique a massivement investi le secteur industriel durant la seconde moitié du siècle dernier. Les robots de production ont ainsi peu à peu remplacé l'homme dans l'exécution de tâches répétitives, pénibles ou dangereuses (soudure, peinture, assemblage de grosses pièces, *etc.*). Évoluant en environnement clos et destinés à exécuter automatiquement des tâches prédéfinies et répétitives, ces robots sont des systèmes incroyablement efficaces mais peu flexibles et peu adaptatifs. Depuis quelques décennies, une partie des recherches en robotique s'oriente vers la conception de machines polyvalentes, capables de réaliser des tâches complexes dans des environnements très variés. Par ailleurs, la robotique actuelle met désormais en jeu un grand nombre de disciplines, telles que la conception mécanique, la théorie du contrôle, le traitement du signal, l'acquisition de données, l'interaction Homme-Machine ou bien encore l'Intelligence Artificielle (planification, apprentissage, *etc.*). Dans ce contexte, le défi de *robots de plus en plus autonomes* ne consiste alors plus à produire de "simples" automates, mais à concevoir, à partir des technologies matérielles et logicielles développées au sein de ces disciplines, des machines *intelligentes* capables de réaliser des missions complexes.

Différentes architectures logicielles de contrôle, telles que l'architecture LAAS [ACF+98], CLARATY [VNE+01] et HARPIC [LD03], ont été proposées pour conférer de l'autonomie à des robots. Ces architectures de contrôle sont utilisées avec succès pour la supervision de robots mobiles (LAAS, HARPIC) et dans le cadre de l'exploration planétaire (CLARATY). Ces architectures de contrôle sont généralement composées de différentes couches (également appelées niveaux). À titre d'exemple, l'architecture LAAS est composée de trois couches : une couche fonctionnelle, une couche de contrôle d'exécution et une couche décisionnelle. La couche fonctionnelle offre un panel de modules temps-réel, qui implémentent les fonctions de base du système. Parmi cet ensemble de modules, on trouve notamment des modules de perception et des modules d'exécution de fonctions sensorimotrices. Les modules de perception encapsulent des procédures de traitement des données issues des capteurs du robots. Les modules d'exécution de fonctions sensorimotrices, que nous nommerons dans la suite *comportements sensorimoteurs*, encapsulent quant à eux des boucles de contrôle sur les actionneurs du robot permettant à ce dernier de se déplacer et d'interagir avec son environnement. Les modules de la couche fonctionnelle sont activés par la couche de contrôle de l'architecture. Cette dernière contrôle et coordonne les requêtes envoyées aux modules en fonction des plans établis par la couche décisionnelle. La couche décisionnelle est la couche la plus haute de l'architecture LAAS. Elle est en charge de la planification de la mission globale du robot et produit pour cela les séquences de déclenchement des modules du niveau fonctionnel. La couche décisionnelle doit par ailleurs être capable de réagir aux événements[1] qui peuvent se produire durant l'exécution de ces séquences, et d'adapter les séquences produites afin de mener à bien la mission globale du robot.

Dans la seconde partie de ce manuscrit, nous focalisons notre attention sur la conception des comportements sensorimoteurs. Bien qu'ils soient considérés par les architectures logicielles de contrôle comme les capacités *de base* d'action du robot, les comportements sensorimoteurs ne sont cependant pas basiques, mais correspondent à la combinaison d'actions élémentaires câblées exécutables par le robot. À titre d'exemple, comme le mentionne [Luz01] (p. 133) pour l'utilisation de l'architecture HARPIC dans le cadre du parking d'un robot mobile, l'entrée dans une place de parking et la manœuvre à l'intérieur de cette place constituent deux comportements sensorimoteurs distincts ; se réorienter ou rejoindre une valeur particulière en abscisse ou en ordonnée constituent quant à elles deux actions élémentaires exécutables par le robot. Dans le cas de robots humanoïdes tels que ASIMO (Honda) ou HRP-2 (Kawada), les actions élémentaires peuvent être par exemple "*Avancer d'un pas*", "*Fermer la main*",

[1] Tels que la mauvaise exécution d'un comportement sensorimoteur, la présence de perturbations extérieures, une modification du contexte (par exemple la détection d'un nouvel élément), *etc.*.

"*Tourner la tête*" ou "*Tendre le bras*" ; La saisie d'un objet situé à une certaine distance du robot constituant un comportement sensorimoteur.

Chaque fois qu'on souhaite superviser un nouveau robot à l'aide d'une architecture logicielle de contrôle, il est nécessaire de combiner les actions élémentaires du robot afin de constituer un panel de comportements sensorimoteurs. À l'heure actuelle, la définition des comportements sensorimoteurs est réalisée de manière *ad-hoc* par les automaticiens. Chaque action élémentaire exécutable par le robot est tout d'abord modélisée à l'aide de systèmes d'équations mathématiques mettant en jeu les lois physiques du monde (loi de conservation des énergies, moments cinétiques, *etc.*). Ensuite, il convient de combiner des actions élémentaires, et par conséquent les systèmes d'équations qui les modélisent, dans l'optique de définir un comportement sensorimoteur. Ainsi, la définition des comportements sensorimoteurs d'un robot constitue une difficulté de taille, qui nécessite un travail de modélisation ardu (parfois fastidieux), ainsi que de nombreux calculs, pouvant aller jusqu'à plusieurs heures. En robotique humanoïde, la constitution de comportements sensorimoteurs est même d'ailleurs, pour l'heure, difficilement envisageable. En effet, la complexité mécanique (une trentaine de degrés de liberté) et la réalité opérationnelle des robots humanoïdes (contrainte d'équilibre, fréquence élevée, nécessité de sécuriser le robot au maximum) constituent, à l'heure actuelle, un rempart rédhibitoire à la constitution de comportements sensorimoteurs exploitables par des architectures de contrôle.[2]

Contribution

Dans la seconde partie de cette thèse, nous nous intéressons à une utilisation pratique de l'apprentissage automatique de réseau de contraintes qui vise à automatiser le processus de définition de comportements sensorimoteurs en robotique.

L'architecture logicielle que nous proposons utilise le paradigme de la programmation par contraintes pour modéliser, planifier et superviser l'exécution de comportements sensorimoteurs. Elle utilise la plate-forme d'acquisition automatique CONACQ, présentée dans la première partie de ce manuscrit, pour abstraire *automatiquement* les actions élémentaires d'un robot sous la forme de réseaux de contraintes. Notre architecture utilise par ailleurs un planificateur de tâches inspiré de CSP-PLAN [LB03] pour combiner les réseaux de contraintes acquis et ainsi définir *automatiquement* des comportements sensorimoteurs. La modélisation sous forme de réseaux de contraintes possède un double avantage. D'une part, elle permet

[2] Les comportements "évolués" (marche, préhension d'objet) actuellement proposés par les humanoïdes ne sont en fait que des *démonstrations* parfaitement sécurisées (afin de limiter les risques de chute dont le coût financier peut être très important), difficilement adaptables aux modifications de contexte et trop peu robustes aux perturbations pour constituer des comportements sensorimoteurs exploitables en pratique.

d'obtenir un *modèle* qui abstrait un espace de solutions possibles pour chaque action élémentaire. D'autre part, les réseaux de contraintes acquis héritent des propriétés de propagation (*c.f.* chapitre 1) qui sont pleinement exploitées durant la planification et l'exécution d'un comportement sensorimoteur.

Nous tenons dès à présent à mettre en lumière deux aspects importants de l'étude que nous proposons dans cette seconde partie. D'une part, nous limitons **strictement** notre approche à la constitution de comportements sensorimoteurs.[3] D'autre part, notre démarche principale concerne une utilisation pratique de l'apprentissage automatique de réseau de contraintes, destinée à **abstraire automatiquement** les actions élémentaires d'un robot sous la forme de CSPs. Par cette approche très prospective, nous cherchons à étudier s'il est possible d'envisager, à long terme, une automatisation du processus de constitution des comportements sensorimoteurs, qui constitue à l'heure actuelle une difficulté de taille.

Plan

La deuxième partie de cette thèse s'articule de la manière suivante. Le chapitre 6 présente les principes d'organisation des architectures logicielles de contrôle proposées en robotique autonome. Dans le chapitre 7, nous présentons formellement l'architecture que nous proposons. Nous validons ensuite, au travers d'une série d'expérimentations (chapitre 8), la capacité de notre architecture à planifier et à superviser automatiquement l'exécution de comportements sensorimoteurs. Enfin, nous concluons notre propos et présentons quelques améliorations futures de notre approche dans le chapitre 9.

[3] Pour s'attaquer à des problèmes de planification de plus haut niveau, on préfèrera les architectures logicielles de contrôle évoquées plus haut qui développent des mécanismes spécifiquement dédiés à ce type de problèmes (comme nous le verrons dans le chapitre 6).

Chapitre 6

Architectures logicielles de contrôle : État de l'art

Depuis quelques décennies, une partie des recherches en robotique s'intéresse à la conception de robots de plus en plus autonomes. Dans cette optique, différentes architectures logicielles de contrôle ont été proposées. L'objectif de ces architectures consiste à développer des systèmes de décision permettant à un robot d'évaluer à tout instant, son état et l'état de son environnement, de les confronter avec la mission qui lui a été confiée et de prendre des décisions cohérentes dans le but d'accomplir cette mission.

Ce chapitre présente les principes d'organisation des trois grandes familles d'architectures logicielles de contrôle : les architectures délibératives (section 6.1), les architectures comportementales (section 6.2) et les architectures hybrides (section 6.3), nous permettant ainsi de situer nos travaux (section 6.4). Pour une présentation plus approfondie des architectures logicielles de contrôle, le lecteur se reportera à [Ark98] qui présente de manière détaillée ces différentes approches.

6.1 Les architectures délibératives

Le modèle d'organisation des architectures **délibératives** (*c.f.* figure 6.1), également appelées architectures *hiérarchisées*, centre la conception sur le système décisionnel. Ces architectures sont organisées, dans la plupart des propositions, en plusieurs couches (ou niveaux) hiérarchisées [ALFW97, Gat98, Alb99]. Une couche communique uniquement avec la couche directement inférieure et la couche directement supérieure. Ces architectures comportent typiquement trois couches :

– La couche *fonctionnelle* contient des modules de perception qui sont en charge de la

transformation des données numériques, provenant des capteurs, en données symboliques. Elle contient également des modules de génération de trajectoires et des modules d'asservissement, qui sont responsables de la transformation des données provenant du niveau supérieur, en données numériques applicables aux actionneurs.

- La couche *exécutive* est en charge de la supervision des tâches du robot, telles que se diriger vers la source de chaleur la plus proche. Elle contient pour cela des modules qui dirigent l'exécution des modules de la couche fonctionnelle.
- La couche *décisionnelle* est en charge des mécanismes de planification (*i.e.* création et supervision de plans). Elle gère la manière dont le robot doit enchaîner différentes tâches afin d'accomplir chaque objectif de sa mission. Les mécanismes de décision de plus haut niveau, appelés traditionellement mécanismes de délibération sont les mécanismes centraux dans ces architectures [TDK94] et sont basés sur des techniques d'intelligence artificielle.

L'information provenant des capteurs du robot est propagée verticalement, de la couche fonctionnelle vers la couche décisionnelle, en traversant potentiellement toutes les couches intermédiaires. L'information est transformée, au fur et à mesure des traitements réalisés dans chaque couche, en une information de plus en plus abstraite. Ceci nécessite des traitements adéquats afin de fusionner les données et d'en extraire les informations plus abstraites (nécessaires au plus haut niveau). Inversement, la propagation de la décision (sous forme de données symboliques) se fait de la couche décisionnelle vers la couche fonctionnelle, en traversant successivement toutes les couches intermédiaires. La couche fonctionnelle applique alors, en fonction de cette décision, les asservissements adéquats en activant les modules d'asservissement correspondants.

Les architectures délibératives sont historiquement les premières à avoir été proposées [Nil80]. Comme le souligne [Luz01], leur modèle d'organisation implique une modélisation analytique, "*où les fonctionnalités de haut niveau que doit remplir un système sont successivement décomposées en fonctionnalités de plus bas niveau jusqu'à un niveau jugé suffisant pour leur résolution.*" Les architectures délibératives proposent en ce sens une approche intuitive et élégante (*i.e.* séparation en couches hiérarchisées, chacune mettant en jeu des mécanismes propres de décision/contrôle).

Le principal problème des architectures délibératives vient cependant de leur manque de réactivité, c'est-à-dire la capacité de réagir en temps voulu à des modifications de l'environnement. Ce manque de réactivité résulte de l'organisation en couches ; le transfert de l'information devant obligatoirement se faire en traversant toutes les couches intermédiaires. Les traitements réalisés dans les couches intermédiaires induisent ainsi des latences d'autant plus

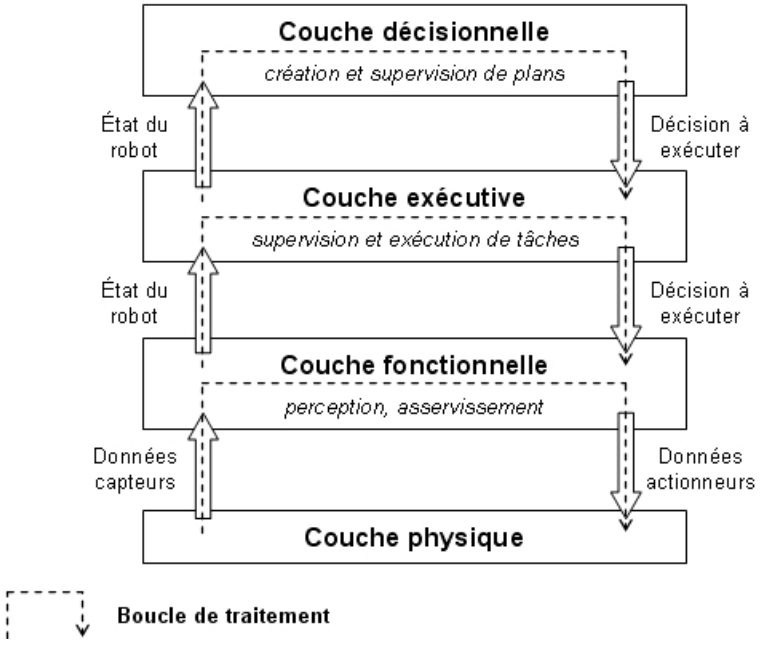

FIG. 6.1 – Principe d'organisation des architectures délibératives.

importantes que le nombre de couches (et donc de traitements) est grand. Il peut ainsi arriver que le robot soit amener à "s'arrêter" en attendant que le processus de planification se termine. D'un point de vue général, il convient donc de noter que l'approche hiérarchisée manque globalement de réactivité. Par ailleurs, déployer une architecture délibérative efficace se révèle difficile lorsque l'on complexifie les problèmes abordés (*i.e.* environnements dynamiques, ouverts, *etc.*).

6.2 Les architectures comportementales

Sur le modèle de la « subsumption architecture » proposée en 1986 par Rodney Brooks [Bro86], les architectures **comportementales** (*c.f.* figure 6.2), aussi appelées architectures réactives, sont tournées vers l'utilisation des réactions réflexes du robot. Ces architectures sont constituées d'un ensemble de modules qui implémentent chacun un comportement du robot, c'est-à-dire une fonctionnalité élémentaire du robot associant à chaque vecteur d'entrée (*i.e.* un ensemble de valeurs capteurs) un vecteur de sortie appliqué aux actionneurs. En ce sens, ces comportements sont dits "réactifs" dans la mesure où ils génèrent immédiatement une réaction au robot dès qu'un ensemble de stimuli se présente en entrée.

Issues de l'observation des comportements des animaux, les architectures comportementales [Bro86, Bro89, Ros97] sont construites selon l'idée qu'un comportement complexe et évolué d'un robot peut émerger de la composition simultanée de plusieurs comportements réactifs simples. Chaque comportement réactif étant défini de manière indépendante, le problème majeur de conception de ces architectures est lié à la manière de composer ces comportements réactifs, dans la mesure où ces derniers peuvent générer, au même moment, des commandes contradictoires. Afin de résoudre ce problème, les architectures comportementales intègrent un *module d'arbitrage* qui se charge de pondérer les sorties de chacun des comportements réactifs et ainsi générer une sortie unique. L'enjeu principal de l'approche comportementale consiste alors à trouver la bonne règle d'arbitrage, permettant au robot d'adopter un comportement global cohérent en fonction de la tâche qu'il a à accomplir, ce qui s'avère souvent particulièrement compliqué en pratique.

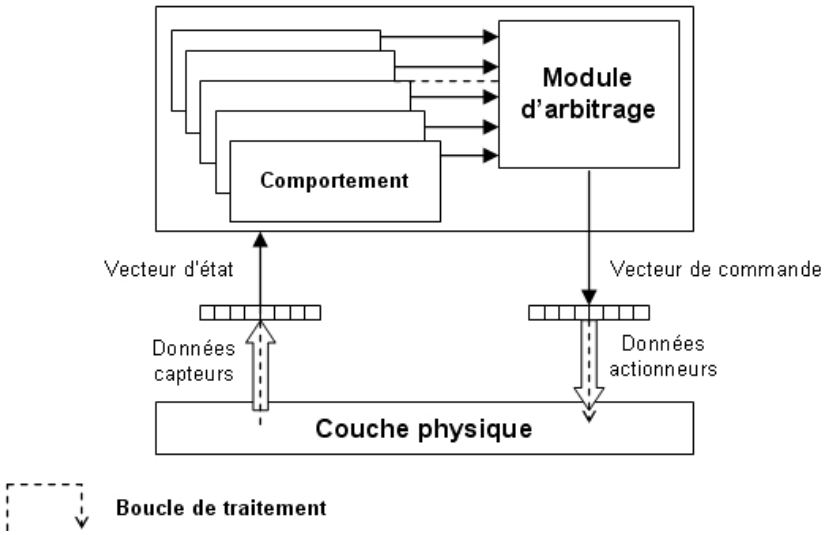

FIG. 6.2 – Principe d'organisation des architectures comportementales.

Plusieurs expériences ont démontré l'efficacité de ces architectures en robotique mobile, notamment en milieu encombré (suivi de route, évitement d'obstacles, *etc.*). Cependant la complexité de conception de ces architectures ne permet pas de les utiliser pour des applications demandant un comportement plus évolué que la navigation. L'objectif global du robot étant implémenté sous forme d'un réseau de modules, il n'est pas possible d'enchaîner plusieurs objectifs ou de changer facilement la mission initiale.

D'un point de vue général, il convient de noter que l'approche comportementale se heurte

au problème dual de l'approche hiérarchisée. En effet, par opposition aux architectures hiérarchisées où il est difficile de décomposer une mission complexe en composantes primitives, il est généralement difficile de remonter depuis les comportements réactifs vers une mission complexe. Les architectures comportementales ont ainsi de la difficulté à réaliser des tâches structurées et ne confèrent, en ce sens que peu d'autonomie à un robot.

6.3 Les architectures hybrides

Comme nous venons de le souligner au travers des deux sections précédentes, les architectures délibératives et comportementales sont le résultat de deux approches "diamétralement" opposées. Elles présentent en ce sens des avantages et des défauts respectifs complémentaires. Les architectures délibératives ont ainsi été développées dans l'optique de doter un robot de capacités décisionnelles importantes, via des mécanismes de planification haut niveau. Ces architectures manquent cependant de réactivité et s'adaptent mal à des modifications de contexte. *A contrario*, les architectures comportementales sont centrées sur la réactivité du système, mais permettent difficilement la réalisation de missions complexes, qui ne peuvent par ailleurs être envisagées sans une planification haut-niveau.

Ce double constat a amené la définition des architectures **hybrides**, qui cherchent à prendre en compte à la fois les aspects décisionnels et réactifs, tentant ainsi de combiner les avantages des architectures délibératives et comportementales. Les architectures hybrides intègrent ainsi une hiérarchisation des couches permettant la symbolisation, et donc la prise de décision (*i.e.* planification haut-niveau, gestion de mission, *etc.*), à laquelle s'ajoutent des boucles réactives imbriquées permettant à chaque couche de fournir des réactions adaptées à sa dynamique et à la réalité opérationnelle du robot. On distingue ainsi deux entités principales dans les architectures hybrides :

1. Un panel de comportements (*i.e.* les modules temps-réel de perception et d'exécution de fonctions sensorimotrices),
2. Un mécanisme explicite ou implicite de choix du ou des comportements actifs.

D'un point de vue général, l'approche hybride cherche, par sa définition, à développer une gestion efficace de la cohésion entre un monde "continu" (*i.e.* le monde physique perçu et contrôlé) et un monde discret (*i.e.* la représentation du monde au sein des couches de décision, nécessaire aux mécanismes de planification) dans l'optique de conférer un maximum d'autonomie à un robot. L'approche hybride semble à ce titre la plus appropriée au défi que constitue la robotique autonome.

Enfin, il convient de noter que de nombreux travaux académiques et industriels actuels portent sur le développement d'architectures hybrides. Ainsi, sans être exhaustifs, nous pouvons citer l'architecture LAAS [ACF+98], développée par le Laboratoire d'Analyse et d'Architecture des Systèmes de Toulouse, CLARATY (Coupled Layer ARchitecture for Robotic Autonomy) [VNE+01], proposée conjoitement par le NASA JET PROPULSION LABORATORY et l'université de Cannergie Mellon de Pittsburg, et HARPIC [LD03], développée au sein de la Délégation Générale pour l'Armement (DGA). Ces architectures hybrides sont utilisées avec succès pour la supervision de robots mobiles (LAAS, HARPIC) et dans le cadre de l'exploration planétaire (CLARATY).

6.4 Problématique de travail

Comme nous l'avons mentionné précédemment, les architectures hybrides disposent d'un panel de modules temps-réel parmi lesquels on trouve des modules de perception et des modules d'exécution de fonctions sensorimotrices. Ces derniers, que nous nommons dans la suite *comportements sensorimoteurs*, encapsulent des boucles de contrôle sur les actionneurs du robot permettant à ce dernier de se déplacer et d'interagir avec son environnement. En ce sens, ils sont considérés par les architectures de contrôle comme les capacités *de base* d'action du robot. Ils ne sont cependant pas basiques, mais correspondent à la combinaison d'actions élémentaires câblées exécutables par le robot.

À l'heure actuelle, la définition des comportements sensorimoteurs est réalisée de manière *ad-hoc* par les automaticiens. Chaque action élémentaire exécutable par le robot est tout d'abord modélisée à l'aide de systèmes d'équations mathématiques mettant en jeu les lois physiques du monde (loi de conservation des énergies, moments cinétiques, *etc.*). Ensuite, il convient de combiner des actions élémentaires, et par conséquent les systèmes d'équations qui les modélisent, dans l'optique de définir un comportement sensorimoteur. En ce sens, la définition des comportements sensorimoteurs d'un robot constitue une difficulté de taille.[1]

Afin de prendre en compte cette problématique, nous proposons, dans la deuxième partie de cette thèse, une approche utilisant l'acquisition automatique de réseaux de contraintes, dont l'objectif consiste à automatiser le processus de constitution de comportements sensorimoteurs.

L'approche que nous proposons est exclusivement dédiée à la constitution de comporte-

[1] Par ailleurs, il convient de noter que la modélisation précise des actions élémentaires, outre le fait de constituer un travail fastidieux, peut être extrêmement (trop) coûteux en terme de puissance de calcul nécessaire à une exécution temps-réel, bridant du même coup la réactivité du robot.

ments sensorimoteurs. Notre approche utilise l'acquisition automatique de réseau de contraintes pour modéliser automatiquement chaque action élémentaire d'un robot sous la forme d'un réseau de contraintes, puis combine par planification les réseaux de contraintes acquis pour constituer un comportement sensorimoteur.

Afin d'éviter toute confusion, nous nous référons à [Luz01] pour distinguer la notion d'action élémentaire de la notion de comportement sensorimoteur, "*ce dernier intégrant une vision téléologique plus explicite*". Ainsi, tout au long de nos travaux, une action élémentaire correspond à la composition d'un ensemble minimal de commandes aux actionneurs permettant une *modification explicite* de l'état du robot. Un comportement sensorimoteur correspond quant à lui à une composition d'actions élémentaires permettant au robot d'*accomplir un but*.

6.5 Conclusion

Pour faire face à la complexité croissante des missions que l'on souhaite voir réaliser par des robots, les architectures délibératives (centrées sur les capacités décisionnelles) et les architectures comportementales (centrées sur la réactivité du système) ont tout à tour été proposées, ayant des défauts et des avantages respectifs complémentaires. Aussi, les recherches s'orientent désormais de plus en plus vers les architectures hybrides, qui cherchent à combiner les avantages des architectures précédemment citées.

La constitution des comportements sensorimoteurs manipulés par les architectures hybrides constitue cependant une difficulté de taille à l'heure actuelle. Réalisée de manière *ad-hoc* par les automaticiens à partir des actions élémentaires d'un robot, la définition des comportements sensorimoteurs nécessite un travail de modélisation ardu (parfois fastidieux), ainsi que de nombreux calculs, pouvant aller jusqu'à plusieurs heures. Afin de prendre en compte cette problématique, la deuxième partie de cette thèse propose une approche utilisant l'acquisition automatique de réseaux de contraintes visant à automatiser le processus de constitution de comportements sensorimoteurs. Au travers de nos travaux, nous chercherons donc :

1. À démontrer la capacité de l'acquisition automatique de réseau de contraintes pour abstraire automatiquement les actions élémentaires sous la forme de réseaux de contraintes,
2. À démontrer qu'il est possible de constituer, par planification, un comportement sensorimoteur à partir des CSPs acquis,
3. Enfin, nous nous attacherons à développer une approche performante en termes de temps de calcul (dans l'optique d'une utilisation temps-réel de notre approche), mais aussi robuste aux incertitudes inhérentes à toute application robotique, telles que le mauvais

fonctionnement d'un actionneur, la saturation d'un capteur ou la présence de perturbations extérieures.

Chapitre 7

Architecture proposée

Dans ce chapitre, nous présentons l'approche logicielle que nous proposons pour planifier et superviser automatiquement des comportements sensorimoteurs en robotique. Nous commençons par exposer l'architecture générale de notre système de supervision (section 7.1). La section 7.2 nous permet d'exposer les motivations de notre approche, que nous définissons ensuite formellement dans la section 7.3 avant de conclure le chapitre.

7.1 Architecture générale du système de supervision

L'approche que nous proposons consiste à modéliser les actions élémentaires d'un robot sous la forme de réseaux de contraintes, puis à combiner ces réseaux de contraintes pour planifier et superviser l'exécution de comportements sensorimoteurs.

Nous utilisons la plate-forme CONACQ étudiée dans la première partie de cette thèse pour modéliser *automatiquement* par apprentissage les actions élémentaires d'un robot. Pour chaque action élémentaire a_i, il suffit de fournir à CONACQ un ensemble d'instances valides de a_i et un ensemble d'instances ne correspondant pas à a_i. CONACQ modélise alors automatiquement a_i sous la forme d'un réseau de contraintes qui modélise les conditions dans lesquelles a_i peut être exécutée (*i.e.* ses pré-conditions), ses effets, ainsi que la manière dont les différents actionneurs du robot doivent réagir pour effectuer a_i. Les réseaux de contraintes ainsi acquis sont alors combinés par planification afin de constituer des comportements sensorimoteurs.

La figure 7.1 illustre le principe de fonctionnement de notre système de supervision. Étant donné un comportement sensorimoteur, notre architecture exprime ce comportement comme une tâche T. Un planificateur de tâches inspiré de CSP-PLAN [LB03] détermine alors une

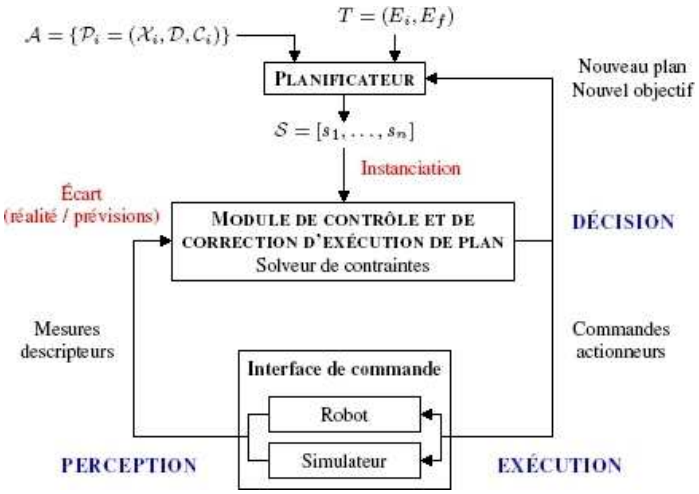

FIG. 7.1 – Contrôle et correction de l'exécution d'une séquence d'actions.

séquence $\mathcal{S}$ d'actions élémentaires à exécuter pour accomplir T.

Une fois calculée, la séquence $\mathcal{S}$ est transmise au module de contrôle qui se charge de faire exécuter cette séquence d'actions au robot ou à son simulateur. Cette exécution se fait alors pas à pas, c'est-à-dire une action après l'autre. Après l'exécution de chaque action élémentaire, le module de contrôle confronte les mesures-capteurs réelles aux prévisions établies via la modélisation en contraintes. Si l'état courant du robot correspond parfaitement aux prévisions, le module de contrôle poursuit l'exécution de la séquence telle qu'elle a été définie initialement. Si l'état courant diffère de la prévision,[1] le module de contrôle détermine par propagation s'il est possible d'ajuster les prochaines actions élémentaires afin d'atteindre l'objectif fixé sans remettre en cause la séquence d'actions. Si de tels ajustements existent, le module de contrôle impose ces corrections *mineures* au robot pour corriger l'exécution de la séquence d'actions. S'il est en revanche impossible d'ajuster la séquence d'actions courante sans la redéfinir, parce que l'écart entre l'état courant et les prévisions est trop grand ou parce que les actions qui restent à exécuter ne le permettent pas, le module de contrôle fait à nouveau appel au planificateur de tâches pour établir une nouvelle séquence d'actions élémentaires $\mathcal{S}'$ permettant d'atteindre l'objectif fixé depuis le nouvel état courant. Le module de contrôle supervise alors l'exécution pas à pas de $\mathcal{S}'$ en contrôlant et corrigeant les éventuels écarts entre la réalité et les prévisions établies par la modélisation en contraintes.

[1] Parce que l'un des actionneurs n'a pas fonctionné correctement, parce qu'une perturbation extérieure a modifié l'exécution de l'action élémentaire précédente, ou bien parce que le réseau de contraintes qui modélisait l'action exécutée n'était pas tout à fait correct.

Il pourra être remarqué que l'architecture que nous proposons s'inspire à la fois des architectures logicielles de contrôle hiérarchisées, dans la mesure où elle met en place un processus de planification, et des architectures logicielles réactives, puisqu'elle cherche à composer des actions élémentaires pour constituer des comportements plus évolués (les comportements sensorimoteurs). Cependant, bien que notre architecture utilise un planificateur de tâches et une boucle de contrôle et de correction d'exécution d'actions, il convient de noter que notre approche se limite à la constitution et à la supervision de comportements sensorimoteurs. Pour s'attaquer à la supervision de missions de plus haut niveau, on préfèrera les architectures logicielles de contrôle présentées dans le chapitre 6, qui mettent notamment en œuvre des méthodes de planification temporelle spécifiquement dédiées à ce type de supervision.[2] À terme, notre objectif vise à ce que ces architectures de contrôle *haut niveau* puissent exécuter des comportements sensorimoteurs automatiquement générés et supervisés par notre architecture logicielle, à la place des comportements sensorimoteurs actuellement définis de manière *ad-hoc* par les automaticiens.

7.2 Pourquoi une approche programmation par contraintes ?

L'utilisation de la programmation par contraintes en robotique a déjà fait l'objet de nombreuses recherches. La programmation par contraintes a ainsi été employée avec succès pour la vérification de trajectoires de robots parallèles [Mer01], le contrôle de robots reconfigurables [ZFCS02] ou la localisation de robots sous-marins [Jau06]. De plus, de nombreux travaux ont porté sur l'utilisation de la programmation par contraintes pour la résolution de problèmes de planification de tâches [SFJ00, NFF+05, BVC07]. Dans ces approches, la programmation par contraintes est utilisée comme un support de calculs pour résoudre efficacement des problèmes combinatoires d'envergure. Tout comme elles, notre architecture de supervision cherche à exploiter pleinement la puissance de résolution de la programmation par contraintes.

Notre architecture modélise par ailleurs les actions élémentaires d'un robot à l'aide de CSPs discrets. L'utilisation de réseaux de contraintes discrets peut paraître un choix audacieux pour s'attaquer à des problèmes mettant en jeu des commandes bas niveau. Cependant, les résultats expérimentaux présentés dans le chapitre 8 démontrent la capacité de notre approche pour planifier efficacement des comportements sensorimoteurs. D'autre part, les réseaux de contraintes manipulés par notre architecture de supervision héritent des propriétés de *propagation* des CSPs discrets (*c.f.* chapitre 1), qui sont pleinement exploitées durant l'exécution d'un comportement sensorimoteur afin d'ajuster ce dernier à la réalité opérationnelle du robot.

[2] L'architecture LAAS utilise ainsi le planificateur IxTeT [GL94].

Enfin, nous utilisons les techniques d'apprentissage automatique de réseau de contraintes afin d'automatiser le processus de modélisation des actions élémentaires, ainsi que la conception des comportements sensorimoteurs, qui constitue à l'heure actuelle une tâche particulièrement ardue.

7.3 Formalisation du processus de supervision

Dans cette section, nous définissons de manière formelle le processus de supervision de notre architecture. Nous considérons pour cela un robot R constitué de p descripteurs (ses capteurs), qui décrivent son état courant, et de q actionneurs (ses moteurs). Soient $\Delta = \{\delta_1, ..., \delta_p\}$ l'ensemble des descripteurs de R, $\alpha = \{\alpha_1, ..., \alpha_q\}$ l'ensemble de ses actionneurs et $\mathcal{A} = \{a_1, \ldots, a_m\}$ l'ensemble des actions élémentaires qu'il peut exécuter.

7.3.1 Modélisation CSP des actions élémentaires

Comme nous l'avons brièvement décrit dans la section 7.1, notre approche consiste à modéliser chaque action élémentaire $a_i \in \mathcal{A}$ à l'aide d'un réseau de contraintes. Pour cela, nous définissons les fonctions $varI$, $varA$ et $varF$ définies sur $\mathcal{A}$ de la manière suivante : $varI(a_i)$ renvoie l'ensemble $\{\delta_1^I, \ldots, \delta_p^I\}$ des variables qui modélisent les descripteurs de R **avant** l'exécution de a_i, $varA(a_i)$ renvoie l'ensemble $\{\alpha_1, \ldots, \alpha_q\}$ des variables qui modélisent les actionneurs de R **pendant** l'exécution de a_i et $varF(a_i)$ renvoie l'ensemble $\{\delta_1^F, \ldots, \delta_p^F\}$ des variables qui modélisent les descripteurs de R **après** l'exécution de a_i.[3]

Chaque action élémentaire $a_i \in \mathcal{A}$ est alors modélisée en suivant la définition 12.

Définition 12 (Modélisation par contraintes) *Une action élémentaire $a_i \in \mathcal{A}$ est modélisée par le réseau de contraintes $\mathcal{P}_i = (X_i, D, C_i)$ tel que :*

$$\begin{cases} X_i = varI(a_i) \cup varA(a_i) \cup varF(a_i), \\ D = \{D(x_j) | x_j \in X_i \text{ et } D(x_j) \text{ est le domaine fini des valeurs possibles pour } x_j\}, \\ C_i = Pre(a_i) \cup Actuators(a_i) \cup Post(a_i). \end{cases}$$

où $Pre(a_i)$ est l'ensemble des contraintes qui modélisent les conditions dans lesquelles a_i peut être exécutée, $Actuators(a_i)$ modélise comment a_i est exécutée, et $Post(a_i)$ modélise les effets de a_i sur le robot et son environnement. $Pre(a_i)$ est défini sur $varI(a_i)$. $Post(a_i)$ est défini sur $varI(a_i)$ et $varF(a_i)$. Enfin, $Actuators(a_i)$ est défini sur $varI(a_i)$, $varA(a_i)$ et $varF(a_i)$.

[3] En fait, $varI$ correspond à l'état Initial du robot, $varF$ à l'état Final, et $varA$ à ses Actionneurs.

Il convient de noter que la définition 12 permet d'obtenir, pour chaque action élémentaire $a_i \in \mathcal{A}$, un modèle qui abstrait un ensemble d'instances possibles pour a_i. En d'autres termes, le réseau de contraintes ainsi défini modélise un *espace* de solutions possibles pour l'exécution de a_i. Cette approche permet de combiner plus facilement différentes actions élémentaires en effectuant la conjonction des réseaux de contraintes correspondants à ces actions. Par ailleurs, cette approche "contraintes" permet d'ajuster le déroulement d'un comportement sensorimoteur après l'exécution de chaque action élémentaire a_i qui le compose, simplement en résolvant à nouveau la conjonction des CSPs après mise à jour des variables modélisant l'état du robot après l'exécution de a_i. À titre d'exemple, imaginons une action élémentaire permettant à un robot mobile de se déplacer d'une distance d comprise entre 1 cm et 40 cm. Le réseau de contraintes correspondant modélisera le lien entre les valeurs des actionneurs pendant l'exécution de l'action et la distance d effectivement couverte par le robot, ainsi que la contrainte $1 \leq d \leq 40$. Imaginons alors que nous souhaitions que le robot se déplace de 55 cm. Le robot peut accomplir ce comportement sensorimoteur en se déplaçant (par exemple) de 30 cm dans un premier temps, puis de 25 cm pour atteindre l'objectif fixé. Imaginons maintenant qu'en raison d'une légère perturbation extérieure, le robot n'ait parcouru que 29 des 30 cm prévus pour la première étape. Le comportement sensorimoteur est cependant toujours réalisable. En effet, un simple appel au résolveur de contrainte en imposant 29 cm (au lieu de 30) comme distance réellement parcourue permet de déterminer que le robot doit désormais se déplacer de 26 cm (au lieu des 25 initialement prévus) afin de couvrir les 55 cm et ainsi accomplir le comportement sensorimoteur souhaité. Ce type de *corrections mineures* sera pleinement utilisé par le module de contrôle de notre architecture de supervision pour ajuster l'exécution d'un comportement sensorimoteur (*c.f.* section 7.3.5).

Dans l'architecture que nous proposons, les actions élémentaires d'un robot constituent des *briques de base* qui doivent être combinées pour générer des comportements sensorimoteurs. Contrairement aux architectures logicielles de contrôle présentées dans le chapitre 6, notre architecture de supervision n'est pas destinée à s'attaquer à des problèmes de planification de haut niveau. En effet, comme nous l'avons mentionné précédemment, les comportements sensorimoteurs constituent les capacités *basiques* de perception et d'action d'un robot et sont exécutés dans des fenêtres temporelles très réduites,[4] au contraire des missions de haut niveau supervisées par les architectures de contrôle précédemment citées. Aussi, bien que l'environnement du robot soit dynamique et qu'il soit donc difficile (voire impossible) de prévoir parfaitement l'évolution du robot, nous avons choisi de limiter notre approche à un problème de planification classique [RN03].

[4] Les comportements sensorimoteurs sont en effet exécutés à l'aide d'un nombre réduit d'actions élémentaires.

Le formalisme que nous utilisons est une extension du langage STRIPS, dans laquelle nous rajoutons la composante *Actuators* qui modélise des ordres au niveau des actionneurs du robot (*i.e.* le comportement des actionneurs durant l'exécution de chaque action élémentaire). Une planification classique en environnement dynamique peut conduire à une production excessive de plans dont l'exécution se solde par un échec. Cependant, les expérimentations que nous avons menées montrent que le nombre de replanifications est réduit et que les temps de planification ne pénalisent pas notre approche.

Enfin, il convient de noter que les réseaux de contraintes manipulés par notre architecture peuvent être vus comme des *fonctions de transition* permettant au robot de passer d'un état à un autre. Les actions élémentaires prises en compte dans le processus de planification de notre architecture constituent par ailleurs le seul moyen de faire évoluer *volontairement* le système.[5] La modélisation CSP des actions élémentaires d'un robot doit être par conséquent la plus précise possible, afin de rendre compte fidèlement du déroulement d'une action élémentaire et de ses conséquences sur le robot et son environnement. En ce sens, on notera que la définition 12 vise la production d'une modélisation complète, puisqu'elle implique l'ensemble des descripteurs (initiaux et finaux) et des actionneurs du robot.

7.3.2 Acquisition automatique des actions élémentaires

L'abstraction des actions élémentaires sous forme de réseaux de contraintes peut être réalisée par un opérateur humain. Cette option n'est cependant pas satisfaisante car elle nécessite une double compétence : une expertise robotique (*i.e.* l'expertise métier) et une expertise en programmation par contraintes. Cette option tendrait par ailleurs à replacer notre approche dans le cadre d'une constitution *manuelle* des comportements sensorimoteurs d'un robot, ce que notre approche tente de contourner. Afin d'automatiser le processus de modélisation des actions élémentaires, nous avons donc choisi d'utiliser la plate-forme d'acquisition automatique de réseau de contraintes CONACQ étudiée dans la première partie de cette thèse.

Pour acquérir le réseau de contraintes modélisant une action élémentaire $a_i \in \mathcal{A}$, nous avons besoin d'une librairie de contraintes $\mathcal{L}$, qui constitue le biais d'apprentissage, ainsi que de deux ensembles disjoints E^+ et E^- qui contiennent respectivement des instances positives de a_i et des instances négatives de cette action élémentaire.

Les données d'entraînement fournies à CONACQ peuvent être constituées, soit par simu-

[5] L'exécution d'actions réflexes, dont nous discuterons en fin de section 7.3.5, et la présence de perturbations extérieures peuvent faire évoluer le système. Elles sont cependant indépendantes de notre processus de planification, dans la mesure où leur exécution ne peut être déclenchée par notre architecture.

lation à l'aide d'un logiciel de Conception Assistée par Ordinateur,[6] soit en effectuant directement des exécutions sur le robot étudié. Il convient alors de distinguer les exécutions qui correspondent à l'action élémentaire a_i, qui constituent l'ensemble E^+ fourni à CONACQ, de celles qui n'y correspondent pas, et qui constituent l'ensemble E^-. Chaque donnée d'entraînement est pour cela décrite sous la forme d'un tuple étiqueté (positivement ou négativement) $(\delta_1^I, \ldots, \delta_p^I, \alpha_1, \ldots, \alpha_q, \delta_1^F, \ldots, \delta_p^F)$, où $\delta_1^I, \ldots, \delta_p^I$ et $\delta_1^F, \ldots, \delta_p^F$ correspondent respectivement à l'état des descripteurs du robot au début (*resp.* à la fin) de l'action, et où $\alpha_1, \ldots, \alpha_q$ correspondent aux commandes envoyées aux actionneurs du robot pour l'exécution de a_i. Les différents descripteurs et actionneurs d'un robot prennent majoritairement leurs valeurs dans des espaces continus. Notre architecture de contrôle manipulant des réseaux de contraintes discrets, il est nécessaire d'effectuer un pré-traitement sur les données issues des descripteurs et des actionneurs afin de les discrétiser. Dans le cadre des expérimentations que nous avons menées, nous avons effectué une discrétisation arbitraire (*i.e.* valeur entière la plus proche). Une telle discrétisation peut cependant trop contraindre l'apprentissage. En conséquence, dans l'optique de projets plus vastes, il conviendra d'opter pour la discrétisation la mieux adaptée en fonction du robot étudié.

À partir de ces données d'entraînement, la plate-forme CONACQ modélisera **automatiquement** un réseau de contraintes répondant à la définition 12.

Le choix du biais d'apprentissage constitue une caractéristique essentielle au succès du processus d'acquisition. En effet, la librairie de contraintes utilisée pour le processus d'apprentissage doit remplir deux exigences partiellement opposées : l'expressivité, qui permet d'abstraire précisément chaque action élémentaire,[7] ainsi que de bonnes propriétés computationnelles permettant de garantir des temps de calculs limités. Le biais d'apprentissage devra en conséquence être choisi en étroite collaboration avec les roboticiens.

Les expérimentations présentées dans le chapitre 8 démontrent que la plate-forme CONACQ est capable d'acquérir un modèle précis de chaque action élémentaire. Cependant, le choix d'un biais d'apprentissage *adéquat* pourrait être difficile à faire dans le cas de robots plus complexes. Dans ce cas, CONACQ pourrait être envisagée en tant qu'outil de pré-traitement, permettant l'acquisition d'un modèle initial (mais imprécis) de chaque action élémentaire. Les réseaux de contraintes acquis devraient alors être améliorés et reformulés en collaboration avec les roboticiens afin d'obtenir des modélisations précises.

[6] L'utilisation d'un simulateur permet en effet la réalisation d'un nombre *infini* de tests sans risque de panne ou de casse du robot, tout en limitant les coûts d'exploitation du robot.

[7] C'est-à-dire établir une modélisation sémantiquement correcte de chaque action élémentaire.

7.3.3 Planification d'un comportement sensorimoteur

Dans cette section, nous présentons le processus de planification utilisé par notre architecture pour établir un comportement sensorimoteur donné. Notre approche traduit tout d'abord ce comportement sensorimoteur sous la forme d'une tâche T, exprimée à l'aide d'un état initial E_I et d'un but E_F (*i.e.* son état final). Déterminer un plan permettant de réaliser T revient alors à trouver une séquence $\mathcal{S}$ d'actions élémentaires issues de $\mathcal{A}$, telle que si ces actions sont exécutées à partir de E_I, elles permettent d'aboutir à un état qui satisfait E_F.

Afin de limiter le temps de résolution de ce problème de planification, nous restreignons la modélisation de chaque action élémentaire a_i à ses pré-conditions et à ses effets, comme dans le cadre STRIPS. Les pré-conditions et les effets de chaque action élémentaire a_i sont données par les ensembles de contraintes $Pre(a_i)$ et $Post(a_i)$ (*c.f.* définition 12). Nous utilisons alors un planificateur de tâches inspiré de CSP-PLAN [LB03] pour résoudre le problème de planification. Conformément à cet algorithme, nous fixons la longueur k du plan recherché. Le problème consiste maintenant à déterminer s'il existe un plan de longueur k permettant d'atteindre le but de la tâche T à partir de l'état initial E_I.

Modélisation CSP du problème de planification

Étant donnée une longueur k, conformément à CSP-PLAN, notre approche modélise le problème de planification sous la forme d'un réseau de contraintes défini comme suit :

1. Définition des variables - Pour modéliser le problème correspondant à un plan de longueur k, nous définissons $k+1$ ensembles $\Delta^t = \{\delta_1^t, \ldots, \delta_p^t\}$ de *variables descripteurs*, où Δ^t correspond à l'ensemble Δ des descripteurs du robot à l'étape t, $0 \leq t \leq k$. Ainsi, $\forall t \in \{0 \ldots k\}$, $\forall i \in \{1 \ldots p\}$, δ_i^t modélise le descripteur δ_i du robot à l'étape t. Nous définissons par ailleurs k ensembles $\mathcal{A}^t = \{a_1^t, \ldots, a_m^t\}$ de *variables actions* où, $\forall j \in \{1 \ldots m\}$, a_j^t est une variable booléenne telle que a_j^t est vraie si et seulement si l'action élémentaire a_j est exécutée de l'étape t à l'étape $t+1$, $0 \leq t \leq k-1$.

2. Définition des contraintes - Pour modéliser correctement (du point de vue sémantique) le problème de planification, nous définissons les ensembles de contraintes suivants :

1. *État initial et but.* Les valeurs des variables descripteurs δ_i^0 de l'étape 0 et les valeurs des variables descripteurs δ_i^k de l'étape k doivent être respectivement compatibles avec l'état initial et le but de la tâche T. Nous définissons pour cela les contraintes correspondantes sur les variables δ_i^0 et δ_i^k, $1 \leq i \leq p$. Ainsi, par exemple, nous définissons la contrainte $(v_{min} \leq \delta_i^k \leq v_{max})$ si le descripteur δ_i du robot doit prendre sa valeur dans $[v_{min}, v_{max}]$

à la fin du plan,

2. *Vérification des pré-conditions.* Une action $a_j \in \mathcal{A}$ peut être exécutée de l'étape t à l'étape $t+1$ uniquement quand *toutes* ses pré-conditions sont vérifiées. Ainsi, pour chaque étape $t \in \{0 \dots k-1\}$ et pour chaque variable action a_j^t, nous ajoutons la contrainte $a_j^t \rightarrow Pre(a_j)^t$, où $Pre(a_j)^t$ est la pré-condition de a_j relativement à l'étape t (*i.e.* définie sur Δ^t),
3. *Effets d'une action.* De manière analogue, $\forall t \in \{0 \dots k-1\}, \forall j \in \{1 \dots m\}$, la contrainte $a_j^t \rightarrow Post(a_j)^t$ est ajoutée, telle que $Post(a_j)^t$ est défini sur Δ^t et Δ^{t+1} et modélise les effets de a_j lorsque cette action élémentaire est exécutée de l'étape t à l'étape $t+1$.

Ainsi définie, notre modélisation est similaire au Base-CSP construit par le planificateur CSP-Plan, étendue aux variables non-booléennes. Notre architecture de contrôle se limite cependant à des plans séquentiels (*i.e.* une action par étape). Pour chaque étape $t \in \{0..k-1\}$, il convient d'ajouter deux contraintes supplémentaires : $atleast(\{a_1^t, .., a_m^t\}, 1))$ impose qu'au moins une action soit exécutée de l'étape t à l'étape $t+1$, et $atmost(\{a_1^t, .., a_m^t\}, 1))$ impose qu'au plus une action soit exécutée de l'étape t à l'étape $t+1$.

Résolution du problème de planification

Pour déterminer s'il existe un plan permettant de réaliser T, conformément à l'algorithme CSP-Plan, nous faisons appel à notre planificateur de tâches en faisant varier la longueur k de la solution recherchée (en commençant par $k=1$). Pour une longueur k donnée, notre planificateur modélise le réseau de contraintes correspondant au problème de planification de taille k, puis fait appel à un résolveur de contraintes pour déterminer si une solution existe. Si le résolveur de contraintes indique qu'il n'existe pas de solution, nous recherchons une solution de longueur $k+1$. Le processus est alors reconduit jusqu'à ce qu'une solution soit identifiée ou jusqu'à ce qu'une borne supérieure sur k soit atteinte (*i.e.* longueur maximale acceptée). En procédant de la sorte, la première solution fournie par notre planificateur de tâches est *minimale* en nombre d'étapes.

S'il existe un plan permettant de réaliser T, le planificateur de tâches renvoie une séquence $\mathcal{S}$ d'actions élémentaires issues de $\mathcal{A}$ que le robot doit exécuter séquentiellement afin d'accomplir T. La modélisation CSP du problème de planification n'ayant pas pris en compte[8] la composante $Actuators(a_i)$ de chaque action élémentaire $a_i \in \mathcal{A}$, il convient cependant de noter que $\mathcal{S}$ ne permet pas, en l'état, de déterminer comment les actionneurs du robot doivent être successivement sollicités pour que ce dernier réalise effectivement T. $\mathcal{S}$ correspond *simplement*

[8] Dans l'optique de limiter les temps de résolution.

à une séquence respectant les conditions d'enchaînement des actions élémentaires permettant de réaliser T.

7.3.4 Instanciation d'un comportement sensorimoteur

Dans cette section, nous présentons comment le module de contrôle de notre architecture détermine comment les actions élémentaires d'une séquence $\mathcal{S}$, planifiée selon le processus précédemment décrit, doivent être successivement exécutées afin d'accomplir T (et donc le comportement sensorimoteur associé). Le module de contrôle modélise la séquence $\mathcal{S}$ sous la forme d'un réseau de contraintes, appelé dans la suite *CSP global*. La modélisation utilisée pour modéliser ce CSP global est très proche de celle utilisée pour la planification. Les variables *actions*, qui caractérisaient l'exécution (ou la non-exécution) des actions $a_i \in \mathcal{A}$ à chaque étape du plan, sont cependant remplacées par des variables *actionneurs*, qui modélisent l'état des actionneurs au cours du temps. En procédant de la sorte, le module de contrôle cherche à déterminer comment les actionneurs du robot doivent être sollicités durant l'exécution de $\mathcal{S}$.

Pour modéliser une séquence $\mathcal{S}$ possèdant k actions élémentaires (*i.e.* $\mathcal{S}$ est de longueur k), nous utilisons $k+1$ ensembles $\Delta^t = \{\delta_1^t, \ldots, \delta_p^t\}$ de *variables descripteurs* et k ensembles $\alpha^t = \{\alpha_1^t, \ldots, \alpha_q^t\}$ de *variables actionneurs*. Comme pour la modélisation du problème de planification, $\forall t \in \{0 \ldots k\}, \forall i \in \{1 \ldots p\}$ δ_i^t modélise le descripteur δ_i à l'étape t. $\forall j \in \{1 \ldots q\}$ α_j^t modélise l'actionneur α_j de l'étape t à l'étape $t+1$, $0 \leq t \leq k-1$.

Les contraintes du CSP global sont données par les actions élémentaires successives de $\mathcal{S}$. La première action élémentaire de $\mathcal{S}$ doit être exécutée de l'étape 0 à l'étape 1, la deuxième action élémentaire de $\mathcal{S}$ doit être exécutée de l'étape 1 à l'étape 2 et, récursivement, la t-ième action élémentaire de $\mathcal{S}$ doit être exécutée de l'étape $t-1$ à l'étape t. Soit $a_j \in \mathcal{S}$ l'action élémentaire devant être exécutée de l'étape t à l'étape $t+1$, $0 \leq t \leq k-1$. Pour modéliser correctement l'exécution de a_j, nous ajoutons au CSP global les ensembles de contraintes suivants. L'ensemble des contraintes $Pre(a_j)^t$, qui modélise les pré-conditions de a_j relativement à l'étape t, est définie sur Δ^t et ajouté au CSP global. $Post(a_j)^t$, qui modélise les effets de a_j, est défini sur Δ^t et Δ^{t+1} et ajouté à la modélisation, exactement comme dans la modélisation du problème de planification. Enfin, $Actuators(a_j)^t$, qui modélise comment les actionneurs du robot doivent réagir pendant l'exécution de a_j de l'étape t à l'étape $t+1$, est défini sur Δ^t, α^t et Δ^{t+1}, et ajouté à la modélisation.

Pour modéliser l'état initial E_I du robot et le but E_F de la tâche T, nous ajoutons enfin les contraintes suivantes au CSP global. Pour chaque variable descripteur δ_i^0, nous ajoutons la contrainte $(\delta_i^0 = v)$, où v est la valeur prise par le descripteur δ_i dans l'état initial. De manière

analogue, si le descripteur δ_j doit prendre sa valeur dans $[v_{min}, v_{max}]$ dans le but de la tâche T, nous ajoutons la contrainte $(v_{min} \leq \delta_j^k \leq v_{max})$.

Ainsi défini, le CSP global modélise correctement la séquence $\mathcal{S}$ d'actions élémentaires et la tâche T. S'il existe une solution au CSP global, les valeurs de la variable descripteur δ_i^t $(0 \leq t \leq k)$ indiquent les valeurs successives prises par le descripteur $\delta_i \in \{\delta_1, \ldots, \delta_p\}$ pendant l'exécution de $\mathcal{S}$. Les valeurs des variables actionneurs α_i^t, $0 \leq t \leq k-1$, indiquent quant à elles comment les actionneurs du robot doivent successivement réagir pour accomplir T.

7.3.5 Supervision d'un comportement sensorimoteur

Une fois modélisé, le réseau de contraintes correspondant à la séquence $\mathcal{S}$ est fourni à un résolveur de contraintes qui détermine si le problème est soluble. Si le résolveur de contraintes renvoie une solution, le module de contrôle de notre architecture supervise l'exécution étape par étape de la séquence $\mathcal{S}$, qu'il ajuste à la réalité opérationnelle du robot.

À l'issue de l'action élémentaire exécutée de l'étape t à l'étape $t+1$, $0 \leq t \leq k-1$, le module de contrôle confronte l'état courant du robot, donné par l'ensemble $\Delta = \{\delta_1, \ldots, \delta_p\}$ de ses descripteurs, aux prévisions établies par les valeurs des variables descripteurs $\{\delta_1^{t+1}, \ldots, \delta_p^{t+1}\}$ du CSP global. L'exécution de la séquence est alors ajustée différemment en fonction de l'écart entre l'état courant les prédictions établies par le CSP global :

- **Exécution identique aux prédictions -** Si l'état courant du robot correspond parfaitement aux prévisions établies, le module de contrôle poursuit l'exécution de la séquence $\mathcal{S}$ telle qu'elle a été définie initialement dans le CSP global. L'action élémentaire suivante est alors exécutée en fournissant aux actionneurs du robot les valeurs des variables actionneurs $\alpha_1^{t+1}, \ldots, \alpha_q^{t+1}$ du CSP global.
- **Corrections mineures -** Si l'état courant du robot diffère des prédictions, le CSP global est réduit aux actions élémentaires de $\mathcal{S}$ n'ayant pas encore été exécutées, et on impose l'état courant aux variables descripteurs δ_i^{t+1}, $\forall i \in \{1..p\}$. Le résolveur de contraintes détermine[9] alors s'il est possible d'ajuster[10] les prochaines actions élémentaires de $\mathcal{S}$ afin d'atteindre l'objectif fixé depuis l'état courant sans remettre en cause $\mathcal{S}$. Si de tels ajustements existent, le module de contrôle rectifie l'exécution de la séquence $\mathcal{S}$ en imposant aux prochaines actions élémentaires à exécuter les corrections mineures calculées par le CSP global résultant.

[9] À l'heure actuelle, la résolution du CSP global résultant est relancée *from scratch* sans pour autant pénaliser les performances du module de contrôle.

[10] Comme nous l'avons illustré dans l'exemple du robot mobile présenté en section 7.3.1.

– **Replanification -** S'il est impossible d'accomplir T à partir de l'état courant au moyen des actions élémentaires de $\mathcal{S}$ non encore exécutées, le module de contrôle fait à nouveau appel au planificateur de tâches pour déterminer une nouvelle séquence $\mathcal{S}'$ d'actions élémentaires permettant d'accomplir T à partir de l'état courant. Le module de contrôle supervise alors l'exécution de $\mathcal{S}'$ en utilisant le même cycle opératoire (*i.e.* exécution d'une action élémentaire, perception de l'état courant puis correction des éventuels écarts entre la réalité et les prévisions), jusqu'à ce que la tâche T, et donc le comportement sensorimoteur associé, soit achevée, ou jusqu'à ce que le planificateur de tâches ne détermine que T ne peut être accomplie à partir de l'état courant.

Les robots actuels, en plus des actions élémentaires pris en compte par notre architecture de supervision, disposent généralement d'un panel d'actions *réflexes*. Exécutées en tâche de fond, les actions réflexes d'un robot, telles que l'arrêt immédiat, sont destinées à répondre de manière immédiate à une situation inattendue susceptible d'être dangereuse pour le robot. Leur exécution est par conséquent prioritaire par rapport à l'exécution des actions élémentaires.

L'exécution d'une action réflexe est *transparente* pour notre architecture. En effet, elle interrompt temporairement l'exécution d'une séquence d'actions pour "sécuriser" le robot avant de rendre la main au module de contrôle de notre architecture une fois cette sécurisation accomplie. L'état courant du robot (résultant de l'exécution de l'action réflexe) ne correspond plus, *a priori*, aux prévisions établies. Cette modification d'état, si elle existe, est alors identifiée par le module de contrôle de notre architecture qui corrige en conséquence l'exécution du comportement sensorimoteur courant selon le cycle de supervision précédemment décrit (*i.e.* corrections mineures ou replanification).

Enfin, il convient de noter que les actions réflexes constituent par ailleurs un complément de sécurité à notre approche, dans la mesure où elles permettent de pallier d'éventuels défauts de modélisation[11] ainsi que d'éventuels temps de résolution prohibitifs lors d'un appel au résolveur de contraintes ou au planificateur de tâches.

7.4 Conclusion

Dans ce chapitre, nous avons proposé une approche programmation par contraintes pour la constitution et la supervision *automatique* de comportements sensorimoteurs en robotique. L'architecture logicielle que nous proposons utilise les réseaux de contraintes pour modéli-

[11] On peut en effet imaginer que l'exécution d'une action élémentaire mal modélisée puisse placer le robot en situation dangereuse, nécessitant alors l'exécution d'une action réflexe.

ser les actions élémentaires d'un robot. Ces réseaux de contraintes, automatiquement acquis par apprentissage, sont ensuite combinés à l'aide d'outils de planification pour constituer des comportements sensorimoteurs exécutables par le robot. Lors de l'exécution d'un comportement sensorimoteur, notre architecture exploite les propriétés calculatoires des réseaux de contraintes pour contrôler en temps réel la réalisation du comportement, en corrigeant les éventuels écarts existants entre l'état effectif du robot et les prévisions fournies par la modélisation CSP.

Il convient de noter que l'architecture de supervision que nous proposons se limite strictement à la constitution de comportements sensorimoteurs. Les actions élémentaires d'un robot sont modélisées à l'aide de réseaux de contraintes discrets puis combinées dans le cadre d'une planification classique. Ce choix vise à fournir une réponse efficace à la réalité opérationnelle d'un robot autonome. Le recours à une plate-forme d'apprentissage vise quant à lui à automatiser le processus de constitution de comportements sensorimoteurs, qui constitue à l'heure actuelle une tâche particulièrement ardue pouvant nécessiter de nombreuses heures de modélisation et de calculs.

La modélisation des actions élémentaires à l'aide de réseau de contraintes ainsi que l'utilisation d'outils de planification pour consituter des séquences d'actions ont été publiées dans [PBDK06] et [Pau06], et ont par ailleurs été présentées aux *Journées nationales d'Intelligence Artificielle Fondamentale 2007* [Pau07]. La formalisation complète de notre approche fait quant à elle l'objet d'un article en cours de soumission, co-signé avec Christian Bessière et Jean Sallantin.

Chapitre 8

Expérimentations

Dans ce chapitre, nous présentons un ensemble d'expérimentations permettant de valider l'intérêt de l'approche proposée dans la seconde partie de cette thèse pour la constitution automatique de comportements sensorimoteurs en robotique. Réalisées dans le cadre d'une collaboration transversale entre les départements robotique et informatique du LIRMM, ces expérimentations permettent par ailleurs de mettre en lumière certains aspects à améliorer dans l'optique d'une utilisation plus large de notre architecture logicielle de supervision.

Pour réaliser ces expérimentations, nous avons choisi d'utiliser la plate-forme CONACQ mentionnée précédemment, ainsi que le résolveur de contraintes CHOCO [Cho07]. Par ailleurs, certaines vidéos des expérimentations présentées dans ce chapitre sont disponibles sur l'url suivante : `http ://www.lirmm.fr/~paulin/-PHD/`.

Ce chapitre commence par présenter deux expérimentations préliminaires (section 8.1), réalisées sur des robots simples, destinées à lever deux verrous techniques. Le reste du chapitre est consacré au déploiement complet de notre architecture sur le robot réel TRIBOT présenté en section 8.2. Dans la section 8.3, nous présentons les actions élémentaires considérées dans notre expérimentation. La section 8.4 présente la modélisation CSP réalisée par CONACQ de ces actions élémentaires, puis nous présentons en section 8.5 la planification et l'exécution effective, supervisées par notre architecture, d'un comportement sensorimoteur pour le TRIBOT. Enfin, nous présentons une synthèse de ces expérimentations dans la section 8.6.

8.1 Expérimentations préliminaires

Nous présentons dans cette section deux expérimentations préliminaires destinées à lever deux verrous techniques liés à l'approche programmation par contraintes proposée dans le

chapitre 7 : D'une part la capacité de CONACQ à modéliser automatiquement par apprentissage des actions élémentaires sous la forme de CSPs, et d'autre part la capacité de CHOCO à contrôler un robot en contexte réel (robot cadencé à $100Hz$, perturbations extérieures).

8.1.1 Validation de la plate-forme CONACQ pour l'acquisition automatique d'actions élémentaires

L'objectif de cette première expérimentation préliminaire s'attache à valider l'utilisation de la plate-forme CONACQ pour l'acquisition automatique d'actions élémentaires. Elle porte sur l'étude du robot sauteur unijambiste TWIG illustré par la figure 8.1.

Le robot TWIG

Dans la cadre de notre collaboration avec le département robotique du LIRMM, et dans l'optique de réduire les coûts de développement de cette première étude, le robot TWIG et les expérimentations ont entièrement été réalisés à l'aide du logiciel CAO SOLIDWORKS [Sol07] et de son simulateur COSMOSMOTION.

TWIG est un *vieux* robot unijambiste défini dans [Rai86], présentant une architecture minimale mais dont les lois de commande sont difficiles à établir malgré sa mécanique simple. Comme le montre la figure 8.1, TWIG est constitué d'un volant d'inertie, d'un ressort hélicoïdal et d'un vérin. Le volant d'inertie permet à TWIG de rester dans un état stable (*i.e.* en position verticale) et lui permet aussi de se pencher dans une direction. Le vérin sert à effectuer une poussée longitudinale sur la jambe alors que le ressort est utilisé pour amortir la réception d'un saut.

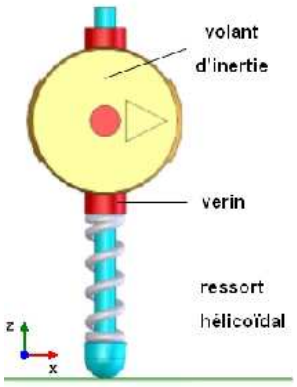

FIG. 8.1 – Le robot TWIG.

Sur recommandation de Sébastien Krut,[1] qui a modélisé TWIG sous SOLIDWORKS, les variables d'état prises en compte dans l'étude de TWIG sont les suivantes. U_θ et U_R correspondent aux tensions qui sont respectivement appliquées au volant d'inertie et au vérin. R représente la distance de déploiement de la partie mobile du vérin par rapport à sa partie fixe. θ caractérise l'angle d'orientation du volant d'inertie par rapport au corps du robot. G est le point de gravité du robot sur lequel est fixé un accéléromètre. Les descripteurs $\ddot{x}_G$ et $\ddot{z}_G$

[1] Sébastien Krut est chargé de recherche CNRS en robotique au LIRMM, `http ://www.lirmm.fr/~krut/`.

caractérisent les accélérations de G et les descripteurs x_G et z_G indiquent la position de G. Pour notre étude, x_G et z_G nous sont donnés directement par le simulateur.[2] Enfin, s est un capteur booléen qui vaut 1 ssi TWIG touche le sol.

Acquisition automatique des actions élémentaires de TWIG par apprentissage

La fréquence du simulateur COSMOSMOTION a été fixée à $100Hz$, c'est-à-dire que pendant une durée de 1 seconde, le simulateur réalise 100 mesures successives de chacun des descripteurs de TWIG. Pour modéliser les différentes actions élémentaires, nous avons restreint notre étude à l'état des descripteurs au début et à la fin de chaque action élémentaire et avons pris en compte le temps nécessaire à la réalisation de chaque action élémentaire. Chacune des actions élémentaires de TWIG a été décrite sous la forme de tuples étiquetés $(t^I, s^I, x_G^I, z_G^I, U_R, U_\theta, t^F, s^F, x_G^F, z_G^F)$ où la variable t référence le temps et où les exposants I et F indiquent respectivement l'état des variables au temps initial et final de chaque action élémentaire. Par souci de simplification des réseaux de contraintes, les descripteurs θ et R n'ont pas été pris en compte. Les valeurs des descripteurs ont par ailleurs été discrétisées arbitrairement en utilisant la valeur entière la plus proche.

À l'aide de COSMOSMOTION, nous avons effectué une série de simulations afin de constituer les données d'entraînement nécessaires à l'apprentissage automatique, à raison de 3 instances positives et 4 instances négatives en moyenne pour chaque action élémentaire. À partir de ces données d'entraînement, CONACQ a **automatiquement modélisé par apprentissage** les quatre actions élémentaires de TWIG à l'aide des réseaux de contraintes suivants :

Saut Vertical (action a_1)

$$\begin{cases} Prec(a_1) = \{(s^I = 1)\} \\ Actuators(a_1) = \{(U_R = 500,\ U_\theta = 0)\} \\ Post(a_1) = \{(s^F = 0,\ x_G^F = x_G^I,\ z_G^F = z_G^I + 3,\ t^F = t^I + 2)\} \end{cases}$$

Atterrissage Vertical (action a_2)

$$\begin{cases} Pre(a_2) = \{(s^I = 0)\} \\ Actuators(a_2) = \{(U_R = 0,\ U_\theta = 0)\} \\ Post(a_2) = \{(s^F = 1,\ x_G^F = x_G^I,\ z_G^F = z_G^I - 3,\ t^F = t^I + 20)\} \end{cases}$$

Saut Horizontal (action a_3)

$$\begin{cases} Pre(a_3) = \{(s^I = 1)\} \\ Actuators(a_3) = \{(U_R = 300,\ U_\theta = 450)\} \\ Post(a_3) = \{(s^F = 0,\ x_G^F = x_G^I + 3,\ z_G^F = z_G^I + 2,\ t^F = t^I + 6)\} \end{cases}$$

Rester stable (action a_4)

$$\begin{cases} Pre(a_4) = \{(s^I = 1)\} \\ Actuators(a_4) = \{(U_R = 0,\ U_\theta = 0)\} \\ Post(a_4) = \{(s^F = 1,\ x_G^F = x_G^I,\ z_G^F = z_G^I,\ t^F = t^I + 1)\} \end{cases}$$

Discussion

L'étude du robot TWIG s'est volontairement concentrée sur l'utilisation de CONACQ en robotique. Ainsi, le biais d'apprentissage utilisé a mis en jeu une librairie limitée de contraintes,

[2] x_G et z_G peuvent cependant être calculés en intégrant deux fois les accélérations de G.

adéquate pour la modélisation des quatre actions élémentaires étudiées mais qui devrait être étoffée pour la modélisation d'actions élémentaires plus complexes. Par ailleurs, les instances fournies à CONACQ sont le résultat d'exécution sous simulateur et représentent en ce sens une vision *idéalisée* du fonctionnement de TWIG (absence de perturbations extérieures, parfait fonctionnement des actionneurs et des descripteurs, *etc.*). Dans le cadre d'expérimentations futures, il pourrait cependant être bénéfique que des instances issues d'exécutions réelles soient ajoutées afin de constituer des données d'entraînement encore plus représentatives.

Cette première expérimentation préliminaire permet néanmoins de valider la capacité de la plate-forme CONACQ à modéliser automatiquement par apprentissage les actions élémentaires d'un robot sous la forme de réseaux de contraintes.

8.1.2 Validation du résolveur de contraintes CHOCO pour une utilisation en contexte réel

L'objectif de la seconde expérimentation s'attache à valider l'utilisation du résolveur de contraintes CHOCO en contexte réel. Nous souhaitons pour cela vérifier que les performances de CHOCO sont compatibles avec la réalité opérationnelle d'un robot en environnement réel. Pour cela, en collaboration avec Sébastien Krut et Sébastien Andary (doctorant au LIRMM), nous avons utilisé CHOCO pour le contrôle de l'équilibre du robot PIRI illustré par la figure 8.2.

Le robot PIRI

Le robot PIRI est un Pendule Inversé stabilisé par une Roue d'Inertie, développé par le département robotique du LIRMM, constitué d'un bâti, d'un pendule, d'un volant d'inertie et d'un inclinomètre. Le pendule constitue le corps du robot. Il est relié au bâti par une liaison pivot. Comme pour le robot TWIG, le volant d'inertie est sollicité pour faire pencher le pendule d'un côté ou l'autre. L'inclinomètre indique quant à lui l'angle du pendule par rapport à la verticale.

PIRI est en perpétuel déséquilibre : en l'absence d'une force de contrôle, il ne peut rester indéfiniment en position verticale. Pour maintenir le robot en position verticale, les roboticiens ont établi une loi de commande, cadencée à $100Hz$, robuste aux perturbations [Gar06]. Cette loi de commande stabilisante met en jeu l'angle θ_1 du pendule par rapport à la verticale, sa vitesse de chute $\dot{\theta_1}$, et à la vitesse de rotation $\dot{\theta_2}$ de la roue d'inertie. Elle s'exprime à l'aide de la somme pondérée $C_2 = -7,5682\ \theta_1 + 1,0373\ \dot{\theta_1} + 0,0032\ \dot{\theta_2}$, où C_2 est la valeur du couple que doit appliquer le moteur du volant d'inertie pour ramener le pendule en position verticale.

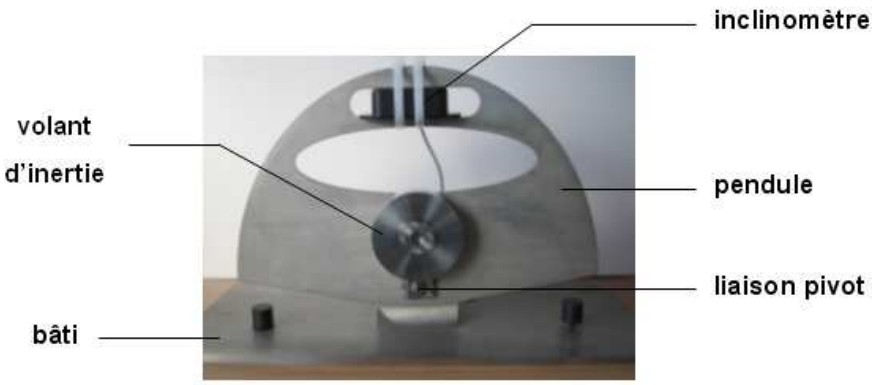

FIG. 8.2 – Le robot PIRI.

Expérimentation

Afin de tester les performances de CHOCO dans le cadre d'une utilisation en robotique, nous avons traduit la loi de commande établie par les roboticiens sous la forme d'un réseau de contraintes. Nous avons ensuite utilisé ce CSP pour commander le moteur du volant d'inertie du robot PIRI, en lieu et place du dispositif mis en place par les roboticiens.[3] La fréquence du robot était par ailleurs fixée à $100Hz$, conformément à la loi de commande stabilisante.

Exceptée une légère perte de fluidité dûe à un manque d'optimisation du canal de communication entre CHOCO et le robot, notre modélisation en contraintes a permis de commander **en temps réel** le contrôle de l'équilibre de PIRI en présence de perturbations extérieures (déstabilisations volontaires du pendule) ou inhérentes au système (imprécision des capteurs et du rendement du moteur), comme le faisait la loi de commande établie par les roboticiens.

Discussion

Le pendule inversé est un outil très utilisé dans le domaine de l'automatique car sa dynamique est très proche de systèmes beaucoup plus complexes. Malgré son apparente simplicité, la loi de commande stabilisante utilisée pour le contrôle du PIRI est le fruit d'un long travail théorique et pratique réalisé au LIRMM [Gar06]. Une étude théorique a tout d'abord permis d'établir le modèle mathématique représentant la dynamique du robot (équations d'Euler-Lagrange, énergies cinétique et potentielle, Lagrangien associé). Le dimensionnement du système a ensuite permis d'ajuster au mieux la loi de commande aux caractéristiques réelles du robot (densité et masse des matériaux utilisés, correction de la géométrie du système, puis-

[3] Le programme utilisé par les roboticiens pour le calcul de la loi de commande est implémenté en langage C et le contrôle temps réel du robot est assuré via l'environnement RTX édité par ARDENCE [Ard07].

sance réelle du moteur du volant d'inertie). Une série de simulations virtuelles et réelles a enfin permis d'ajuster la fréquence de la loi de commande à $100Hz$ afin de garantir une bonne fluidité dans la recherche d'équilibre.

Dans le cadre de notre seconde expérimentation préliminaire, notre intervention a simplement consisté à coder en contraintes la loi de commande stabilisante établie par les roboticiens. Les résultats obtenus, sensiblement équivalents à ceux obtenus par le système logiciel des roboticiens, démontrent la capacité du résolveur de contraintes Choco à commander un robot cadencé à une fréquence élevée, et ainsi sa capacité à répondre efficacement à la réalité opérationnelle d'un robot en environnement réel.

8.2 Le robot Tribot

Dans le reste de ce chapitre, nous présentons les expérimentations que nous avons menées sur le robot Tribot afin de valider l'intérêt de notre architecture logicielle pour la planification automatique d'un comportement sensorimoteur. Illustré par la figure 8.3, Tribot est un robot mobile proposé dans le kit Lego Mindstorms NXT.[4]

8.2.1 Dispositif expérimental

Fig. 8.3 – Le robot Tribot.

Tribot est constitué de 3 servomoteurs. Deux d'entre eux permettent au robot de se déplacer, tandis que le troisième est utilisé pour ouvrir et fermer la pince du Tribot. Tribot est par ailleurs constitué des 4 descripteurs suivants :

– Le capteur d'ultrasons (*ultrasonic* sensor) permet de détecter un obstacle situé en face du Tribot et permet d'estimer la distance séparant le robot de cet obstacle (jusqu'à $255cm$),

[4]Plus de détails sur : `http ://mindstorms.lego.com/`.

- Le capteur photosensible (*light* sensor), situé sous le robot, détermine la couleur du sol, en mesurant l'intensité lumineuse de la surface sur laquelle évolue le TRIBOT,
- Le capteur de pression (*touch* sensor) détecte quand il est pressé par quelque chose, et quand la pression est interrompue. Le capteur de pression est utilisé comme un capteur tactile qui permet au TRIBOT de déterminer s'il est en contact avec un objet,
- Le capteur sonore (*sound* sensor) permet quant à lui d'effectuer des mesures de pression acoustique.

Les trois servomoteurs et les quatre descripteurs du TRIBOT sont par ailleurs connectés à la brique programmable NXT. Ce micro-contrôleur 32-bits permet d'envoyer des commandes aux servomoteurs et de récupérer les valeurs des différents descripteurs du robot.

Le robot TRIBOT est une plate-forme mécanique très simple,[5] composée d'éléments plastiques LEGO TECHNIC. Il n'est capable d'effectuer qu'une gamme réduite de mouvements (avancer, reculer, tourner, ouvrir/fermer la pince) et ses capteurs ne permettent de le doter que de capacités d'interactions limitées. La programmation TRIBOT suggérée par le manuel de l'utilisateur du kit LEGO MINDSTORMS NXT se limite ainsi à la définition de *tâches programmables* relativement rudimentaires, telle que la saisie d'une balle ou le suivi d'une ligne au sol. De ce point de vue, les capacités du robot TRIBOT sont limitées. Elles correspondent en revanche à des comportements sensorimoteurs, ce qui inscrit pleinement l'étude du TRIBOT dans le cadre de nos expérimentations.

8.2.2 Programmation du TRIBOT via le logiciel LEGO MINDSTORMS NXT

En configuration standard (commercialisée par la société LEGO), la programmation d'un comportement sensorimoteur pour le robot TRIBOT s'effectue en deux étapes. Dans un premier temps, on programme les actions du robot sur ordinateur à l'aide du logiciel de programmation LEGO MINDSTORMS NXT, qui propose une palette de *blocs de programmation*. Chaque bloc de programmation détermine comment le robot agit ou réagit. Pour programmer un comportement sensorimoteur, il suffit alors de définir une séquence (aussi appelé *programme*) de blocs de programmation (modélisant chacun une étape du comportement sensorimoteur). Une fois la création du programme terminée, la deuxième étape consiste à télécharger ce programme sur le micro-contrôleur NXT, via une connexion USB ou Bluetooth, et il devient possible de lancer son exécution. Le micro-contrôleur coordonne alors automatiquement les commandes aux servomoteurs et la lecture des valeurs des descripteurs, conformément à la suite de blocs

[5] Le kit LEGO MINDSTORMS NXT est ainsi commercialisé en tant que "robot mécano jouet".

de programmation définie sur ordinateur. Si la séquence de blocs de programmation a été correctement définie et qu'elle est correctement exécutée, le TRIBOT accomplit le comportement sensorimoteur désiré.

En configuration standard, la définition d'un comportement sensorimoteur est assujettie à deux limitations de taille. D'une part, la programmation du robot TRIBOT, via le logiciel LEGO MINDSTORMS NXT se révèle relativement fastidieuse. La probabilité d'une exécution correcte au premier essai est en effet très faible. La programmation du robot nécessite alors un travail de tâtonnement pour déterminer un comportement sensorimoteur exploitable. D'autre part, les comportements sensorimoteurs définis à l'aide du logiciel de programmation constituent des séquences pré-enregistrées qu'un événement imprévu peut le rendre inopérant.[6]

8.2.3 Programmation du TRIBOT via le langage URBI

Dans le cadre de notre expérimentation, nous avons choisi d'utiliser le langage URBI [Bai05] pour implémenter facilement les actions élémentaires du robot TRIBOT.

Édité par la société GOSTAI, URBI (*Universal Robotic Body Interface*) est un langage de script conçu pour fonctionner selon un mode client/serveur dans le but de contrôler un robot. L'idée générale est d'avoir un serveur URBI chargé et démarré sur le robot, et d'utiliser un client pour envoyer des commandes au robot, demander la valeur d'un descripteur et la recevoir et plus généralement, recevoir des messages provenant du robot et réagir de la manière la plus appropriée. Le langage URBI présente différentes qualités : il est à la fois simple (facile à comprendre et à utiliser), flexible (URBI est indépendant du robot et de l'OS), modulaire (architecture de composants objet interfaçable avec les langages *C++*, *Java*, *Python*, *etc.*) et réalise des traitements parallèles (exécution parallèle de commandes & gestion des accès concurrentiels).

Pour notre étude, nous utilisons la version dédiée aux MINDSTORMS du langage URBI.[7] Chaque partie matérielle du robot (capteur ou servomoteur), appelé *device*, est un objet et possède un nom, reporté dans le tableau 8.1. On commande alors facilement le robot en affectant et en récupérant des valeurs aux différentes devices du tribot. À titre d'exemple, pour lire la valeur courante du capteur d'ultrasons, on envoie la commande `sonar.val ;` au serveur URBI. La valeur courante du capteur d'ultrasons est alors renvoyée au client immédiatement

[6] Un événement imprévu peut en effet faire échouer une exécution si le programme correspondant n'est pas assez robuste.

[7] Cette version est gratuitement téléchargeable sur le site internet de la société GOSTAI, Url : *http ://www.gostai.com/*.

après que le serveur a réussi à accéder au capteur.

Le langage URBI permet de commander un robot à l'aide de fonctionnalités avancées, telles que les branchements conditionnels, les boucles et les capteurs d'événements. Dans le cadre de l'expérimentation TRIBOT, URBI nous sert cependant *simplement* à implémenter un ensemble d'actions élémentaires pour le robot. Nous limitons ainsi l'utilisation d'URBI à des commandes basiques aux actionneurs et aux descripteurs du TRIBOT, comme nous le verrons dans la section 8.3.

Device du TRIBOT	**Objet URBI associé**
capteur d'ultrasons	`sonar`
capteur photosensible	`light`
capteur de pression	`bumper`
capteur sonore	`decibel`
servomoteur de la roue droite	`wheelR`
servomoteur de la roue gauche	`wheelL`
servomoteur de la pince	`claw`

TAB. 8.1 – Tableau récapitulatif des objets URBI associés aux devices du TRIBOT.

Dans le cadre de notre expérimentation, le langage URBI est utilisé comme *interface* pour commander le TRIBOT, mais n'intervient en rien dans le processus de supervision d'un comportement sensorimoteur. Conformément à l'approche proposée au chapitre 7, c'est en effet notre architecture de supervision qui se charge de planifier une séquence d'actions élémentaires permettant la réalisation d'un comportement sensorimoteur. Une fois le processus de planification achevé, notre architecture supervise la réalisation de cette séquence pas à pas en commençant par la première action élémentaire de la séquence planifiée. Notre architecture en délègue l'exécution effective au serveur URBI (embarqué sur le TRIBOT), qui exécute alors les commandes aux actionneurs correspondant à cette première action élémentaire. Après l'exécution de celle-ci, le serveur URBI renvoie l'état courant du robot puis se met en attente. Notre architecture reprend alors la main et *décide* des éventuelles corrections à apporter à la la séquence d'actions élémentaires en cours d'exécution, conformément au cycle de supervision exposé dans la section 7.3.5 du chapitre 7. Le serveur URBI est à nouveau sollicité une fois terminée l'analyse du module de contrôle (*i.e.* analyse de l'écart réalité/prévisions, calcul des corrections mineures ou replanification éventuelle). Il se charge de l'exécution effective d'une nouvelle action élémentaire, puis rend la main à notre architecture une fois l'action achevée et se met une nouvelle fois en attente des décisions de l'architecture de supervision. Ce processus

continue ainsi jusqu'à la réalisation complète du comportement sensorimoteur.

8.2.4 Objectif de l'expérimentation TRIBOT

Au travers de l'étude du robot TRIBOT, nous souhaitons tester, en contexte réel, l'architecture de supervision proposée dans le chapitre 7. Dans un premier temps, nous utiliserons donc la plate-forme CONACQ pour modéliser automatiquement par apprentissage un ensemble d'actions élémentaires exécutables par le TRIBOT. Dans un second temps, nous utiliserons notre architecture de supervision pour constituer et superviser automatiquement un comportement sensorimoteur à partir des réseaux de contraintes précédemment acquis par CONACQ. Dans le cadre de notre expérimentation, le comportement sensorimoteur que le TRIBOT devra accomplir consistera à se saisir (à l'aide de sa pince) d'un mug situé à proximité du robot.

Au travers de cette expérimentation, nous souhaitons confronter notre architecture à la réalité opérationnelle d'un robot évoluant en environnement réel. En ce sens, la qualité moyenne[8] des servomoteurs et des capteurs du kit LEGO MINDSTORMS NXT nous permettront de tester réellement la robustesse de notre architecture face aux imprécisions de descripteurs et au mauvais fonctionnement d'un actionneur, qui sont inhérents à toute expérimentation en robotique. Nous perturberons par ailleurs volontairement l'exécution des comportements sensorimoteurs afin d'évaluer la capacité (et les performances) de notre architecture à replanifier un comportement lorsqu'il s'agit de s'adapter à une modification de contexte.

8.3 Actions élémentaires étudiées dans le cadre de l'expérimentation TRIBOT

Pour l'expérimentation TRIBOT, nous avons choisi de modéliser les 5 actions élémentaires suivantes : *Careful move*, *Standard move*, *Find target*, *Open claw* et *Close claw*. Les deux premières (*i.e. Careful move* & *Standard move*) implémentent des capacités de déplacement. L'action *Find target* implémente une fonction de détection, tandis que les deux dernières sont dédiées à la commande de la pince du robot. Ces cinq actions élémentaires encapsulent ainsi, chacune, une fonction de *base* du TRIBOT. Elles correspondent en ce sens à des actions élémentaires qui peuvent (doivent) être combinées pour constituer des comportements plus évolués, tels que la saisie d'un objet.

[8] Cette qualité moyenne étant liée au prix de vente relativement faible du kit LEGO MINDSTORMS NXT (prix public moyen : 299 euros).

Comme nous l'avons mentionné précédemment, ces cinq actions élémentaires sont implémenté à l'aide du langage URBI. Les paragraphes suivants décrivent précisément chacune de ces actions élémentaires, ainsi que le code URBI associé.

L'action élémentaire *Careful move* a_1

L'action élémentaire *Careful move* permet de faire avancer avec précision, mais à faible allure le robot TRIBOT. Le code URBI permettant d'implémenter cette action élémentaire est le suivant :

```
function global.carefulMove(angle) {
    move : {
       wheelL.val= wheelL.val+angle & wheelR.val= wheelR.val+angle ; wait(2s) ;
    };
    wait(0.2s) ;
} ;
```

Durant l'action élémentaire *Careful move*, les servomoteurs des roues du TRIBOT sont commandés en position, c'est-à-dire qu'ils doivent tourner jusqu'à une position donnée exprimée en degré. Les deux servomoteurs `wheelL` et `wheelR` sont ainsi sollicités pour tourner de l'angle de rotation `angle` fourni en paramètre de la fonction URBI et doivent atteindre leur nouvelle position en deux secondes. Cette dernière condition est imposée par la ligne de commande "`wait(2s) ;`". Par ailleurs, il convient de noter que nous avons étiqueté cet ensemble de commandes aux actionneurs à l'aide du drapeau `move` et nous avons implémenté une action réflexe,[9] exécutée en tâche de fond, permettant l'arrêt immédiat de l'exécution de ce bloc d'instructions (et donc l'arrêt immédiat du TRIBOT) dès que le robot touche un obstacle. Enfin, nous imposons un délai supplémentaire (*i.e.* `wait(0.2s) ;`) avant de rendre la main afin de ne pas perturber les mesures faites par les descripteurs. Ces mesures sont en effet complètement erronées[10] si elles sont effectuées immédiatement après le déplacement du robot ; le délai imposé permet ainsi une "stabilisation" du système avant la récupération des données capteurs.

L'action élémentaire *Standard move* a_2

L'action élémentaire *Standard move* implémente elle aussi une fonction de déplacement pour le TRIBOT. À ce titre, son exécution peut être interrompue par l'action réflexe précé-

[9] `at (bumper==1) {stop move ; wheels=0 ;} ;`

[10] À titre d'exemple, le capteur d'ultrasons peut ainsi indiquer une distance complètement erronée, parfois jusqu'à plus d'une vingtaine de centimètres en plus ou en moins que la distance réelle.

demment introduite. Par ailleurs, la commande des servomoteurs des roues gauche et droite est cette fois effectuée en vitesse. Il est par ailleurs à noter que la vitesse de déplacement du TRIBOT est plus élevée que dans le cas de l'action élémentaire *Careful move*, le déplacement effectué est cependant moins précis. Le code URBI associé à cette action élémentaire est le suivant :

```
function global.standardMove(speed) {
    move : { wheels = speed; wait(2s); wheels=0; };
    wait(0.2s);
};
```

L'action élémentaire *Find target* a_3

L'action élémentaire *Find target* (a_3) implémente une fonction de détection. Elle doit ainsi être exécutée lorsque l'on souhaite que le TRIBOT se place en face du premier objet qu'il détecte à l'aide de son capteur d'ultrasons.

```
function global.findTarget() {
    wheelL = -5 & wheelR = 5; waituntil(sonar <= distance); wheels=0;
    wheelL = -5 & wheelR = 5; wait(0.5s);
    wheels = 0; wait(0.2s);
};
```

Durant l'exécution de cette action élémentaire, le TRIBOT tourne sur lui-même (`wheelL = -5 & wheelR = 5;`) jusqu'à ce qu'il détecte un objet (`waituntil(sonar <= distance);`). Il continue alors sa rotation sur lui-même pendant une demi-seconde afin de se positionner parfaitement en face de cet objet. Comme les deux actions élémentaires précédentes, on ajoute un délai supplémentaire avant de rendre la main dans l'optique de ne pas perturber les données capteurs. Il convient par ailleurs de noter que cette action élémentaire se place *simplement* en face du premier objet détecté. Elle ne permet cependant pas de déterminer la nature de cet objet.[11] Dans le cadre de notre expérimentation, il sera donc essentiel d'assurer que le seul objet aux alentours du TRIBOT sera bien le mug à attraper.

Les actions élémentaires *Open Claw* a_4 & *Close claw* a_5

Comme leur nom l'indique, les actions *Open Claw* et *Close claw* permettent respectivement d'ouvrir et de fermer la pince du TRIBOT. Pour exécuter ces actions élémentaires, seul le servomoteur de la pince est sollicité. Le code URBI permettant d'implémenter ces deux actions élémentaires est le suivant :

[11] Le premier obstacle détecté est ainsi considéré comme la cible en face de laquelle le TRIBOT doit se placer ;

```
function global.openClaw() {
    claw=10 ; wait(1s) ;
    claw=0 ; wait(1s) ;
} ;
```

```
function global.closeClaw() {
    claw=-50 ; wait(0.5s) ;
    claw=0 ; wait(1.5s) ;
} ;
```

8.4 Acquisition automatique des actions élémentaires du robot TRIBOT sous la forme de réseaux de contraintes

Dans cette section, nous décrivons la première étape de l'expérimentation TRIBOT : l'acquisition automatique faite par la plate-forme CONACQ des cinq actions élémentaires du robot sous la forme de réseaux de contraintes.

8.4.1 Objectif de l'acquisition automatique

Comme nous l'avons vu dans la section 8.3, les actions élémentaires considérées dans le cadre de l'expérimentation TRIBOT correspondent à des capacités d'action et de perception *basiques*. Dans l'optique de planifier, à l'aide de ces actions élémentaires, des comportements sensorimoteurs, l'objectif de l'acquisition automatique des actions élémentaires sous la forme de réseaux de contraintes est double. L'acquisition automatique doit en effet :

1. Identifier le contexte et la plage d'utilisation de chaque action élémentaire. En d'autres termes, l'acquisition automatique doit permettre de déterminer le *domaine de viabilité* de chacune des actions élémentaires étudiées,
2. Déterminer les commandes aux actionneurs permettant d'exécuter ces actions élémentaires. À titre d'exemple, l'acquisition automatique de l'action élémentaire *Careful move* devra permettre de déterminer le lien existant entre l'angle de rotation fourni en paramètre de la fonction URBI et la distance que l'on souhaite faire parcourir au TRIBOT.

Dans la seconde partie de notre expérimentation (*c.f.* section 8.5), notre architecture de contrôle sera amenée à manipuler les réseaux de contraintes acquis pour planifier et superviser l'exécution du comportement sensorimoteur de saisie d'un objet. En ce sens, le processus de planification et de supervision fournira une *mesure de qualité* des réseaux de contraintes acquis. En effet, il permettra de déterminer si la modélisation CSP établie par l'acquisition automatique permet de combiner convenablement entre elles les différentes actions élémentaires du TRIBOT dans l'optique de faire accomplir à ce dernier un comportement plus évolué.

8.4.2 Format des données d'entraînement fournies à CONACQ

Pour modéliser les actions élémentaires précédemment citées, nous avons utilisé les descripteurs suivants : le capteur d'ultrasons indique la distance d séparant le TRIBOT du mug à saisir. Nous utilisons le capteur de pression pour déterminer si le robot est en contact avec le mug ($b = 1$ dans ce cas, 0 sinon) et c est un descripteur booléen valant 1 si et seulement si la pince du TRIBOT est ouverte.[12] Dans un souci de clarté, le capteur photosensible et le capteur sonore n'ont pas été pris en compte durant l'expérimentation. Enfin, les trois actionneurs du TRIBOT sont nommés de la manière suivante. α_R (*resp.* α_L) correspond au servomoteur de la roue droite (*resp.* gauche) du robot et α_C au servomoteur utilisé pour ouvrir et fermer la pince du TRIBOT.

Il convient par ailleurs de noter que les servomoteurs du TRIBOT et les descripteurs utilisés dans le cadre de notre expérimentation sont à valeur entière. Ainsi, contrairement à l'expérimentation TWIG (*c.f.* section 8.1.1), il n'a pas été nécessaire de discrétiser les variables étudiées.

Dans le cadre de cette expérimentation, chacune de ces données d'entraînement fournie à la plate-forme CONACQ est un tuple étiqueté (positivement ou négativement) de la forme $(d^I, b^I, c^I, \alpha_R, \alpha_L, \alpha_C, d^F, b^F, c^F)$ où les exposants indiquent l'état des descripteurs du robot au début et à la fin de l'exécution étudiée.

8.4.3 Acquisition automatique de l'action élémentaire *Careful move*

La modélisation des actions élémentaires au moyen de la plate-forme CONACQ a débuté par l'étude de l'action élémentaire *Careful move*, qui doit être exécutée lorsque l'on souhaite faire avancer le TRIBOT doucement sur une distance précise.

L'ensemble des données d'entraînement fournies CONACQ est reporté dans le tableau 8.2. Dans l'optique de fournir des données d'entraînement représentatives de l'action élémentaire étudiée, la majorité des instances fournies à CONACQ est le résultat d'exécutions réelles effectuées sur le robot TRIBOT. Seules les instances négatives e_6^-, e_8^- et e_{10}^- ont été ajoutées manuellement car elles ne peuvent résulter d'une exécution réelle, mais permettent cependant de modéliser plus précisément l'action élémentaire *Careful move*. e_6^- et e_{10}^- caractérisent ainsi la modification de l'ouverture de la pince du robot durant l'exécution de a_1, sans que le servomoteur associé ne soit sollicité. L'instance négative e_8^- retrace quant à elle un mauvais fonctionnement des servomoteurs ou du capteur de distance du TRIBOT, dans la mesure où la distance parcourue (10 cm) ne correspond pas à la commande imposée aux actionneurs du

[12] Cette information est déduite directement de l'angle de rotation du servomoteur de la pince.

robot.

	d^I	b^I	c^I	α_R	α_L	α_C	d^F	b^F	c^F
e_1^+	80	0	1	240	240	0	70	0	1
e_2^-	120	0	1	240	240	0	110	0	1
e_3^+	50	0	1	24	24	0	49	0	1
e_4^-	60	0	1	24	24	10	59	0	1
e_5^+	15	0	1	48	48	0	13	0	1
e_6^-	70	0	0	120	120	0	65	0	1
e_7^+	20	0	1	72	72	0	17	0	1
e_8^-	50	0	1	200	200	0	40	0	1
e_9^+	35	0	1	120	120	0	30	0	1
e_{10}^-	80	0	1	24	24	0	79	0	0
e_{11}^-	101	0	1	240	240	0	91	0	1
e_{12}^+	8	0	1	24	24	0	7	0	1
e_{13}^-	0	0	1	0	0	0	0	0	1

TAB. 8.2 – Données d'entraînement fournies à CONACQ pour la modélisation de l'action élémentaire *Careful move*.

À partir de cette quinzaine d'instances, la plate-forme a automatiquement construit, par apprentissage, le réseau de contraintes suivant :

$$a_1 \textbf{ : Careful move}$$

$$\begin{cases} Pre(a_1) = \{(d^I < 100), (b^I = 0), (c^I = 1)\} \\ \\ Actuators(a_1) = \{(\alpha_R = \alpha_L = 24 \times (d^I - d^F)), (\alpha_C = 0)\} \\ \\ Post(a_1) = \{(d^I - 10 \leq d^F \leq d^I - 1), (d^F = 7 \Rightarrow b^F = 1), (c^F = c^I)\} \end{cases}$$

Ce réseau de contraintes modélise précisément l'action élémentaire *Careful move* a_1, dont les principales caractéristiques suivent : a_1 peut être exécutée si et seulement si la distance séparant le TRIBOT du mug est inférieure ou égale à 1 mètre (*i.e.* $(d^I < 100)$) et si et seulement si la pince du robot est ouverte (*i.e.* $(c^I = 1)$). La contrainte $(\alpha_C = 0)$ modélise ensuite le fait que le servomoteur α_C, qui permet l'ouverture et la fermeture de la pince du robot, n'est pas sollicité durant l'exécution de l'action élémentaire *Careful move*. La contrainte $(c^F = c^I)$ modélise le fait que la pince du TRIBOT reste ouverte durant l'exécution de cette action élémentaire. Durant l'exécution de l'action élémentaire *Careful move*, les servomoteurs α_R et α_L

sont par ailleurs contrôlés en position. La contrainte $(\alpha_R = \alpha_L = 24 \times (d^I - d^F))$ modélise alors le fait que les servomoteurs α_R et α_L doivent tourner jusqu'à atteindre une position égale à $24 \times d$ pour que le robot TRIBOT parcoure une distance d. De plus, les contraintes de $Post(a_1)$ impliquant d^F (*i.e.* $(d^I - 10 \leq d^F \leq d^I - 1)$) modélisent le fait que la distance parcourue par le tribot lors de l'action élémentaire *Careful move* est comprise entre 1 et 10 cm. Enfin, la contrainte $(d^F = 7 \Rightarrow b^F = 1)$ modélise correctement le fait que le TRIBOT est en contact avec le mug à la fin de l'exécution de l'action élémentaire a_1 si la valeur finale du capteur d'ultrasons (*i.e.* la distance finale) est égale à 7 cm.[13]

8.4.4 Acquisition automatique des autres actions élémentaires de TRIBOT

En utilisant le même protocole d'expérimentation, les quatre autres actions élémentaires du robot TRIBOT ont été automatiquement modélisées par la plate-forme CONACQ sous la forme des réseaux de contraintes suivants :

a_2 : Standard move

$$\begin{cases} Pre(a_2) = \{(10 < d^I < 100), (b^I = 0)\} \\ Actuators(a_2) = \{(\alpha_R = \lceil (d^I - d^F)/0.9 \rceil), (\alpha_C = \alpha_R), (\alpha_C = 0)\} \\ Post(a_2) = \{(d^I - 36 \leq d^F \leq d^I - 27), (d^F > 10), (b^F = 0), (c^F = c^I)\} \end{cases}$$

a_3 : Find target

$$\begin{cases} Pre(a_3) = \{(d^I > 100), (b^I = 0)\} \\ Actuators(a_3) = \{(\alpha_R = 5), (\alpha_L = -5), (\alpha_C = 0)\} \\ Post(a_3) = \{(d^F \leq 100), (c^F = c^I)\} \end{cases}$$

a_4 : Open claw

$$\begin{cases} Pre(a_4) = \{(d^I > 10), (b^I = 0), (c^I = 0)\} \\ Actuators(a_4) = \{(\alpha_R = \alpha_L = 0), (\alpha_C = 10)\} \\ Post(a_4) = \{(d^F = d^I), (b^F = b^I), (c^F = 1)\} \end{cases}$$

a_5 : Close claw

$$\begin{cases} Pre(a_5) = \{(d^I < 100), (c^I = 1)\} \\ Actuators(a_5) = \{(\alpha_R = \alpha_L = 0), (\alpha_C = -50)\} \\ Post(a_5) = \{(d^F = d^I), (d^F = 7 \Rightarrow b^F = 1), (c^F = 0)\} \end{cases}$$

8.5 Planification et supervision d'un comportement sensorimoteur : saisie d'un objet par TRIBOT

Dans cette section, nous présentons la seconde étape de l'expérimentation TRIBOT : la planification et la supervision d'un comportement sensorimoteur permettant la saisie d'un mug situé à proximité du TRIBOT.

[13] Le capteur d'ultrasons se trouve en effet en léger retrait (environ $7cm$) par rapport au capteur de contact.

Au travers de cette expérimentation, nous testons d'une part la capacité de notre architecture à combiner des actions élémentaires modélisées sous la forme de réseaux de contraintes. D'autre part, cette expérimentation doit nous permettre, soit de valider la modélisation CSP des actions élémentaires établie par CONACQ, soit de détecter des erreurs de modélisation et de les corriger afin d'obtenir une modélisation *fidèle* des actions élémentaires du TRIBOT.[14] Nous avons pour cela procédé à deux séries d'expérimentations.

8.5.1 Volet expérimental 1 : Saisie d'un objet *sans* perturbation extérieure volontaire

Dans la première série d'expérimentations, le mug est positionné en face du TRIBOT, pince fermée, à une distance n'excédant pas 1 mètre. Pour accomplir le comportement sensorimoteur consistant à se saisir du mug, la séquence *type* d'actions élémentaires planifiée par notre architecture de supervision se présente de la manière suivante : la séquence débute tout d'abord par l'action élémentaire *Open claw* et se poursuit par un certain nombre de *Standard move* (en fonction de la distance séparant le TRIBOT du mug). L'action élémentaire *Careful move* doit être exécutée afin que le robot touche le mug. Enfin, la séquence se termine par l'exécution de l'action *Close claw*. Durant cette première série d'expérimentations, le comportement sensorimoteur de saisie du mug est accompli en effectuant exclusivement des corrections mineures (*i.e.* aucune replanification nécessaire) afin d'adapter la séquence établie par planification à l'exécution réelle et effective du comportement.

La figure 8.4 retranscrit une trace d'exécution du comportement sensorimoteur de saisie du mug par le TRIBOT, réalisée dans le cadre de cette première série d'expérimentations. Les lignes 1 à 8 permettent de visualiser l'initialisation du serveur URBI. Une fois notre architecture connectée à ce dernier, le processus de supervision commence en récupérant l'état initial du robot (ligne 9) : le TRIBOT se trouve pince fermée, en face du mug, à une distance de 52 cm. Notre architecture propose alors en ligne 10 la séquence d'actions élémentaires $\mathcal{S} = \{a_4, a_2, a_1, a_5\}$ pour se saisir du mug. L'exécution effective de l'ouverture de la pince (action élémentaire a_4, lignes 11 à 15) est conforme aux prévisions établies. En conséquence, on continue l'exécution de la séquence $\mathcal{S}$ telle qu'elle a été initialement calculée. L'exécution de l'action élémentaire *Standard move* (lignes 16 à 20), prévue pour faire parcourir 36 cm au TRIBOT, permet à ce dernier de se déplacer d'une distance légèrement plus réduite. L'action élémentaire suivante (*i.e. Careful move*), initialement prévue pour parcourir 9 cm, est

[14] Afin de garantir, *in fine*, une planification sémantiquement et opérationnellement correcte.

en conséquence ajustée (ligne 20) par propagation dans l'optique de "rattraper" le retard. À l'issue de son exécution, notre architecture détecte un écart entre l'état courant et les prévisions (ligne 25). Il s'agit en fait d'une imprécision du capteur d'ultrasons. Le TRIBOT étant cependant en contact avec le mug (*i.e.* $bumper = 1$), l'action élémentaire *Close claw* peut être exécutée, ce qui finalise la réalisation du comportement sensorimoteur de saisie du mug (ligne 30).

```
1  Connected to 127.0.0.1.
2  *** ********************************************************
3  *** URBI Language specif 1.0 - Copyright 2006-2007 Gostai SAS
4  *** URBI Kernel version 1.0 rev.979
5  *** URBI Engine version 1.0 rev. 232
6  *** (C) 2005-2007 Gostai SAS
7  *** ********************************************************
8  *** ID: U11870800
9  Initial State : Desc\_sonar = 52 Desc\_bumper = 0 Desc\_clawAngle = 0
10       Séquence proposée : {a4  a2  a1  a5  }
11 t = 0 > Début exécution a4 global.openclaw()
12 t = 2 > Fin   exécution a4 global.openclaw()
13    Etat courant : (52 ; 0 ; 1)
14    Prévisions :   (52 ; 0 ; 1)
15       (Etat courant == Prévisions) => Continuation
16 t = 2 > Début exécution a2 global.standardMove(40)
17 t = 4 > Fin   exécution a2 global.standardMove(40)
18    Etat courant : (17 ; 0 ; 1)
19    Prévisions :   (16 ; 0 ; 1)
20       Ecart mineur => Correction mineure...
21 t = 4 > Début exécution a1 global.carefulMove(240)
22 t = 6 > Fin   exécution a1 global.carefulMove(240)
23    Etat courant : (6 ; 1 ; 1)
24    Prévisions :   (7 ; 0 ; 1)
25       Ecart mineur => Correction mineure...
26 t = 6 > Début exécution a5 global.closeClaw()
27 t = 8 > Fin   exécution a5 global.closeClaw()
28    Etat courant : (7 ; 1 ; 0)
29    Prévisions :   (6 ; 1 ; 0)
30       Etat courant compatible avec but => Expérimentation menée avec succès!!!
```

FIG. 8.4 – Trace d'une exécution du comportement sensorimoteur de saisie du mug sans perturbation extérieure volontaire.

Comme l'illustre l'exécution précédemment commentée, il est à noter que les écarts entre la réalité et les prévisions sont systématiquement corrigés par *ajustement* des actions élémentaires suivantes, via les propriétés de propagation des CSPs (acquis par CONACQ) manipulés

par notre architecture de supervision. Par ailleurs, une analyse de la trace de la figure 8.4 nous indique que les temps de calcul correspondant à ces corrections mineures ne ralentissent en rien l'exécution du comportement sensorimoteur de saisie du mug.

8.5.2 Volet expérimental 2 : Saisie d'un objet *avec* perturbations extérieures volontaires

Durant la seconde série d'expérimentations, nous perturbons volontairement l'exécution du comportement sensorimoteur afin de tester le cycle de replanification de notre architecture de supervision. Les conditions initiales sont les mêmes que précédemment (*i.e.* pince fermée, mug situé en face du robot à moins d'un mètre) et le TRIBOT planifie en conséquence le même type de séquence que celle établie dans la première série d'expérimentations. Nous déplaçons cependant volontairement le mug durant l'exécution d'une séquences d'actions élémentaires, ce qui force une replanification. En fonction de l'état courant du TRIBOT, notre planificateur de tâches planifie alors une nouvelle séquence d'actions élémentaires, qui débute par l'action élémentaire *Find target*. La nouvelle séquence est alors supervisée par le module de contrôle de notre architecture, qui adapte son exécution en fonction d'éventuels écarts entre la réalité et les prévisions établies à l'aide de la modélisation CSP.

La figure 8.5 retranscrit une trace d'exécution réalisée dans le cadre de la seconde série d'expérimentations. Les lignes 1 à 15, qui correspondent à l'initialisation du serveur URBI et à l'exécution de l'action élémentaire *Open claw*, sont similaires à celles reportées dans la figure 8.4. Nous déplaçons ensuite volontairement le mug durant l'exécution de l'action élémentaire *Standard move* (lignes 16-17). L'état courant diffère alors totalement des prévisions établies et une replanification est nécessaire (ligne 21). Comme nous le mentionnions précédemment, la séquence d'actions élémentaires issue de cette replanification débute par l'action *Find target*. Une fois le TRIBOT à nouveau en face du mug (ligne 24), notre architecture planifie[15] une nouvelle séquence d'actions élémentaires dont elle supervise l'exécution jusqu'à la saisie effective du mug par le TRIBOT(lignes 28 à 42).

Comme nous pouvons le visualiser sur la vidéo, le robot TRIBOT réussit parfaitement à saisir le mug, même lorsque ce dernier est déplacé à plusieurs reprises durant l'exécution du comportement sensorimoteur. Au travers de cette seconde série d'expérimentations, nous avons par ailleurs remarqué que la boucle de replanification n'est exploitée qu'en cas de pertur-

[15]Une replanification est ici nécéessaire car, contrairement aux prévisions établies, le TRIBOT n'est pas en contact avec le mug à l'issue de l'exécution de *Find target*.

```
1  Connected to 127.0.0.1.
2  *** ************************************************************
3  *** URBI Language specif 1.0 - Copyright 2006-2007 Gostai SAS
4  *** URBI Kernel version 1.0 rev.979
5  *** URBI Engine version 1.0 rev. 232
6  *** (C) 2005-2007 Gostai SAS
7  *** ************************************************************
8  *** ID: U11870800
9  Initial State : Desc\_sonar = 60 Desc\_bumper = 0 Desc\_clawAngle = 0
10       Séquence proposée : {a4  a2  a1 a1  a5  }
11 t = 0 > Début exécution a4 global.openclaw()
12 t = 2 > Fin   exécution a4 global.openclaw()
13    Etat courant : (60 ; 0 ; 1)
14    Prévisions :   (60 ; 0 ; 1)
15       (Etat courant == Prévisions) => Continuation
16 t = 2 > Début exécution a2 global.standardMove(40)
17 t = 4 > Fin   exécution a2 global.standardMove(40)
18    Etat courant : (255 ; 0 ; 1)
19    Prévisions :   (24 ; 0 ; 1)
20       Ecart majeur => Replanification...
21       Séquence proposée : {a3 a5  }
22 t = 4 > Début exécution a3 global.findTarget()
23 t = 9 > Fin   exécution a3 global.findTarget()
24    Etat courant : (43 ; 0 ; 1)
25    Prévisions :   ( 0 ; 0 ; 1)
26       Ecart majeur => Replanification...
27 Séquence proposée : {a2 a1 a5  }
28 t = 9 > Début exécution a2 global.standardMove(30)
29 t = 11 > Fin   exécution a2 global.standardMove(30)
30    Etat courant : (14 ; 0 ; 1)
31    Prévisions :   (16 ; 0 ; 1)
32       Ecart mineur => Correction mineure...
33 t = 11 > Début exécution a1 global.carefulMove(168)
34 t = 13 > Fin   exécution a1 global.careful(168)
35    Etat courant : (7 ; 1 ; 1)
36    Prévisions :   (7 ; 0 ; 1)
37       Ecart mineur => Correction mineure...
38 t = 13 > Début exécution a5 global.closeClaw()
39 t = 15 > Fin   exécution a5 global.closeClaw()
40    Etat courant : (7 ; 1 ; 0)
41    Prévisions :   (7 ; 1 ; 0)
42       Etat courant compatible avec but => Expérimentation menée avec succès!!!
```

FIG. 8.5 – Trace d'une exécution du comportement sensorimoteur de saisie du mug avec perturbation extérieure volontaire.

bations volontaires ; les légers disfonctionnements des servomoteurs du TRIBOT ou les erreurs de mesure des capteurs étant "rattrapés" par corrections mineures. À l'image des temps de calcul correspondant à des corrections mineures, il s'avère que le processus de replanification ne ralentit pas l'exécution du comportement sensorimoteur. Enfin, il convient de noter qu'aucune erreur de modélisation n'a pour l'heure été identifiée ; l'acquisition automatique réalisée par CONACQ semble en ce sens fiable et appropriée.

8.6 Synthèse

Les expérimentations présentées dans ce chapitre nous ont permis de confronter l'architecture proposée dans la chapitre 7 à la réalité opérationnelle de la robotique.

Les expérimentations préliminaires présentées dans la section 8.1 ont tout d'abord permis de lever les verrous techniques liés à notre approche. L'étude du robot TWIG a dans un premier temps permis de valider la capacité de la plate-forme CONACQ à abstraire automatiquement par apprentissage les actions élémentaires d'un robot sous la forme de réseaux de contraintes. Dans un second temps, l'expérimentation réalisée sur le robot PIRI a permis de démontrer la capacité du résolveur de contraintes CHOCO à commander un robot cadencé à une fréquence élevée.

L'expérimentation TRIBOT a quant à elle permis de tester notre architecture de supervision en contexte réel. Nous avons dans un premier temps utilisé le langage URBI pour implémenter facilement 5 actions élémentaires exécutables par le TRIBOT (section 8.3). Dans un deuxième temps (section 8.4), nous avons utilisé la plate-forme CONACQ pour modéliser automatiquement par apprentissage 5 actions élémentaires exécutables par le robot TRIBOT. Enfin (section 8.5), notre architecture a planifié, via notre planificateur de tâches inspiré de CSP-PLAN, une séquence d'actions élémentaires dont l'exécution, supervisée par notre architecture, a permis d'accomplir le comportement sensorimoteur de saisie d'un objet.

Chapitre 9

Conclusion et perspectives

Dans la seconde partie de cette thèse, nous nous sommes intéressés à une utilisation pratique de l'acquisition automatique de réseaux de contraintes en proposant une architecture logicielle qui vise à automatiser la planification de comportements sensorimoteurs en robotique. Notre architecture utilise l'acquisition automatique de réseaux de contraintes pour abstraire *automatiquement* par apprentissage les actions élémentaires d'un robot sous la forme de réseaux de contraintes. Une fois ce processus de modélisation achevé, un planificateur de tâches inspiré de CSP-Plan [LB03] combine les réseaux de contraintes acquis pour définir une séquence d'actions élémentaires dont l'exécution doit permettre au robot d'accomplir un comportement sensorimoteur. Cette exécution est alors supervisée par un module de contrôle, qui utilise les propriétés de propagation des CSPs acquis pour ajuster la séquence lorsque les écarts entre son exécution effective et les prévisions sont mineurs. Une boucle de replanification est par ailleurs déployée en cas d'écarts significatifs.

Les expérimentations menées dans le cadre de la seconde partie de cette thèse (*c.f.* chapitre 8) nous ont permis de confronter notre architecture aux exigences de l'exploitation de robots en conditions réelles. Les résultats obtenus nous permettent désormais de mettre en lumière la portée et les limites actuelles de l'approche que nous proposons, ainsi que de dégager quelques perspectives de travail.

Contribution

L'expérimentation réalisée sur le robot Tribot Mindstorms NXT nous permet de dégager deux enseignements principaux concernant l'architecture logicielle que nous avons proposée dans la seconde partie de nos travaux :

1. D'une part, l'expérimentation Tribot nous a montré la capacité de la plate-forme Conacq à modéliser automatiquement par apprentissage les actions élémentaires d'un ro-

bot réel sous la forme de réseaux de contraintes représentant fidèlement[1] et efficacement (*c.f.* item suivant) ces actions élémentaires.

2. D'autre part, l'étude de TRIBOT nous a permis de valider la capacité de notre architecture à planifier et à superviser l'exécution d'un comportement sensorimoteur. Les réseaux de contraintes acquis se combinent en effet aisément entre eux pour constituer un comportement sensorimoteur effectif et robuste aux perturbations. Il est par ailleurs à noter que les incertitudes inhérentes à toute application en robotique, telles que le mauvais fonctionnement d'un actionneur ou la saturation d'un capteur, ont été parfaitement maîtrisées et corrigées en exploitant pleinement les propriétés de propagation des réseaux de contraintes acquis. La boucle de replanification n'a quant à elle été exploitée qu'en cas d'écart significatif - engendré par une perturbation extérieure volontaire - entre les prévisions établies via la modélisation CSP et l'exécution effective du robot. Enfin, les temps de calcul nécessaires aux mécanismes d'ajustements ainsi que les temps de calcul nécessaires à la production d'un nouveau plan n'ont ralenti en rien l'exécution du comportement sensorimoteur du robot TRIBOT.

En ce sens, les résultats obtenus dans le cadre de nos expérimentations répondent à la motivation initiale de la seconde partie de cette thèse : une utilisation *pratique* de l'apprentissage automatique de réseaux de contraintes visant à automatiser efficacement le processus de constitution de comportements sensorimoteurs en robotique.

Limites actuelles de l'approche proposée

Dans cette section, nous souhaitons cependant mettre en lumière les deux principales limites actuelles de notre architecture logicielle.

La première limitation de notre approche est relative au passage à l'échelle. En effet, le robot TRIBOT étudié dans le chapitre 8 est un robot jouet ayant un nombre limité de degrés de liberté et de descripteurs, et ne pouvant exécuter qu'un nombre très limité d'actions élémentaires. Ainsi, bien que les performances de notre architecture soient pour l'heure parfaitement compatibles avec les expérimentations que nous avons menées, des travaux portant à la fois sur les aspects conceptuels de notre approche ainsi que sur sa mise en œuvre pratique devront être menés dans l'optique de répondre efficacement à la réalité opérationnelle de robots plus complexes.

La seconde limitation actuelle de notre approche réside dans le fait que l'approche proposée dans la seconde partie de cette thèse s'est limitée à la modélisation et à la planification

[1] Il est en effet à noter qu'aucune erreur de modélisation n'a pour l'heure été identifiée.

d'actions élémentaires exécutées de manière séquentielle. Les robots actuels sont cependant généralement conçus de manière à pouvoir exécuter des actions en parallèle.[2] Il conviendra donc d'étendre notre approche en conséquence, en déployant notamment d'autres mécanismes d'apprentissage (capables de modéliser les exclusions mutuelles entre actions élémentaires) ainsi qu'en étendant notre algorithme de planification à la manipulation d'actions exécutables en parallèle. D'un point de vue général, comme en témoigne l'ouvrage [GNT04], la planification de tâches regroupe aujourd'hui un large panel de techniques et d'approches. Aussi, bien que le recours à une planification classique se soit révélé adéquat dans le cadre de nos expérimentations, il n'est pas exclu de recourir à d'autres approches à l'avenir.

Perspectives

En plus des réflexions précédemment citées, et bien que les performances de notre architecture soient pour l'heure parfaitement compatibles avec nos expérimentations, nous envisageons dès à présent des améliorations concernant trois aspects distincts de notre approche : les performances du processus de planification, les performances du module de contrôle et enfin l'expressivité des réseaux de contraintes manipulés par notre architecture.

L'implémentation actuelle de notre planificateur de tâches est une extension du Base-CSP construit par l'algorithme CSP-Plan. Nous envisageons donc de l'améliorer en implémentant l'ensemble des transformations proposées dans [LB03] pour augmenter les performances de CSP-Plan. Par ailleurs, il convient de noter qu'il peut exister plusieurs séquences d'actions élémentaires permettant d'accomplir un même comportement sensorimoteur. Il nous paraît en ce sens intéressant de réfléchir à l'utilisation d'un module de gestion de la planification qui, étant données plusieurs séquences d'actions élémentaires permettant d'accomplir un même comportement sensorimoteur, serait capable de "préférer" une séquence à une autre. Le recours à des techniques d'apprentissage par renforcement [WD92] [SB98] pourrait alors être étudié, en gardant cependant pour objectif principal le déploiement d'approches efficaces en termes de temps de calcul.

Actuellement, l'analyse de l'écart entre l'état courant du robot et les prévisions - permettant de déterminer les corrections mineures ou de demander une replanification - s'effectue via un redémarrage complet du résolveur de contraintes (*i.e.* modélisation du CSP-global correspondant à la séquence restant à exécuter, puis résolution de ce CSP-global). À terme, nous envisageons donc d'étudier la façon de modifier le module de contrôle de notre architecture de manière à réaliser cette opération de manière incrémentale.

[2] Il est par ailleurs à noter que certaines actions élémentaires du Tribot pourraient être exécutées en parallèle comme, par exemple, ouvrir la pince et se déplacer.

Enfin, comme nous l'avons mentionné à plusieurs reprises dans nos propos, l'expressivité des réseaux de contraintes acquis et manipulés par notre architecture de contrôle constitue une caractéristique essentielle au succès de notre approche. Ils doivent en ce sens représenter le plus fidèlement et le plus efficacement possible les actions élémentaires. Nous étudierons donc avec une grande attention les réseaux de contraintes hybrides, qui mettent à la fois en jeu des variables à valeurs discrètes et des variables à valeurs continues. Les CSPs hybrides pourraient en effet permettre d'obtenir un meilleur modèle de chaque action élémentaire (au regard des théories du contrôle et de la commande), et ainsi améliorer les comportements sensorimoteurs obtenus par planification. Il convient cependant de noter qu'à ce jour, la plate-forme CONACQ ne permet pas l'acquisition de réseau de contraintes hybrides et que les performances des résolveurs de CSPs hybrides actuels sont prohibitives dans notre cadre de travail.

Chapitre 10

Conclusion générale

Dans le cadre de cette thèse, nous nous sommes intéressés à l'acquisition automatique de réseau de contraintes, dont l'objectif consiste à développer des solutions capables d'aider un utilisateur à modéliser un problème combinatoire sous la forme d'un réseau de contraintes. Notre travail s'est plus précisément focalisé sur la plate-forme d'acquisition automatique de réseau de contraintes CONACQ.

Le travail effectué dans la première partie de cette thèse s'est attaché à développer une version *interactive* de la plate-forme CONACQ. Dans cette optique, nous avons proposé différentes stratégies de questionnement présentées dans le chapitre 3. Au cours du processus d'apprentissage, CONACQ est désormais capable de poser à l'utilisateur des questions dont le but est d'augmenter plus rapidement et de manière conséquente la connaissance de la plate-forme. L'évaluation empirique des différentes stratégies de questionnement a permis d'établir que le recours à la stratégie *optimale-en-espérance* diminue drastiquement le nombre d'instances nécessaires à la convergence du processus d'apprentissage, notamment lorsque le problème cible est structuré, ce qui est souvent le cas des problèmes réels.

Dans la seconde partie de cette thèse, nous nous sommes intéressés à une utilisation pratique de l'apprentissage automatique de réseau de contraintes dans le domaine de la robotique. Dans cette optique, l'approche proposée dans le chapitre 7 consiste à utiliser la plate-forme d'acquisition automatique CONACQ pour modéliser automatiquement, sous la forme de réseaux de contraintes, les actions élémentaires d'un robot. Les réseaux de contraintes ainsi acquis sont ensuite combinés par planification pour définir automatiquement une séquence d'actions élémentaires dont l'exécution doit permettre au robot d'accomplir un comportement sensorimoteur. Les résultats expérimentaux présentés dans le chapitre 8 ont par ailleurs permis de démontrer la capacité de notre architecture, d'une part à abstraire automatiquement, grâce à CONACQ, les actions élémentaires d'un robot sous la forme de réseaux de contraintes, et d'autre part à planifier et à superviser efficacement l'exécution d'un comportement sensorimo-

teur, en maîtrisant parfaitement les incertitudes inhérentes à toute application en robotique. Le travail proposé dans la seconde partie de cette thèse constitue cependant une étude très prospective qu'il conviendra de poursuivre dans l'optique de déployer l'architecture proposée sur des robots plus complexes.

Bibliographie

[ACF+98] R. Alami, R. Chatila, S. Fleury, M. Ghallab, and F. Ingrand. An Architecture for Autonomy. *International Journal of Robotics Research*, 17(4) :315–337, April 1998.

[AFM02] J. Amilhastre, H. Fargier, and P. Marquis. Consistency restoration and explanations in dynamic CSPs-Application to configuration. *Artificial Intelligence*, 135(1-2) :199–234, February 2002.

[Alb99] James S. Albus. 4-D/RCS : A Reference Model Architecture for Unmanned Ground Vehicles. In *Proceedings of the SPIE AeroSense Technical Conference 3693*, Orlando, April 1999.

[ALFW97] J.S. Albus, R. Lumia, J. Fiala, and A. Wavening. Nasrem : The nasa/nbs standard reference model for telerobot control system architecture. Technical report, Robot System Division, National Institute of Standards and Technologies, 1997.

[Ang04] Dana Angluin. Queries revisited. In *Theoritical Computer Science*, volume 313, pages 175–194, February 2004.

[Ard07] Ardence. Software Platforms for the On-Demand World. Url : *http ://www.ardence.com/*, 2007.

[Ark98] Ronald C. Arkin. *Behavior-Based Robotics*. A Bradford Book, The MIT Press, 1998.

[Bai05] Jean-Christophe Baillie. URBI : Towards a Universal Robotic Low-Level Programming Language. In *Proceedings of IROS'05 (International Conference on Intelligent Robots and Systems)*, pages 820–825, August 2005.

[BCFO04] C. Bessiere, R. Coletta, E.C. Freuder, and B. O'Sullivan. Leveraging the Learning Power of Examples in Automated Constraint Acquisition. In *Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming (CP'04)*, pages 123–137, Toronto, Canada, September 2004.

[BCKO05] C. Bessiere, R. Coletta, F. Koriche, and B. O'Sullivan. A SAT-Based Version Space Algorithm for Acquiring Constraint Satisfaction Problems. In *Proceedings*

of the 16th European Conference on Machine Learning (ECML'05), pages 747–751, Porto, Portugal, October 2005.

[BCO+03] C. Bessiere, R. Coletta, B. O'Sullivan, E.C. Freuder, S. O'Connell, and J. Quinqueton. Semi-Automatic Modeling by Constraint Acquisition. In *Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming (CP'03)*, pages 812–816, Springer Kinsale, Cork, Ireland, September 2003.

[BCOP07] C. Bessiere, R. Coletta, B. O'Sullivan, and M. Paulin. Query-driven Constraint Acquisition. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI'07)*, pages 50–55, Hyderabad, India, January 2007.

[BCP07] C. Bessiere, R. Coletta, and T. Petit. Learning Implied Global Constraints. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI'07)*, pages 44–49, 2007.

[BD08] N. Beldiceanu and S. Demassey. Global Constraints Catalog. Url : *http ://www.emn.fr/x-info/sdemasse/gccat/index.html*, 2008.

[Bes91] Christian Bessière. Arc-Consistency in Dynamic Constraint Satisfaction Problems. In *AAAI*, pages 221–226, 1991.

[BFR95] C. Bessiere, E.C. Freuder, and J.C. Regin. Using Inference to Reduce Arc Consistency Computation. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI'95)*, pages 592–599, Montreal, Canada, August 1995.

[BGLC04] F. Benhamou, F. Goualard, E. Languénou, and M. Christie. Interval constraint solving for camera control and motion planning. *ACM Trans. Comput. Logic*, 5(4) :732–767, 2004.

[BH03] C. Bessiere and P. Van Hentenryck. To be or not to be...a global constraint. In *Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming (CP'03)*, pages 789–794, Springer Kinsale, Cork, Ireland, September 2003.

[BLV06] M. Benedetti, A. Lallouet, and J. Vautard. Reusing CSP propagators for QCSPs. In *Proceedings of the 11th Annual ERCIM Workshop on Constraint Solving and Constraint Programming (CSCLP)*, pages 201–215, Lisbonne, Portugal, June 2006.

[BM02] L. Bordeaux and E. MontFroy. Beyond NP : Arc-Consistency for Quantified Constraints. In *Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming (CP'02)*, pages 371–386, Cornell University, Ithaca, NY, USA, September 2002.

[BR01] C. Bessiere and J.C. Regin. Refining the Basic Constraint Propagation Algorithm. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI'01)*, pages 309–315, Seattle, Washington, USA, August 2001.

[Bro86] Rodney Brooks. A Robust Layered Control System for a Mobile Robot. *IEEE Journal of Robotics and Automation*, 2(1) :14–23, March 1986.

[Bro89] Rodney Brooks. A robot that walks : Emergent behavior from a carefully evolved network. *Neural Computation*, 1 :253 – 262, 1989.

[BVC07] G. Beaumet, G. Verfaillie, and M.C. Charmeau. Estimation of the Minimal Duration of an Attitude Change for an Autonomous Agile Earth-Observing Satellite. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming (CP'07)*, pages 3–17, Providence, USA, September 2007.

[Cam06] Hadrien Cambazard. *Résolution de problèmes combinatoires par des approches fondées sur la notion d'explication.* PhD thesis, Université de Nantes, Faculté des Sciences et des Techniques, Nantes, France, 2006.

[CCLW99] B. M. W. Cheng, Kenneth M. F. Choi, Jimmy Ho-Man Lee, and J. C. K. Wu. Increasing Constraint Propagation by Redundant Modeling : an Experience Report. *Constraints*, 4(2) :167–192, 1999.

[CGJ95] D. Cohn, Z. Ghahramani, and M. Jordan. Active Learning with Statistical Models. In *Advances in Neural Information Processing Systems*, volume 7, pages 705–712. The MIT Press, 1995.

[CHIP08] CHIP. Système de résolution de contraintes développé par la société COSYTEC. url : *http ://www.cosytec.fr*, 2008.

[Cho07] Choco. A Java library for constraint satisfaction problems, constraint programming and explanation-based constraint solving. Url : *http ://-choco.sourceforge.net/*, 2007.

[CMK02] A. Cornuéjols, L. Miclet, and Y. Kodratoff. *Apprentissage artificiel - Concepts et algorithmes.* Eyrolles, 2002.

[CP07] ILOG CP. Solve Complex Scheduling and Routing Problems. Url : *http ://www.ilog.com/products/cp/*, 2007.

[Dav87] Ernest Davis. Constraint propagation with interval labels. *Artificial Intelligence*, 32(3) :281–331, 1987.

[Dec03] Rina Dechter. *Constraint Processing (The Morgan Kaufmann Series in Artificial Intelligence).* Morgan Kaufmann, 2003.

[FW02] E.C. Freuder and R. Wallace. Suggestion strategies for constraint-based matchmaking agents. *International Journal on Artificial Intelligence Tools*, 11(1) :3–18, 2002.

[Gar06] Bruno Garabedian. Etude de faisabilité d'un tramway monorail. Master's thesis, Master STPI EEA 1^{ere} année, Université Montpellier II, September 2006. Sous la direction de S. Krut.

[Gat98] Erann Gat. *On Three-Layer Architectures* - Artificial Intelligence and Mobile Robots, chapter 8, pages 195–210. AAAI Press, 1998.

[GB65] S. Golomg and L. Baumert. Backtrack programming. *Journal of the Association for Computing Machinery*, 12 :516–524, 1965.

[Gee92] Pieter Andreas Geelen. Dual viewpoint heuristics for binary constraint satisfaction problems. In *ECAI '92 : Proceedings of the 10th European conference on Artificial intelligence*, pages 31–35, New York, NY, USA, 1992. John Wiley & Sons, Inc.

[GJ79] M. R. Garey and David S. Johnson. *Computers and Intractability : A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[GL94] M. Ghallab and H. Laruelle. Representation and Control in IxTeT, a Temporal Planner. In *AIPS*, pages 61–67, 1994.

[GNT04] M. Ghallab, D. Nau, and P. Traverso. *Automated Planning : Theory & Practice*. Morgan Kaufmann Publishers Inc., 2004.

[Jau06] Luc Jaulin. Localization of an underwater robot using interval constraints propagation. In *Proceedings of the 12th International Conference on Principles and Practice of Constraint Programming (CP'06)*, pages 244–255, Nantes, France, September 2006.

[Jun01] Ulrich Junker. Quickxplain : Conflict detection for arbitrary constraint propagation algorithms. In *IJCAI'01 Workshop on Modelling and Solving problems with constraints (CONS-1)*, Seattle, WA, USA, August 2001.

[Jus97] Narendra Jussien. *Relaxation de Contraintes pour les problèmes dynamiques*. PhD thesis, Université de Rennes I, Rennes, France, 1997.

[Jus03] Narendra Jussien. *The versatility of using explanations within constraint programming*. PhD thesis, Habilitation thesis of Université de Nantes, 2003.

[Lar02] Javier Larrosa. Node and arc consistency in weighted csp. In *Eighteenth national conference on Artificial intelligence*, pages 48–53, Menlo Park, CA, USA, 2002. American Association for Artificial Intelligence.

[LB03] A. Lopez and F. Bacchus. Generalizing GraphPlan by Formulating Planning as a CSP. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI'03)*, pages 954–960, Acapulco, Mexico, August 2003.

[LD03] D. Luzeaux and A. Dalgalarrondo. Hybrid architecture for autonomous robots, based on representation, perception and intelligent control. *Recent advances in intelligent paradigms and applications*, 113 :37–56, 2003.

[Lho93] Olivier Lhomme. Consistency techniques for numeric csps. In *IJCAI*, pages 232–238, 1993.

[Lit88] Nick Littlestone. Learning Quickly When Irrelevant Attributes Abound : A New Linear-Threshold Algorithm. *Machine Learning*, 2(4) :285–318, 1988.

[LL05] A. Lallouet and A. Legtchenko. Two Contributions of Constraint Programming to Machine Learning. In *Proceedings of the 16th European Conference on Machine Learning (ECML'05)*, pages 617–624, Porto, Portugal, October 2005.

[LS06] J. Larrosa and T. Schiex. Soft constraint solving. In *Tutorial of the 12th International Conference on Principles and Practice of Constraint Programming (CP'06)*, Nantes, France, September 2006.

[Luz01] Dominique Luzeaux. *De la commande à l'autonomie des systèmes robotisés*. PhD thesis, Habilitation à diriger les recherches - Université de Paris-Sud, Centre d'Orsay, 2001.

[Mac77] Alan K. Mackworth. Consistency in Networks of Relations. *Artificial Intelligence (AI Journal)*, 8(1) :99–118, 1977.

[McA80] David McAllester. An outlook on Truth Maintenance. Technical Report Memo 551, MIT Artificial Intelligence Laboratory, Boston, 1980.

[Mer01] Jean-Pierre Merlet. A Generic Trajectory Verifier for the Motion Planning of Parallel Robots. *Journal of Mechanical Design*, 123 :510–515, 2001.

[Mit80] Tom M. Mitchell. The need for biases in learning generalizations. Technical Report CBM-TR-117, Rutgers University, New Brunswick, 1980.

[Mit82] Tom M. Mitchell. Generalization as Search. *Artificial Intelligence (AI Journal)*, 18(2) :203–226, 1982.

[Mit97] Tom M. Mitchell. *Machine Learning*. McGraw Hill Science, 1997.

[Mon74] Ugo Montanari. Networks of Constraints : Fundamental Properties and Applications to Picture Processing. *Information Sciences*, 7(2) :95–132, April 1974.

[MP06] I. Miguel and S. Prestwich, editors. *Proceedings of The Fifth International Workshop on Constraint Modelling and Reformulation*, Nantes, France, September 2006.

[MR94] S. Muggleton and L. De Raedt. Inductive Logic Programming : Theory and Methods. *Journal of Logic Programming*, 19/20 :629–679, 1994.

[MZK+99] R. Monasson, R. Zecchina, S. Kirkpatrick, B. Selman, and L. Troyansky. Determining computational complexity from characteristic 'phase transitions'. *Nature*, 400 :133–137, 1999.

[NFF+05] A. Nareyek, E.C. Freuder, R. Fourer, E. Giunchiglia, R. Goldman, H. Kautz, J. Rintanen, and A. Tate. Constraints and AI Planning. *IEEE Intelligent Systems*, 20(2) :62–72, 2005.

[Nil80] N. J. Nilsson. *Principles of artificial intelligence.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1980.

[Pau05] Mathias Paulin. Apprentissage interactif de réseau de contraintes. In *Actes des 7èmes Rencontres nationales des Jeunes Chercheurs en Intelligence Artificielle (RJCIA 2005)*, pages 225–238, Nice, France, June 2005.

[Pau06] Mathias Paulin. Supervision of robot tasks planning through constraint networks acquisition. In *Proceedings of the CP 2006 Doctoral Programme*, pages 180–185, Nantes, France, September 2006.

[Pau07] Mathias Paulin. Une approche programmation par contraintes pour la modélisation et la planification de mouvements en robotique humanoïde. In *Journées nationales d'Intelligence Artificielle Fondamentale (JIAF'07)*, Grenoble, France, 02-03 July 2007.

[PBDK06] M. Paulin, E. Bourreau, C. Dartnell, and S. Krut. Modélisation et planification d'actions élémentaires robotiques par apprentissage de réseaux de contraintes. In *Actes des deuxièmes Journées Francophones de Programmation par Contraintes (JFPC'2006)*, pages 405–414, Nîmes, France, June 2006.

[PwM07] Palm (Propagation and Learning with Move). An explanation-based constraint programming system using the Choco constraints library. Url : *http ://www.e-constraints.net/palm/palm.html*, 2007.

[RÓ1] Jean-Charles Régin. Minimization of the number of breaks in sports scheduling problems using constraint programming. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 57 :115–130, 2001.

[Rae97] Luc De Raedt. Logical settings for concept-learning. *Artificial Intelligence*, 95(1) :187–201, 1997.

[Rai86] Marc H. Raibert. *Legged robots that balance.* Series In Artificial Intelligence. The MIT Press, 1986.

[RBQ06] G. Raymond, C. Bessiere, and J. Quinqueton. Mining historical data to build constraint viewpoints. In *Proceedings CP'06 Workshop on Modelling and Reformulation*, pages 1–16, Nantes, France, September 2006.

[RN03] S. Russell and P. Norvig. *Artificial Intelligence - A Modern approach.* Second Edition. Prentice Hall, 2003.

[Ros97] Julio Rosenblatt. DAMN : A Distributed Architecture for Mobile Navigation. *Journal of Experimental and Theoretical Artificial Intelligence*, 9 :339 – 360, 1997.

[RS98] F. Rossi and A. Sperduti. Learning solution preferences in constraint problems. *Journal of Theoretical and Experimental Artificial Intelligence (JETAI)*, 10(1) :103–116, 1998.

[RS04] F. Rossi and A. Sperduti. Acquiring Both Constraint and Solution Preferences in Interactive Constraint Systems. *Constraints*, 9(4) :311–332, 2004.

[RvBW06] F. Rossi, P. van Beek, and T. Wash. *Handbook of Constraint Programming (Foundations of Artificial Intelligence).* Elsevier Science Inc., 2006.

[Sat07] Sat4J. A satisfiability library for Java. Url : *http ://www.sat4j.org/*, 2007.

[SB98] R. Sutton and A. Barto. *Reinforcement Learning : An Introduction.* The MIT Press, 1998.

[SF94] D. Sabin and E.C. Freuder. Contradicting Conventional Wisdom in Constraint Satisfaction. In *Proceedings of the Second International Workshop on Principles and Practice of Constraint Programming (PPCP'94)*, pages 10–20, Rosario, Orcas Island, Washington, USA, May 1994.

[SFJ00] D. Smith, J. Frank, and A. Jónsson. Bridging the Gap Between Planning and Scheduling. *Knowledge Engineering Review*, 15(1) :47–83, 2000.

[Sol07] SolidWorks. Logiciel de conception mécanique 3D. Url : *http ://www.solidworks.fr*, 2007.

[SSW99] B. Smith, K. Stergiou, and T. Walsh. Modelling the Golomb Ruler Problem. In *16th International Joint Conference on Artificial Intelligence (IJCAI'99), Workshop on Non Binary Constraints*, Stockholm, Sweden, August 1999.

[TDK94] A. Tate, B. Drabble, and R. Kirby. O-Plan 2 : an Open Architecture for Command, Planning and Control. In M. Fox and M. Zweben, editors, *Knowledge Based Scheduling.* Morgan Kaufmann, San Mateo, California, 1994.

[VB06] G. Verger and C. Bessiere. BlockSolve : a Bottom-Up Approach for Solving Quantified CSPs. In *Proceedings of the 12th International Conference on Principles*

and Practice of Constraint Programming (CP'06), pages 635–649, Nantes, France, September 2006.

[VJ03] G. Verfaillie and N. Jussien. Dynamic Constraint Solving. In *Tutorial of the 9th International Conference on Principles and Practice of Constraint Programming (CP 2003)*, Kinsale, Ireland, September 2003.

[VNE+01] R. Volpe, I. Nesnas, T. Estlin, D. Mutz, R. Petras, and H. Das. The claraty architecture for robotic autonomy. In *Proceedings of the 2001 IEEE Aerospace Conference*, pages 121–132, Big Sky, Montana (USA), March 2001.

[Wal96] Mark Wallace. Practical Applications of Constraint Programming. *Constraints*, 1(1/2) :139–168, 1996.

[Wal01] Toby Walsh. Permutation problems and channelling constraints. *Lecture Notes in Computer Science*, 2250 :377–391, 2001.

[WD92] C. Watkins and P. Dayan. Q-learning. *Machine Learning*, 8 :279–292, 1992.

[ZFCS02] Y. Zhang, M.P.J. Fromherz, L. Crawford, and Y. Shang. A general constraint-based control framework with examples in modular self-reconfigurable robots. In *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2002)*, pages 2163–2168, Lausanne, Switzerland., October 2002.

[Zyt07] Matthias Zytnicki. *Localisation d'ARN non-codants par réseaux de contraintes pondérées.* PhD thesis, Université Toulouse III - Paul Sabatier, Toulouse, France, 2007.

Résumé : Dans le cadre de cette thèse, nous nous intéressons à l'acquisition automatique de réseau de contraintes, aussi appelée apprentissage automatique de réseau de contraintes, qui consiste à développer des solutions capables d'aider un utilisateur à modéliser un problème combinatoire sous la forme d'un réseau de contraintes. Notre travail se focalise plus précisément sur la plate-forme d'acquisition automatique de réseau de contraintes Conacq. Dans son implémentation standard, Conacq est passive vis-à-vis de l'utilisateur, c'est-à-dire basée sur la capacité de ce dernier à fournir des instances significatives de son problème. Dans la première partie de cette thèse, nous proposons une version interactive de Conacq capable de poser à l'utilisateur des questions dont le but est d'augmenter plus rapidement et de manière conséquente la connaissance acquise par la plate-forme. Afin de limiter le nombre d'interactions, nous proposons différentes stratégies de questionnement que nous validons ensuite empiriquement. Dans la seconde partie, nous nous intéressons à une utilisation pratique de l'acquisition automatique de réseau de contraintes qui vise à automatiser le processus de définitions de comportements sensorimoteurs en robotique. Dans cette optique, nous proposons une architecture logicielle, complémentaire aux architectures de contrôle existantes, qui utilise le paradigme de la programmation par contraintes pour modéliser, planifier et superviser l'exécution de comportements sensorimoteurs. Elle utilise la plate-forme Conacq étudiée dans la première partie de cette thèse pour modéliser automatiquement les actions élémentaires d'un robot sous la forme de réseaux de contraintes. Notre architecture utilise par ailleurs un planificateur de tâches inspiré de CSP-Plan pour combiner les réseaux de contraintes acquis et ainsi définir automatiquement des comportements sensorimoteurs. Différents résultats expérimentaux sont par ailleurs présentés afin de valider notre approche.
Mots-clés : Programmation par contraintes, Apprentissage, Robotique, Planification, Contrôle, Comportements sensorimoteurs.

Abstract : This thesis deals with constraint network acquisition, which consists in automatically acquiring a constraint network formulation of a problem from a subset of its solutions and non-solutions. Our work is mainly focussed on the constraint network acquisition platform Conacq. In the standard version of Conacq, the choice of the subset of solutions and non-solutions to use for learning was assumed to be made before and independently of the acquisition process. In the first part of this thesis, we present an interactive version for Conacq in which the acquisition system actively assists in the selection of the set of examples used to acquire the constraint network through the use of learner-generated queries. We show that the number of examples required to acquire a constraint network is significantly reduced if queries are selected carefully. In order to decrease the number of interactions, we provide a theoretical and empirical evaluation of query generation strategies for interactive constraint acquisition, with very positive experimental results. In the second part of this thesis, we are interested in a pratical use of constraint network acquisition for robotics. Thus, we propose a Constraint Programming-based framework for modelling, planning and supervising sensorimotor behaviors. Our approach uses the constraint network acquisition platform Conacq to automatically encode by Machine Learning each elementary action of a robot as a CSP. The acquired CSPs are then automatically combined by planning to constitute sensorimotor behaviors using a CSP-based planner inspired by CSP-Plan. Moreover, our architecture exploits the propagation properties of the acquired CSPs to supervise the execution of a given sensorimotor behavior. Some experimental results are presented to validate our approach.
Keywords : Constraint Programming, Machine Learning, Robotics, Planning, Supervision, Sensorimotor behaviors.