N° d'ordre : 4460 **ANNÉE 2011**

THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Européenne de Bretagne

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Informatique

Ecole doctorale Matisse

présentée par

Nadjib LAZAAR

préparée à l'unité de recherche 6074 IRISA
Institut de Recherche en Informatique et Systèmes Aléatoires
ISTIC

Méthodologie et outil de Test, de localisation de fautes et de correction automatique des programmes à contraintes

Thèse soutenue à Rennes le 5 décembre 2011

devant le jury composé de :

Mireille DUCASSE
Professeur à l'INSA de Rennes / *président*

Nicolas BELDICEANU
Professeur EMN Nantes/ *rapporteur*

Arnaud LALLOUET
Professeur GREYC Caen / *rapporteur*

Patrick ALBERT
Ingénieur rech. IBM-Ilog Paris / *examinateur*

Yahia LEBBAH
Professeur LITIO Oran / *examinateur*

Thomas JENSEN
Directeur de recherche INRIA/ *directeur de thèse*

Arnaud GOTLIEB
Chargé de recherche INRIA / *co-directeur de thèse*

Remerciements

Tout d'abord, je tiens à remercier les membres de mon jury de thèse pour l'intérêt qu'ils ont porté à ce travail : Mireille Ducassé qui m'a fait l'honneur de présider ce jury, Nicolas Beldiceanu et Arnaud Lallouet pour avoir accepter la charge de rapporteur, et Patrick Albert qui a bien voulu examiner mon travail. Je remercie Thomas Jensen d'avoir accepté de diriger cette thèse.

Je remercie particulièrement Arnaud et Sophie Gotlieb pour leur aide et leur soutien tout au long de mon master recherche et de cette thèse. Arnaud, avec qui nos discussions furent des plus enrichissantes, qu'elles soient scientifiques ou autres.

Je remercie également Yahia Lebbah qui a toujours été présent tout au long de mes années universitaires à Oran, durant mon Master recherche et de cette thèse. Je le remercie également d'avoir répondu présent en faisant partie de mon jury.

Je remercie les membres des équipes CeP de Nice, Contraintes de Nantes, CRIL de Lens, Constraints de Microsoft Cambridge, LITIO d'Oran, à savoir Michel Rueher, Claude Michel, Jean-Charles Régin, Olivier Lhomme, Laurent Granvilliers, Lakhdar Sais, Youssef Hamadi, pour les différentes discussions qui ont enrichi le contenu de cette thèse.

Je remercie Michèle Sebag et Marc Schoenauer ainsi que tous les membres de l'équipe TAO au LRI pour leur récent accueil et leur soutien moral durant les derniers instants de ma thèse.

Je remercie ceux qui ont contribué à la bonne ambiance quotidienne : les membres de l'équipe LANDE puis CELTIQUE pour leur accueil amical, à Drid Hamza pour nos bavardages et pour les pauses à la cafétéria, à Mathieu Petit, Tristan Denmat, Florence Charreteur Schadle et Matthieu Carlier pour tous les bons moments partagés, à Pierre Riteau pour nos échanges sur l'enseignement et sur bien d'autres sujets. Je remercie également Lydie Mabil pour sa gentillesse et son aide précieuse.

Je remercie les enseignants de l'ISTIC avec qui j'ai eu la chance de travailler : ce fut un plaisir d'être intégré dans une équipe pédagogique si chaleureuse.

Je termine par remercier ma famille et mes amis : mes parents qui ont toujours cru en moi, mes deux sœurs si cheres à mon cœur Farah et Hadjer, ma grand-mère que j'aime tant, une pensée à mon petit frère Abdou. Merci les amis, merci en particulier à Fayçal, Raouf, Anes d'Oran. Jawad, Mehdi, Faiz, Abdellah de Rennes. Adnan, Mohammed, Alia,

Remerciements

Tassadit, Djawida, Emna, Houssem de l'IRISA. A la fin, je ne saurais autant remercier mes parents qui ont toujours été présents, qui ont toujours cru en moi et qui ont fait de moi ce que je suis aujourd'hui...leur enfant.

Á Abdou...

Table des matières

Introduction

Le logiciel est devenu un produit omniprésent dans notre vie quotidienne (e.g., transport, finance, santé, etc.) et son importance en fonction des tâches qu'il remplit ne fait que croître. Le logiciel est *critique* lorsqu'une défaillance peut entrainer des conséquences dramatiques, telle une perte humaine, financière, ou encore une catastrophe environnementale. La qualité d'un logiciel se mesure par son niveau de correction, de fiabilité, de sécurité et de sa capacité à répondre à toutes les exigences de l'utilisateur. Plusieurs approches et méthodes sont utilisées pour assurer un certain niveau de qualité du logiciel. Les méthodes basées sur la *preuve* essayent d'établir une démonstration formelle de l'exactitude du code par rapport à une spécification mathématique du programme. Ces méthodes s'appuient sur une logique mathématique et un haut niveau d'abstraction :

- preuve formelle sur les programmes (Dijkstra, 1997),
- analyse statique par interprétation abstraite (Cousot & Cousot, 1977),
- vérification de modèles (*model checking*) (Sifakis, 1982).

Ces techniques permettent d'assurer des propriétés pour toutes les exécutions du programme, mais bien qu'elles soient très rigoureuses, elles sont aussi couteuses et difficiles à mettre en œuvre dans un cadre industriel.

Dans ce contexte, le **test logiciel** reste la méthode la plus aisée à mettre en œuvre et la plus répandue dans l'industrie, même si elle offre moins de garanties que la vérification formelle. Elle consiste à exécuter un *programme sous test* avec des données d'entrée particulières. Pour chacune de ces exécutions, un *oracle*, humain ou automatisé, détermine si le comportement du programme est conforme à sa spécification. Contrairement à la preuve, le test ne permet pas de garantir l'absence de fautes, mais augmente la confiance portée au programme. Le test reste, la plupart du temps, une activité manuelle, laborieuse et onéreuse. Généralement 30 à 50% du budget global du développement d'un logiciel est dédié à l'effort de test.

La programmation par contraintes (PPC) est un paradigme de programmation déclarative, où le problème est décrit à l'aide d'un ensemble de contraintes bien formalisées. Résoudre ce problème revient à satisfaire l'ensemble des contraintes en laissant un *solveur* explorer l'espace de recherche. Lors de ces dernières années, les langages de modélisation des programmes à contraintes ont connu un développement spéctaculaire. Ils ont été conçus pour un haut niveau de modélisation et d'abstraction, et permettent une grande expressivité et une expérimentation facile avec différents solveurs. Des langages comme OPL (Optimization Programming Language) de IBM Ilog, Comet de Dynadec, Sicstus Prolog, Gecode, Choco, Zinc, ainsi que d'autres, proposent des solutions robustes aux problèmes du monde réel. De plus, ces langages commencent à être utilisés dans des applications critiques comme la gestion et le contrôle du trafic aérien (Flener *et al.*, 2007a;

Junker & Vidal, 2008), le e-commerce (Holland & O'Sullivan, 2005) et le développement de programmes critiques (Collavizza *et al.*, 2008; Gotlieb, 2009). Ainsi, comme tout processus de développement logiciel effectué dans un cadre industriel, les développements de ces programmes à contraintes doivent désormais inclure une phase de test, de vérification formelle et/ou de validation. Ceci ouvre la voie à des recherches orientées vers les aspects génie logiciel dédiés à la PPC, et en particulier le test des programmes à contraintes.

Problématique : le test des programmes à contraintes

Le développement rapide des langages de modélisation des programmes à contraintes a été le fruit de l'exploitation des aspects et des techniques de génie logiciel. Á notre connaissance, il n'existe pas d'outils et d'environnements dédiés au test de ces langages de haut niveau. Par ailleurs, les théories de test des programmes conventionnels existantes semblent être des schémas inopérants pour capter les spécificités du test des programmes à contraintes pour différentes raisons : le modèle de fautes en PPC est différent de celui des programmes à contraintes, les optimisations en PPC sont différentes, pas de notion de séquentialité des programmes conventionnels. Nous y reviendrons plus loin.

Á l'inverse, la **mise-au-point** de ce type de programmes a fait l'objet de nombreux travaux de recherche qui ont abouti à la définition de modèles de trace génériques (Deransart *et al.*, 2000c; Langevine *et al.*, 2001) et à la réalisation d'outils d'observation et d'analyse de traces post-mortem. Ces outils aident à comprendre les comportements d'un programme à contraintes et facilitent son optimisation. En revanche, ces outils ne sont pas dédiés à la détection de fautes. En effet, celle-ci exige la donnée d'une référence (un oracle) afin de déterminer la divergence entre une implantation et cette référence.

Aucun des langages de modélisation des programmes à contraintes cités précédemment ne présentent, à notre connaissance, de théorie et/ou d'outil de test des programmes à contraintes à proprement parler.

La problématique étudiée dans cette thèse est la définition d'un premier cadre de test pour les programmes à contraintes. En d'autres termes, il s'agit de poser les jalons d'une théorie de test qui puisse servir de socle à la vérification de ce type de programmes.

Contributions

Notre thèse s'intéresse au test des programmes à contraintes, à la localisation et la correction automatique des fautes dans ces programmes. Elle contient trois contributions principales.

Notre première contribution est **la définition d'une théorie de test pour les programmes à contraintes** (Lazaar *et al.*, 2009; Lazaar *et al.*, 2010b). Cette dernière s'appuie sur l'observation du processus de développement des programmes à contraintes tel qu'il semble être pratiqué dans les milieux industriels. Il est habituel de démarrer la modélisation d'un problème par un programme très déclaratif qui ne fait que poser les

contraintes du problème telles qu'elles se présentent. Ce premier modèle est une traduction fidèle de la spécification du problème, sans pour autant accorder grand intérêt à ses performances. Puis, des techniques puissantes de raffinement de modèles sont mises en œuvre. Par exemple un codage approprié du problème à l'aide de structures de données optimisées est proposé, ainsi qu'une reformulation des contraintes d'origine. Des *contraintes globales* sont utilisées afin de maximiser le pouvoir de déduction du modèle, ainsi que des *contraintes redondantes ou impliquées* qui visent à accélérer la résolution. De plus, l'espace de recherche peut être considérablement réduit à l'aide de contraintes qui cassent des symétries dans le problème. Ce processus de raffinement peut introduire des fautes dans le programme final. Un *modèle de fautes*, propre aux programmes à contraintes, est donc défini dans ce manuscrit. Nous avons également défini des *relations de conformité* entre le premier modèle qui nous sert de référence de test et le programme optimisé qui est sous test. Ces relations sont définies selon la classe des problèmes abordés. En effet, une relation de conformité des problèmes de satisfaction de contraintes est différente de celle des problèmes d'optimisation. Aussi, une même relation de conformité ne peut être appliquée sur un problème qui cherche une solution et un autre qui cherche toutes les solutions possibles. À travers le modèle de fautes et ces relations de conformité, nous sommes parvenus à définir un processus de test dont l'objectif est de trouver des données de test qui révèlent l'existence de fautes dans le programme.

Notre deuxième contribution concerne la mise au point de ces programmes (i.e., **localisation et correction automatique des fautes**) (Lazaar *et al.*, 2010a; Lazaar *et al.*, 2011a). Si la phase de test révèle la présence d'une faute dans le programme, la phase de localisation essaie de trouver l'origine de la faute. Dans le cadre de la théorie du test proposée dans cette thèse, nous introduisons un algorithme qui calcule un ensemble de contraintes susceptibles de contenir la faute (contraintes suspectes). Ces contraintes représentent des candidats potentiels pour être à l'origine des fautes. En d'autres termes, faire une révision sur l'une des contraintes suspectes permet probablement de corriger la faute. Une révision de contrainte dans un but de correction est une tâche manuelle qui nécessite une grande expertise, mais avoir des propositions de correction de façon automatique permet de réduire le coût de la mise-au-point des programmes. Nous proposons donc un processus qui permet de générer des corrections possibles aux fautes.

La troisième contribution dans ce manuscrit est le développement **d'un prototype de test et de mise-au-point** basé sur le cadre formel de test des programmes à contraintes (Lazaar, 2011). Ce prototype, nommé CPTEST, permet de faire de la détection, de la localisation et de la correction automatique des fautes des programmes à contraintes écrit en OPL. Développé en Java, CPTEST représente 25 000 lignes de code avec un analyseur complet des programmes OPL. Durant la vérification et la mise-au-point d'un programme OPL, le prototype fait appel au solveur CP Optimizer 2.3 d'IBM Ilog. CPTEST nous a permis de faire une étude expérimentale sur des problèmes académiques (règles de golomb, n-reines, social golfer), ainsi que sur un exemple dérivé d'un problème réel qui est l'ordonnancement de véhicules. Les résultats de cette étude montrent que CPTEST peut être utile pour détecter et corriger automatiquement des fautes dans les programmes

OPL.

Organisation de la thèse

Cette thèse est divisée en cinq parties et comprenant dix chapitres.

Dans **la partie I**, le premier chapitre présente le contexte des travaux avec un état de l'art sur les langages de modélisation en PPC et la mise au point dans ce type de langages. Le second est dédié aux théories de tests existantes et les travaux récents sur la mise au point (localisation et correction) des programmes conventionnels. Cette partie a pour objectif la mise en évidence de l'absence des techniques de test dans la PPC.

Dans **la partie II** le troisième chapitre présente les différents raffinements en PPC et le modèle de fautes résultant. Le quatrième présente les relations de conformité que nous avons proposées. Le cinquième chapitre, en se basant sur les relations de conformité, comporte la description du processus de test.

La partie III est dédiée à la mise-au-point des programmes à contraintes, le sixième chapitre traite la question de la localisation de fautes avec des calculs de contraintes suspectes. Le septième chapitre faisant suite à la localisation traite la correction automatique des programmes à contraintes où une révision est proposée pour toute contrainte suspecte.

Dans **la partie IV**, le huitième chapitre décrit le prototype CPTEST implanté pour offrir un environnement complet de test et de mise au point des programmes à contraintes écrits en OPL. Enfin le chapitre neuf présente notre étude expérimentale sur plusieurs exemples, avec comme objectif de montrer que la détection, la localisation et la correction des fautes dans les programmes à contraintes étaient possibles et utiles.

la partie V contient un dernier chapitre qui clôture ce manuscrit avec des conclusions et des perspectives liées aux travaux de test et de mise-au-point des programmes à contraintes.

Première partie .

État de l'art

1. Langages de modélisation des programmes à contraintes

1.1. Programmation par contraintes

La programmation par contraintes (PPC) est un paradigme de programmation puissant qui permet d'exprimer des problèmes en un ensemble de contraintes avec une expressivité difficile à atteindre avec un programme *algorithmique* ou *conventionnel*[1]. La PPC est considérée comme une programmation *non-algorithmique* car elle pose le problème sous forme de contraintes sans implémenter l'algorithme ou la façon d'atteindre une solution. Ce paradigme de programmation est apparu vers la fin des années 70 du siècle dernier, il traite la manière dont l'ensemble des solutions d'un problème donné peut être codé et non pas le calcul d'une solution particulière.

Les premiers travaux en PPC remontent à la proposition du système ALICE de J-L Laurière (Laurière, 1976) qui est un système pour énoncer et résoudre des problèmes combinatoires. ALICE est aussi considéré comme un langage de modélisation avec une représentation déclarative et concise des problèmes. Le système est basé sur des notions mathématiques et logiques ainsi que des contraintes formelles (logico-algébriques) qui ne sont pas nécessairement linéaires. Le module de résolution introduit les bases de *filtrage*, de *propagation*, de choix et d'heuristique *d'énumération*, qui font le cœur de la résolution PPC de nos jours. Une des implémentations s'inspirant du système ALICE en programmation logique avec contraintes est un système issu des travaux de P. Van Hentenryck (Van Hentenryck, 1989a). Ce travail a donné naissance par la suite au langage CHIP (Constraint Handling In Prolog) (Dincbas *et al.*, 1988).

La PPC peut être perçue comme une convergence croisée de trois principaux domaines : la programmation logique, la satisfaction des contraintes et les techniques de recherche opérationnelle. La PPC se base sur un fondement théorique solide et offre un intérêt commercial répandu, elle est devenue maintenant une méthode incontournable pour la modélisation de beaucoup de types de problèmes combinatoires qui surgissent de l'optimisation, la planification (*scheduling*), la gestion, la production, etc.

Une des caractéristiques intéressantes de la PPC est cette séparation entre la partie modélisation et la partie résolution (Van Hentenryck & Michel, 2005). La première partie a pour vocation d'exprimer le problème à l'aide de contraintes. Dès lors, la deuxième partie prend l'ensemble de ces contraintes et essaie de trouver une solution qui le satisfait. Ceci est résumé par l'équation suivante :

1. On notera dans ce document programme conventionnel tout programme impératif ou fonctionnel.

PPC= Modèle + Résolution

1.1.1. Partie modélisation

Face à un système du monde réel, l'étape de modélisation permet de représenter le comportement et l'interaction des objets de ce dernier à travers des contraintes. Une *contrainte* en PPC représente une *relation logique* entre des variables, exprimée avec des opérateurs (arithmétiques, logiques, ensemblistes, etc). Chaque variable peut avoir un domaine différent de valeurs. La spécificité des contraintes est qu'elles sont *déclaratives* car elles expriment la relation entre les variables sans présenter de procédure opérationnelle qui vérifie cette relation. Une contrainte peut être de type linéaire ou non, discrète ou continue. Elle peut aussi prendre une forme spéciale en représentant une sémantique bien décrite comme c'est le cas des contraintes globales (e.g., *allDifferent*).

Une contrainte C peut également être définie en extension avec son ensemble de solutions, noté $sol(C)$, qui représente l'ensemble des instantiations qui satisfont la contrainte C. De plus, on dit qu'une contrainte C est satisfiable si son ensemble de solutions n'est pas vide ($sol(C) \neq \emptyset$).

1.1.2. Partie résolution

Lorsque les domaines sont finis, il est possible d'atteindre une instantiation qui satisfait l'ensemble des contraintes avec une recherche exhaustive sur toutes les combinaisons possibles des valeurs des variables. Cette résolution est trop couteuse dans le cas général (i.e., problème NP-difficile). En revanche, il existe des mécanismes qui permettent de réduire ce nombre de combinaisons et atteindre rapidement une solution.

Algorithme de filtrage

Chaque contrainte a son propre algorithme de filtrage qui permet de réduire les domaines des variables. Le filtrage a pour objectif ultime de supprimer toute valeur d'une variable donnée qui ne peut participer à une solution.

Exemple 1 $\begin{cases} D_x = \{1,3,5,9\}, \\ D_y = \{0,1,4\} \end{cases} \xrightarrow[x=y+1]{Filtrage} \begin{cases} D_x = \{1,5\}, \\ D_y = \{0,4\} \end{cases}$

L'exemple 1 montre un filtrage sur la contrainte binaire $x = y + 1$ qui réduit les domaines de x et de y en supprimant toutes les valeurs dites *inconsistantes*. Cet exemple illustre une propriété intéressante des algorithmes de filtrage, la *consistance d'arc* (AC). Un algorithme de filtrage assure une consistance d'arc pour une contrainte binaire donnée s'il supprime toute valeur inconsistante. Cette consistance peut être généralisée pour les contraintes portantes sur plusieurs variables (contraintes n-aires) avec une consistance dite d'*hyper-arc* ou *généralisée* (GAC) (exemple 2).

Exemple 2 $\begin{cases} D_x = \{0, 4, 7\}, \\ D_y = \{1, 3, 5\}, \\ D_z = \{0, 1, 4\} \end{cases} \xrightarrow[x=y+z]{Filtrage} \begin{cases} D_x = \{4, 7\}, \\ D_y = \{3\}, \\ D_z = \{1, 4\} \end{cases}$

Une consistance d'arc est forte dans le sens où toute valeur inconsistante est supprimée. En revanche, elle peut s'avérer très couteuse quand on a de grande plages de valeurs à supprimer une à une. Dans ce cas, une autre consistance moins forte peut être utilisée, la *consistance de borne*. Cette consistance permet de réduire les bornes des domaines des variables de façon plus rapide qu'avec une consistance d'arc mais sans pouvoir atteindre les valeurs inconsistantes comprises dans les domaines. Prenons l'exemple 3, les domaines de x et de y sont réduits en deux temps avec une consistance de borne.

Exemple 3 $\begin{cases} D_x = [1\ 000, 3\ 000], \\ D_y = [0, 2\ 000] \end{cases} \xrightarrow[x<y+1]{Filtrage} \begin{cases} D_x = [1\ 000, 2\ 000], \\ D_y = [1\ 000, 2\ 000] \end{cases}$

De plus, plusieurs types de consistance existent (Debruyne & Bessiére, 2001), chacune essaie d'enlever le maximum de valeurs inconsistantes. Les algorithmes de filtrage ont pour objectif d'assurer une ou plusieurs consistances.

Mécanisme de propagation

La réduction du domaine d'une variable due au filtrage d'une contrainte peut avoir des conséquences sur d'autres contraintes impliquant cette même variable. Ainsi, une modification d'un domaine réveille d'autres contraintes en appelant le filtrage associé. En d'autres termes, cette modification est propagée sur les autres contraintes. Le mécanisme de propagation permet d'avoir un enchainement de réductions avec des appels successifs des algorithmes de filtrage ce qui donne une puissance à la réduction des domaines et de l'espace de recherche.

Mécanismes de recherche de solutions

Le filtrage et la propagation des contraintes sont des mécanismes qui réduisent l'espace de recherche. Au moment où aucune réduction n'est possible, on procède à une énumération et un parcours de l'espace de recherche. Le mécanisme le plus simple pour trouver une instantiation qui satisfait l'ensemble des contraintes est l'algorithme du *retour-en-arrière* ou *backtrack chronologique* (pour plus de détails voir (Marriott & Stuckey, 1998)). L'idée est de sélectionner une variable x puis une valeur d du domaine de la variable $D(x)$ qui détermine la satisfiabilité des contraintes. Ceci est fait avec des appels récursifs de l'algorithme du *backtrack* sur un ordre prédéfini de variables. Dans le cas où une contrainte est insatisfiable, on doit procéder à un retour arrière (*backtrack*) et sélectionner une autre valeur pour la variable x.

Le parcours de l'espace de recherche peut être représenté par un arbre où chaque noeud représente une affectation valeur/variable. L'algorithme du *backtrack* fait un parcours en

profondeur. De plus, cet algorithme peut être amélioré avec des méthodes rétrospectives (*look-back*) et prospectives (*look-ahead*).

Les méthodes rétrospectives, au lieu de faire un retour arrière naïf, essayent d'identifier la cause de l'échec pour sélectionner un meilleur point de retour (*backjumping (Dechter, 1990), dynamic backtraking (Ginsberg, 1993)*, etc.) ou de supprimer des sous-arbres ne menant à aucune solution (*backmarking, conflict-set*, etc.) (Jiang *et al.*, 1994).

Les méthodes prospectives examinent les valeurs dans le but d'éliminer celles ne participant à aucune solution, comme par exemple le *forward-cheking* (Bessière *et al.*, 2002).

Dans la majorité des applications PPC, le problème n'est pas seulement de trouver une solution, mais de trouver une *solution optimale* qui répond à des critères. L'expression de ces critères se fait à l'aide d'une ou de plusieurs *fonctions objectif*. L'objectif du solveur est alors de trouver des solutions où la valeur de la fonction objectif est optimale. L'algorithme de base utilisé pour cette classe de problèmes est l'algorithme par *séparation et évaluation* ou *Branch&Bound*. Cet algorithme permet de déterminer la solution optimale par énumération de sous-problèmes (pour une lecture détaillée, nous recommandons (Winston, 2004)). La trace de la résolution est représentée par une arborescence où l'union des feuilles représente l'ensemble des solutions possibles. L'objectif derrière cet algorithme est d'économiser l'évaluation de sous-problèmes qui ne peuvent contenir la solution optimale. Dans le cadre général du *Branch&Bound*, l'algorithme implique :

- une méthode pour décomposer l'espace de recherche.
- une stratégie d'exploration de l'arbre des sous-problèmes.
- un algorithme de calcul de la borne supérieure.
- un algorithme de calcul de la borne inférieure.

Dès lors, l'efficacité du *Branch&Bound* dépend de tous ces éléments.
Quelques variantes de cet algorithme utilise une technique appelée *partitionnement optimiste* (i.e., *optimistic partitioning*) (Marriott & Stuckey, 1998) et dont l'idée est de partitionner les domaines des variables de la fonction objectif dans le but de converger le plus rapidement possible. Une autre variante de l'algorithme *Branch&Bound* est l'algorithme des *poupées russes* (i.e., *RDS : Russian Doll Search* (Verfaillie *et al.*, 1996)) qui a été proposé dans le cadre des problèmes d'ordonnancement en PPC. L'idée est de résoudre successivement des sous-problèmes de plus en plus imbriqués, à commencer par ordonner les dernières tâches et à terminer par résoudre le problème dans son ensemble. Chaque sous-problème retourne une bonne borne prise par le sous-problème suivant.

Un solveur de contraintes représente une implémentation des mécanismes vus précédemment. Il existe une palette assez variée de solveurs qu'on distingue selon des choix d'implémentation : les heuristiques de recherche, le nombre et le type de contraintes prises en compte, les algorithmes de filtrages, etc.

1.2. Langages PPC

Les langages PPC sont de plus en plus nombreux, une partie de ceux-ci étant commercialisés comme Sicstus Prolog, OPL ou Comet. D'autres sont plus académiques comme

Choco, Zinc ou Gecode. Nous décrivons brièvement ces langages et leur outils de mise-au-point associés.

Langage OPL

OPL est un langage de modélisation de haut niveau qui combine la modélisation en programmation mathématique, proche de celle de AMPL (Fourer *et al.*, 2003), et en programmation par contraintes (Van Hentenryck, 1999). Le langage est purement déclaratif avec une grande puissance d'expression et de modélisation de contraintes, tout en incorporant les techniques de résolution PPC.

La version industrielle d'OPL s'appuie sur CP Optimizer de IBM Ilog[2] qui offre un environnement de développement pour des applications d'optimisation combinatoire (OPL Studio). Cet environnement inclut des fonctionnalités de base de mise-au-point avec un vérificateur syntaxique et une visualisation de trace post-mortem sur les états des variables, des domaines, etc. En revanche, l'environnement de développement ne propose aucune fonctionnalité de test des programmes à contraintes.

Langage Comet

Le langage Comet est un langage orienté-objet avec un haut niveau de modélisation de contraintes. Il implémente une hybridation de méthodologies avec la programmation par contraintes, la programmation linéaire et entière, l'optimisation combinatoire stochastique et le paradigme de recherche locale basée sur les contraintes (CBLS *: Constraint Based Local Search*) (Van Hentenryck & Michel, 2005).

Ce langage regroupe les travaux de Pascal Van Hentenryck dans les différentes méthodologies citées dessus. La compagnie Dynadec[3] propose des solutions robustes à des problèmes combinatoires avec sa plateforme d'optimisation hybride sous Comet. Cette plateforme propose une interface à gdb, le débogueur standard de la suite gcc. Elle offre aussi un outil de visualisation (Dooms *et al.*, 2007) qui permet un affichage et un suivi d'exécution en temps réel.

Langage Zinc

Zinc est un langage de modélisation en PPC développé dans le cadre du projet G12[4] (Marriott *et al.*, 2008). Il permet une modélisation de haut-niveau indépendante des solveurs, avec une notation proche de la programmation mathématique. Il permet aussi bien l'utilisation que la définition de nouvelles contraintes, de fonctions et de prédicats. MiniZinc représente une couche intermédiaire sur un sous-ensemble du langage. Ce langage offre un bon niveau d'abstraction pour exprimer facilement un système de contraintes, et pour passer facilement d'un solveur à un autre. La couche basse est représentée par FlatZinc. Étant donné un solveur de contrainte, FlatZinc génère une version du système de contraintes correspondante aux exigences du solveur.

2. www-01.ibm.com/software/integration/optimization/cplex-cp-optimizer/
3. dynadec.com/
4. www.g12.csse.unimelb.edu.au/

Langage Gecode

Gecode[5](Schulte, 2002) est un solveur de contraintes écrit en C++. C'est aussi un environnement de modélisation en PPC modulable et extensible. L'environnement est ouvert car il peut facilement s'interfacer avec d'autres systèmes. Il permet la définition de nouvelles contraintes, de nouveaux propagateurs ainsi que des stratégies de recherche. Les contraintes et les stratégies de recherche sont exprimées dans une syntaxe proche du langage impératif implémentant le solveur (C++). La distribution de Gecode est gratuite et assure la portabilité du code sur de nombreuses plateformes. La mise-au-point des modèles écrits en Gecode se fait grâce au débogueur sous-djacent (gdb).

Langage Choco

Le solveur Choco[6] (CHOCO-Team, 2010) est développé en Java sous forme de librairies open-source. Les librairies de Choco sont assez riches avec la possibilité d'exprimer un grand nombre de contraintes et de stratégies de recherche. La mise-au-point sous Choco se fait sous le jdb de Java avec une analyse syntaxique du code. La mise-au-point des programmes Choco a fait l'objet de quelques travaux de recherche. Ces travaux ont comme objectif la définition d'un modèle de trace et d'outils d'observation et d'analyse post-mortem basés sur l'explication des échecs (Jussien & Barichard, 2000).

1.3. Mise-au-point des programmes à contraintes

Les premiers travaux en matière de mise-au-point des programmes logiques à contraintes (CLP) sont ceux de M. Ducassé (Ducassé, 1986) avec le système OPIUM, un outil de trace et de mise-au-point des programmes Prolog. Par la suite, il y a eu les travaux de Dahmen (Dahmen, 1991) et Meier (Meier, 1995) qui présentent un paradigme de trace des programmes à contraintes avec la conception et l'implémentation de l'outil Grace qui est un environnement graphique pour tracer des programmes CLP(FD) écrits en ECL^{i}PS^e. Ces travaux ont été pionniers et représentent le point de départ de beaucoup de travaux dans la mise au point en PPC.

Deux projets phares ont eu comme objectif de définir une méthodologie de débogage et la réalisation d'outils d'analyse et de mise au point des programmes à contraintes. Le premier est le projet européen DiSCiPL (Deransart *et al.*, 2000a) (*Debugging Systems for Constraint Programming*) à la fin des années 90. Le deuxième est OADymPPaC (Rocquencourt *et al.*, 2001) (*Outils pour l'Analyse Dynamique et la mise au Point de Programmes avec Contraintes*) qui est un projet national RNTL au début des années 2000. Ces projets ont abouti à des modèles formels de trace et la réalisation d'outils d'observation.

La section suivante introduit les différentes mises-au-point, à savoir la mise-au-point pour la *correction* et la mise-au-point pour la *performance*.

5. www.gecode.org/
6. www.emn.fr/z-info/choco-solver/

1.3.1. Mise-au-point pour la correction

La mise-au-point de correction, telle détaillée dans (Deransart *et al.*, 2000b), a pour objectif de corriger des erreurs d'ordre syntaxique, l'absence de réponse, mauvais résultat, une non-terminaison du programme ainsi que d'autres comportements inattendus.

Mise-au-point statique des programmes à contraintes. Le principe ici est le même que dans une *analyse statique* de programmes conventionnels. L'idée est de trouver des fautes dans le programme sans l'exécuter. Pour cela, on utilise des assertions qui permettent de décrire des propriétés des contraintes. Un écart entre une propriété inférée et celle spécifiée ou attendue est appelé *un symptôme abstrait* (Deransart *et al.*, 2000b). *Le diagnostic abstrait* est le processus (manuel ou automatique) qui permet de localiser et d'expliquer l'origine du *symptôme*.

Une des limitations de ces techniques est qu'elles n'arrivent pas à capter l'interaction entre les contraintes dans la phase de propagation. Ceci limite ces techniques au niveau de la logique des programmes. De plus, les techniques statiques de mise-au-point demandent un effort considérable d'annotation de code.

Mise-au-point dynamique des programmes à contraintes. Contrairement à l'approche statique, le programme est exécuté et le symptôme observé devient *un symptôme d'exécution* ((Deransart *et al.*, 2000b), projet DiSCiPL) qui peut être détecté durant une analyse de la trace de résolution en temps réel ou lors d'une analyse post-mortem. Une trace de résolution représente un ensemble d'événements qui peuvent être liés aux réveils des contraintes ou à la phase d'énumération. Ce type de mise-au-point peut être basée sur une sémantique déclarative qui donne un diagnostic dit *déclaratif* permettant de localiser la faute (Shapiro, 1983). Dans cette optique du *diagnostic déclaratif*, on trouve les travaux de Ferrand et al. du projet *OADymPPaC* (Ferrand *et al.*, 2003) qui proposent une approche de localisation de contrainte qui est à l'origine d'un *symptôme* basée sur les explications. Le *symptôme* traité ici représente l'absence de solution. En partant d'une sémantique attendue qui est un sous-ensemble de solutions acceptables et non-acceptables, on parcourt un arbre de preuve pour trouver une explication minimale au retrait d'une solution valide.

Le premier cadre pointillé de la figure 1.1 représente la méthodologie générale suivie dans une mise-au-point pour la correction des programmes à contraintes.

1.3.2. Mise-au-point pour la performance

Une mauvaise modélisation et/ou une stratégie de recherche inadéquate affectent les performances. Il n'existe aucune méthode générale qui permet de donner les raisons précises d'une mauvaise performance d'un programme donné, néanmoins, on trouve une palette d'outils qui aident le développeur à mieux comprendre l'exécution complexe de son programme et voir ses faiblesses. Certains d'entres eux présentent le contrôle de l'exécution, d'autres suivent de près *le store* des contraintes ou encore les contraintes globales. En général, dans une mise-au-point pour la performance, il est rare d'expliquer

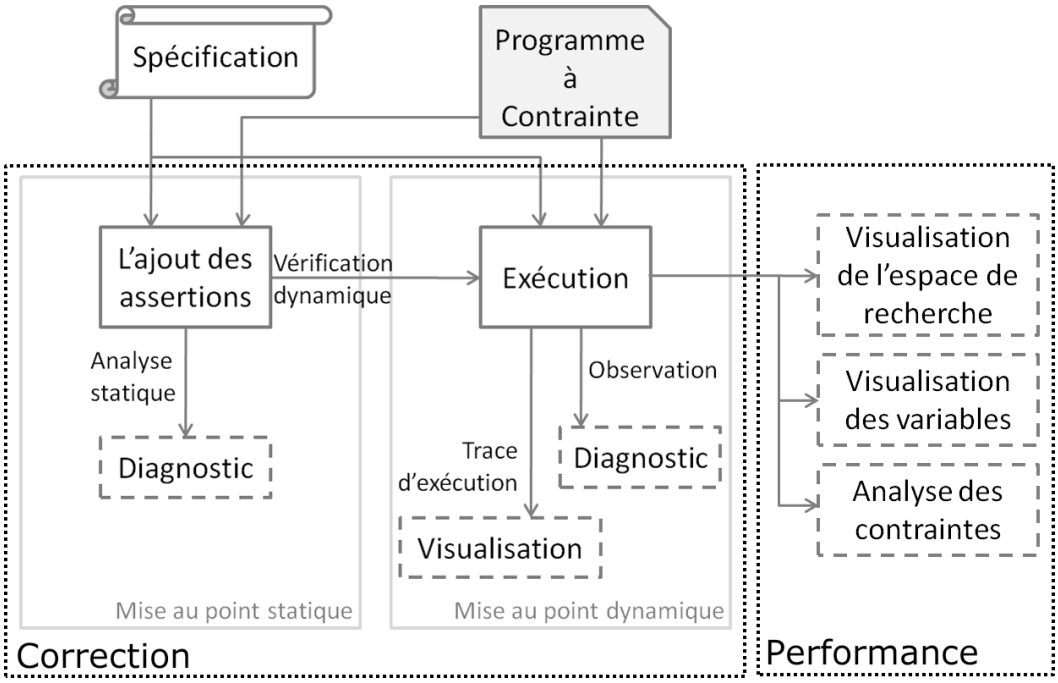


FIGURE 1.1.: Méthodologie de base pour la mise-au-point de correction (Deransart *et al.*, 2000b).

une mauvaise performance à travers une seule contrainte, mais plutôt avec l'interaction entre plusieurs contraintes.

1.3.3. Outils de mise-au-point des programmes à contraintes

Les travaux autour de mise-au-point des programmes à contraintes ont produit un éventail assez large d'outils.

Analyseurs syntaxiques

La plupart des langages de modélisation en PPC proposent un environnement de développement avec des analyseurs syntaxiques. Par exemple, les distributions d'OPL et de Comet incluent des analyseurs propres à la syntaxe du langage. D'autres langages comme Choco et Gecode font appel aux analyseurs des langages sous-djacents (resp. Java et C++).

La mise-au-point, à travers ces analyseurs, permet de remédier aux fautes d'ordre syntaxique, comme l'oublie d'un " ;" ou une mauvaise déclaration de variable. Ce type de fautes est facile à détecter et à corriger. Les plus difficiles à atteindre sont les fautes d'ordre sémantique qui peuvent altérer une solution attendue.

Analyseurs de trace post-mortem

Les travaux sur la mise-au-point des programmes à contraintes ont abouti à la définition de modèles de trace génériques (Deransart *et al.*, 2000b; Langevine *et al.*, 2001) et à la réalisation d'outils d'observation et d'analyse de traces post-mortem, tels que Codeine pour Prolog, Morphine (Langevine *et al.*, 2001) pour Mercury, ILOG Gentra4CP, ou encore JPalm/JChoco. SICStus Prolog intègre désormais un outil d'analyse de trace nommé fdbg pour les programmes clpfd qui est capable de montrer les étapes de propagation de contraintes ainsi que les réductions des domaines effectuées.

Ces outils permettent de suivre l'exécution, avoir un aperçu sur les variables, les domaines et les contraintes. En revanche, la partie la plus dure dans ce type de mise-au-point est laissée au développeur qui analyse le modèle pour savoir si les contraintes posées sont celles répondant au problème ou non.

Outils d'explication des échecs

Une faute dans un système de contraintes peut réduire l'ensemble des solutions à être vide comme elle peut introduire et/ou enlever des "solutions". Pour le premier cas de figure, il existe des travaux qui essaient de trouver l'explication minimale de l'insatisfiabilité du système. L'algorithme QuickXplain de Junker (Junker, 2004) est l'une des références majeures. Cet algorithme utilise un schéma dichotomique pour retourner ce qu'on appelle le noyau insatisfiable. Bailey et Stuckey proposent dans (Bailey & Stuckey, 2005) un algorithme qui retourne toutes les explications minimales en utilisant la dualité qui existe entre un ensemble minimal insatisfiable et un ensemble maximal satisfiable. Les explications des échecs sont utilisées pour bien choisir des points de retours dans la

recherche, ainsi faire du backtrack intelligent. Le système Palm pour Choco (Jussien & Barichard, 2000) est une implémentation de ce principe.

Ces outils et techniques sont dédiées à la détection des échecs des programmes à contraintes pour une correction et/ou une optimisation du modèle et de la propagation. Par contre, ces techniques sont inopérantes dans le cas où le programme à contraintes accepte une *solution* qui ne répond pas au problème posé.

Outils de visualisation de trace

Il existe des outils qui permettent de visualiser à posteriori une exécution du programme avec un suivi de trace des domaines des variables, des contraintes, etc. On peut citer l'outil CLPGUI (Fages *et al.*, 2004) qui offre une interface graphique et intuitive à la visualisation des traces, l'outil de visualisation de Comet (Dooms *et al.*, 2007) qui permet un affichage et un suivi des algorithmes CBLS (Constraint-Based Local Search) en temps réel. CP-Viz (Simonis *et al.*, 2010) est une autre plateforme de visualisation qui est générique et peut s'interfacer avec différents systèmes de contraintes comme Sicstus Prolog ou ECL^{i}PS^e. Cette plateforme permet de visualiser l'arbre de recherche ainsi que l'état des contraintes et des variables indépendamment des systèmes de contraintes. L'échange d'information entre les différents solveurs et CP-Viz est sous forme XML.

Encore une fois, ces outils de visualisation des traces offrent une assistance au développeur pour mieux cerner les fautes et comprendre un défaut de performance, mais ne comparent pas les résultats observés à une référence.

1.4. La PPC dans le monde industriel

Cette section a pour objectif de faire un survol rapide des domaines d'application de la PPC et de son impact dans le monde industriel. La PPC a fait ses preuves sur différents terrains d'applications qu'on peut regrouper par secteur : transport, production, etc.

Un des secteurs importants qui touche le quotidien de chacun d'entre nous est *le transport*. Ce secteur fait appel à des solutions informatisées sur différents niveaux. La PPC est présente dans des applications de gestion comme la gestion du trafic ferroviaire (Chiu *et al.*, 2002; Rodriguez & Kermad, 1998) et la gestion des aéroports : du service tapis roulant des bagages à l'affectation d'équipages (e.g., Airbus). La PPC répond aussi à des besoins de planification dans ce secteur. P. Flener et al. proposent dans (Flener *et al.*, 2007a) un modèle à contraintes qui modélise la planification et le contrôle du trafic aérien sur un espace aérien dense.

Ces dernières années, plusieurs projets collaboratifs entre des industriels et des chercheurs concernaient l'intérêt de la PPC dans le secteur de la gestion du traffic. A titre d'exemple, INRETS (Institut national de recherche sur les transports et leur sécurité) et la SNCF étaient impliqués dans un projet en l'an 2000 qui avait comme objectif de développer un support de système de décision pour le contrôle du trafic ferroviaire (Rodriguez, 2000). Le but spécifique du projet était le développement de différents modèles

à contraintes qui gèrent la circulation des trains. Un autre exemple est TESTEC qui représente un projet ANR en collaboration avec Geensys. Un des objectifs du projet est le développement de stratégies dynamiques, basées sur des contraintes, de génération de données de test des programmes critiques écrits en C ou en Java (Collavizza *et al.*, 2011).

Dans le secteur de *la production*, la PPC a permis aux grands industriels de l'automobile de réduire considérablement les coûts de leurs productions comme Daimler-Chrysler, Nissan, Peugeot-Citroen et Renault. Ces constructeurs font appel à la PPC pour la planification des tâches de production et de séquencement des opérations.

Par ailleurs, la Société Française de Recherche Opérationnelle et d'Aide à la Décision (ROADEF) a l'habitude de lancer des challenges dans le but de permettre aux industriels d'avoir une meilleure perception des développements récents dans le domaine de la recherche, ainsi confronter les chercheurs à de nouvelles problématiques. Le challenge de 2005 (Sanlaville, 2005) avait comme sujet l'ordonnancement des véhicules pour une chaine de montage automobile, sujet proposé par Renault.

D'autre part, beaucoup d'entreprises et d'ingénierie logicielle spécialisées en optimisation, planification et gestion proposent des solutions PPC. Les services peuvent inclure la gestion des entrepôts (e.g., KLS Optim [7]), l'affectation de fréquences et de bande passante, les emplois du temps (e.g., Dynadec [8] et IBM ILOG [9]), etc.

Un autre secteur comprenant un enjeu financier majeur est *le commerce électronique*. La PPC a permis de définir des applications en ligne qui gèrent les stocks, les commandes, les comptes des clients, etc. Á titre d'exemple, le marché du crédit connait des solutions PPC avec de nouvelles méthodes de conception de portefeuille virtuel dans la finance, les *Portfolio Designs* (Flener *et al.*, 2007b).

On trouve aussi les systèmes de recommandation (*RS*) à base de contraintes. Ces systèmes aident les clients dans leurs choix de produits et de services. Les problèmes de recommandation sont des problèmes qui ont connu un intérêt majeur des applications IA depuis les années 90 avec le début du *e-commerce*. Au départ, les RSs étaient assez simples avec des recommandations sur des livres ou des films ; les systèmes actuels sont plus complexes avec une multitude de produits et/ou de services. Ce qui augmente aussi la complexité de ces systèmes est notamment la notion de dominance et de préférence (Felfernig & Burke, 2008) sur les produits et les services. Il existe également des systèmes qui expriment les critiques postées sur un produit donné à l'aide de contraintes pour définir de nouvelles dominances (McSherry & Aha, 2007).

7. www.klsoptim.com
8. dynadec.com
9. www-01.ibm.com/software/fr/websphere/ilog/

1.5. Conclusion

Nous avons essayé à travers ce chapitre de faire un survol rapide de la PPC, ses différents langages et sa mise-au-point. Nous avons également essayé de mettre en évidence l'impact de cette programmation sur l'industrie actuelle, plus particulièrement, les applications critiques qui engendrent le besoin d'approche plus systématique pour leurs validations. š

2. Test et mise-au-point des programmes conventionnels

2.1. Théories de test existantes

Théorie de test de Goodenough et Gerhart

Le premier formalisme du test de logiciel est la théorie du test de Goodenough et Gerhart (Goodenough & Gerhart, 1975). Elle décrit un cadre général du test et introduit la notion de *critère* de sélection des données de test qui permet de spécifier formellement un objectif de test. Un des concepts fondamentaux dans cette théorie est la sélection de données de test : Soit P un programme sous test et D le domaine d'entrée du programme, le sous-ensemble $T \subseteq D$ représente un ensemble de données de test sélectionnées par un critère C. Le critère de sélection C étant un ensemble de prédicats de test c. On dit qu'un ensemble T est une sélection de données de test du critère C si et seulement si le prédicat $COMPLET(T, C)$ est satisfait :

$$COMPLETE(C, T) \equiv (\forall c \in C, \exists t \in T \ : c(t)) \wedge (\forall t \in T, \exists c \in C \ : c(t))$$

où $c(t)$ dénote que la donnée de test t est validée par le prédicat c.

L'entité *oracle* dans cette théorie est représentée par un prédicat OK qui permet de dire si une exécution du programme sous une donnée de test $P(t)$ atteint un *succès* (i.e., $OK(t) = vrai$) ou un *échec* (i.e., $OK(t) = faux$). Une suite de tests T est réussie si toute donnée de test t est validée par l'oracle (i.e., $SUCCESSFUL(T) \equiv \forall t \in T : \ OK(t)$). Cette théorie introduit également la notion de validité et de fiabilité d'un critère de test. Un critère de test C est dit *valide* si pour tout programme incorrect, il existe une suite de tests non réussie satisfaisant le critère (i.e., $VALIDE(C)$). On dit qu'un critère de test est *fiable* s'il produit uniquement des suites de tests réussies ou des suites de test non réussies (i.e., $RELIABLE(C)$). Le théorème fondamental dans cette théorie définit le test *idéal* :

$$(\exists T \subseteq D)(COMPLETE(T, C) \wedge RELIABLE(C) \wedge VALID(C) \wedge SUCCESSFUL(T))$$
$$\Rightarrow (\forall d \in D) OK(d)$$

La théorie de Goodenough et Gerhart a fait l'objet de plusieurs critiques. En particulier, elle a été remise en cause par Howden dans (Howden, 1976) et Ostrand et Weyuker (Weyuker & Ostrand, 1980) en montrant qu'il est difficile d'avoir un critère de test *idéal* qui soit à la fois fiable et valide et qui permet de prouver la correction du programme.

Théorie de test de Gourlay

On note également la théorie de test définie par Gourlay (Gourlay, 1983) qui présente un cadre formel de test. Ce cadre unifie les travaux précédents de Goodenough, Gerhart et Howden sur le test et définit un système de test par un 5-uplet $< \mathcal{P}, \mathcal{S}, \mathcal{T}, corr, ok >$ avec $\mathcal{P}$, $\mathcal{S}$ et $\mathcal{T}$ des ensembles de programmes, de spécification et de suites de tests. Le prédicat $corr(p, s)$ signifie que le programme p est correct par rapport à une spécification s. De même, le prédicat $ok(p, s, t)$ permet de dire que le test t sur p est validé par s. On trouve également une définition d'une méthode de test par une fonction $\mathcal{M}$ qui permet de générer des suites de test (i.e., $\mathcal{M} : \mathcal{P} \times \mathcal{S} \rightarrow \mathcal{T}$).
Dans cette théorie, on s'intéresse à la puissance d'une méthode de test et à détecter des fautes. On dit qu'une méthode M est aussi puissante qu'une méthode N si et seulement si les fautes détectées par N sont également détectées par M. La figure 2.1 montre deux cas possibles où une méthode M est meilleure qu'une méthode N avec, respectivement, (T_M, F_M) et (T_N, F_N) la suite de tests et les fautes détectées par M et N.

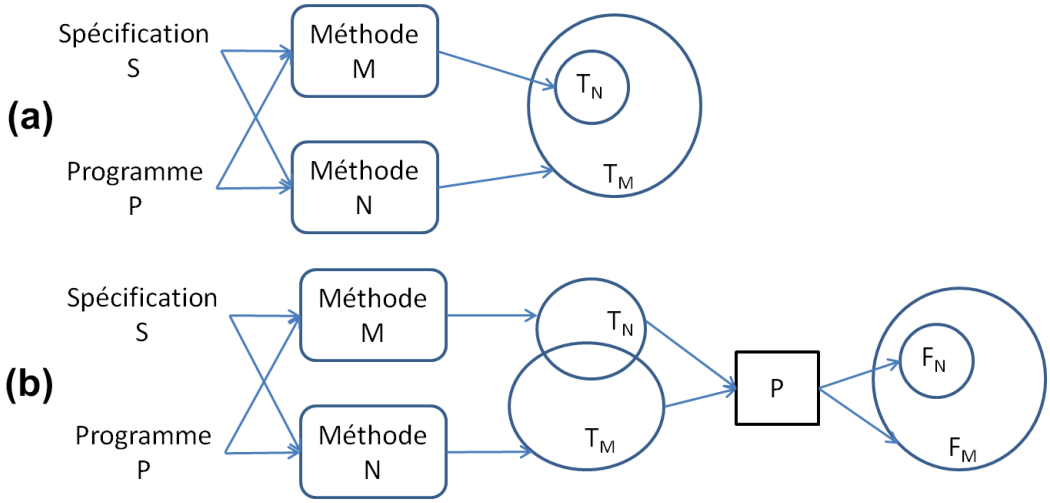


FIGURE 2.1.: Comparaison des méthodes de test.

Par ailleurs, la théorie de test de Bernot et al. (Bernot *et al.*, 1991) s'intéresse au problème de l'oracle. Cette théorie montre qu'à partir d'une spécification algébrique, il est possible de définir un oracle de test.

La figure 2.2 donne un schéma général du test basé sur les critères de sélection.

Les théories de test discutées précédemment semblent être des schémas inopérants pour capter les spécificités de test des programmes à contraintes et cela pour différentes raisons :

– En PPC, un problème est modélisé par un ensemble de contraintes non ordonné. Il

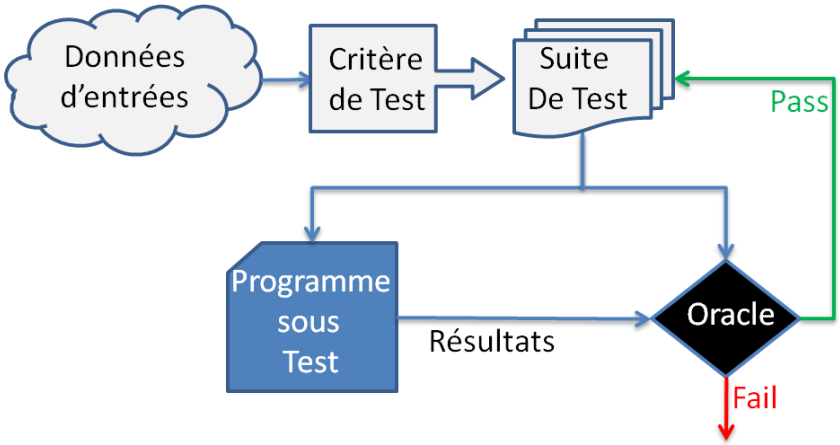


FIGURE 2.2.: Schéma général d'une méthode de Test.

n'y a donc pas de notion de séquence, comme on en trouve dans les programmes conventionnels. Le flot de contrôle dans les programmes conventionnels est guidé par les données où toute trace d'exécution dépend des entrées du programme. En revanche, le flot de contrôle dans un programme à contraintes est guidé par les contraintes où, durant une résolution, l'ordre dans lequel les contraintes sont traitées ne dépend pas des entrées du programme.

- Les cinq éléments de base pour examiner l'hypothèse $\mathcal{H}$, à savoir *(données en entrée, objet à tester, oracle de test, résultats obtenus, résultats attendus)*, doivent être bien définis pour les programmes à contraintes, alors que pour les programmes conventionnels seul l'oracle est à définir, les autres entités étant triviales.

- Un programme conventionnel essaie d'encoder une solution à un problème donné suivant une algorithmique définie préalablement, alors qu'un programme à contraintes modélise le problème et capture implicitement son ensemble de solutions.

- L'optimisation et le raffinement du code est une phase susceptible d'introduire des fautes dans n'importe quel type de programmation. Les techniques appliquées sur des programmes conventionnels sont complètement différentes de celles des programmes à contraintes. Nous y reviendrons sur ce point un peu plus loin dans ce manuscrit.

Ainsi, les points précédents nous conduisent à des modèles de fautes différents. Une faute dans un programme à contraintes peut ajouter ou enlever des *solutions* au problème.

2.2. La localisation des fautes

Une des tâches couteuses dans la mise-au-point des programmes est la localisation des fautes (Vessey, 1985). Cette section donne un bref aperçu de cette activité. Cette localisation entre dans un cadre de test bien défini où elle exploite les suites de test et les verdicts pour apprendre le maximum sur la faute commise dans le programme. Une suite de tests met en évidence l'écart entre le résultat attendu et celui obtenu, cet écart révèle l'existence de faute dans le programme. Le développeur est amené par la suite à chercher l'endroit (instruction, module, fichier) où la faute a été introduite, sachant que le test ne détecte que les effets de la faute. Réduire le temps de localisation d'une faute est un réel besoin dans le développement logiciel.

Face à un programme conventionnel qui contient une faute, le développeur doit identifier le bloc d'instructions qui est à l'origine de cette faute. Aucune technique ne permet une telle identification de manière sûre. Les techniques existantes renvoient des instructions de la plus suspecte à la moins suspecte. La plus intéressante des approches est celle proposée par Jones, Harrold et Stasko (Jones *et al.*, 2002). Sous l'hypothèse que les fautes ne se trouvent que sur des traces de cas de test non réussi, l'approche prend une suite de tests et compare les traces d'exécution réussie et non-réussie pour détecter les instructions susceptibles de contenir la faute. Cette approche est implémentée dans l'outil de visualisation Tarantula (Jones & Harrold, 2005) qui aide à la localisation des fautes. Cet outil prend toutes les traces d'exécution d'une suite de tests (succès/échec) et retourne un ordre sur des instructions suspectes. Une instruction est d'autant plus suspecte si elle apparait dans la plupart des traces de tests non-réussis. Dans la même optique, le système DeLLIS utilise la fouille de données pour analyser les différentes traces d'exécutions du programme. L'idée est de construire un *treillis des défaillances* à partir d'un ensemble de règles d'association de type ($trace_i \rightarrow Echec$), la faute est alors localisée suite à un parcours du treillis du bas vers le haut (Cellier *et al.*, 2009).

Une autre démarche qui est assez proche de celle de Jones et al. est celle proposée par Cleve et Zeller (Cleve & Zeller, 2005) qui compare les états entre une exécution réussie et une autre non-réussie. Dans un travail précédent (Zeller, 2002), Zeller montre qu'une identification de la faute en se basant sur une recherche dans l'espace (variables/valeurs) n'est pas suffisante. En effet, il est possible de retourner des instructions fautives en comparant les états des variables et des valeurs sur des traces réussies et non-réussies. Seulement, une faute dans un programme peut entrainer une différence sur tous les états du programme. Dans (Cleve & Zeller, 2005), une autre recherche dans le temps est considérée. Cette recherche détermine la première transition qui mène vers un échec et élimine par la suite toute conséquence de cette transition qui peut produire une différence sur les états du programme.

La technique de localisation des programmes conventionnels qui consiste à comparer les traces, les états et les transitions des exécutions ne peut prendre en compte les programmes à contraintes. Ces techniques exploitent la séquentialité des programmes et le

flot de contrôle qui est dirigé par les données. Or, il n'y a pas de séquentialité dans un système de contraintes, car le flot de contrôle est dirigé par les contraintes et l'interaction entre elles.

2.3. La correction automatique

Le travail de mise-au-point logicielle ne s'arrête pas à la localisation des fautes. Le développeur est amené à corriger ces fautes. La correction automatique des programmes conventionnels est une nouvelle tendance dans le développement du logiciel. De récents travaux témoignent de l'intérêt que portent les chercheurs en génie logiciel à ce nouveau domaine. Toujours dans un cadre de test, étant donné un programme contenant une faute, l'objectif est de corriger systématiquement la faute de sorte à faire passer les tests non-réussis.

On peut distinguer deux approches différentes pour une correction automatique, la première propose une correction à partir du code existant, la deuxième fait appel à une spécification formelle.

2.3.1. Correction depuis un code existant

Dans (Weimer *et al.*, 2010), Westley et al. proposent une méthode basée sur la localisation et un algorithme évolutionniste (i.e., algorithme génétique). La méthode génère des variants du programme à l'aide d'opérateurs génétiques de croisement et de mutation. Parmi les variants on sélectionne un programme qui contient une correction possible.

Une autre méthode, un peu différente, est celle proposée par Debroy et Wong dans (Debroy & Wong, 2010). La méthode combine la localisation de faute avec Tarantula (Jones & Harrold, 2005) et des techniques de mutation pour produire automatiquement une correction éventuelle.

Dans la même optique, on peut noter l'outil BugFix (Jeffrey *et al.*, 2009) qui offre une assistance aux développeurs dans la phase de correction. L'outil implémente une approche basée sur l'*apprentissage automatique* à partir d'un débogage statique et/ou dynamique. En analysant les situations de débogage, l'outil met à jour la base d'apprentissage pour pouvoir proposer par la suite des suggestions de correction.

2.3.2. Correction avec spécification formelle

Un exemple de ce type de correction est l'outil AutoFix-E. Cet outil permet une correction automatique des programmes écrits en Eiffel (Wei *et al.*, 2010). L'approche peut être appliquée sur d'autres langages qui implémentent le concept de programmation par contrat comme Spec#, ou sur des langages existants avec une extension pour exprimer des contrats (e.g., JML pour Java). Un contrat d'une classe donnée représente sa spécification formée de pré-conditions, post-conditions et des assertions intermédiaires. En analysant les contrats et la différence entre les tests réussis et non réussis, l'outil génère des propositions de correction de façon automatique. La première étape est la génération de cas de test en utilisant l'outil AutoTest de Eiffel. À partir de ces cas de test, des états

d'objet sont extraits en utilisant des requêtes booléennes. Puis, un profil de faute est généré après une comparaison entre des traces d'exécution réussie et non-réussie. Ensuite, à partir des transitions d'état d'une exécution réussie on produit un *modèle de comportement à état fini* qui capture le comportement normal en termes de contrôle. Ce modèle et le profil de faute permettent par la suite de générer de corrections éventuelles. Seules les corrections qui passent les tests de régression sont retenues à la fin. Parmi 42 fautes dans des programmes en Eiffel, AutoFix-E a permis de corriger 16 d'entre elles (Wei *et al.*, 2010). Ces corrections sont identiques, ou du moins proches des corrections proposées par les développeurs.

L'approche de correction, qui se base sur les traces d'exécution des cas de test et des opérateurs de mutation, semble être inopérante dans le cas d'un programme à contraintes sans un cadre de test bien défini préalablement. En revanche, la méthode formelle pour la correction automatique reste possible si un langage en PPC propose un langage d'assertions pour décrire les contraintes.

2.4. Résumé

Dans les deux premiers chapitres, nous avons essayé de donner le contexte de travail avec un état de l'art diversifié. Cette diversité est liée à la nature de nos contributions. Le premier chapitre fait un survol rapide sur la PPC, sa mise-au-point et son impact industriel qui justifie pleinement le besoin en génie logiciel autour de la validation des programmes à contraintes. Le deuxième chapitre concerne les théories de test et la mise-au-point des programmes conventionnels. Á notre connaissance, aucune théorie et/ou outil de test spécifique n'existe pour la détection de fautes dans les programmes à contraintes. La mise au point dynamique des programmes conventionnels a connu une avancée importante qui a produit plusieurs outils de test, de localisation et, récemment, de correction automatique, mais aucun d'entre eux n'est applicable aux modèles à contraintes écrits dans les langages OPL, Comet, etc.

Deuxième partie .

Théorie de Test des Programmes à Contraintes

3. Modélisation en PPC

3.1. Introduction

En programmation par contraintes, la partie modélisation est très importante. Une expertise est exigée pour avoir un modèle à contraintes qui respecte la spécification de départ et qui conduit à une résolution efficace du problème.

Tout au long de ce chapitre, nous allons détailler la modélisation en PPC telle qu'elle se pratique avec les différents raffinements. De plus, nous illustreront cette modélisation sur un problème académique qui présente des caractéristiques intéressantes en PPC, le problème des *règles de Golomb* (Rankin, 1993).

Ces règles trouvent leur terrain d'application dans des domaines variés tels les communications radio, rayons X en cristallographie, les codes convolutionnels doublement orthogonaux, tableaux des antennes linéaires, communications PPM (Pulse Phase Modulation). Pour une lecture approfondie sur les différentes façons de modéliser le problème des règles de Golomb, nous invitons le lecteur à consulter les références suivantes (Smith *et al.*, 1999; Cotta *et al.*, 2007).

3.2. Modèle-Oracle

Étant donnée une spécification d'un problème, le développeur commence par traduire cette spécification en un premier modèle à contraintes sans se soucier des questions de performance, d'optimisation et de raffinement. Ce premier modèle représente un point de départ pour une modélisation optimisée.

Une spécification du problème des règles de Golomb est donnée comme suit :

> *« Une règle de Golomb est définie comme un ensemble de m entiers $0 \leq x_1 < x_2 < ... < x_m$ tel que les $m(m-1)/2$ différences $(x_j - x_i)$, $1 \leq i < j \leq m$, sont distinctes. On dit qu'une telle règle est d'ordre m, et qu'elle est de longueur x_m. L'objectif est de trouver une règle de longueur minimale ».*

Traduire le texte précédent en un modèle à contraintes revient à :
- faire sortir les variables de décision et les paramètres d'entrée,
- exprimer l'ensemble des contraintes,
- exprimer la fonction d'optimisation.

Pour les règles de Golomb, les variables de décision représentent l'ensemble des marques (vecteur x) alors que la longueur de la règle m représente un paramètre du problème.

Le premier ensemble de contraintes à traduire concerne l'ordre sur les marques :

$$\forall i, j \in 1..m : \qquad i < j \Rightarrow x_i < x_j$$

Un autre ensemble de contraintes à traduire est celui des distances entre les marques :

$$\forall i, j, k, l \in 1..m : \qquad (i < j) \wedge (k < l) \wedge (i \neq k) \wedge (j \neq l) \Rightarrow x_j - x_i \neq x_l - x_k$$

Les règles de Golomb est un problème d'optimisation où on cherche une règle de longueur minimale, ce défi de minimisation revient à définir une fonction objectif notée $f(x)$. La longueur d'une règle est exprimée par la dernière marque x_m, notre fonction objectif est par conséquent $f(x) = x_m$: $minimize\ f(x)$

```
using CP;

int m=...;

dvar int x[1..m] in 0..m *m;

minimize x[m];

subject to{

c1: forall(i in 1..m - 1)
     x[i]  < x[i+1] ;

c2: forall(ordered i,j in 1..m)
    forall(ordered k, l in 1..m:
                  (i!=k||j!=l))
     !((x[j]-x[i]) == (x[l]-x[k]));
}
```

(A)

```
using CP;

int m=...;
int nbDist= m*(m-1)div 2;

dvar int x[1..m] in 0..m*m;
dvar int d[1..nbDist];

minimize d[m-1];

subject to {

cc1: forall (i in 1..m-1)
      x[i] < x[i+1];

cc2: forall(ordered i,j in 1..m)
      d[(nbDist-((m-i+1)*(m-i)div 2))+(j-i)]
                                  == x[j]-x[i];
//cc2': forall(ordered i,j in 1..nbDist div m)
//      d[nbDist-(j-i)] == x[j]-x[i];
                                          faute
cc3: x[1]==0;

cc4: x[2] <= d[nbDist];

cc5: x[m] >= nbDist;

cc6: allDifferent(d);

cc7: forall(ordered i,j in 2..m, k in 1..m*m)
      x[i]==x[i-1]+k => x[j]!=x[j-1]+k;
}
```

(B)

FIGURE 3.1.: Premier modèle **(A)** et programme à contraintes optimisé **(B)** du problème des règles de Golomb.

La partie **(A)** de la figure 3.1 donne ce premier modèle écrit en OPL. Un développeur prend ce premier modèle et l'optimise en appliquant différentes techniques de raffinement PPC dans le but d'avoir à la fin un programme à contraintes prêt à résoudre des instances difficiles du problème. La partie **(B)** de la figure 3.1 montre un programme à contraintes

des règles de Golomb en OPL suite à des raffinements. Ces raffinements représentent un ajout d'une nouvelle variable de décision `d` des distances entre les marques, des contraintes de connexion `cc2` pour relier `d` aux variables de base, un raffinement sur la fonction objectif, des contraintes de symétries (i.e ., `cc3` et `cc4`), des contraintes redondantes (i.e ., `cc5` et `cc7`) et une contrainte globale `cc6`.

Soumettre le modèle **(A)** au solveur de contraintes peut s'avérer couteux (mémoire / temps d'exécution). Un exemple serait une instance de règle $m = 11$, le programme **(B)** arrive à rendre une solution (x_i = [0 1 9 19 24 31 52 56 58 69 72]) [1] après un temps de résolution de 7.83 *minutes* et une consommation mémoire de 735.2 Ko [2]. Le modèle **(A)** retourne la même solution après 46.39 *minutes* et une consommation mémoire de 2.3 Mo.

Le modèle (A) de Golomb traduit une vue possible sur le problème. Il est possible de traduire de plusieurs manières un problème en un modèle à contraintes selon les points de vue des développeurs. Á titre d'exemple et dans (Nadel, 1990), Nadel donne neuf manières différentes pour modéliser le problème des *n-reines*.

L'*oracle* est un mécanisme important dans un processus de test, il représente la référence de test qui permet de déterminer si un test est validé ou pas. L'oracle permet de définir des procédures décisionnelles qui interprètent les résultats des tests. Il peut prendre différentes formes : un ensemble de propriétés exprimées dans une certaine logique, une spécification algébrique, un ensemble de post-conditions, etc.

Notre référence de test pour les programmes à contraintes, nommée *Modèle-Oracle*, est un modèle à contraintes qu'on note $\mathcal{M}_x(k)$. Ce Modèle-Oracle caractérise l'ensemble des solutions du problème tout en réstant conforme aux exigences de l'utilisateur. Il est déclaratif, simple, fidèle à la spécification où chaque contrainte traduit une partie importante de la spécification. Par exemple, le modèle **(A)** de la figure 3.1 est un Modèle-Oracle pour le problème des règles de Golomb.

Définition 1 (Modèle-Oracle) *Soit un problème P défini par une spécification $\mathcal{S}$. Un modèle-Oracle $\mathcal{M}_x(k)$ du problème P est un ensemble de contraintes C_i qui traduisent la spécification $\mathcal{S}$:*

$$\mathcal{M}_x(k) \equiv C_1 \wedge \ldots \wedge C_n$$

Avec x un vecteur de variables et k un vecteur de paramètres.

Le Modèle-Oracle ici est paramétré par k qui prend des valeurs dans un ensemble noté $\mathcal{K}$. Le vecteur k représente l'ensemble des paramètres du problème. Ces paramètres instanciés caractérisent une instance faisable du problème. Ce paramétrage peut affecter le nombre de variables, le nombre de contraintes, la taille des domaines, l'expression des contraintes, l'expression des domaines des variables, etc. Le vecteur x représentent l'ensemble des variables du modèle qui prennent un domaine donné (i.e., $\forall x_i : \ x_i \in Dom_{x_i}$).

1. Solveur : IBM Ilog CP optimizer 2.3
2. Machine : Intel Core2 Duo CPU 2.40Ghz, 2 GB de RAM

Hypothèse 1 *Le Modèle-Oracle $\mathcal{M}_x(k)$ modélise fidèlement la spécification du problème.*

Hypothèse 2 *Le Modèle-Oracle $\mathcal{M}_x(k)$ représente au moins une solution pour chaque instance k_i du problème.*

L'hypothèse 2 permet d'exclure du cadre de test qu'on veut définir les instances du problème qui sont insatisfiables de nature. Ceci nous évitera de traiter la question d'équivalence dans la classe des problèmes insatisfiables.

3.3. Raffinement des modèles en PPC

3.3.1. Structures de données et contraintes de connexion

La première technique de raffinement est l'ajout de nouvelles structures de données, de variables auxiliaires et de contraintes de connexion. Enrichir le modèle avec de tels éléments permet d'augmenter l'expressivité et donne la possibilité de définir de nouvelles contraintes. Ajouter de nouvelles variables de décision qui peuvent être des structures de données de haut niveau (tableaux, matrices, structures composées, etc.) augmente la taille initiale de l'espace de recherche. Il permet aussi bien une factorisation au niveau des contraintes existantes, ce qui augmente la puissance de filtrage et de propagation, ainsi, une réduction rapide de l'espace de recherche durant la résolution. Ajouter des variables de décision tout en augmentant la puissance de résolution, avec de nouvelles contraintes, sans pour autant exploser l'espace de recherche de départ semble être une tâche qui demande une bonne expertise.

Les variables de base donnent un support aux solutions du problème. Par contre, les variables auxiliaires capturent une relation globale entre les variables de base.

Exemple 4 (Factorisation des variables) *Le problème de l'ordonnancement de véhicules (car sequencing) revient à placer des véhicules sur une chaine de montage de sorte à respecter les contraintes de capacité des installateurs des options et la demande en option des véhicules. La solution est représentée par une chaine de montage (instantiation du vecteur* **slot***). L'ajout d'une variable auxiliaire* **setup[o,s]** *capture la relation « l'option* **o** *est installée au véhicule placé dans le slot* **s** *».*

Pour Golomb, le programme (B) introduit une nouvelle variable de décision `d` pour capturer les distances entre les marques ($d_{ij} = x_i - x_j$). Étant donnée une instantiation sur les variables de base `x`, ceci donne une valeur au vecteur `d`.

L'ajout de variables auxiliaires implique nécessairement un ajout de *contraintes de connexion*. Ces contraintes ont pour objectif d'exprimer la connexion entre les variables de base et ces nouvelles variables auxiliaires. Ainsi, ces contraintes permettent d'utiliser ces nouvelles variables dans de nouvelles contraintes. Par exemple, utiliser un *allDifferent* sur les distances `d` (i.e., `cc6`).

La contrainte `cc2` du programme **(B)** (Figure 3.1) est un exemple de contraintes de connexion. Ce type de contrainte relie les variables de distance `d` aux variables de base `x`

où une variable de distance d_k représ ente la distance $x_j - x_i$ (i.e., $k = nbDist - ((m - i + 1) * (m - i)/2) + (j - i)$

Exemple 5 *Soit une règle de Golomb de $m = 4$ marques (i.e., $x_1, \ldots, x_4$), nous obtenons 6 différentes distances (i.e., $d_1, \ldots, d_6$) :*

$$(d_k = x_j - x_i) \wedge (k = 6 - ((4 - i + 1) * (4 - i/2) + (j - i))$$

$$\left| \begin{array}{l} d_1 = x_2 - x_1 \\ d_2 = x_3 - x_1 \\ d_3 = x_4 - x_1 \end{array} \right| \left| \begin{array}{l} d_4 = x_3 - x_2 \\ d_5 = x_4 - x_2 \end{array} \right| \left| \begin{array}{l} d_6 = x_4 - x_3 \end{array} \right|$$

3.3.2. Contraintes impliquées ou redondantes

Une contrainte impliquée, aussi appelée *redondante*, est le résultat d'une déduction ou une inférence sémantique sur les contraintes d'origine. Ces contraintes sont redondantes dans le sens sémantique où elles peuvent exprimer la sémantique (ou une partie) d'une ou plusieurs contraintes d'origine. En d'autres termes, ce sont des contraintes qui peuvent être omises sans affecter l'ensemble des solutions (Van Hentenryck, 1989b). Par exemple, si dans notre ensemble de contraintes nous avons $x < y$ et $y < z$, une conséquence logique est que $x < z$. Cette contrainte n'altère pas l'ensemble des solutions du problème, mais donne plus de puissance à la résolution. Dans (Cheng *et al.*, 1996) on retrouve la notion de *modélisation redondante* où on ajoute au modèle initial des modèles redondants qui sont sémantiquement équivalents. Les liens entre les modèles sont alors assurés à l'aide des contraintes de connexion.

La résolution peut être amenée à chercher un support de solution en instanciant des variables. Un long support qui ne conduit pas à une solution peut prendre un temps important. Dans ce cas, une décomposition de ce support avec une contrainte impliquée peut être utile.

Prenons la contrainte impliquée `cc7` du programme **(B)** de Golomb. Durant une instantiation du vecteur de variables x, si $(x_i = v) \wedge (x_{i+1} = v + k) \wedge (x_j = v')$, alors toute instantiation sur la variable x_{j+1} doit être différente de $v' + k$. Ceci permet d'éviter une instantiation complète sur x qui n'amène pas à une solution. Une telle contrainte permet de gagner du temps de recherche de solution dans des sous-problèmes insatisfiables.

D'autre part, les contraintes impliquées peuvent être considérées comme des contraintes qui éliminent, durant la recherche, des sous-arbres sans solution (i.e., *nogoods*). Dès lors, trouver des contraintes impliquées durant la recherche accélère la résolution.

Les contraintes impliquées qui sont des conséquences logiques des contraintes existantes (Charnley *et al.*, 2006) peuvent être générées de façon automatique. Ceci ne représente qu'une classe de contraintes impliquées. Hnich et al. ont donné une première classification des contraintes impliquées dans (Hnich *et al.*, 2003) avec des discussions sur la génération

automatique de chaque classe. Parmi ces classes, on trouve les contraintes globales qui peuvent remplacer plusieurs contraintes, des contraintes résultantes d'une élimination de variables, etc.

3.3.3. Contraintes de symétries

Les symétries dans les programmes à contraintes sont assez fréquentes. Elles peuvent être liées à la nature du problème comme elles peuvent être introduites durant la phase de modélisation. Par exemple, le problème des n-reines comprend de nature 16 symétries selon les rotations et les réflexions de l'échiquier. Une modélisation possible de ce problème est de définir une variable pour chaque position dans l'échiquier ($n \times n$ variables). Pour $n = 8$, le nombre de symétries est dans ce cas $2(9!)^2$ (Chapitre 10 : Symmetry in Constraint Programming (Rossi *et al.*, 2006)).

En PPC, les symétries représentent un sujet de recherche très actif (Puget, 2006; Cohen *et al.*, 2006; Flener *et al.*, 2009). Une *symétrie* est définie comme un groupe de *permutations*. Une permutation représente une correspondance entre deux éléments du même ensemble. De plus, l'*identité* d'une permutation est une correspondance entre un élément et lui-même. L'*inverse* d'une permutation est une permutation. La *composition* de plusieurs permutations donne aussi une permutation. Ceci dit, une symétrie préserve les solutions (non-solutions) d'un problème.

Une symétrie augmente artificiellement l'ensemble des solutions aussi bien que l'ensemble des non-solutions (l'espace de recherche). Le plus grand défi est de pouvoir *identifier* et *casser* les symétries.

La phase d'identification des symétries est la plus importante (Rossi *et al.*, 2006). J-F. Puget (Puget, 2005) a proposé une approche pour rendre cette phase automatique. Cette approche génère un graphe incluant toutes les symétries du programme à contraintes. Á partir de ce graphe, des générateurs de groupes de symétries sont calculés à l'aide d'un algorithme d'automorphisme de graphe. Cependant, il reste difficile de découvrir des symétries non-évidentes fortement liées à la nature du problème. La seconde phase est celle qui casse les symétries. Fondamentalement, il existe trois principales approches pour casser une symétrie. Les deux premières approches sont dites statiques. La première consiste à ajouter des contraintes dans le programme lors de son écriture. La seconde est une reformulation du problème de sorte à casser le maximum de symétries. La troisième est dynamique et consiste à ajouter des contraintes durant la résolution. La technique la plus connue dans cette catégorie est SBDS (*Symmetry Breaking During Search*) (Backofen & Will, 1998; Gent & Smith, 2000). Nous illustrons par la suite les approches statiques qui interviennent dans la phase de modélisation des règles de Golomb.

Il existe deux types de symétrie, symétrie de valeurs (i.e., permutations sur les valeurs) et symétrie de variables (i.e., permutations sur les variables).

Le problème de Golomb contient une symétrie de chaque type (valeurs/variables).

La première symétrie est une symétrie de valeurs. On peut passer d'une solution à

une autre solution en appliquant un décalage des valeurs (partie **(1)** figure 3.2). Casser cette symétrie revient à fixer la première marque de la règle à zéro (contrainte cc3 du programme **(B)** figure 3.1).

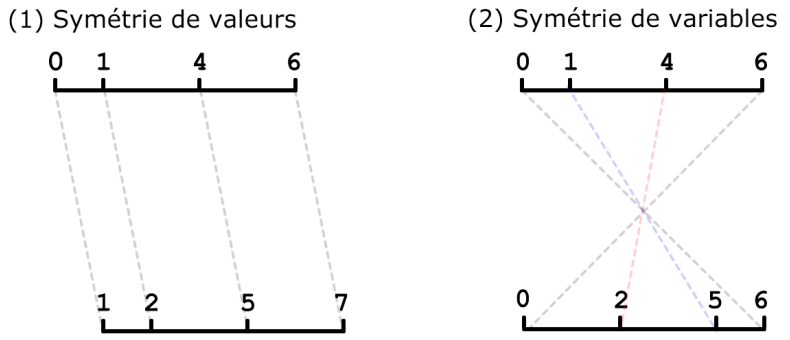


FIGURE 3.2.: Les symétries dans règles de Golomb.

La deuxième est une symétrie de variables. Une rotation des variables, comme la montre la partie **(2)** de la figure 3.2, permet d'obtenir des instantiations symétriques. La contrainte cc4 du programme **(B)** casse cette symétrie en imposant le fait que la distance entre les deux premières marques ($x_2 - x_1$) est strictement inférieure à la distance des deux dernières marques ($x_m - x_{m-1}$).

Casser les deux symétries précédentes du problème réduit drastiquement l'espace de recherche de Golomb. Pour un problème à huit marques, notre étude montre que l'espace de recherche dépasse la taille de 2.10^{14} avec 10^8 solutions faisables et seulement deux solutions optimales (problème d'optimisation). Casser la première symétrie réduit l'espace de recherche à 4.10^{12} avec 2.10^7 solutions faisables et conserve les deux solutions optimales. De plus, casser la deuxième symétrie réduit l'espace à 2.10^{12}, 10^7 solutions faisables et une seule solution optimale.

3.3.4. Contraintes globales

Les contraintes globales représentent en grande partie la puissance de la PPC. Le filtrage d'une contrainte permet d'avoir une vision locale sur un sous-problème (i.e., la contrainte en question), alors que la propagation permet d'avoir une vision globale sur tout le problème. Plus le filtrage au niveau contrainte est puissant, plus la propagation est forte sur les autres contraintes. Une contrainte globale permet de capturer l'interaction entre plusieurs contraintes simples et d'avoir un algorithme de filtrage plus puissant qu'un enchainement de filtrage sur ces contraintes simples. La contrainte globale est donc redondante dans le sens où elle représente une conjonction de contraintes simples.

Définir des algorithmes de filtrage efficaces et puissants pour des contraintes globales est un thème de recherche qui a bénéficié d'un intérêt important et qui a attiré l'attention de nombreux chercheurs (Régin, 2011).

D'un point de vue expressivité, il est plus simple d'avoir une contrainte globale qui modélise une partie importante du problème abordé et qui capture tout un ensemble de contraintes simples. Prenons par exemple la contrainte *allDifferent(X)* qui assure une valeur différente pour chaque variable $x_i \in X$. Il est plus simple d'utiliser cette contrainte qu'un ensemble de contraintes de différence ($x_i \neq x_j$).

Du point de vue performance, avoir un algorithme de filtrage pour une contrainte globale est plus efficace d'un point de vue déductif que faire du filtrage sur les contraintes simples. Un filtrage global prend en compte une relation globale entre les variables et les contraintes simples. Satisfaire la contrainte *allDifferent(X)* revient à trouver un couplage biparti entre les variables et les valeurs possibles (Régin, 1994). Ceci permet de filtrer un maximum de valeurs qui ne peuvent participer à une solution, ce qui n'est pas le cas si on prend les contraintes de différence une à une.

Exemple 6
$$\begin{cases} D(x_1) = \{1,2\} \\ D(x_2) = \{2,3\} \\ D(x_3) = \{1,3\} \\ D(x_4) = \{3,4\} \\ D(x_5) = \{2,4,5,6\} \\ D(x_6) = \{5,6,7\} \end{cases} \xrightarrow[allDifferent(x)]{Filtrage} \begin{cases} D(x_1) = \{1,2\} \\ D(x_2) = \{2,3\} \\ D(x_3) = \{1,3\} \\ D(x_4) = \{4\} \\ D(x_5) = \{5,6\} \\ D(x_6) = \{5,6,7\} \end{cases}$$
Un filtrage de la contrainte `allDifferent` *réduit le domaine de* x_4 *et* x_5 *alors qu'un filtrage local sur les contraintes de différences ne réduit le domaine d'aucune variable.*

Notre programme de Golomb (figure 3.1, partie (B)) fait appel à la contrainte globale `cc6` pour reformuler les contraintes de différence sur les distances.

Autre que la contrainte *allDifferent*, une des contraintes les plus utilisées est la contrainte *sum* :

$$sum(\vec{x}, \vec{a}, b) \equiv (b = \sum_{i=1}^{n} a_i x_i)$$

Aussi, la contrainte *element* qui indique que le i^{me} élément d'un vecteur x est égal à une valeur v (Hentenryck & Carillon, 1988).

La contrainte $pack(l, p, w)$ (voir *bin_packing* et *bin_packing_capa* dans le catalogue (Beldiceanu *et al.*, n.d.)) assure le chargement de m containers avec n articles de différent poids. Soit un container i et un article j, l_i représente la capacité du container, alors que p_j et w_j représentent respectivement l'emplacement et le poids de l'article :

$$pack(l, p, w) \equiv \forall i \in 1..m \sum_{j \in 1}^{n} ((p_j = i) * w_j) = l_i$$

On peut citer d'autres comme $atLeast(n, X, val)$, $atMost(n, X, val)$) ou $exactly(n, X, val)$) qui assurent qu'au moins, qu'au plus ou exactement `n` variables du vecteur `X` prennent la valeur `val`.

Le catalogue (Beldiceanu *et al.*, n.d.) liste plus de 350 contraintes globales. Le filtrage de ces contraintes fait appel à des techniques en théorie des graphes (i.e., couplage, flot), des techniques IA et de la programmation linéaire.

Ces dernières années, on retrouve de nouvelles représentations des contraintes globales par des automates ou des MDD (*Multivalued Decision Diagram*) qui sont des graphes acycliques. Ceci a permis de définir des contraintes globales, dites génériques, comme *regular* (Pesant, n.d.), *grammar* (Quimper & Walsh, n.d.), etc.

Prenons l'exemple de *regular* (Pesant, n.d.). Cette contrainte est définie sur une séquence finie de variables à domaines finis. Elle permet de vérifier si une séquence de valeur appartient à un langage régulier. Elle trouve son application dans des problèmes d'ordonnancement et de planification.

$$regular(\mathtt{x}, \mathcal{A}) = \{\tau \ : \ \tau \textit{ tuple de } \mathtt{x} \textit{ reconnu par } \mathcal{A}\}$$

Le vecteur x représente une séquence de variables, $\mathcal{A}$ un automate d'états finis (DFA).

3.3.5. Fonction objectif

Les problèmes d'optimisation définissent une fonction objectif f qui exprime le coût d'une solution donnée. Un programme à contraintes qui résout un problème d'optimisation doit trouver une solution qui satisfait l'ensemble des contraintes tout en minimisant (ou maximisant) le coût de cette solution. Le raffinement en PPC touche aussi cette partie du programme. Cette fonction peut être exprimée de différentes manières. Une bonne fonction objectif est une fonction qui mesure des caractéristiques du problème. De plus, une fonction f est plus performante si elle est exprimée avec des variables de décision qui sont présentes dans des contraintes importantes dans le programme. La résolution de tels problèmes fait appel à des cadres *Branch&Bound* (section 1.1.2) avec des ajouts de contraintes de type $f(x) < T$ (contrainte objectif), ce qui fait du choix de la fonction objectif un choix important.

Par exemple, dans le programme **(B)** de Golomb, la fonction objectif est exprimée avec les variables de distance $f(d) = d_{m-1}$, alors qu'on retrouve une autre fonction objectif sur les marques $f(x) = x_m - x_1$ dans le modèle **(A)**. La variable de distance d est présente dans la plupart des contraintes du programme. De plus, la contrainte *allDifferent* est appliquée sur d. Un filtrage sur une contrainte ajoutée $f(d) < T$ a un impact direct sur la contrainte $allDifferent(d)$, contrairement à un filtrage sur $f(x) < T$.

Ici, les deux fonctions $f(x)$ et $f(d)$ retournent un coût dans un même domaine de variation. Dans le cadre de notre travail et pour ne pas avoir à traiter deux modèles de deux problèmes d'optimisation différents, nous posons l'hypothèse de travail suivante :

Hypothèse 3 *Un raffinement sur une fonction objectif conserve son domaine de variation.*

3.4. Programme à Contraintes Sous Test (CPUT)

Appliquer les différents raffinements, détaillés précédemment, permet d'avoir un programme à contraintes, noté $\mathcal{P}_z(k)$, qui utilise la puissance de la PPC.

Selon l'enchainement des raffinements et la manière dont ils sont appliqués, un programme à contraintes $\mathcal{P}_z(k)$ peut être plus performant que $\mathcal{P}'_{z'}(k)$, sachant que tout deux résolvent un même problème et font appel à un même solveur. La performance ici est en fonction de la consommation temps/mémoire pour atteindre une solution. C'est l'expertise du développeur dans le choix des variables, des contraintes à ajouter et des reformulations possibles qui fait la différence.

Définition 2 (Raffinement unitaire) *Un raffinement unitaire, noté $\mathcal{R}$, représente toute reformulation, ajout et/ou suppression d'une contrainte.*

En partant du Modèle-Oracle $\mathcal{M}_x(k)$, un raffinement unitaire $\mathcal{R}$ apporte une modification sur une seule contrainte dans $\mathcal{M}_x(k)$. Ainsi, une succession finie de raffinements $\mathcal{R}^i$ nous permet d'avoir un programme $\mathcal{P}_z(k)$:

$$\mathcal{M}_x(k) \xrightarrow{\mathcal{R}} \mathcal{M}'_{x'}(k) \xrightarrow{\mathcal{R}} \ldots \xrightarrow{\mathcal{R}} \mathcal{P}'_{z'}(k) \xrightarrow{\mathcal{R}} \mathcal{P}_z(k)$$

où :

$$\mathcal{P}_z(k) \equiv \mathcal{R}^n(\mathcal{M}_x(k))$$

Le vecteur de variable z peut être différent de x suite aux ajouts de variables auxiliaires.

Un programme à contraintes sous test, noté CPUT, représente l'objet à tester (i.e., $\mathcal{P}_z(k)$). Ce programme est le résultat d'une suite de raffinements sur le Modèle-Oracle et vise à résoudre des instances difficiles du problème. Prenons le programme **(B)** de la figure 3.1, la question qui se pose est la suivante : est-ce que le programme **(B)** résout bien le problème des règles de Golomb ? Un mauvais raffinement dans **(B)** peut donner la contrainte `cc2'` au lieu de `cc2`. Si on lance le CPUT avec `cc2'` au lieu de `cc2` pour une instance de 6 marques, le solveur nous retourne la solution suivante :

$$x_i = [0\ 1\ 3\ 6\ 10\ 15]$$

La solution retournée n'est pas une règle de Golomb (i.e., $(6-3) = (3-0)$), ceci est due à une faute introduite par `cc2'`.

Un CPUT peut également être inconsistant et ne pas avoir de solutions ($sol(\mathcal{P}) = \emptyset$) à cause de raffinements qui réduisent l'ensemble des solutions à vide.

3.5. Hypothèses de base

Dans cette section, nous allons détailler les hypothèses de base sur lesquelles nos travaux de test et de mise-au-point des programmes à contraintes reposent, à savoir la correction du solveur sous-jacent, l'hypothèse du programmeur compétant et l'effet de couplage.

Hypothèse 4 *Le solveur de contraintes sous-jacent est correct.*

La plupart des implémentations des solveurs de contraintes sont minutieusement testés à l'aide d'outil de test des langages sous-jacent (e.g., C++ pour Gecode, Java pour Choco). De plus, on trouve également plusieurs outils de mise-au-point qui permettent une analyse ainsi qu'une visualisation de la trace (e.g., Ilog solver debbuger (ILOG, 2003)) avec l'arbre de recherche et les décisions prisent à chaque nœud, la réduction des domaines ... L'hypothèse 4 nous permet de nous concentrer sur les aspects de correction des modèles à contraintes sans se soucier des fautes liées au solveur. Ceci dit, la correction des solveurs reste un sujet important en PPC, mais qui sort du contexte des travaux présentés dans cette thèse.

Hypothèse 5 *Programmeur Compétent (DeMillo* et al.*, 1978; Acree* et al.*, 1979).*

L'hypothèse du programmeur compétent (i.e., *CPH : the Competent Programmer Hypothesis*) a été introduite pour la première fois par DeMillo et al. en 1978 (DeMillo *et al.*, 1978). Elle témoigne de la compétence du programmeur, ce qui implique que le programmeur aura tendance à produire un programme très proche de la version correcte. Ceci dit, bien que le programmeur soit compétent, ce dernier peut produire une version du programme contenant des fautes. Cette hypothèse nous permet de dire que la suite des raffinements apportés par le développeur produit un CPUT très proche de la version correcte, ainsi le Modèle-Oracle et le CPUT sont fortement liés en termes de problématique [3].

Hypothèse 6 *Effet de couplage (DeMillo* et al.*, 1978).*

L'hypothèse sur l'effet de couplage (i.e., *CEH : Coupling Effect Hypothesis*) a aussi été introduite par DeMillo et al. dans (DeMillo *et al.*, 1978). Contrairement à l'hypothèse 5 qui est liée au comportement du programmeur, l'hypothèse 6 est liée aux fautes utilisées en test de mutation. Cette hypothèse avance le fait qu'une donnée de test qui détecte une faute simple due à un changement syntaxique dans le programme, peut également détecter des fautes complexes. Dans notre cadre de travail, ceci nous permet de dire qu'une faute peut être complexe et due, non pas à un seul raffinement unitaire, mais à plusieurs à la fois. En se basant sur cette hypothèse, le travail de localisation s'intéresse aux fautes simples introduites par une seule contrainte ainsi qu'aux fautes multiples qui sont introduites par un sous-ensemble de contraintes.

3.6. Résumé

Dans ce chapitre, nous avons détaillé la modélisation en PPC avec les différents raffinements et techniques d'optimisation de l'ensemble des contraintes de sorte à avoir un programme à contraintes performant. Il est usuel de démarrer à partir d'un premier

3. Par exemple, on ne peut avoir un Modèle-Oracle qui résout les règles de Golomb et un CPUT qui résout les n-reines

modèle simple et très déclaratif, une traduction fidèle de la spécification du problème (Modèle-Oracle), sans pour autant accorder d'intérêt à ses performances. Par la suite, nous avons montré comment ce premier modèle peut être raffiné par l'introduction de contraintes impliquées ou reformulées, de contraintes globales, etc.

Définir des opérateurs automatiques pour les différents raffinements permet de faire de la preuve sur les transformations de modèles et sur la correction du programme à contraintes. En revanche, la plupart des raffinements nécessitent encore une intervention humaine : bien que la détection automatique de quelques symétries du problème est possible (Puget, 2005), la nature du problème et l'expertise humaine sont importantes pour détecter des symétries non-triviales. Aussi, les formalismes de génération de contraintes impliquées ne prennent pas en compte toute catégorie de ce type de contraintes. Notre démarche de test des programmes à contraintes permet de détecter des fautes introduites par tout raffinement possible.

4. Relations de Conformité (RC)

4.1. Introduction

Dans le chapitre précédent, nous avons défini les deux entités essentielles au test. Á savoir l'oracle ou la référence de test (Modèle-Oracle) et l'objet à tester (le CPUT). Dès lors, nous introduisons la notion de conformité entre ces deux entités selon la classe des problèmes abordés : les problèmes de satisfaction de contraintes (recherche d'une ou de toutes les solutions) et les problèmes d'optimisation (recherche d'une solution faisable dans un intervalle donné ou optimale).

À la fin de ce chapitre, nous donnerons un modèle de faute propre aux programmes PPC.

4.2. Notations

Dans cette section nous rappelons l'essentiel des notations qui sont utilisées dans les différentes définitions des relations de conformité.

Nous avons vu précédemment les notations $\mathcal{M}_x(k)$ et $\mathcal{P}_z(k)$ qui représentent respectivement le Modèle-Oracle et le CPUT. Les deux portent sur un même vecteur de paramètres $k \in \mathcal{K}$ ($\mathcal{K}$ étant l'ensemble des instances faisables du problème). Pour Golomb nous avons un seul paramètre qui est la longueur de la règles m (pour $m = 3$ nous avons une règle $\mathtt{x} = [\mathtt{0\ 1\ 3}]$ alors que pour $m = 4$ nous avons $\mathtt{x} = [\mathtt{0\ 1\ 4\ 6}]$). Comme vu précédement, le vecteur de variables z n'est pas forcement égal à x.

Nous rappelons ici que la notation $sol(Q)$ représente l'ensemble des solutions faisables du système de contraintes Q. Si Q modélise un problème d'optimisation avec une fonction objective à minimiser, on note alors $optim(Q)$ l'ensemble des minimums globaux.

Nous introduisons également la notation $Proj_X(Q)$ qui représente l'ensemble des projections des solutions de Q sur le vecteur de variables X.

Exemple 7 (Projection) *Soit un programme à contraintes :*

$$Q \equiv x, y, z \in 1..4 : allDifferent(x, y, z)$$

- $sol(Q) = \{(1,2,3),(1,2,4),(1,3,2),(1,3,4),(1,4,2),(1,4,3),(2,1,3),(2,3,1),(2,1,4),(2,3,4),(2,4,3),(2,4,1),(3,1,2),(3,2,1),(3,4,1),(3,1,4),(3,2,4),(3,4,2),(4,2,1),(4,3,1),(4,1,2),(4,3,2),(4,1,3),(4,2,3)\}$

- $Proj_{\{x,y\}}(sol(Q)) = \{(1,2),(1,3),(1,4),(2,1),(2,3),(2,4),(3,1),(3,2),(3,4),(4,2),(4,3),(4,1)\}$

Un programme à contraintes inclut implicitement une procédure générique *solve*() qui représente l'appel au solveur de contraintes ou à une procédure d'optimisation. La procédure d'optimisation a pour objective de réduire le coût d'une solution, ce coût est exprimé à l'aide d'une fonction objective f. Pour les problèmes d'optimisation, généralement on cherche des solutions faisables dans un intervalle de coût $[l,u]$. Par conséquent, nous introduisons l'ensemble suivant :

$$bounds_{f,l,u}(Q) = \{x | x \in sol(Q), f(x) \in [l,u]\}$$

Exemple 8 *Soit le programme à contrainte suivant :*

$$Q \equiv x,y,z \in 0..5 : \begin{cases} x+y<z \\ allDifferent(x,y,z) \\ minimize(x+y+z) \end{cases}$$

- $optim(Q) = \{(0,1,2),(1,0,2)\}$
- $bounds_{f,5,6}(Q) = \{(0,1,4),(1,0,4),(0,2,3),(2,0,3),(0,1,5),(1,0,5),(0,2,4),(2,0,4)\}$

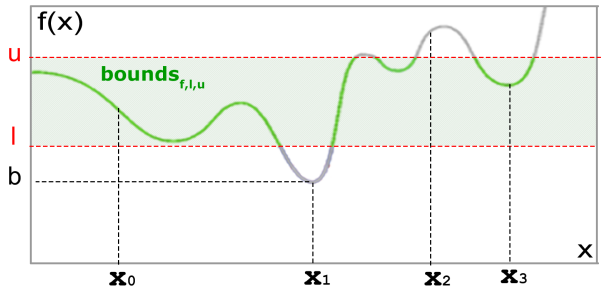


FIGURE 4.1.: L'ensemble $bounds_{f,l,u}$.

La figure 4.1 illustre l'ensemble $bounds_{f,l,u}$ sur une fonction objective f. Le point x_1 représente un minimum global avec un coût égal à b. L'ensemble $bounds_{f,l,u}$ inclut les points x_0 et x_3 sans compter x_1 et x_2.

La correction d'un CPUT par rapport à son Modèle-Oracle peut être évaluée à travers des relations de conformité. Ces relations sont exprimées sur différents niveaux : satisfaction de contraintes ou optimisation et/ou une ou toutes solutions possibles. Nous proposons donc quatre différentes relations de conformité.

4.3. RC : Problèmes de satisfaction de contraintes

4.3.1. $conf_{one}$

Les problèmes de satisfaction de contraintes cherchent en général une solution possible qui ne viole aucune contrainte du programme. Pour une instance k du problème, une conformité entre le Modèle-Oracle et le CPUT ($conf^k_{one}$) est établie si et seulement si le CPUT retourne des solutions (au moins une) validées par le Modèle-Oracle. En d'autres termes, la relation $conf^k_{one}$ assure que l'ensemble des solutions du CPUT n'est pas vide et qu'il est un sous-ensemble des solutions du Modèle-Oracle. Cette relation représente une *correction partielle* du CPUT :

Définition 3 ($conf_{one}$)

$$\mathcal{P}\ conf^k_{one}\ \mathcal{M} \triangleq \emptyset \subsetneq Proj_x(sol(\mathcal{P}_z(k))) \subseteq sol(\mathcal{M}_x(k))$$
$$\mathcal{P}\ conf_{one}\ \mathcal{M} \triangleq (\forall k \in \mathcal{K}, \mathcal{P}\ conf^k_{one}\ \mathcal{M})$$

La figure 4.2 montre des cas de conformité et de non-conformité entre un CPUT et son Modèle-Oracle. L'ensemble de solutions $sol(\mathcal{M}_x(k))$ (resp. $Proj_x(sol(\mathcal{P}_z(k)))$) est noté **M** (resp. **P**) dans la figure. Les **x** représentent des points de non-conformité entre le CPUT et le Modèle-Oracle (i.e., des fautes dans le CPUT), les **o** représentent des conformités. Les parties **(a)** et **(b)** de la figure 4.2 sont des cas de non-conformité où la relation $conf_{one}$ n'est pas vérifiée : le CPUT peut retourner des solutions qui ne sont pas des solutions du Modèle-Oracle. La partie **(c)** représente aussi un cas de non-conformité sans exhiber de solution non-conforme, en revanche la faute introduite dans le CPUT a réduit l'ensemble des solutions à vide. La dernière partie **(d)** montre un cas de conformité où toute solution du CPUT est conforme au Modèle-Oracle.

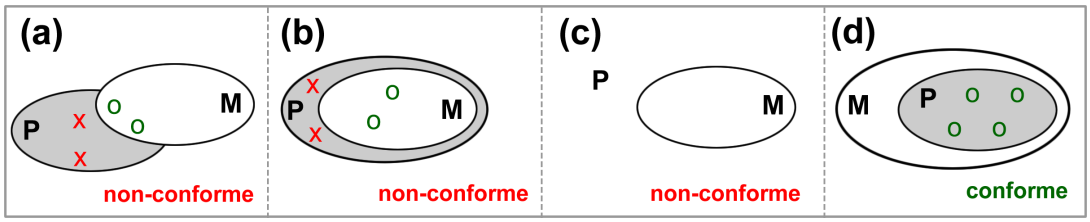


FIGURE 4.2.: Relation de conformité $conf_{one}$ sur $\mathcal{P}_z(k)$ et $\mathcal{M}_x(k)$.

4.3.2. $conf_{all}$

Il existe aussi des problèmes de satisfaction de contraintes qui cherchent à énumérer toutes les solutions d'un problème donné. Pour cette classe de problèmes, nous définissons la relation de conformité qui représente une *correction totale* du CPUT :

Définition 4 ($conf_{all}$)

$$\mathcal{P}\ conf^{k}_{all}\ \mathcal{M} \triangleq Proj_x(sol(\mathcal{P}_z(k))) = sol(\mathcal{M}_x(k))\ (\neq \emptyset)$$
$$\mathcal{P}\ conf_{all}\ \mathcal{M} \triangleq (\forall k \in \mathcal{K}, \mathcal{P}\ conf^{k}_{all}\ \mathcal{M})$$

Cette relation permet de conserver l'ensemble des solutions du Modèle-Oracle au CPUT. Une telle relation n'accepte pas de raffinement qui peut réduire le nombre de solutions comme l'ajout de contraintes des symétries.

Les parties **(a)**, **(b)** et **(c)** de la figure 4.3 montrent des cas de non-conformité, alors que le cas **(d)** assure une $conf_{all}$. Ceci dit, tout CPUT sous-contraint (le cas **(b)**) ou sur-contraint (le cas **(c)**) est non-conforme au Modèle-Oracle selon la relation $conf_{all}$.

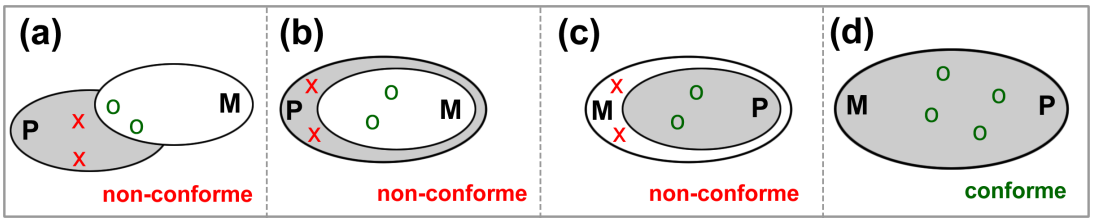


FIGURE 4.3.: Relation de conformité $conf_{all}$ sur $\mathcal{P}_z(k)$ et $\mathcal{M}_x(k)$.

4.4. RC : Problèmes d'optimisation

Dans un processus de résolution globale d'un problème d'optimisation, l'intervalle des valeurs de la fonction objective $[l, u]$ (i.e., l : borne inférieure, u : borne supérieure) est itérativement réduit jusqu'à atteindre un intervalle [l*, u*] de taille nulle ou inférieure à une précision donnée par l'utilisateur (pour une lecture détaillée, nous proposons (Sahinidis, 2002)). De ce fait, nous proposons la relation $conf_{bounds}$ qui, justement, assure la conformité des solutions dans un intervalle donné des valeurs de la fonction objective.

4.4.1. $conf_{bounds}$

Les solutions faisables sont des solutions localement optimales. En d'autres termes, ce sont des solutions optimales dans un intervalle donné et assez proches du minimum global qui est difficile à atteindre dans le cas général. Nous proposons une relation de conformité qui, dans un intervalle donné, assure la conformité du CPUT au Modèle-Oracle. Cette relation permet d'assurer que toute solution faisable du CPUT (au moins une) est une solution du Modèle-Oracle dans un intervalle $[l, u]$.

Définition 5 ($conf_{bounds}$)

$$\mathcal{P}\ conf^{k}_{bounds}\ \mathcal{M} \triangleq \emptyset \subsetneq Proj_x(bounds_{f',l,u}(\mathcal{P}_z(k))) \subseteq bounds_{f,l,u}(\mathcal{M}_x(k))$$
$$\mathcal{P}\ conf_{bounds}\ \mathcal{M} \triangleq (\forall k \in \mathcal{K}, \mathcal{P}\ conf^{k}_{bounds}\ \mathcal{M})$$

Les fonctions objectives f et f' du Modèle-Oracle et du CPUT peuvent être deux fonctions différentes (i.e., un raffinement sur f donne f', section 4.5). De plus, sous l'hypothèse 3 l'intervalle $[l, u]$ représente un intervalle de coût des solutions calculées par f et f'.

La relation $conf_{bounds}$ est intéressante dans le sens où on peut aborder des problèmes d'optimisation difficiles sans se soucier du minimum global. Choisir un intervalle qui n'inclut pas forcément le minimum global est une manière d'étudier la correction de problèmes d'optimisation globale en faisant abstraction de l'optimalité.

La figure 4.4 illustre des cas de conformité $conf_{bounds}$ entre un Modèle-Oracle et un CPUT. La notation B_M et B_P représentent respectivement l'ensemble $bounds_{f,l,u}(\mathcal{M})$ et $Proj_x(bounds_{f',l,u}(\mathcal{P}))$. D'autre part, B représente l'ensemble des minimums globaux $optim(\mathcal{M})$. Les parties **(a)** et **(b)** de la figure 4.4 illustrent deux cas de non-conformité où il existe des solutions faisables du CPUT qui ne sont pas des solutions du Modèle-Oracle dans l'intervalle $[l, u]$. Dans le cas **(c)**, L'ensemble B_P est réduit à vide, ce qui représente un cas de non-conformité. En revanche, la relation $conf_{bounds}$ est vérifiée dans le cas **(d)** où toute solution faisable du CPUT est une solution du Modèle-Oracle.

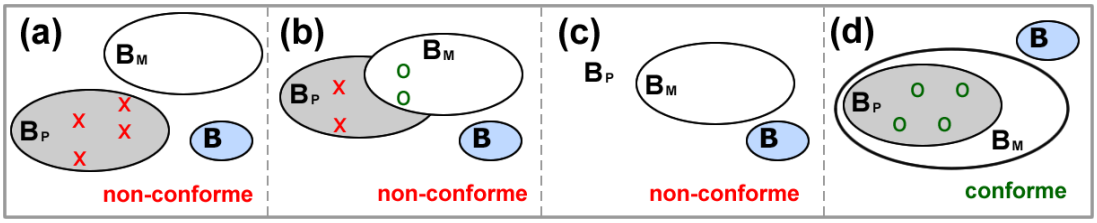


FIGURE 4.4.: Relation de conformité $conf_{bounds}$ sur $\mathcal{P}_z(k)$ et $\mathcal{M}_x(k)$.

4.4.2. $conf_{best}$

Pour certains problèmes d'optimisation, il est important de vérifier la correction d'un programme non seulement sur des intervalles donnés, mais aussi voir si ce dernier calcule bien des minimums globaux. Une conformité, notée $conf_{best}$, permet d'assurer une telle correction.

Une conformité $conf_{best}$ est alors équivalente à $conf_{one}$ vue précédemment où, tout minimum global du CPUT $\mathcal{P}$ est un minimum global du Modèle-Oracle $\mathcal{M}$ (i.e., $optim(\mathcal{P}) \subseteq optim(\mathcal{M})$).

Définition 6 ($conf_{best}$)

$$\mathcal{P}\ conf^k_{best}\ \mathcal{M} \triangleq \emptyset \subsetneq Proj_x(optim(\mathcal{P}_z(k))) \subseteq optim(\mathcal{M}_x(k))$$
$$\mathcal{P}\ conf_{best}\ \mathcal{M} \triangleq (\forall k \in \mathcal{K}, \mathcal{P}\ conf^k_{best}\ \mathcal{M})$$

4.5. Modèle de faute

Les raffinements unitaires $\mathcal{R}$ ont pour objective de donner au programme résultant un maximum de performance (Figure 4.5). En revanche, ils sont susceptibles d'introduire des fautes. Dans cette section, nous définissons le modèle de faute qui est une conséquence des raffinements unitaires.

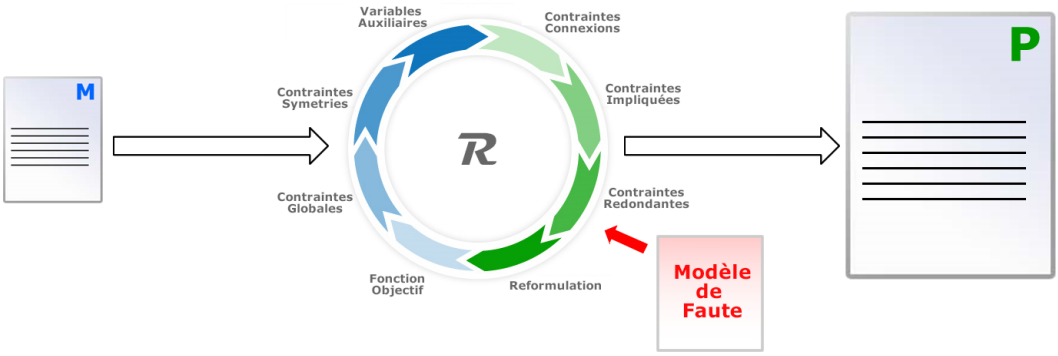


FIGURE 4.5.: Raffinements en PPC.

Une faute dans un programme à contraintes peut être de type syntaxique ou sémantique. En général, les fautes syntaxiques sont les plus faciles à détecter et à corriger à l'aide d'un vérificateur syntaxique dédié. Les fautes difficiles à détecter sont les fautes qui touchent la logique du programme. Le modèle de faute présenté ici concerne tout type de fautes qui peut apporter une modification sur l'ensemble des solutions du CPUT. Cette modification peut être un ajout comme une suppression de solutions possibles.

Définition 7 (Faute positive) *Une faute positive sur un CPUT $\mathcal{P}_z(k)$ par rapport à son Modèle-Oracle $\mathcal{M}_x(k)$, notée $\phi^+_{\mathcal{M}}(\mathcal{P})$, est une faute qui ajoute des solutions à $sol(\mathcal{P}_z(k))$:*

$$\phi^+_{\mathcal{M}}(\mathcal{P}) \triangleq Proj_x(sol(\mathcal{P}_z(k)))\backslash sol(\mathcal{M}_x(k)) \neq \emptyset$$

Exemple 9 *Soit* `c1`$:x < y$ *et* `c2`$:z < t$ *deux contraintes, une reformulation $\mathcal{R}$ des deux contraintes en* `c3`$:x + z < y + t$ *introduit une faute positive ϕ^+ car une intantiation comme $(x, y, z, t) = (0, 1, 4, 2)$ est une solution de* `c3` *qui ne satisfait pas* `c1` $\wedge$ `c2`.

En général, une faute positive sur un CPUT est une forme de relaxation qui permet à ce dernier d'accepter plus de solutions.

Définition 8 (Faute négative) *Une faute négative sur un CPUT $\mathcal{P}_z(k)$ par rapport à son Modèle-Oracle $\mathcal{M}_x(k)$, notée $\phi^-_{\mathcal{M}}(\mathcal{P})$, est une faute qui supprime des solutions de $sol(\mathcal{P}_z(k))$:*

$$\phi^-_{\mathcal{M}}(\mathcal{P}) \triangleq sol(\mathcal{M}_x(k))\backslash Proj_x(sol(\mathcal{P}_z(k))) \neq \emptyset$$

Exemple 10 *Soit* c1:$x \neq y$ *et* c2:$y \neq z$ *deux contraintes, une reformulation* $\mathcal{R}$ *en* c3:*allDifferent(x,y,z) introduit une faute négative* ϕ^- *où* $(x, y, z) = (1, 2, 1)$ *est une solution de* c1 $\wedge$ c2 *qui ne satisfait pas* c3. *Aussi, par définition et de façon générale, les contraintes de symétrie introduisent des fautes négatives.*

Définition 9 (Faute zéro) *Une faute zéro sur un CPUT* $\mathcal{P}_z(k)$ *par rapport à son Modèle-Oracle* $\mathcal{M}_x(k)$, *notée* $\phi^*_{\mathcal{M}}(\mathcal{P})$, *est une faute qui réduit l'ensemble des solutions* $sol(\mathcal{P}_z(k))$ *à vide :*

$$\phi^*_{\mathcal{M}}(\mathcal{P}) \triangleq (sol(\mathcal{P}_z(k)) = \emptyset)$$

Exemple 11 *Soit* $x, y, z \in 1..3$ *et* c1:$atLeast(1, [x, y, z], 1)$, c2:$atLeast(1, [x, y, z], 2)$ *deux contraintes, un raffinement* $\mathcal{R}$ *qui ajoute la contrainte* c3:$atMost(2, [x, y, z], 3)$ *introduit une faute zéro* ϕ^* *où* c1 $\wedge$ c2 $\wedge$ c3 *n'a pas de solution.*

5. Processus de test

5.1. Introduction

Dans ce chapitre, nous allons détailler un processus de test des programmes à contraintes. Ce processus est basé sur un générateur automatique de données de test, les relations de conformité et le modèle de faute.

5.2. Données de test

Tout processus de test exécute un programme sous une donnée de test puis, à l'aide d'un oracle, donne un verdict : *non-réussi* pour dire que le test a permis de révéler une faute, *réussi* dans le cas contraire. Pour un programme conventionnel, la donnée de test représente l'entrée/sortie du programme. En PPC, on ne retrouve pas cette notion d'entrée/sortie, mais on parle plus de paramétrage et d'ensemble de solutions.

Définition 10 (Données de Test) *Soit un CPUT $\mathcal{P}_z(k)$ et son Modèle-Oracle $\mathcal{M}_x(k)$, une donnée de test est une paire d'instantiation des paramètres et des variables (k, x).*

Il est à noter qu'à partir d'une donnée de test (k_0, x_0), il est possible de déduire (k_0, z_0) (voir section 3.3.1).

Définition 11 (Suite de Test) *Une suite de test, notée $\mathcal{T}$, est un ensemble fini de données de test :*

$$\mathcal{T} = \{(k_0, x_0), \ldots, (k_n, x_n)\}$$

Dès lors, la fiabilité du processus de test dépend fortement du processus de génération de données de test. La façon la plus simple et la plus courante est l'utilisation de générateur aléatoire (Duran & Ntafos, 1984). Cependant, cette méthode ne donne aucune garantie sur la validité de la suite générée.

5.3. Verdict et non-conformité

Étant donné (k_0, x_0), donner un verdict à cette donnée de test revient à consulter le tableau suivant :

Modèle-Oracle	CPUT	verdict ($conf_{one}$, $conf_{bounds}$, $conf_{best}$)	verdict ($conf_{all}$)
solution	solution	réussi	réussi
solution	non-solution	réussi	non-réussi
non-solution	solution	non-réussi	non-réussi
non-solution	non-solution	réussi	réussi

Prenons le deuxième cas de figure du tableau où la donnée de test représente une solution acceptable par le Modèle-Oracle et non-acceptable par le CPUT. Ce cas dépend de la conformité en question : si on cherche une conformité $conf_{all}$ entre le CPUT et le Modèle-Oracle alors le verdict de test est *non-réussi*, autrement le verdict est *réussi*.

Si le test échoue en retournant un verdict *non-réussi* à une donnée de test, cette donnée de test représente une non-conformité entre le CPUT et le Modèle-Oracle.

Définition 12 (non-conformité) *Une non-conformité entre un CPUT $\mathcal{P}$ et le Modèle-Oracle $\mathcal{M}$,due à une faute ϕ, est une donnée de test qui retourne le verdict "non-réussi". La non-conformité est notée comme suit :*

$$nc(\phi, k_i, x_i)$$

$conf_{one}$: Un CPUT $\mathcal{P}$ est non-conforme au Modèle-Oracle $\mathcal{M}$ si le CPUT comporte une faute de type ϕ^+ ou ϕ^* :

$$\phi^+_{\mathcal{M}}(\mathcal{P}) \vee \phi^*_{\mathcal{M}}(\mathcal{P}) \Rightarrow \mathcal{P} \neg conf_{one}\ \mathcal{M}$$

Une faute négative (ϕ^-) n'affecte pas la conformité $conf_{one}$ puisque l'objectif est d'atteindre au moins une solution acceptable par le Modèle-Oracle. Dès lors, tout raffinement qui peut supprimer des solutions (non pas toutes), comme la cassure de symétrie, est acceptable.

$conf_{all}$: Le CPUT est non-conforme au Modèle-Oracle s'il contient une faute de tout type :

$$\phi_{\mathcal{M}}(\mathcal{P}) \Rightarrow \mathcal{P} \neg conf_{all}\ \mathcal{M}$$

Toute modification de l'ensemble des solutions $sol(\mathcal{P})$ (ϕ^+ : ajout, ϕ^- : suppression, ϕ^* : réduction à vide) affecte la relation $conf_{all}$.

$conf_{bounds}$: Pour les problèmes d'optimisation, la conformité $conf_{bounds}$ est établie dans un intervalle $[l, u]$. Le CPUT est non-conforme au Modèle-Oracle si le CPUT contient une faute de type ϕ^+ ou ϕ^* dans l'intervalle $[l, u]$:

$$\phi^+_{\mathcal{M}}(\mathcal{P}) \vee \phi^*_{\mathcal{M}}(\mathcal{P}) \Rightarrow \mathcal{P} \neg conf_{bounds}\ \mathcal{M}$$

$conf_{best}$: On note une non-conformité entre le CPUT $\mathcal{P}$ et le Modèle-Oracle $\mathcal{M}$, dans le cadre d'une optimisation globale, si le CPUT comporte une faute de type ϕ^+ ou ϕ^* :

$$\phi^+_{\mathcal{M}}(\mathcal{P}) \vee \phi^*_{\mathcal{M}}(\mathcal{P}) \Rightarrow \mathcal{P} \neg conf_{best}\ \mathcal{M}$$

5.4. Test en PPC : Problèmes de satisfaction des contraintes

L'objectif du test des programmes à contraintes est d'atteindre un cas de non-conformité entre le CPUT et le Modèle-Oracle. Nous avons vu précédemment que le type de faute à détecter peut être différent selon la classe de problèmes abordée.

Pour les problèmes de satisfaction de contraintes, on peut être amené à chercher une solution au problème ou tout l'ensemble de solutions. Ici, le processus de test est différent pour les deux classes de problèmes.

5.4.1. Processus de test basé sur $conf_{one}$

Le processus de test basé sur $conf_{one}$ cherche à trouver une non-conformité qui révèle la présence de faute de type ϕ^+ ou ϕ^* dans le CPUT.

Algorithm 1: $algo_{one}$

Entrée: Modèle-Oracle $\mathcal{M}_x(k)$, CPUT $\mathcal{P}_z(k)$, ensemble d'instances K
Sortie : $nc(\phi, k_i, x_i)$, *conf-certificat*(K)

while $K \neq \emptyset$ **do**
  $k_i \leftarrow one\_of(K)$
  $K \leftarrow K \backslash \{k_i\}$
  if $sol(\mathcal{P}_z(k_i)) = \emptyset$ **then return** $nc(\phi^*, k_i, -)$
  $V \leftarrow one\_negated(\mathcal{P}_z(k_i), \mathcal{M}_x(k_i))$
  if $V \neq \emptyset$ **then** $x_i \leftarrow one\_of(V)$
  return $nc(\phi^+, k_i, x_i)$
end
return *conf-certificat*(K)

$one\_of(X)$ dénote une fonction qui retourne de façon non-déterministe un élément de l'ensemble X.

L'algorithme 1 décrit un processus de test qui prend en entrée le Modèle-Oracle, le CPUT et un sous-ensemble des instances du problème K (i.e., $K \subset \mathcal{K}$). Pour une instance $k_i \in K$, l'algorithme vérifie si l'ensemble des solutions du CPUT n'est pas réduit à vide dû à une faute ϕ^*. Autrement, un générateur de non-conformité est appelé (*one_ negated*). Un appel de ce processus *one_ negated(A,B)* retourne une solution de A qui n'est pas une solution de B. Une solution x_i du CPUT qui n'est pas une solution du Modèle-Oracle forme avec k_i une non-conformité qui révèle la présence d'une faute de type ϕ^+.

Dans le cas où $algo_{one}$ ne retourne aucune non-conformité pour un sous-ensemble d'instances K, l'algorithme délivre un certificat de conformité pour ce sous-ensemble K pour dire qu'aucune faute n'est présente dans ces instances. En revanche, ce certificat ne peut pas être valide pour une instance $k_j \in \mathcal{K} \backslash K$.

5.4.2. Processus de test basé sur $conf_{all}$

Pour le cas de $conf_{all}$, l'ensemble des solutions du CPUT doit être équivalent à l'ensemble du Modèle-Oracle. Le processus de test cherche alors des cas de non-conformité

où une solution est ajoutée et/ou supprimée.

Algorithm 2: $algo_{all}$

Entrée: Modèle-Oracle $\mathcal{M}_x(k)$, CPUT $\mathcal{P}_z(k)$, ensemble d'instances K
Sortie : $nc(\phi,k_i,x_i)$, *conf-certificat*(K)

while $K \neq \emptyset$ **do**
  $k_i \leftarrow one\_of(K)$
  $K \leftarrow K \backslash \{k_i\}$
  if $sol(\mathcal{P}_z(k_i)) = \emptyset$ **then return** $nc(\phi^*, k_i, -)$
  $V \leftarrow one\_negated(\mathcal{P}_z(k_i), \mathcal{M}_x(k_i))$
  if $V \neq \emptyset$ **then** $x_i \leftarrow one\_of(V)$
  return $nc(\phi^+, k_i, x_i)$
  $V \leftarrow one\_negated(\mathcal{M}_x(k_i), \mathcal{P}_z(k_i))$
  if $V \neq \emptyset$ **then** $x_i \leftarrow one\_of(V)$
  return $nc(\phi^-, k_i, x_i)$
end
return *conf-certificat*(K)

Les algorithmes 1 et 2 cherchent tous les deux à détecter des fautes de type ϕ^+ et ϕ^*. De plus, l'algorithme 2 cherche à trouver une solution du Modèle-Oracle qui n'est pas une solution du CPUT avec un appel à $one\_negated(\mathcal{M}, \mathcal{P})$. Cette non-conformité valide la présence de faute de type ϕ^- dans le CPUT.

5.5. Test en PPC : Problèmes d'optimisation

En optimisation, le test des programmes à contraintes a pour objectif la détection des fautes introduites dans l'ensemble des contraintes aussi bien que dans la fonction objectif. Le Modèle-Oracle et le CPUT d'un problème d'optimisation sont formés d'un ensemble de contraintes et d'une fonction objectif à minimiser :

$$\mathcal{M}_x(k) \equiv \mathcal{C}_x(k) \wedge minimize(f(x)) \qquad \mathcal{P}_z(k) \equiv \mathcal{C'}_z(k) \wedge minimize(f'(z))$$

5.5.1. Processus de test basé sur $conf_{bounds}$

Une solution faisable est une solution avec un coût minimal dans un intervalle $[l, u]$. Une faute de type ϕ^+ ou ϕ^* sur un CPUT provoque une non-conformité du CPUT avec le Modèle-Oracle dans un intervalle donné.

Algorithm 3: $algo_{bounds}$

Entrée: Modèle-Oracle $(\mathcal{M}_x(k) \equiv \mathcal{C}_x(k) \wedge minimize(f(x)))$,
CPUT $(\mathcal{P}_z(k) \equiv \mathcal{C}'_z(k) \wedge minimize(f'(z)))$,
ensemble d'instances K, bornes $[l, u]$

Sortie : $nc(\phi, k_i, x_i)$, *conf-certificat*(K)

while $K \neq \emptyset$ **do**
  $k_i \leftarrow one\_of(K)$
  $K \leftarrow K \backslash \{k_i\}$
  $A \leftarrow \mathcal{C}_x(k_i) \wedge (l \leq f(x) \leq u)$
  $B \leftarrow \mathcal{C}'_z(k_i) \wedge (l \leq f'(z) \leq u)$
  if $sol(B) = \emptyset$ **then return** $nc(\phi^*, k_i, -)$
  $V \leftarrow one\_negated(B, A)$
  if $V \neq \emptyset$ **then** $x_i \leftarrow one\_of(V)$
  return $nc(\phi^+, k_i, x_i)$
end
return *conf-certificat*(K)

L'algorithme 3 vérifie en premier lieu si l'ensemble des solutions du CPUT comprises dans un intervalle $[l, u]$ est vide. Si c'est le cas, il retourne la non-conformité $nc(\phi^*, k_i, -)$. Autrement, $algo_{bounds}$ tente de trouver une non-conformité qui représente une solution du CPUT et non pas du Modèle-Oracle dans l'intervalle $[l, u]$ (i.e., ϕ^+).

5.5.2. Processus de test basé sur $conf_{best}$

Dans le cas d'une minimisation globale (i.e., $conf_{best}$), le processus est équivalent à $algo_{one}$ qui cherche à détecter des fautes de type ϕ^+ et ϕ^*. Ici, le générateur $one\_negated(A, B)$ cherche à trouver un minimum global de A qui n'est pas un minimum global de B.

5.6. Générateur *one_negated*

Les processus de test présentés précédemment (algorithmes 1 à 3) se basent sur le générateur de non-conformité $one\_negated(A, B)$ qui retourne une solution de A qui n'est pas une solution de B.

Algorithm 4: one_negated

Input : A, $B \equiv C_1 \wedge \ldots \wedge C_n$
Output: un singleton x_i sinon $\emptyset$

```
1 foreach C_i ∈ B do
2 |   V ← vars(C_i)\vars(A)
3 |   if V = ∅ then set ← sol(A ∧ ¬C_i)
4 |   else set ← sol(A ∧ Proj_{vars(A)}(¬C_i))
5 |   if set ≠ ∅ then return singleton(one_of(set))
6 end
7 return ∅
```

L'algorithme 4 prend deux systèmes de contraintes notés A et B. L'objectif de cet algorithme est de trouver une solution au système :

$$A \wedge \neg B \equiv (A \wedge \neg C_1) \vee \ldots \vee (A \wedge \neg C_i) \vee \ldots \vee (A \wedge \neg C_n)$$

Ce système caractérise l'ensemble des solutions de A qui ne sont pas des solutions de B. l'algorithme 4 sélectionne une contrainte C_i à nier et appelle le solveur de contraintes pour résoudre le système $A \wedge \neg C_i$. Dès qu'une solution est trouvée, l'algorithme s'interrompt en renvoyant cette solution.

Une projection sur les contraintes est nécessaire dans le cas où l'ensemble des variables du système A est un sous-ensemble de B ($vars(A) \subset vars(B)$). Cette projection est nécessaire dans un seul cas, le deuxième appel de *one_negated* dans le processus de test $algo_{all}$. Dans ce cas particulier, on cherche une solution du Modèle-Oracle qui n'est pas une solution d'un CPUT incluant des variables auxiliaires (i.e., $one\_negated(\mathcal{M}, \mathcal{P})$ et $vars(\mathcal{M}) \subset vars(\mathcal{P})$).

5.6.1. Analyse de *one_negated*

Nous allons dans un premier temps détailler deux points essentiels dans l'algorithme *one_negated* : la projection de variables sur les contraintes pour répondre au problème des variables auxiliaires ; la négation de contrainte pour atteindre des solutions d'un système A qui ne sont pas des solutions de B.

Problème des variables auxiliaires

L'ajout de variables auxiliaires donne de nouvelles dimensions au CPUT par rapport au Modèle-Oracle comme le montre l'exemple suivant :

Exemple 12

Modèle-Oracle : $\mathcal{M}$	*CPUT* : $\mathcal{P}$
$c_1 : \ x - y \neq y - z$	$cc_1 : \ d_1 = x - y$
$c_2 : \ x - y \neq x - z$	$cc_2 : \ d_2 = y - z$
$c_3 : \ x - z \neq y - z$	$cc_3 : \ d_3 = x - z$
	$cc_4 : \ allDifferent(d_1, d_2, d_3)$

Dans cet exemple, le CPUT définit trois variables auxiliaires ($d_1,\ d_2,\ d_3$) à travers des contraintes de connexion. Ces deux systèmes sont sémantiquement équivalents.

Il est évident que le CPUT est conforme au Modèle-Oracle et préserve l'ensemble des solutions. En revanche, l'algorithme *one_ negated*, sans une conditionnelle sur les variables auxiliaires (ligne 5), peut retourner des fausses alarmes. Ces fausses alarmes peuvent être retournées dans un seul cas de figure : une conformité $conf_{all}$ revient à trouver une solution du CPUT qui n'est pas une solution du Modèle-Oracle et vice-versa. La deuxième étape du processus de test dans $algo_{all}$ cherche une solution du Modèle-Oracle qui n'est pas une solution du CPUT ($\mathcal{M} \wedge \neg\mathcal{P}$).

Dans ce cas, si on sélectionne une contrainte de connexion, par exemple la deuxième contrainte, le système résultant aura des solutions qui représentent des fausses alarmes :

$$\mathcal{M} \wedge (d_2 \neq y - z)$$

Pour remédier à ces fausses alarmes, l'algorithme *one_ negated* fait appel à un opérateur générique de projection qui peut être implémenté différemment.

Une façon de faire est l'utilisation de l'élimination de Fourier (Schrijver, 1986). Ce procédé permet d'éliminer des variables d'un système linéaire en créant un système S' équivalent à celui de départ S, en éliminant certaines variables.

Toutefois, utiliser une telle élimination peut être très couteux dans le cas où on a plusieurs variables où la méthode n'est plus polynomiale (Schrijver, 1986). De plus, une telle projection n'est possible que sur des systèmes linéaires, ce qui restreint le champ d'application de cette méthode.

La façon la plus simple pour remédier au problème des variables auxiliaires est de pouvoir distinguer entre les contraintes de connexion et le reste des contraintes. Dans l'exemple 12, seule la contrainte *allDifferent* est à nier. De plus, il faudra ajouter au système résultant les contraintes de connexion :

$$[\mathcal{M} \wedge (cc_1 \wedge cc_2 \wedge cc_3)] \wedge \neg allDifferent(d_1, d_2, d_3)$$

Négation des contraintes

L'algorithme *one_ negated* cherche une solution d'un système A qui n'est pas une solution d'un autre système B en se basant sur une démarche par réfutation avec de la négation de contraintes.

Les contraintes primitives sont des contraintes simples de type $\{\wedge, \vee, =, \neq, <, \geq\}$. Avec un solveur de contrainte assez riche, la négation de toute contrainte primitive (resp. combinaison de contraintes primitives) est une contrainte primitive (resp. combinaison de

contraintes primitives). Par exemple, la négation de la contrainte primitive $(x - y < z)$ nous donne la contrainte $(x - y \geq z)$. De plus, si on prend une conjonction de contraintes primitives $(c_1 \wedge \ldots \wedge c_n)$, la négation nous donne la disjonction suivante : $(\neg c_1 \vee \ldots \vee \neg c_n)$.

La difficulté d'une négation est celle appliquée sur des contraintes globales. Les contraintes globales sont des contraintes spéciales qui encapsulent la sémantique de plusieurs contraintes primitives avec un algorithme de filtrage dédié. Dès lors, une manière de procéder à la négation revient à appliquer l'opérateur de négation sur la spécification déclarative de cette contrainte.

Exemple 13

$$pack(l, p, w) \equiv \forall i \in 1..m \sum_{j \in 1}^{n} ((p_j = i) * w_j) = l_i$$

La négation nous donne :

$$\neg pack(l, p, w) \equiv \exists i \in 1..m \sum_{j \in 1}^{n} ((p_j = i) * w_j) \neq l_i$$

Cette négation est sémantiquement correcte, mais d'un niveau opérationnel peut couter cher. La puissance de la PPC est en grande partie dans les contraintes globales et le filtrage sous-jacent. La négation au niveau spécifications déclaratives des contraintes globales perd le filtrage dédié. De plus, la négation donne en général des disjonctions difficiles à gérer par un solveur.

Encore une fois, si le solveur de contraintes est suffisamment riche, il est possible d'exprimer la négation d'un ensemble de contraintes globales à l'aide de contraintes globales existantes.

Exemple 14

$$\neg atLeast(n, X, val) \equiv atMost(n - 1, X, val)$$

$$\neg atMost(n, X, val) \equiv atLeast(n + 1, X, val)$$

$$\neg exactly(n, X, val) \equiv exactly(m, X, val) \wedge m \neq n$$

Nous avons noté également sur la dernière version du catalogue des contraintes globales (Beldiceanu *et al.*, n.d.) plus de 20 contraintes dont la négation est une contrainte globale présente dans le catalogue (e.g., la négation de `allDifferent` est `some_equal` et vice versa).

Les contraintes globales représentent en grande partie la puissance de la PPC. Ces dernières années, on retrouve de nouvelles représentations des contraintes globales par des automates à états finis (i.e., DFA) (Carlsson & Beldiceanu, 2004; Beldiceanu *et al.*, 2005) ou encore des MDD (Multivalued Decision Diagram) (Andersen *et al.*, 2007) avec du filtrage générique. Á partir d'une représentation DFA, nous avons proposé une approche pour la négation des contraintes globales. En prenant une contrainte globale C, l'idée est

de définir des opérateurs de calcul d'automate qui reconnaissent seulement les solutions de $\neg C$. Pour le filtrage, nous utilisons la contrainte générique `regular` (Pesant, n.d.) qui prend l'automate de la version niée de la contrainte. Nous avons expérimenté cette approche sur deux contraintes globales (i.e., `global_contiguity` et `Lex`), les résultats sont comparés à ceux de la négation syntaxique. Il nous est apparu comme trop ambitieux de vouloir intégrer la présentation de ce travail dans la thèse qui porte sur le test et la mise-au-point des programmes à contraintes. En revanche, le compromis que nous avons trouvé consistera à inclure le rapport de recherche (Lazaar *et al.*, 2011b) en annexe.

5.6.2. Correction et complétude

L'algorithme *one_ negated(A,B)* est correct dans le sens où il retourne une solution de A qui n'est pas une solution de B si cette dernière existe. L'algorithme est également complet dans le sens où il ne retourne pas de fausses alarmes si $A \wedge \neg B$ n'a pas de solution.

S'il existe une solution au système A, notée s, qui n'est pas une solution de B, cela veut dire que cette solution viole au moins une contrainte C_i du système B. En d'autres termes, cette solution satisfait $\neg C_i$ et elle satisfait également le système $A \wedge \neg C_i$. En somme, la ligne 5 de l'algorithme *one_ negated* (ligne 6 dans le cas $vars(B) \subset vars(A)$) permet de retourner la solution s (correction).

Maintenant, s'il n'existe pas une solution s du système $A \wedge \neg B$, ceci implique que tout système $A \wedge \neg C_i$, avec $C_i \in B$, est un système insatisfiable. Dès lors, *one_ negated* retourne vide (complétude).

5.7. Résumé

Les deux chapitres précédents posent les jalons d'une théorie de test des programmes à contraintes pour construire édifice plus grand qui concerne la validation et la vérification des programmes déclaratifs. Un premier pas était la définition d'une référence de test (i.e., le Modèle-Oracle) ainsi que l'objet à tester (i.e., le CPUT). Par la suite, nous avons défini différentes relations de conformité établies entre le Modèle-Oracle et le CPUT. Dès lors, une faute dans un CPUT provoque une non-conformité avec son Modèle-Oracle. Nous avons vu également comment cette faute peut être de type différent selon son impact sur les solutions du CPUT (i.e., ϕ^+, ϕ^-, ϕ^*) , ce qui nous a conduit à définir un modèle de faute ϕ propre aux fautes qui peuvent être introduites dans un programme à contraintes.

Le cadre précédent nous a permis par la suite de proposer différents processus de test selon la conformité en question. Ces processus font appel à un même générateur de données de test (i.e., *one_ negated*). Les données de test retournées par ces processus de test représentent des non-conformités qui révèlent la présence de faute ϕ^+, ϕ^- ou encore ϕ^* dans le CPUT.

Une fois que le test détecte une faute dans le CPUT, il est important de localiser cette faute dans le CPUT pour rétablir la conformité entre le Modèle-Oracle et le CPUT. Nous avons consacré la partie suivante de ce manuscrit à cette mise-au-point des programmes à contraintes (i.e., localisation de fautes et correction automatique).

Troisième partie .

Localisation et Correction des fautes

6. Localisation des fautes dans les programmes à contraintes

6.1. Introduction

Les langages PPC actuels bénéficient d'un haut niveau de modélisation qui donne des aspects génie logiciel au développement dans ce type de programmation (section 1.2). D'autres parts, Nous avons vu en section 1.4 l'impact industriel de ce type de programmation dans différents secteurs (e.g., la gestion, la production, le transport, etc.). Ainsi, le besoin d'outils de maintenance et de mise au point de ce type de programmes est réel.

Nous avons présenté dans la partie précédente de ce manuscrit un cadre de test des programmes à contraintes qui permet de faire de la détection de fautes. Nous proposons maintenant une étude sur la localisation et la correction automatique des programmes à contraintes.

La localisation de faute est une tâche importante, coûteuse (manuelle de manière générale) et nécessaire pour la mise au point de tout programme. Les techniques automatiques de localisation de fautes sont plus ou moins précises. Nous avons vu en section 2.2 différents travaux qui renvoient une liste d'instructions suspectes d'un code conventionnel (e.g., Tarrantula (Jones & Harrold, 2005), DeLLiS (Cellier *et al.*, 2009)). Dans le cas des programmes à contraintes, une faute est due à une mauvaise formulation d'une ou de plusieurs contraintes. Dès lors, la localisation de faute dans ce type de programmes doit pouvoir retourner la ou les contraintes fautives.

6.2. Exemple illustratif

Dans cette section, nous allons prendre comme exemple illustratif de localisation le problème des n-reines. Nous rappelons que ce problème consiste à placer n reines sur un échiquier de taille $n \times n$ sans que les reines ne puissent se menacer mutuellement. La figure 6.1 montre un Modèle-Oracle (partie **(A)**) et un CPUT (partie **(B)**) écrits en OPL. Nous avons introduit une faute dans notre CPUT sous forme d'une mauvaise formulation de la contrainte cc4.

Phase de test

Tester notre CPUT et voir s'il est $conf_{one}$ au Modèle-Oracle revient à appeler $algo_{one}$ vu précédemment. Ce processus de test arrive à détecter la faute introduite en retournant la non-conformité suivante :

```
using CP;

int n=...;

dvar int queens[1..n] in 1..n;

subject to{

c1: forall(ordered i,j in 1..n)
     queens[i]!=queens[j];

c2: forall(ordered i,j in 1..n)
     queens[i]+i!=queens[j]+j;

c3: forall(ordered i,j in 1..n)
     queens[i]-i!=queens[j]-j;
}
```

(A)

```
using CP;

int n=...;
range Domain=1..n;

dvar int queens[1..n] in 1..n;
dvar int diag1[Domain];
dvar int diag2[Domain];

subject to{

cc1: forall( i in Domain)
      diag1[i]==queens[i]+i;

cc2: forall( i in Domain)
      diag2[i]==queens[i]-i;

cc3: sum(i in Domain)
       queens[i]==(n*(n+1) div 2);

cc4: forall(ordered i,j in Domain)
       diag1[i]<=diag1[j];

cc5: allDifferent(queens);

cc6: allDifferent(diag2);

}
```

(B) ϕ^+

FIGURE 6.1.: Un Modèle-Oracle (**A**) et un CPUT (**B**) avec une faute ϕ^+ du problème des n-reines.

$$nc(\phi^+, 4, [4\ 3\ 2\ 1])$$

Ici, la faute est de type ϕ^+ : Pour une instance du problème $n = 4$, l'instantiation [4 3 2 1] représente une solution du CPUT qui n'est pas une solution du Modèle-Oracle. Le rôle de la contrainte `cc4` est de ne pas avoir deux reines sur une même diagonale descendante. La mauvaise formulation sur `cc4` a introduit une faute, le test a détecté cette faute avec une solution qui positionne toutes les reines sur la même diagonale descendante.

Phase de localisation

La phase de localisation doit pouvoir retourner un sous-ensemble de contraintes suspectes incluant `cc4`. Le pire des cas serait que la localisation retourne tout l'ensemble des contraintes du CPUT comme suspectes. La localisation peut également retourner une seule contrainte suspecte. Dans ce dernier cas, la localisation est précise au point où la contrainte suspecte retournée représente la contrainte fautive. Nous allons détailler maintenant ce processus de localisation qui, pour cet exemple des n-reines, arrive à retourner uniquement `cc4` comme contrainte suspecte.

6.3. Notations et définitions

Les fautes ϕ introduites dans un CPUT sont la conséquence de mauvais raffinements. Ces fautes peuvent augmenter l'ensemble des solutions en acceptant de nouvelles (i.e., ϕ^+), comme elles peuvent le réduire en supprimant quelques solutions (i.e., ϕ^-) ou la totalité des solutions (i.e., ϕ^*). Une localisation de ce type de faute revient à trouver une origine possible à la faute.

Soit un Modèle-Oracle $\mathcal{M}$, un CPUT $\mathcal{P}$ et une faute ϕ dans $\mathcal{P}$. On appelle une localisation de ϕ, un sous-ensemble de contraintes de $\mathcal{P}$ noté Loc_ϕ. Si la faute est introduite par un seul raffinement unitaire, alors Loc_ϕ est réduit à une seule contrainte :

$$\mathcal{M} \xrightarrow[C_i]{\mathcal{R}_i} \mathcal{M}' \xrightarrow[C_j]{\mathcal{R}_j} \dots \xrightarrow[C_k]{\mathcal{R}_k} \mathcal{P}' \dots \xrightarrow[C_l]{\mathcal{R}_l} \mathcal{P}$$

Ici, le raffinement $\mathcal{R}_i$ permet d'avoir la contrainte C_i dans le nouveau modèle et ainsi de suite. Si la faute ϕ est introduite par le raffinement $\mathcal{R}_k$ alors la localisation $Loc_\phi = C_k$. De plus, si la faute ϕ est introduite par $\mathcal{R}_j$, $\mathcal{R}_k$ et $\mathcal{R}_l$, alors $Loc_\phi = \{C_j, C_k, C_l\}$.

Définition 13 (Localisation minimale) *Soit Loc_ϕ une localisation d'une faute ϕ dans un CPUT $\mathcal{P}$. Loc_ϕ est dite localisation minimale si et seulement s'il n'existe pas une localisation Loc'_ϕ telle que $Loc'_\phi \subsetneq Loc_\phi$.*

Exemple 15 (Localisation minimale) *Prenons notre exemple sur les n-reines de la section 6.2. Une localisation minimale serait* `{cc4}` *alors qu'une localisation* `{cc3, cc4}` *n'est pas une localisation minimale.*

Dans la section 1.3.3, nous avons abordé des travaux qui cherchent des localisations minimales pour des fautes de type ϕ^* (systèmes insatisfiables). Dans la suite de ce chapitre, nous allons nous intéresser aux localisations minimales pour tout type de faute ϕ.

Une faute ϕ dans un CPUT peut avoir plusieurs localisations possibles où chaque localisation Loc_ϕ représente un sous-ensemble de contraintes dit *suspect*.

Définition 14 (*SuspiciousSet) Soit ϕ une faute dans un CPUT $\mathcal{P}$. Un SuspiciousSet représente l'ensemble des localisations minimales possibles Loc_ϕ :*

$$SuspiciousSet = \{Loc_1, \ldots, Loc_n\}$$

Dès lors, l'objectif d'une localisation de faute est de pouvoir calculer le *SuspiciousSet* d'une faute ϕ.

6.4. Localisation de faute simple

Une faute simple représente la conséquence d'un mauvais raffinement unitaire $\mathcal{R}$ qui touche une seule contrainte du CPUT (définition 2). Dans cette section, nous allons présenter un processus de localisation de faute sous l'hypothèse d'avoir une seule contrainte fautive dans le CPUT.

Hypothèse 7 *Une faute ϕ dans un CPUT est introduite suite à un raffinement unitaire $\mathcal{R}$ donné.*

6.4.1. Intuition

L'intuition derrière une localisation de faute de type ϕ^+ et ϕ^* est la suivante :
- une contrainte contenant une faute ϕ^* réduit l'ensemble des solutions du CPUT $\mathcal{P}$ à vide. Enlever cette contrainte de $\mathcal{P}$ permet d'avoir des solutions partagées entre le Modèle-Oracle $\mathcal{M}$ et le CPUT $\mathcal{P}$, ainsi rétablir un degré de conformité.
- une contrainte contenant une faute ϕ^+ augmente l'ensemble des solutions du CPUT $\mathcal{P}$ avec des solutions non-acceptables par le Modèle-Oracle $\mathcal{M}$. Enlever cette contrainte de $\mathcal{P}$ peut également augmenter l'ensemble solutions acceptables par le Modèle-Oracle et le CPUT.

L'intuition derrière une localisation de ϕ^- est la suivante :
- une contrainte $C_i \in \mathcal{P}$ contenant une faute ϕ^- réduit l'ensemble des solutions de $\mathcal{P}$. Dans une phase de test, cette contrainte permet de retourner une non-conformité x et par conséquent détecter cette faute (i.e., $x \in sol(\mathcal{M} \wedge \neg C_i)$).

Dès lors, nous arrivons à la définition de *contrainte suspecte*.

Définition 15 (Contrainte suspecte) *Soit une faute ϕ sur un CPUT ($\mathcal{P} \equiv C_1 \wedge \ldots \wedge C_n$) sous l'hypothèse 7 et un Modèle-Oracle $\mathcal{M}$. Une contrainte $C_i \in \mathcal{P}$ est suspecte si elle représente une localisation potentielle (i.e., $Loc_\phi = \{C_i\}$) de la faute ϕ. On note :*

$$C_i \in \mathcal{P} \text{ est suspecte } \triangleq \begin{cases} \phi^+, \phi^* : sol(\mathcal{M}) \cap sol(\mathcal{P} \backslash C_i) \neq \emptyset \\ \phi^- \quad\quad : sol(\mathcal{M} \wedge \neg C_i) \neq \emptyset \end{cases}$$

Nous rappelons ici que la phase de test qui précède la phase de localisation arrive à détecter la nature de la faute à travers la non-conformité retournée (e.g., $nc(\phi^+, k_i, x_i)$).

6.4.2. Procéssus de localisation sous l'hypothèse 7

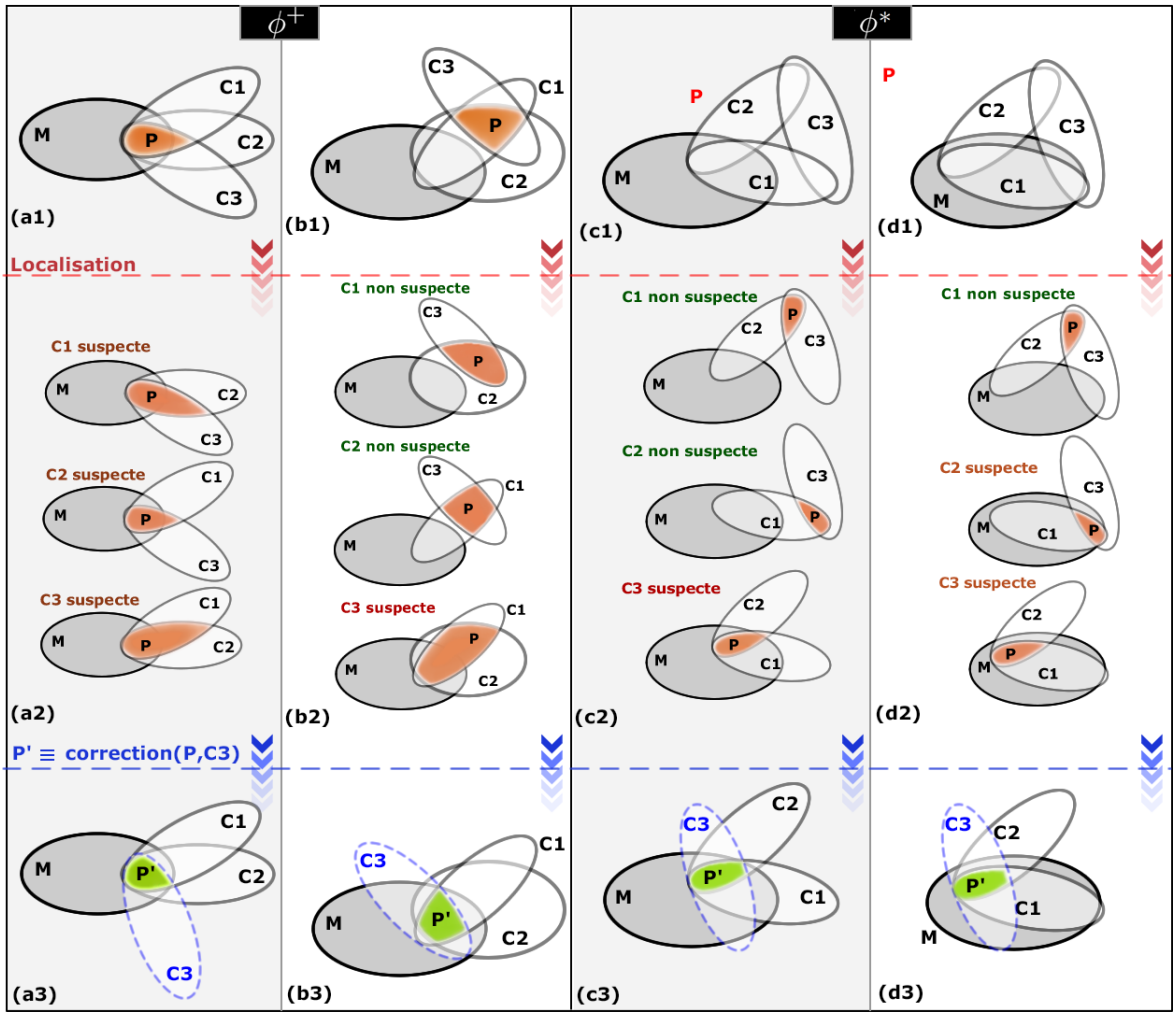


FIGURE 6.2.: Localisation des fautes ϕ^+ et ϕ^* sous l'hypothèse 7.

La figure 6.2 présente quatre cas de non-conformité entre un CPUT $\mathcal{P}$ et le Modèle-Oracle $\mathcal{M}$. Dans cet exemple, le CPUT est une conjonction de trois contraintes ($\mathcal{P} \equiv$

$C_1 \wedge C_2 \wedge C_3$). La figure 6.2 donne une représentation graphique des ensembles de solutions : $sol(\mathcal{M})$, $sol(\mathcal{P})$, $sol(C_1)$, $sol(C_2)$ et $sol(C_3)$. Dans chacun des quatre cas, nous introduisons une faute de type ϕ^+ ou ϕ^* due à un mauvais raffinement sur C_3.

Fig.6.2 (a) La faute introduite dans ce cas est une faute ϕ^+. On retrouve alors des solutions du CPUT $\mathcal{P}$ qui ne sont pas acceptables par le Modèle-Oracle $\mathcal{M}$ **(a1)**. Les trois contraintes, dont la contrainte fautive C_3, sont des contraintes suspectes (définition 15, **(a2)**).

$$\begin{cases} sol(\mathcal{M}) \cap sol(\mathcal{P} \backslash C_1) \neq \emptyset \\ sol(\mathcal{M}) \cap sol(\mathcal{P} \backslash C_2) \neq \emptyset \\ sol(\mathcal{M}) \cap sol(\mathcal{P} \backslash C_3) \neq \emptyset \end{cases}$$

Une correction possible suite à une révision d'une des contraintes suspectes, par exemple C_3, peut corriger la faute ϕ^+ (**a3**). Il est à noter qu'une révision sur C_1 ou C_2 peut aussi rétablir la conformité entre le CPUT et le Modèle-Oracle ; ce qui fait de C_1 et C_2 des localisations potentielles de la faute ϕ^+. Dans le cas où toute contrainte de $\mathcal{P}$ est suspecte, la localisation indique que $\mathcal{P}$ *est sous-contraint.*

Fig.6.2 (b) Ce deuxième cas représente aussi une faute de type ϕ^+. Ici, $\mathcal{M}$ et $\mathcal{P}$ ne partagent aucune solution **(b1)**. D'après la définition 15, C_3 est la seule contrainte suspecte et représente l'unique localisation de la faute ϕ^+ **(b2)** :

$$sol(\mathcal{M}) \cap sol(\mathcal{P} \backslash C_3) \neq \emptyset$$

Une éventuelle révision de la contrainte C_3 permet de corriger ϕ^+ **(b3)**.

Fig.6.2 (c) L'ensemble des solutions de $\mathcal{P}$ est réduit à vide, ici la faute est de type ϕ^* **(c1)**. Pour ce cas, une seule contrainte est suspecte : C_3 (définition 15, **(c2)**).

$$sol(\mathcal{M}) \cap sol(\mathcal{P} \backslash C_3) \neq \emptyset$$

Une éventuelle révision de la contrainte C_3 permet de corriger la faute **(c3)**.

Fig.6.2 (d) Dans ce dernier cas, nous avons introduit une autre faute de type ϕ^* dans C_3 où le CPUT devient insatisfiable **(d1)**. En revanche, les contraintes C_2 et C_3 sont suspectes d'après la définition 15 **(d2)** :

$$\begin{cases} sol(\mathcal{M}) \cap sol(\mathcal{P} \backslash C_2) \neq \emptyset \\ sol(\mathcal{M}) \cap sol(\mathcal{P} \backslash C_3) \neq \emptyset \end{cases}$$

Une révision de C_3 (comme sur C_2) permet donc de rétablir la conformité entre le Modèle-Oracle et le CPUT **(d3)**.

Dans les quatre cas précédents, la contrainte C_3 qui est à l'origine des fautes, est toujours considérée comme une contrainte suspecte (définition 15).

Les contraintes suspectes forment l'ensemble *SuspiciousSet* d'une faute ϕ. Par exemple pour le dernier cas de la figure 6.2, nous avons :

$$SuspiciousSet = \{\{C_2\}, \{C_3\}\}$$

Nous allons à présent donner l'algorithme de localisation sous l'hypothèse 7 basée sur la définition 15 pour le calcul du *SuspiciousSet*.

6.4.3. Algorithme *locate*

L'algorithme *locate* a pour objectif le calcul de l'ensemble des contraintes suspectes (*SuspiciousSet*) sous l'hypothèse 7, où chaque contrainte représente une localisation potentielle de ϕ.

Algorithm 5: *locate*

Input : Modèle-Oracle :$\mathcal{M}$, CPUT :$\mathcal{P}$, $nc(\phi, k_i, x_i)$
Output: *SuspiciousSet* : contraintes suspectes

```
1  SuspiciousSet ← ∅
2  if (φ+ ∨ φ*) then
3  |  foreach C_i ∈ P do
4  |  |  if sol(M ∧ P\C_i) ≠ ∅ then
5  |  |  |  SuspiciousSet ← SuspiciousSet ∪ {C_i}
6  |  |  end
7  |  end
8  end
9  else
10 |  foreach C_i ∈ P do
11 |  |  if sol(M ∧ ¬C_i) ≠ ∅ then
12 |  |  |  SuspiciousSet ← {C_i}
13 |  |  |  break
14 |  |  end
15 |  end
16 end
17 if check(M, P, SuspisiousSet) then SuspisiousSet ← ∅
18 return SuspiciousSet
```

L'algorithme *locate* prend le Modèle-Oracle, le CPUT et une non-conformité retournée par le test. Selon la faute détectée, la localisation procède au calcul de l'ensemble des contraintes suspectes (*SuspiciousSet*). Si la faute est de type ϕ^+ ou ϕ^*, on vérifie si les contraintes du CPUT (une à une) sont suspectes selon la définition 15. Si la faute est de type ϕ^-, on procède à la négation de contraintes (définition 15) pour localiser l'unique contrainte qui explique la faute ϕ^-.

Dans le cas où le CPUT est sous-contraint (i.e., $sol(\mathcal{M} \wedge \mathcal{P}) \neq \emptyset$ et $sol(\mathcal{P} \wedge \neg\mathcal{M}) \neq \emptyset$) , l'algorithme *locate* reconnait chaque contrainte du CPUT comme suspecte et retourne

vide (ligne 17) pour dire que le CPUT a besoin d'ajout de contraintes. En revanche, il existe un cas pathologique où la localisation est inefficace comme celui montré en figure 6.3. Ici, le CPUT n'est pas sous-contraint et les trois contraintes C_1, C_2 et C_3 sont suspectes. Si toute contrainte du CPUT est retournée comme suspecte, la procédure *check* de la ligne 17 vérifie si le CPUT partage des solutions avec le Modèle-Oracle (i.e., $sol(\mathcal{M} \wedge \mathcal{P}) \neq \emptyset$) et dans ce cas le *SuspiciousSet* est remis à vide. Autrement, on garde l'état du *SuspiciousSet* qui dit que toute contrainte est suspecte.

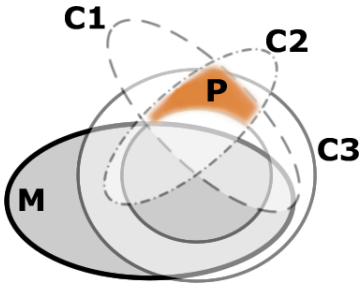


FIGURE 6.3.: Cas pathologique.

Correction de *locate*

La pré-condition de l'algorithme *locate* est l'existence d'une seule contrainte fautive dans le CPUT (i.e., hypothèse 7). L'algorithme *locate* doit alors retourner un sous ensemble de contraintes suspectes où l'une d'elles représente la contrainte fautive (i.e., post-condition). Pour traiter la question de correction de l'algorithme *locate*, il suffit donc prouver le théorème suivant.

Théorème 1 *Soit $\mathcal{M}$, $\mathcal{P}$ et ϕ un Modèle-Oracle, un CPUT et une faute simple :*

$$\mathcal{M}\ \neg conf\ \mathcal{P} \rightarrow \begin{cases} (\phi^+ \vee \phi^*) \wedge \left(\exists C_i \in \mathcal{P} : sol(\mathcal{M} \wedge \mathcal{P} \backslash C_i)) \neq \emptyset\right) \\ \vee \\ \phi^- \wedge \left(\exists C_i \in \mathcal{P} : sol(\mathcal{M} \wedge \neg C_i) \neq \emptyset\right) \end{cases}$$

Ici, la conformité *conf* peut être de type *one*, *all*, *bounds* ou *best* et ne peut être que de type *all* dans le cas d'une faute ϕ^-.

Prenons le cas d'une faute simple de type ϕ^+ ou ϕ^* introduite dans $C_i \in \mathcal{P}$. Soit $\mathcal{P}'$ une version correcte du CPUT avec une contrainte C'_i à la place de C_i, $\mathcal{P}'$ est conforme à $\mathcal{M}$ où toute solution de $\mathcal{P}'$ est une solution de $\mathcal{M}$:

$$sol(\mathcal{M} \wedge \mathcal{P}') \neq \emptyset \Rightarrow sol(\mathcal{M} \wedge \mathcal{P}' \backslash C'_i) \neq \emptyset$$

Sachant que $(\mathcal{P}' \backslash C'_i = \mathcal{P} \backslash C_i)$, ceci nous amène à :

$$sol(\mathcal{M} \wedge \mathcal{P} \backslash C_i) \neq \emptyset$$

Ainsi, la conditionnelle à la ligne 4 de l'algorithme *locate* est vérifiée et $C_i \in SuspiciousSet$.

Prenons maintenant une faute de type ϕ^-, introduite dans $C_i \in \mathcal{P}$, où il existe des solutions de $\mathcal{M}$ qui ne sont pas des solutions de $\mathcal{P}$:

$$sol(\mathcal{M} \wedge \neg\mathcal{P}) \neq \emptyset$$

Sachant que la faute est introduite dans C_i :

$$sol(\mathcal{M} \wedge \neg C_i) \neq \emptyset$$

Ceci assure la conditionnelle à la ligne 11 de l'algorithme *locate* et nous permet d'avoir $C_i \in SuspiciousSet$.

$\Diamond$

Nous rappelons que toute localisation minimale d'une faute ϕ sous l'hypothèse 7 est formé d'une seule contrainte du CPUT. Ainsi, notre algorithme *locate* prend les contraintes du CPUT une à une et ne retourne que des localisations minimales dans $SuspiciousSet$.

6.5. Localisation de faute multiple

Dans cette section, nous allons enlever l'hypothèse 7 où une faute multiple ϕ dans un CPUT peut être due à plusieurs raffinements unitaires $\mathcal{R}$. Dès lors, une localisation d'une faute ϕ représente un sous-ensemble du CPUT. Une révision possible sur ce sous-ensemble de contraintes permet de corriger la faute ϕ et de rétablir la conformité du CPUT au Modèle-Oracle.

6.5.1. Intuition

Une faute ϕ peut être introduite par une, deux ou plusieurs contraintes du CPUT. Dès lors, on ne parle plus de contrainte suspecte, mais d'ensemble suspect.

Définition 16 (Ensemble suspect) *Soit une faute ϕ sur un CPUT ($\mathcal{P} \equiv C_1 \wedge \ldots \wedge C_n$) et un Modèle-Oracle $\mathcal{M}$. Un sous-ensemble de contraintes $\{C_1 \ldots C_l\} \subset \mathcal{P}$, à un renomage près, est suspect s'il représente une localisation potentielle (i.e., Loc_ϕ) de la faute ϕ. On note :*

$$\{C_1, \ldots, C_l\} \subset \mathcal{P} \text{ est suspect } \triangleq \begin{cases} \phi^+, \phi^* : sol(\mathcal{M}) \cap sol(\mathcal{P} \backslash \{C_1, \ldots, C_l\}) \neq \emptyset \\ \phi^- : \forall C_k \in \{C_1 \ldots C_l\},\ sol(\mathcal{M} \wedge \neg C_k) \neq \emptyset \end{cases}$$

6.5.2. Processus de localisation sans l'hypothèse 7

Prenons la figure 6.4, nous avons un exemple de CPUT ($\mathcal{P} \equiv C_1 \wedge C_2 \wedge C_3$) qui contient une faute de type ϕ^+ détecter par le test. Cette faute provoque une non-conformité avec le Modèle-Oracle et cela, quel que soit le type de conformité en question. La faute ϕ^+ dans cet exemple est introduite en C_1 et C_3 (les deux en même temps) suite à de mauvais raffinements sur ces contraintes. L'intuition dans le calcul des localisations potentielles *SuspiciousSet* passe par des niveaux. Le premier niveau revient à prendre des sous-ensembles de cardinalité 1 et vérifier s'ils sont suspects en respectant la définition 16. Pour cet exemple, aucun sous-ensemble de taille 1 n'est suspect :

$$\forall C_i \in \mathcal{P} : sol(\mathcal{M} \wedge \mathcal{P} \backslash C_i) = \emptyset$$

Dès lors, on passe au niveau suivant en prenant des sous-ensembles de cardinalité 2 et ainsi de suite. Pour notre exemple, le sous-ensemble $\{C_1, C_3\}$ est considéré comme suspect selon la définition 16 :

$$sol(\mathcal{M} \wedge \mathcal{P} \backslash \{C_1, C_3\}) \neq \emptyset$$

Une révision possible des deux contraintes permet de corriger la faute ϕ^+. Nous allons maintenant donner l'algorithme de localisation de faute ϕ sans l'hypothèse 7.

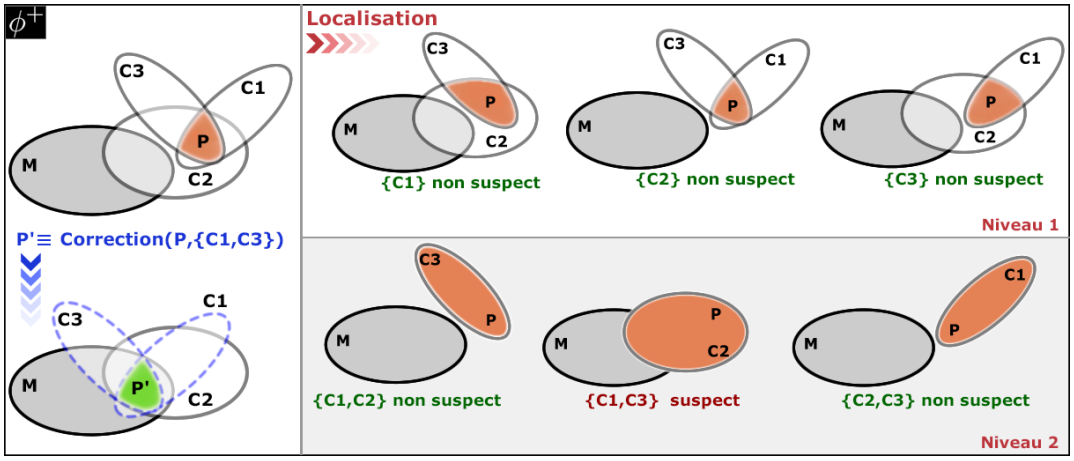


FIGURE 6.4.: Localisation de faute ϕ^+ sans l'hypothèse 7.

6.5.3. Algorithme ϕ-*locate*

Algorithm 6: ϕ-*locate*

Input : Modèle-Oracle :$\mathcal{M}$, CPUT :$\mathcal{P}$, $nc(\phi, k_i, x_i)$, λ : nombre de niveaux
Output: *SuspiciousSet* : contraintes suspectes

```
1  SuspiciousSet ← ∅
2  if (φ+ ∨ φ*) then
3  |  LOC ← powset(P, λ)
4  |  for i ∈ 1..λ do
5  |  |  foreach Loc_j ∈ LOC : |Loc_j| = i do
6  |  |  |  if sol(M ∧ P\Loc_j) ≠ ∅ then
7  |  |  |  |  SuspiciousSet ← SuspiciousSet ∪ Loc_j
8  |  |  |  |  foreach Loc_l ∈ LOC : Loc_j ⊂ Loc_l do LOC ← LOC\Loc_l
9  |  |  |  end
10 |  |  end
11 |  end
12 end
13 else
14 |  Set_φ- ← ∅
15 |  foreach C_i ∈ P do
16 |  |  if sol(M ∧ ¬C_i) ≠ ∅ then
17 |  |  |  Set_φ- ← Set_φ- ∪ C_i
18 |  |  end
19 |  end
20 |  SuspiciousSet ← {Set_φ-}
21 end
22 if check(M, P, SuspisiousSet) then SuspisiousSet ← ∅
23 return SuspiciousSet
```

L'algorithme ϕ-*locate* est une généralisation de *locate* avec un paramètre de plus (i.e., le paramètre λ). Ce paramètre représente la taille maximale que peut avoir une localisation de faute ϕ et il est strictement inférieur à la taille du CPUT (i.e., $\lambda < n$), ainsi il défini le nombre de niveaux que l'algorithme va devoir parcourir.
L'opérateur $powset(\mathcal{P}, \lambda)$ retourne un sous-ensemble du *powerset* de $\mathcal{P}$ (i.e., ensemble des parties de $\mathcal{P}$). Les éléments de $powset(\mathcal{P}, \lambda)$ sont tous des sous-ensembles de $\mathcal{P}$, de taille comprise entre 1 et λ.

Exemple 16 *Soit* $Q \equiv C_1, C_2, C_3, C_4$
$powset(Q, 2) = \{\{C_1\}, \{C_2\}, \{C_3\}, \{C_4\}, \{C_1, C_2\}, \{C_1, C_3\}, \{C_1, C_4\}, \{C_2, C_3\}, \{C_2, C_4\}, \{C_3, C_4\}\}$

Dans le cas d'une faute ϕ^+ ou ϕ^*, on vérifie les sous-ensembles possibles du CPUT de taille 1 à λ (définition 16).

Le cas de faute ϕ^- est plus simple à calculer où le *SuspiciousSet* est réduit à une seule localisation Loc_ϕ. Cette localisation contient toute contrainte qui n'accepte pas de solutions de $\mathcal{M}$.

Le cas pathologique de la figure 6.3 peut également se produire lorsque la faute est multiple. La procédure *check* (i.e., ligne 22) assure le traitement de ce cas de la même manière que dans l'algorithme *locate*.

Complexité de ϕ-*locate*

La taille de l'ensemble LOC calculé par $powset(\mathcal{P}, \lambda)$ à la ligne 4 est de :

$$\Sigma_{i=1}^{\lambda}(C_n^i)$$

où n représente le nombre de contraintes de $\mathcal{P}$. Ainsi, la complexité de l'algorithme ϕ-*locate* est de nature exponentielle. En pratique, nous avons observé qu'une faute dans un CPUT n'est en général due qu'à un nombre restreint de contraintes où $\lambda \ll n$. Ainsi, en choisissant de petites valeurs pour λ, on peut maîtriser l'explosion combinatoire dû à ce calcul.

Correction de ϕ-*locate*

La pré-condition de l'algorithme ϕ-*locate* est l'existence d'un sous-ensemble de contraintes fautives de taille inconnue dans le CPUT. L'algorithme ϕ-*locate* doit alors retourner un ensemble de sous-ensembles suspects où un sous-ensemble représente les contraintes fautives (i.e., post-condition). Pour traiter la question de correction de l'algorithme ϕ-*locate*, il suffit donc prouver le théorème suivant.

Théorème 2 *Soit $\mathcal{M}$, $\mathcal{P}$ et ϕ un Modèle-Oracle, un CPUT et une faute multiple :*

$$\mathcal{M}\ \neg conf\ \mathcal{P} \rightarrow \begin{cases} (\phi^+ \vee \phi^*) \wedge \left(\exists\{C_1, \ldots, C_l\} \subset \mathcal{P} : sol(\mathcal{M} \wedge \mathcal{P}\backslash\{C_1, \ldots, C_l\})) \neq \emptyset\right) \\ \vee \\ \phi^- \wedge \left(\exists C_i \in (\{C_1, \ldots, C_l\} \subset \mathcal{P}) : sol(\mathcal{M} \wedge \neg C_i) \neq \emptyset\right) \end{cases}$$

Soit $Loc_\phi \subset \mathcal{P}$ une localisation de ϕ où $|Loc_\phi| \leq \lambda$. Si on considère la conjonction de contraintes Loc_ϕ comme une seule contrainte, le théorème 2 est équivalent au théorème 1. Dès lors, les éléments de preuve du théorème 1 sont réutilisables pour le théorème 2.

L'algorithme ϕ-*locate* assure l'optimalité dans le calcul du *SuspiciousSet*. La ligne 8 permet de retirer de l'ensemble LOC toute localisation non-minimale. Dans le cas d'une faute ϕ^-, l'ensemble *SuspiciousSet* ne contiendra qu'une seule localisation minimale de la faute.

7. Correction de fautes dans les programmes à contraintes

7.1. Introduction

La mise-au-point de tout programme a besoin d'une révision de code une fois que la faute est localisée. La correction automatique des programmes est un sujet émergeant dans le génie logiciel comme le témoigne la section 2.3 de ce manuscrit. Les travaux existants qui concernent la correction des programmes conventionnels se basent sur la localisation de fautes, la mutation et les verdicts sur des suites de test pour proposer des corrections (Jones & Harrold, 2005; Weimer *et al.*, 2010). D'autre part, on trouve une autre démarche qui se base sur une spécification formelle avec des assertions (pré- et post-conditions) (Wei *et al.*, 2010). Dans l'optique d'une hybridation des deux démarches, nous proposons une méthodologie de correction automatique des programmes à contraintes basée sur : la localisation de fautes telle abordée dans le chapitre précédent, la proposition de corrections des contraintes suspectes à partir du Modèle-Oracle.

7.2. Exemple illustratif

Nous allons dans cette section reprendre l'exemple des n-reines présenté en section 6.2 du chapitre précédent.

7.2.1. Phase de correction

La phase de localisation a permis de retourner `cc4` comme contrainte suspecte. Á présent, la phase de correction a pour objectif de proposer des corrections possibles de `cc4`. L'idée est de pouvoir calculer à partir du Modèle-Oracle l'ensemble de contraintes qui pourra être une correction de `cc4`. Nous rappelons que `cc4` a comme rôle d'assurer que sur toute diagonale descendante, une reine au plus est positionnée. La contrainte qui a le même rôle dans le Modèle-Oracle est `c2` :

```
c2 :   forall(ordered i,j in 1..n)
           queens[i]+i != queens[j]+j;
```

Dès lors, cette contrainte est candidate pour être une correction de `cc4` dans le CPUT. Ceci permet de corriger la faute ϕ^+ pour toute instante du problème des n-reines en remplaçant la contrainte `cc4` du CPUT par `c2` du Modèle-Oracle. On peut également

proposer des corrections propres à des instances du problème. Par exemple pour une instance $n = 4$, une correction possible serait :

```
queens[3]+3 != queens[4]+4;
queens[2]+2 != queens[4]+4;
queens[1]+1 != queens[4]+4;
queens[2]+2 != queens[3]+3;
queens[1]+1 != queens[3]+3;
queens[1]+1 != queens[2]+2;
```

Dans la suite de ce chapitre, nous allons étudier la possibilité de faire une telle correction de façon automatique.

7.3. Intuition

La figure 7.1 montre un CPUT, noté $\mathcal{P}$, qui ne partage aucune solution avec le Modèle-Oracle $\mathcal{M}$ (cas de non-conformité). Le CPUT en question est constitué de sept contraintes (C_1 à C_7). Par souci de clarté, le CPUT est représenté par trois sous-ensembles : $Loc_1 = \{C_1, C_2\}$, $Loc_2 = \{C_3, C_4, C_5\}$, $Loc_3 = \{C_6, C_7\}$.

La phase de test **(A)** détecte une faute ϕ^+ dans notre CPUT en retournant une solution de $\mathcal{P}$ qui n'est pas une solution de $\mathcal{M}$.
La phase de localisation **(B)** retourne une seule localisation : $SuspiciousSet = \{Loc_3\}$.

L'objectif de correction **(C)** est de trouver un sous-ensemble de contraintes $Corr$ du Modèle-Oracle qui corrige Loc_3. Les contraintes qui forment $Corr$, permettent de rétablir la conformité si elles remplacent Loc_3 dans le CPUT :

$$\begin{cases} \mathcal{P}' \equiv (\mathcal{P} \backslash Loc_3) \wedge Corr \\ \mathcal{P}' \quad conf \quad \mathcal{M} \end{cases}$$

Remplacer Loc_3 par $Corr$ permet d'enlever de l'ensemble des solutions du CPUT, toute solution non-acceptable par le Modèle-Oracle. Ces solutions qui sont à retirer, correspond à $(\mathcal{P} \backslash Loc_3) \wedge \neg Corr$.

Définition 17 (Correction) *Soit Loc_ϕ une localisation de ϕ dans $\mathcal{P}$. Une correction possible de la faute ϕ est un sous-ensemble $Corr_\phi$ du Modèle-Oracle $\mathcal{M}$ tel que :*

$$Corr_\phi = \{C_i \in \mathcal{M} : sol(\mathcal{P} \backslash Loc_\phi \wedge \neg C_i) \neq \emptyset\}$$

Propriété 1 *Remplacer Loc_ϕ par $Corr_\phi$ permet d'avoir un CPUT conforme au Modèle-Oracle.*

Preuve : Les élements de preuve de la propriété 1 est détaillée un peu plus loin dans ce chapitre en section 7.4.2.

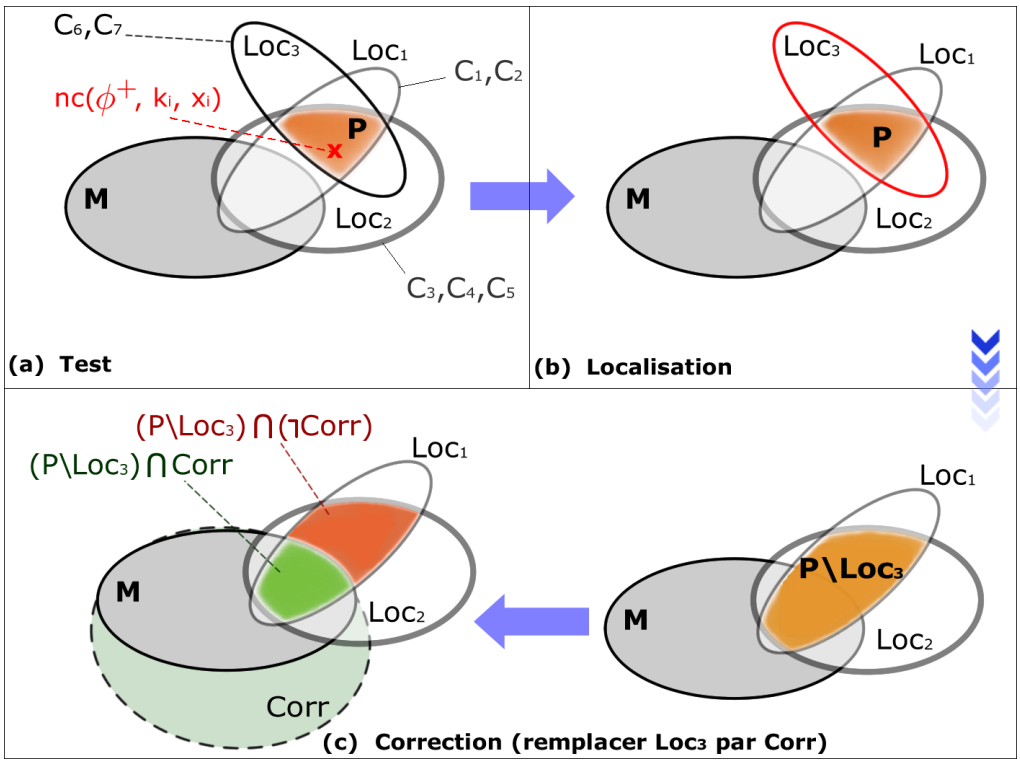


FIGURE 7.1.: Correction de faute ϕ^+.

Définition 18 (Correction minimale) *Soit $Corr_\phi$ une correction possible d'une faute ϕ dans un CPUT $\mathcal{P}$. $Corr_\phi$ est dite une correction minimale si et seulement s'il n'existe pas une correction $Corr'_\phi$ telle que $Corr'_\phi \subsetneq Corr_\phi$.*

Définition 19 (*CorrectionSet) Soit ϕ une faute dans un CPUT $\mathcal{P}$ et un Modèle-Oracle $\mathcal{M}$. Un CorrectionSet représente un ensemble de paires (localisation minimale, correction minimale) de ϕ :*

$$CorrectionSet = \{(Loc_\phi, Corr_\phi) : Loc_\phi \in SuspiciousSet,\ Corr_\phi \subset \mathcal{M}\}$$

Dès lors, l'objectif d'une correction automatique des programmes à contraintes est de trouver, pour toute localisation possible Loc_ϕ de faute ϕ, une correction minimale $Corr_\phi$.

7.4. Processus de correction

En se basant sur la définition 17, nous avons défini un processus de correction automatique qui propose, pour toute localisation possible de faute ϕ, une correction possible.

7.4.1. Algorithme ϕ-*correction*

Algorithm 7: ϕ-correction($\mathcal{M}$, $\mathcal{P}$, *SuspiciousSet*)

Input : Modèle-Oracle :$\mathcal{M}$, CPUT :$\mathcal{P}$, *SuspiciousSet*
Output: *CorrectionSet*

```
1  if SuspiciousSet = ∅ then
2  |  return (−, Corr(M, P))
3  end
4  else
5  |  CorrectionSet ← {(∅, ∅)}
6  |  foreach Loc_φ ∈ SuspiciousSet do
7  |  |  CorrectionSet ← CorrectionSet ∪ {(Loc_φ, Corr(M, P\Loc_φ))}
8  |  end
9  |  return CorrectionSet
10 end

11 Corr(A, B) :
12 Corr_φ ← ∅
13 foreach C_i ∈ A do
14 |  if sol(B ∧ ¬C_i) ≠ ∅ then
15 |  |  Corr_φ ← Corr_φ ∪ {C_i}
16 |  end
17 end
18 return Corr_φ
```

La procédure $Corr(\mathcal{A}, \mathcal{B})$ permet de calculer l'ensemble des contraintes $Corr_\phi \subset \mathcal{A}$ qui corrige $\mathcal{B}$. Corriger $\mathcal{B}$ consiste à enlever les solutions de $\mathcal{B}$ qui ne sont pas des solutions de $\mathcal{A}$. En d'autres termes, réduire l'ensemble des solutions de $\mathcal{A} \wedge \neg\mathcal{B}$ à vide ($sol(\mathcal{A} \wedge \neg\mathcal{B}) = \emptyset$).

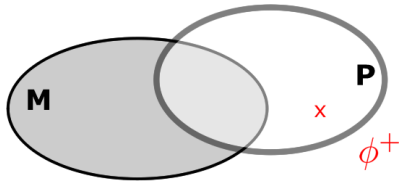
L'algorithme *ϕ-correction* prend en entrée le *SuspiciousSet* retourné par *ϕ-locate*. Si cet ensemble est vide (i.e., CPUT sous-contraint), *ϕ-correction* calcule l'ensemble des contraintes du Modèle-Oracle à ajouter au CPUT avec un appel de $Corr(\mathcal{M}, \mathcal{P})$. Autrement, pour chaque élément $Loc_\phi \in SuspiciousSet$, l'algorithme calcule une correction possible avec $Corr(\mathcal{M}, \mathcal{P} \backslash Loc_\phi)$.

7.4.2. Analyse de *ϕ-correction*

Prouver la correction de notre algorithme *ϕ-correction* revient à vérifier s'il calcule, pour tout élément $Loc_\phi \in SuspiciousSet$, une correction possible.

Un cas particulier est d'avoir une faute ϕ^+ tout en ayant des solutions partagées entre le CPUT et le Modèle-Oracle :

$$(sol(\mathcal{M}) \cap sol(\mathcal{P}) \neq \emptyset) \wedge (sol(\mathcal{P}) \backslash sol(\mathcal{M}) \neq \emptyset)$$



Ici, le $SuspiciousSet = \emptyset$ pour dire que le CPUT est sous-contraint où un renforcement et/ou un ajout de contraintes peut être une correction. L'appel de la procédure $Corr(\mathcal{M}, \mathcal{P})$, ligne 2 de l'algorithme, permet de retourner les contraintes $Corr_\phi$ à ajouter :

$$sol(\mathcal{P} \wedge Corr_\phi) \subseteq sol(\mathcal{M})$$

Dès lors, ceci permet d'avoir $(-, Corr_\phi) \in CorrectionSet$.

Pour les différents cas qui restent, nous avons au moins une localisation $Loc_\phi \in SuspiciousSet$. Enlever Loc_ϕ du CPUT augmente l'ensemble des solutions du système $(\mathcal{P} \backslash Loc_\phi)$:

$$(sol(\mathcal{M}) \cap sol(\mathcal{P} \backslash Loc_\phi) \neq \emptyset) \wedge (sol(\mathcal{P} \backslash Loc_\phi) \backslash sol(\mathcal{M}) \neq \emptyset)$$

Dès lors, on retrouve notre cas particulier vu précédemment et l'appel de $Corr(\mathcal{M}, \mathcal{P} \backslash Loc_\phi)$ permet d'avoir $(Loc_\phi, Corr_\phi) \in CorrectionSet$.

La preuve d'exactitude revient à vérifier si $Corr(\mathcal{A}, \mathcal{B})$ calcule bien des corrections minimales $Corr_\phi$. La ligne 14 assure cette optimalité car l'algorithme ne considère que des $C_i \in \mathcal{A}$ qui réduisent l'ensemble des solutions non-acceptables par $\mathcal{A}$.

Quatrième partie .

Implantation, étude expérimentale

8. Le prototype CPTEST

Ce chapitre a comme objectif la description de l'outil de test et de mise-au-point des programmes à contraintes CPTEST. Cet outil représente une première implémentation des différentes méthodologies décrites dans ce manuscrit, à savoir, la détection, la localisation et la correction des fautes dans les programmes à contraintes en OPL.

8.1. Description générale de CPTEST

CPTEST[1] est un outil écrit en Java qui totalise environ 25 000 lignes de code. Il comprend un analyseur syntaxique complet du langage OPL ainsi que des générateurs de programmes OPL pour la détection, la localisation et la correction des fautes. CPTEST fait appel au solveur de contraintes CP Optimizer 2.3 de la distribution IBM ILOG pour résoudre les programmes OPL.
Le code source de CPTEST est réparti en quatre *packages* (figure 8.1) :

`cptest` contient le *main* de l'application et 6 autres classes pour l'interface graphique de l'outil.

`cptest.ast` contient 29 classes différentes qui permettent la construction de l'arbre de syntaxe abstraite des programmes OPL.

`cptest.core` représente le cœur de l'outil avec 19 classes assurant les différentes fonctionnalités de CPTEST détaillées dans la suite de ce chapitre.

`cptest.parser` contient un analyseur syntaxique et un analyseur lexical.

L'utilisation de l'outil est simplifiée avec une interface graphique dont la figure 8.2 donne un aperçu. Les zones 1, 2 et 3 permettent de charger et/ou modifier le Modèle-Oracle, le CPUT et l'instance du problème ; la zone 4 est dédiée au paramétrage de l'outil : le type de conformité et/ou l'intervalle pour $conf_{bounds}$, la stratégie de recherche du solveur, négation avec ou sans dépliage des contraintes et un champ pour les contraintes de connexion ; la zone 5 est dédiée au test, localisation et correction de fautes.

8.2. Architecture générale de CPTEST

La figure 8.4 présente l'architecture générale de CPTEST. L'outil prend en entrée trois fichiers OPL (`m-oracle.mod, cput.mod, instance.dat`) qui représentent, respectivement, le Modèle-Oracle, un CPUT et une instance du problème.

CPTEST comprend les modules suivants :

1. CPTEST est disponible en ligne sur : www.irisa.fr/celtique/lazaar/CPTEST/

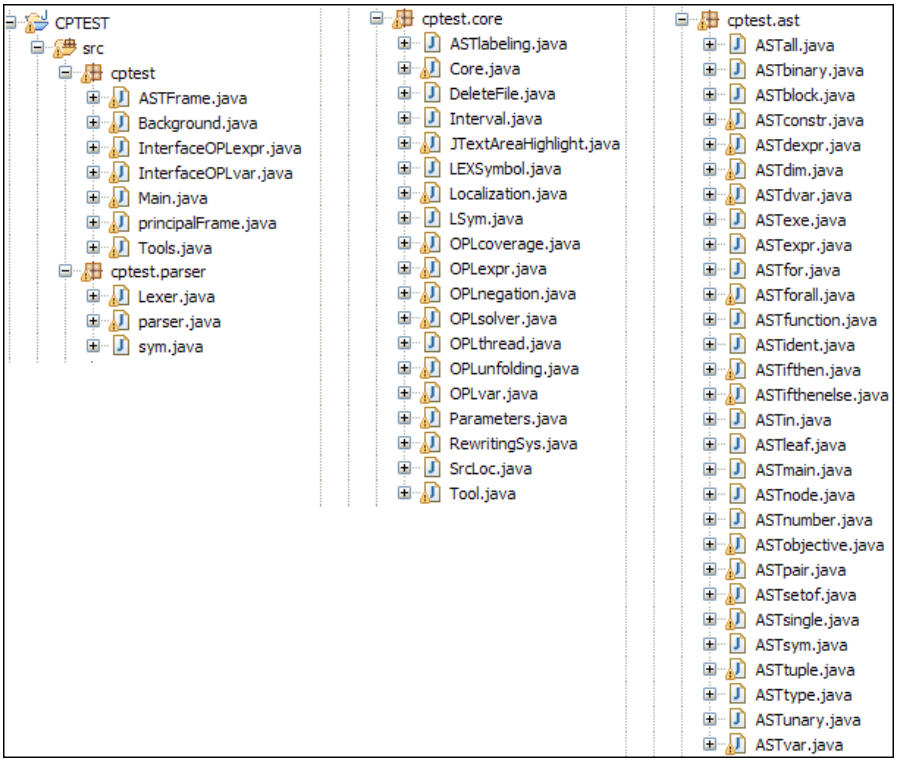


FIGURE 8.1.: Les packages de CPTEST.

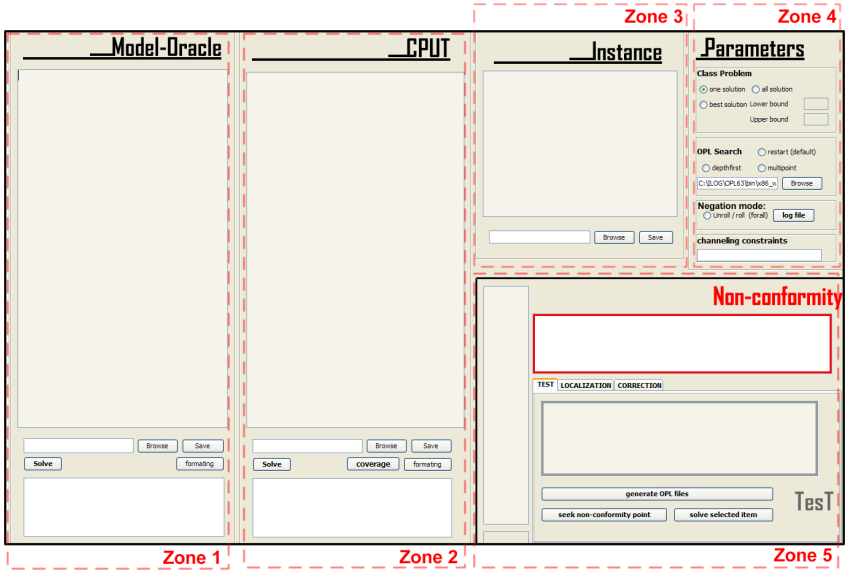


FIGURE 8.2.: L'interface graphique de CPTEST.

8.2.1. Modules de base

Analyseur syntaxique

La grammaire du langage OPL est disponible dans la distribution de IBM ILOG OPL v6.3 [2]. Á partir de cette grammaire, nous avons construit une grammaire LALR pour l'analyseur lexical JFlex [3] et l'analyseur syntaxique CUP [4] qui contient plus de 200 règles. Les actions qui accompagnent chaque règle de la grammaire permettent de construire un arbre de syntaxe abstraite (i.e., `AST`) des fichiers OPL (i.e., les fichiers $*.mod$ et $*.dat$).

Module de test

Dans le package `cptest.core` on trouve une implémentation des processus de test $algo_{one}$, $algo_{all}$ et $algo_{bounds}$ du chapitre 5, ainsi que le générateur des non-conformités *one_ negated*. Un appel de *one_ negated* sur deux modèles OPL $\mathcal{A}$ et $\mathcal{B}$ permet, à partir des AST de $\mathcal{A}$ et $\mathcal{B}$ de produire des AST intermédiaires de $\mathcal{A} \wedge \neg C_i$ avec $C_i \in \mathcal{B}$. Nous détaillerons un peu plus loin dans ce chapitre comment la négation et la projection dans *one_ negated* sont implémentées.

Module de localisation

Le module `cptest.core.Localization` implémente les deux algorithmes *locate* et ϕ-*locate* du chapitre 6. Á partir des AST du Modèle-Oracle et du CPUT, ce module produit des AST de programmes intermédiaires qui permettent de faire de la localisation telle décrite dans les algorithmes précédents. Un *powerSet* est également implémenté dans le cas d'une faute multiple avec une certaine limite pour éviter l'explosion combinatoire.

Module de correction

Le package `cptest.core` inclut une implémentation de l'algorithme de correction automatique (i.e., ϕ-*correction* du chapitre 7). Ce module prend chaque contrainte suspecte et génère des fichiers OPL intermédiaires pour proposer des corrections.

8.2.2. Modules auxiliaires

Module de paramétrage

Les modules de base détaillés précédemment ont besoin de paramètres. Le premier paramètre, concernant le module de test, est le type de conformité en question (i.e., *one*, *all*, *best* et *bounds* avec les bornes l et u). Le deuxième paramètre concerne le type de négation à appliquer (avec ou sans dépliage, voir section 8.4). Un troisième paramètre concerne les contraintes de connexion qui sont nécessaires pour le traitement de la projection (voir section 8.4). On note également le paramètre λ qui permet de fixer la taille maximale d'une localisation.

2. publib.boulder.ibm.com/infocenter/oplinfoc/v6r3/
3. jflex.de/
4. www.cs.princeton.edu/ appel/modern/java/CUP/

Module de formatage de texte

Pour une plus grande lisibilité des modèles à contraintes, nous avons développé un module d'affichage (i.e., `cptest.core.ASTlabeling`) qui permet un formatage de texte en étiquetant chaque contrainte.

Module de résolution

Pour assurer les différentes fonctionnalités de l'outil, on fait appel au solveur de contraintes CP Optimizer d'IBM ILOG. De ce fait, nous avons développé un module qui assure la résolution des différents modèles intermédiaires de façon séquentielle et/ou parallèle avec du multi-theading (i.e., `cptest.core.OPLsolver` et `cptest.core.OPLthread`).

Module de couverture des contraintes

Comme plusieurs langages de modélisation à contraintes, OPL permet d'avoir des structures de contrôle avec des boucles et des branches *if-then-else*. Pour une instance donnée, le solveur OPL fait un pré-traitement qui permet de générer l'ensemble des contraintes. Nous avons développé un module (i.e., `cptest.core.OPLcoverage`) qui permet de savoir préalablement les contraintes qui vont être générées et postées dans le *store* de contraintes. La figure 8.3 montre un exemple de couverture sur un CPUT des règles de Golomb. Avec une instance $m = 6$, on a la partie *then* de la contrainte `cc4` qui sera postée alors que la partie *else* n'est pas couverte par $m = 6$. Aussi, la contrainte `cc6` ne sera pas postée.

8.3. Session de travail sous CPTEST

Nous allons présenter dans cette section une session de travail détaillée de CPTEST.

Chargement des fichiers et paramétrage de CPTEST

Une fois qu'on a lancé CPTEST, on peut charger les fichiers correspondants au Modèle-Oracle, CPUT et l'instance du problème. Nous avons pris comme exemple le problème des n-reines dont la figure 8.5 montre le chargement des différents fichiers. L'utilisateur a également la possibilité d'utiliser l'interface de CPTEST comme éditeur de programme OPL.

Dans cette session de travail, nous allons commencer par tester le CPUT en question sous $conf_{one}$. Dans la partie paramétrage de l'outil, on sélectionne donc $conf_{one}$. On sélectionne également une stratégie de recherche du solveur CP Optimizer, la stratégie par défaut est une recherche en profondeur d'abord avec des redémarrages en cas d'échec. On sélectionne également le mode de négation avec dépliage que nous détaillerons par la suite dans la section 8.4. Dans le cas où on procède à un test sous $conf_{all}$, nous avons réservé un champ où l'utilisateur peut fournir les numéros des contraintes de connexion nécessaires pour le calcul de la projection (section 8.4).

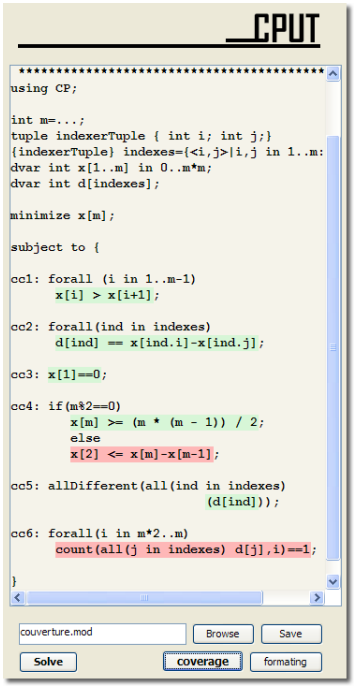


FIGURE 8.3.: Module de couverture des contraintes dans CPTEST.

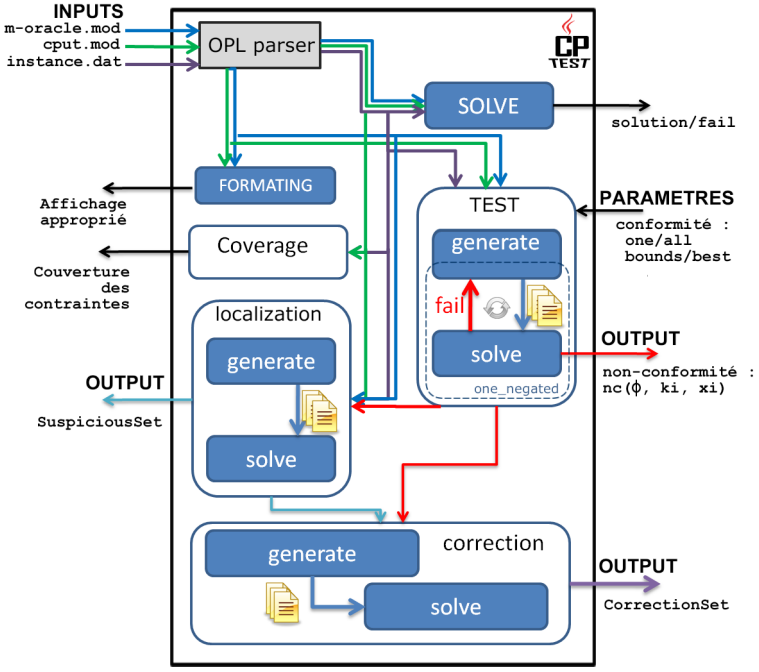


FIGURE 8.4.: Architecture générale de CPTEST.

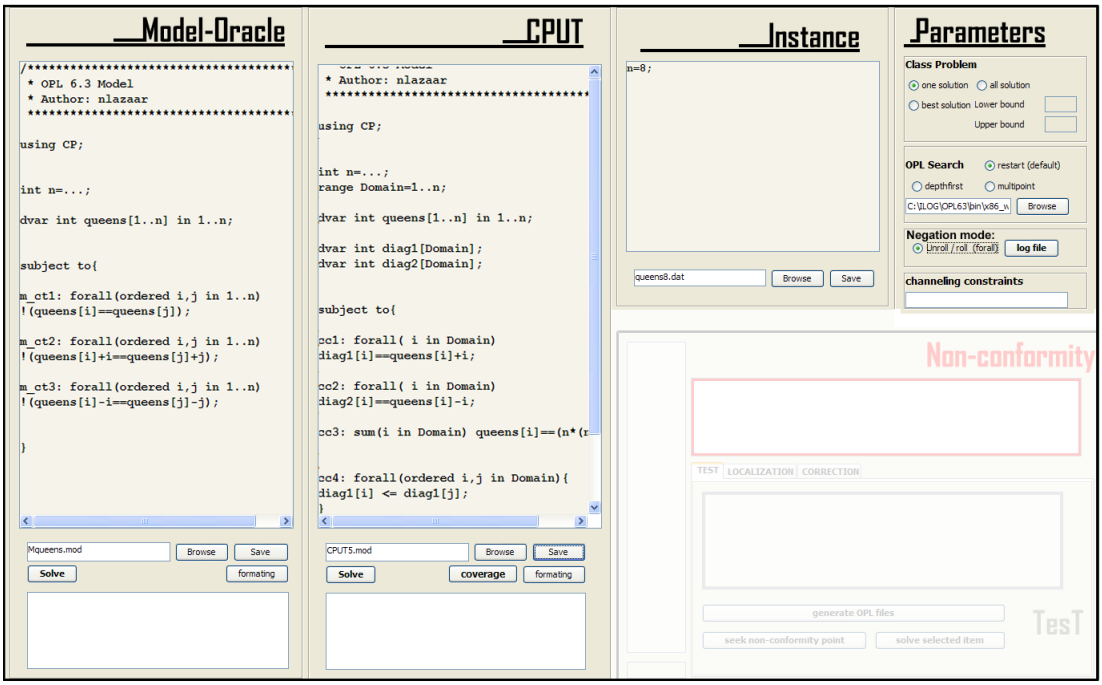


FIGURE 8.5.: Chargement des fichiers n-reines dans CPTEST.

Maintenant que les fichiers sont chargés et les paramètres fixés, on va passer à une phase de test.

8.3.1. Phase de test

La figure 8.6 montre la zone 5 de l'outil dans une phase de test. On commence par générer les fichiers OPL correspondant au test avec de la négation de contraintes (i.e., $\mathcal{P} \wedge \neg C_i : \ C_i \in \mathcal{M}$). Dès lors, on lance le test à la recherche d'une non-conformité entre le CPUT et le Modèle-Oracle. Pour cet exemple, CPTEST retourne une non-conformité après 4.5 *secondes* de temps de calcul. Cette non-conformité révèle la présence d'une faute de type ϕ^+ dans le CPUT en question.

8.3.2. Phase de localisation

Dans la même zone 5 de CPTEST, nous avons consacré un volet spécial pour la localisation des fautes. Comme le montre la figure 8.7, on peut donner une valeur à λ qui est un paramètre important de l'algorithme *ϕ-locate*. Pour notre exemple des n-reines, CPTEST retourne un *SuspiciousSet* avec comme seule contrainte suspecte `cc4` en moins d'une seconde de calcul.

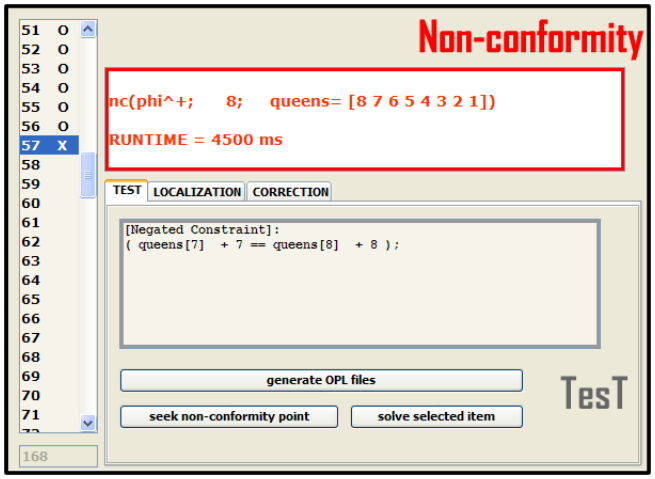


FIGURE 8.6.: Zone 5 de CPTEST : phase de test.

FIGURE 8.7.: Zone 5 de CPTEST : phase de localisation.

8.3.3. Phase de correction

On retrouve également un autre volet pour la correction qui permet de calculer et d'afficher le *CorrectionSet*. Pour notre exemple des n-reines, la figure 8.8 montre le *CorrectionSet* retourné. Ici, l'option dépliage des contraintes qu'on trouve en paramètre (i.e., zone 4) est activée. La correction calculée est donc propre à l'instance 8-reines. Il suffit donc de remplacer la contrainte `cc4` par l'ensemble des contraintes élémentaires $Corr_\phi$ à droite de la figure 8.8 pour corriger la faute ϕ^+ du CPUT.

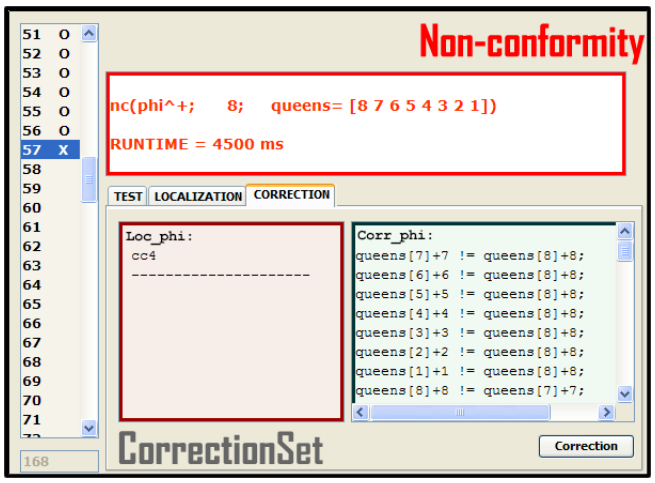


FIGURE 8.8.: Zone 5 de CPTEST : phase de correction.

8.4. Choix d'implémentation

8.4.1. Problème des variables auxiliaires

Dans la section 5.6.1, nous avons décrit le problème des variables auxiliaires et comment une projection de contraintes remédier à ce problème. Cette projection de contraintes revient en premier lieu à séparer les contraintes de connexion du reste des contraintes. CPTEST vérifie en premier si l'utilisateur à annoter les contraintes de connexion dans le champ réservé. Autrement, il procède à une identification de contraintes de connexion qui sont généralement des contraintes d'égalité impliquant d'une part des variables de base et de l'autre part les variables auxiliaires.

8.4.2. Négation de contraintes

Le module de test et de correction se base sur la négation de contraintes. CPTEST assure une négation sémantique des contraintes OPL telle décrite en section 5.6.1. Le tableau 8.1 présente une syntaxe réduite des contraintes qui peuvent être exprimées en OPL version 6.x, CPTEST traite la négation de toutes ces contraintes dans la partie

TABLE 8.1.: Syntaxe réduite des contraintes OPL

Ctrs ::=	*Ctr* | *Ctrs*
Ctr ::=	*rel* | `forall(` *rel* `)` *Ctrs*| `or(` *rel* `)` *Ctrs* | `if(` *rel* `)` *Ctrs* `else` *Ctrs*
	| `allDifferent(`*rel*`)` | `allMinDistance(`*rel*`)` | `inverse(`*rel*`)`| `forbiddenAssignments(`*rel*`)`
	| `allowedAssignments(`*rel*`)`| `pack(`*rel*`)`

rel : non-terminal sur les relations.

`cptest.core.OPLnegation` qui prend le sous-arbre abstrait AST d'une contrainte et génère automatiquement le sous-arbre de la négation AST' (e.g., figure 8.9).

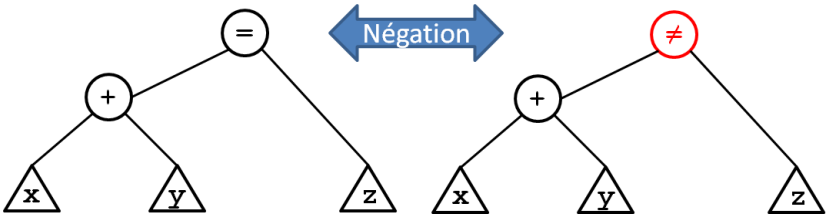


FIGURE 8.9.: AST de la négation de $(x + y = z)$.

Le module d'analyse syntaxique de CPTEST prend des programmes OPL de la version 6.x. Cette dernière version a connu une migration de plusieurs contraintes globales présentes dans la version 3.x (IBM, 2009) (`distribute`, `sequence`, `atLeast`, `atMost`, etc.). Ces contraintes sont exprimées désormais à l'aide de l'agrégateur `forall`.

Exemple 17 (contrainte `distribute`)

En OPL v3.x *:*

```
distribute(cardi, value, base)
```

En OPL v6.x *:*

```
forall(i in RangeOfValue)
    count(base, value[i]) == cardi[i];
```

L'implémentation actuelle de l'outil réécrit les contraintes globales à l'aide des `forall`. Prenons par exemple la contrainte `allDifferent((i in RangeOfValue) x[i])`, cette contrainte peut être réécrite en :

```
forall(ordered i,j in RangeOfValue) x[i] != x[j] ;
```

Négation sans dépliage

La négation des contraintes écrites avec des `forall` nous donne des contraintes écrites avec des `or` et vice-versa. Prenons l'exemple du `allDifferent` qui est réécrit avec un `forall`, la négation nous donne :

```
or (ordered i,j in RangeOfValue) x[i] == x[j] ;
```

Négation avec dépliage

Sous une instance donnée du problème, les `forall` peuvent être dépliés avant d'appliquer la négation. Prenons par exemple le `forall` suivant :

```
forall(ordered i,j in 1..3) x[i] != x[j] ;
```

un dépliage nous donne :

```
x[1] != x[2] ;
x[1] != x[3] ;
x[2] != x[3] ;
```

Dès lors, la négation devient plus intéressante qu'un `or` qui nous donne, dans le cas général, un arbre de recherche très large.

Négation du if-then-else

Le langage OPL permet d'avoir dans les modèles à contraintes des conditionnelles :

```
If(cond)
  bloc_contrs1 ;
else
bloc_contrs2 ;
```

Sachant que la condition `cond` ne peut contenir de variables de décision, la négation est équivalente à la négation du bloc *then* et du bloc *else* :

```
If(cond)
  Neg(bloc_contrs1) ;
else
Neg(bloc_contrs2) ;
```

8.4.3. Optimisation dans l'implémentation du *one_ negated*

Nous rappelons ici que le générateur *one_ negated(A, B)* (section 5.6) permet de retourner une solution du système de contraintes A qui n'est pas une solution de B en se basant sur la négation de contraintes. Nier une contrainte $C_i \in B$ qui appartient également au système A nous donne un système insatisfiable (i.e., $A \wedge \neg C_i \equiv fail$) .
Dans `cptest.core.RewritingSys`, on trouve une implémentation d'un sélectionneur de contraintes à nier qui inclut un système de réécriture modulo une théorie $\mathcal{T}$:

$$\mathcal{T} : \left\{ \begin{array}{ccc} x \circ y \to y \circ x, & (x \circ y) \circ z \to x \circ (y \circ z), & x + 0 \to x, \\ x * 1 \to x, & x * 0 \to 0, & x \times (y \bullet z) \to (x \times y) \bullet (x \times z), \\ x < y \leftrightarrow y > x, & x \leq y \leftrightarrow y \geq x, & x - 0 \to x, \end{array} \right\}$$

où $\circ \in \{+, *, \wedge, \vee\}$, $\times \in \{*, \wedge, \vee\}$ et $\bullet \in \{+, \wedge, \vee\}$.

Ceci nous permet d'éviter le cas où une contrainte appartient aux deux systèmes :

$$(C_i \in A) \wedge (C_j \in B) \wedge (C_i \equiv_{\mathcal{T}} C_j) \Rightarrow A \wedge \neg C_j \equiv B \wedge \neg C_i \equiv fail$$

Exemple 18 *Soit C_1 et C_2 respectivement deux contraintes de A et B avec :*

$$C_1 : \ (x + y) * z > t + 0 \qquad\qquad C_2 : \ t < (x * z) + (y * z)$$

Dans un appel de one_negated(A, B), la contrainte C_2 ne sera pas sélectionner car $(C_2 \equiv_{\mathcal{T}} C_1)$ et $(A \wedge \neg C_2 \equiv fail)$.

8.5. Limitations

Le point à améliorer dans CPTEST est la négation des contraintes globales. Ceci implique une étude approfondie et un travail théorique sur la négation de contraintes globales. L'idée est d'avoir un opérateur générique de négation qui permettra de prendre une contrainte globale et de générer un filtrage dédié à sa version niée.

Dans la présente implémentation, nous avons mit en place à différents endroits des verrous et des *timeout* pour ne pas tomber dans des dépassements de capacité en mémoire et en temps de calcul. Ceci est dû à la complexité de quelques algorithmes comme *ϕ-locate* avec son *powerset*. De plus, la négation avec dépliage des `forall` peut aussi tomber dans une explosion combinatoire.

Le système de réécriture implémenté dans CPTEST peut également être amélioré. La théorie $\mathcal{T}$ de ce système peut être étendue sur d'autres règles qui peuvent, à titre d'exemple, exprimer une équivalence entre un ensemble de contraintes élémentaires et une contrainte globale (e.g., $(x \neq y) \wedge (y \neq z) \wedge (x \neq z) \rightarrow$ *allDifferent*(x, y, z)).

9. Expérimentations

Dans ce chapitre, nous présentons les premiers résultats expérimentaux de CPTEST sur des problèmes académiques. Nous avons séléctionné les règles de golomb, les n-reines, golfeurs sociables (social golfers) ainsi que l'ordonnancement de véhicules qui est dérivé d'un problème réel des chaînes de montage des véhicules. Les résultats que nous donnons sur la détection des fautes ainsi que la mise-au-point des programmes à contraintes ont été produit sur une machine standard Intel Core2 Duo CPU 2.40Ghz avec 2 Go de RAM.

9.1. Procédure d'expérimentation

L'objectif de notre étude expérimentale est multiple : valider nos approches de test, de localisation des fautes et de correction automatique ; valider la capacité de CPTEST à détecter des fautes dans des programmes OPL ; montrer que le test d'un programme à contraintes est moins coûteux que le résoudre ; étudier le temps nécessaire à la localisation et la correction automatique des fautes.

Étant donné un problème, la procédure que nous avons adoptée est la suivante :
- Traduire, en premier lieu, la spécification du problème en un premier modèle à contraintes $\mathcal{M}$ en OPL (Modèle-Oracle).
- Appliquer les différents raffinements possibles pour obtenir un programme à contraintes $\mathcal{P}$ dédié à résoudre des instances difficiles du problème.
- Injecter de manière systématique des fautes significatives dans $\mathcal{P}$ et obtenir des CPUTs incluant ces fautes.
- Soumettre chaque CPUT à CPTEST pour une phase de test sous une conformité donnée.
- Dans le cas où une faute ϕ est détectée, lancer CPTEST pour une phase de localisation.
- Lancer la correction automatique de CPTEST.

Les fautes significatives ici représentent une relaxation et/ou un renforcement de contraintes, une mauvaise connexion d'une variable auxiliaire, l'ajout d'une contrainte de symétrie et sa négation, mauvaise formulation de contrainte redondante, utilisation inadaptée d'une contrainte globale, etc.

Le tableau 9.1 présente les résultats de CPTEST sur les règles de Golomb, n-reines, golfeurs sociables et ordonnancement de véhicules[1]. Le tableau est constitué de quatre

1. Les résultats, ainsi que d'autres, sont disponible en ligne : `www.irisa.fr/celtique/lazaar/CPTEST`

colonnes :

CPUTs : cette colonne décrit les différents CPUT. Chaque CPUT est composé d'un nombre de contraintes regroupées en `sets`. Pour une instance donnée, (e.g., m=6 des règles de Golomb), ces ensembles ou `sets` totalisent un nombre précis de variables (i.e., `# vars`) et de contraintes élémentaires et globales (i.e., `# ctrs`). La partie ϕ`-injection` indique dans quelle contrainte du CPUT la faute ϕ est introduite.

Test : Cette colonne est dédiée à la phase de test de CPTEST. Elle présente les résultats de $algo_{one}$ et $algo_{all}$ avec les temps en seconde.

Localisation : Cette partie du tableau présente les *SuspiciousSet* retournés par ϕ-*locate* avec les temps en seconde.

Correction : Les résultats de la phase de correction de CPTEST avec des *CorrectionSet* et des temps en secondes.

Nous avons sélectionné particulièrement ces quatre problèmes PPC à cause des caractéristiques que présente chacun :

- Le problème des règles de Golomb représente un exemple idéal pour appliquer les différents raffinements possibles. De plus, il représente un problème d'optimisation qui nous permet d'étudier $conf_{bounds}$ et $conf_{best}$.

- Le problème des n-reines peut être modélisé à l'aide de contraintes de différence où un raffinement peut être l'ajout d'une contrainte *allDifferent*. Ceci nous permettra d'expérimenter notre négation syntaxique de cette contrainte globale.

- Le problème des golfeurs sociables représente un bon exemple pour illustrer les symétries de valeurs/variables en PPC et comment des raffinements avec des cassures de symétries peuvent introduire des fautes.

- L'ordonnancement de véhicules est dérivé d'un problème réel qui consiste à trouver une chaîne de montage de véhicules tout en satisfaisant les demandes en options des véhicules ainsi qu'un ensemble de contraintes de capacité des installateurs d'options. La modélisation de ce problème est très intéressante où différents raffinements peuvent être appliqués. Particulièrement, l'utilisation de la contrainte globale *pack*. Ceci nous permettra de tester le bon usage de cette contrainte ainsi que sa négation syntaxique

Les résultats du tableau 9.1 sont commentés dans les sections suivantes.

TABLE 9.1.: Résultats de CPTEST sur le Test et de la mise-au-point des programmes à contraintes.

	CPUTs					Test			Localisation		Correction	
	sets	# vars	# ctrs	ϕ-Inj.	$algo_{one}$	T(s)	$algo_{all}$	T(s)	Susp.Set	T(s)	CorrectionSet	T(s)
6-golomb												
CPUT1	2	6	187	cc2	nc(ϕ^+,6,g1)	0.86	nc(ϕ^+,6,g4)	11.43	–	0.96	(–, c2::50ct)	17.43
CPUT2	4	21	204	cc3	nc(ϕ^+,6,g2)	1.32	nc(ϕ^+,6,g5)	25.61	–	2.65	(–, c2::49ct)	27.03
CPUT3	5	21	279	cc3	nc(ϕ^+,6,g1)	0.67	nc(ϕ^+,6,g1)	14.48	–	4.96	(–, c2::50ct)	46.70
CPUT4	9	21	290	cc5	nc(ϕ^+,6,g3)	1.01	nc(ϕ^-,6,g6)	1.71	–	10.54	(–, c2::5ct)	29.63
CPUT5	9	21	283	cc1	nc(ϕ^*,6,–)	10.46	nc(ϕ^-,6,g7)	0.42	cc1	9.57	(cc1, c1::3ct)	1 698.04
CPUT6	8	21	262	cc7	nc(ϕ^*,6,–)	9.37	nc(ϕ^-,6,g8)	0.48	cc7	9.36	(cc7, c2::22ct)	107.81
CPUT7	9	21	1 183	cc9	nc(ϕ^*,6,–)	35.10	nc(ϕ^-,6,g9)	0.51	cc9	13.50	(cc9, –)	108.45
g1=[0 1 3 6 13 20], g2=[0 1 3 10 13 20], g3=[0 9 11 12 15 19], g4=[0 1 5 7 10 15], g5=[0 1 3 8 12 15], g6=[0 2 7 13 16 17], g7=[0 5 7 13 16 17], g8=[0 3 5 9 16 17], g9=[0 1 4 10 12 17]												
8-reines												
CPUT1	12	48	236	cc11	nc(ϕ^*,8,–)	8.32	nc(ϕ^-,8,q4)	0.54	cc11	7.53s	(cc11, –)	18.21s
CPUT2	12	48	236	cc12	nc(ϕ^*,8,–)	8.21	nc(ϕ^-,8,q4)	0.40	cc12	8.10	(cc12, –)	18.10
CPUT3	12	48	236	cc11	nc(ϕ^*,8,–)	7.85	nc(ϕ^-,8,q4)	0.50	cc11	8.14	(cc11, –)	18.35
CPUT4	12	48	236	cc12	nc(ϕ^*,8,–)	8.06	nc(ϕ^-,8,q4)	0.52	cc12	8.16	(cc12, –)	18.23
CPUT5	6	24	50	cc4	nc(ϕ^+,8,q1)	2.78	nc(ϕ^-,8,q5)	0.45	cc4	2.32	(cc4, c2::28ct)	7.21
CPUT6	3	72	211	cc3	nc(ϕ^+,8,q2)	3.34	nc(ϕ^-,8,q5)	0.21	–	0.98	(–, c2|c3::56ct)	9.28
CPUT7	2	24	202	cc1	nc(ϕ^+,8,q3)	3.07	nc(ϕ^-,8,q6)	0.22	–	0.53	(–, c2|c3::56ct)	9.12
q1=[8 7 6 5 4 3 2 1], q2=[7 2 3 6 8 1 5 4], q3=[8 4 3 6 5 7 2 1], q4=[7 3 1 6 8 5 2 4], q5= [5 7 1 3 8 6 4 2], q6=[4 2 7 3 6 8 5 1]												
sI-golfeurs												
CPUT1	5	36	2 738	cc1	nc(ϕ^*,sI,–)	10.28	nc(ϕ^-,sI,s2)	0.45	cc1	3.65	(cc1, c1|c2::58ct)	32.95
CPUT2	5	36	2 738	cc2	nc(ϕ^+,sI,s1)	0.31	nc(ϕ^-,sI,s2)	0.82	–	3.51	(–, c1|c2::75ct)	18.32
CPUT3	5	36	2 738	cc3	nc(ϕ^*,sI,–)	9.85	nc(ϕ^-,sI,s2)	0.54	cc3	3.71	(cc3, c1|c2::58ct)	36.28
CPUT4	5	36	2 738	cc4	nc(ϕ^*,sI,–)	9.37	nc(ϕ^-,sI,s2)	0.40	cc4	3.64	(cc4, c1|c2::58ct)	67.39
									cc5		(cc5, c1|c2::58ct)	
CPUT5	5	36	2 738	cc5	nc(ϕ^*,sI,–)	9.21	nc(ϕ^-,sI,s2)	0.39	cc5	3.75	(cc5, c1|c2::58ct)	34.18
sI: weeks = 4; groups = 3; groupSize = 3; s1=[[1 1 1 1][1 2 2 2][1 3 3 3][2 1 3 3][2 2 2 1][2 2 1 1][3 3 3 2][3 1 1 2][3 3 2 3]], s2=[[3 3 1 1][2 2 2 1][1 2 1 2][2 1 1 3][3 1 2 2][3 2 3 3][1 3 2 3][2 3 3 2][1 1 3 1]]												
cI-ord. véhic.												
CPUT1	2	10	88	cc2	nc(ϕ^+,cI,o1)	5.68	nc(ϕ^+,cI,o1)	8.54	–	1.34	(–, c2::32ct)	6.09
CPUT2	3	60	139	cc3	nc(ϕ^+,cI,o2)	5.82	nc(ϕ^+,cI,o3)	17.95	–	1.54	(–, c2::32ct)	6.28
CPUT3	3	60	139	cc2	nc(ϕ^+,cI,o3)	7.18	nc(ϕ^+,cI,o5)	9.84	–	1.51	(–, c2::31ct)	3.28
CPUT4	4	66	113	cc2	nc(ϕ^*,cI,–)	2.78	nc(ϕ^-,cI,o6)	0.32	cc2	3.07	(cc2, c2::37ct)	34.95
CPUT5	6	66	185	cc1	nc(ϕ^*,cI,–)	2.62	nc(ϕ^-,cI,o7)	0.11	cc1	4.96	(cc1, –)	9.08
CPUT6	6	66	185	cc6	nc(ϕ^*,cI,–)	2.64	nc(ϕ^-,cI,o7)	0.32	cc6	5.09	(cc6, –)	9.23
CPUT7	6	66	185	cc5	nc(ϕ^*,cI,–)	2.40	nc(ϕ^-,cI,o6)	0.29	cc5	5.09	(cc5, –),	14.98
									cc6		(cc6, –)	
cI: nbSlots = 10; nbCars = 6; nbOptions = 5; demand = [1,1,2,2,2,2]; capacity = [[1,2],[2,3],[1,3],[2,5],[1,5]]; optionDemand = [5,6,3,4,2]; option = [[1,0,0,0,1,1],[0,0,1,1,0,1],[1,0,0,0,1,0],[1,1,0,1,0,0],[0,0,1,0,0,0]]; o1=[4 5 4 6 3 6 5 1 3 2], o2=[4 1 6 4 3 5 3 6 2 5], o3=[4 6 2 5 3 6 1 3 5 4], o4=[1 6 4 3 2 5 3 5 4 6], o5=[5 2 6 3 1 4 3 6 4 5], o6=[5 4 3 5 4 6 2 6 3 1], o7=[1 3 6 2 6 4 5 3 4 5]												

Susp.Set: SuspiciousSet | ϕ-inj. : injection de fautes

9.2. Les règles de Golomb

Le problème des règles de Golomb est un problème d'optimisation académique difficile (Prob006 de la CSPLib[2]).

Le chapitre 3 introduit et prend comme fil rouge ce problème pour illustrer les différents raffinements possibles. La partie **(A)** de la figure 3.1 présente le Modèle-Oracle avec deux ensembles de contraintes élémentaires. La partie **(B)** présente le programme à contraintes suite à des raffinements $\mathcal{R}^n$ avec de nouvelles variables, six différents ensembles de contraintes élémentaires et une contrainte globale `cc6`.

En altérant la suite de raffinements $\mathcal{R}^n$, nous avons introduit des fautes ϕ, ce qui nous a permis de produire sept CPUT différents qu'on retrouve en annexe. Nous avons sélectionné le CPUT4 et le CPUT6 de Golomb car leurs résultats sont significatifs.

Test

Le CPUT4 est constitué de neuf ensembles de contraintes différentes où une instance de règles de six marques ($m = 6$) génère 21 variables et 290 contraintes. Une faute ϕ est introduite manuellement en `cc5` qui remplace la contrainte globale `allDifferent` sur les distances :

```
cc5: forall(i in m..m*2)
        count(all(j in indexes) d[j],i)==1;
```

Cette contrainte assure une valeur différente pour quelques distances $(x_j - x_i)$ mais pas toutes.

Sous la relation de conformité $conf_{one}$, CPTEST retourne, en une seconde, la non-conformité suivante :

`nc(`ϕ^+`,6, g3)` avec `g3=[0 9 11 12 15 19]`

Dès lors, une faute de type ϕ^+ est détectée par CPTEST où `g3` représente une solution de CPUT4 qui n'est pas une solution du Modèle-Oracle.

Sous une conformité $conf_{all}$ où on cherche à avoir toutes les solutions du problème, CPTEST retourne en $1.71s$ la non-conformité suivante :

`nc(`ϕ^-`,6, g6)` avec `g6=[0 2 7 13 16 17]`

Ici la faute est de type ϕ^-. En effet, `g6` représente une solution du Modèle-Oracle (une vraie règle de Golomb) qui n'est pas une solution du CPUT4. Cette non-conformité n'est pas due à la faute que nous avons introduite en `cc5` du CPUT4, mais plutôt à l'ajout de la contrainte `cc4` :

```
cc4: x[2] >= x[m] - x[m-1];
```

2. `www.cs.st-andrews.ac.uk/~ianm/CSPLib/prob/prob006/`

Cette contrainte casse la symétrie du problème. Ce type de raffinement rend le CPUT4 non-conforme au Modèle-Oracle ($conf_{all}$) où la solution `g6` est supprimée de l'ensemble des solutions de CPUT4.

CPTEST arrive à détecter également les fautes de type ϕ^*. Prenons maintenant le CPUT6 dont la faute introduite dans `cc7` le rend insatisfiable. CPTEST retourne, en moins de $10s$ la non-conformité :

$$\texttt{nc}(\phi^*,\texttt{6, }-)$$

Localisation et correction

Pour le CPUT4, CPTEST retourne un *SuspiciousSet* vide. Ceci montre que le CPUT4 a besoin d'ajout et/ou renforcement de contraintes pour rétablir une conformité $conf_{one}$ avec le Modèle-Oracle. En phase de correction, CPTEST retourne le *CorrectionSet* suivant après $30s$ de temps de calcul :

$$CorrectionSet = \{(-,\ \texttt{c2::5ct})\}$$

Pour une instance $m = 6$, la contrainte `c2` du Modèle-Oracle encapsule 221 contraintes élémentaires. Un ajout de cinq contraintes élémentaires de `c2` permet de corriger l'instance $m = 6$ du CPUT4 alors que la contrainte `c2` corrige toute instance de CPUT4 :

```
(x[6] - x[4]) != (x[4] - x[3]);
(x[6] - x[3]) != (x[3] - x[1]);
(x[5] - x[3]) != (x[6] - x[5]);
(x[5] - x[3]) != (x[3] - x[2]);
(x[4] - x[2]) != (x[5] - x[4]);
```

Concernant le CPUT6, la phase de localisation retourne :

$$SuspiciousSet = \{\texttt{cc7}\}$$

En effet, la faute a été introduite dans la contrainte `cc7` de CPUT6 et CPTEST arrive à retourner cette contrainte comme suspecte.

La phase de correction retourne, en moins de $2min$, 22 contraintes élémentaires qui peuvent remplacer `cc7` dans l'instance $m = 6$ du CPUT6 pour rétablir la conformité avec le Modèle-Oracle :

$$CorrectionSet = \{(\texttt{cc7}, \texttt{c2 :: 22ct})\}$$

De plus, remplacer `cc7` par `c2` du Modèle-Oracle permet de corriger CPUT6 pour toute instance.

Le cas du CPUT5 représente le pire des cas de la phase de correction automatique. CPTEST prend $30min$ pour trouver une correction possible à la faute injectée dans `cc1`. Ceci est dû au nombre de résolutions lancées.

TABLE 9.2.: Résultats de Test de CPTEST sur 8-Golomb.

		$conf_{best}$	$conf_{bounds}$
non-conformité		nc(ϕ^+,8,[0 1 3 6 10 15 24 33])	nc(ϕ^+,8,[0 2 3 6 11 58 72 86])
CPUT1	T(s)	7.31s	5.64s
non-conformité		nc(ϕ^+,8,[0 3 4 9 13 15 24 33])	nc(ϕ^+,8,[0 18 39 43 45 46 55 64])
CPUT2	T(s)	174.43s	4.64s
non-conformité		nc(ϕ^+,8,[0 3 4 9 13 15 24 33])	nc(ϕ^+,8,[0 18 39 43 45 46 55 64])
CPUT3	T(s)	389.04s	7.15s
non-conformité		nc(ϕ^+,8,[0 6 13 21 22 25 27 32])	nc(ϕ^+,8,[0 21 30 32 42 45 46 50])
CPUT4	T(s)	12.53s	9.01s

$[l, u] = [50, 100]$

Dans le cas général, les temps de réponse de CPTEST sont acceptables pour faire du test et de la mise-au-point des programmes à contraintes. Un des objectifs de nos expérimentations est de montrer que tester un programme à contraintes est moins coûteux que le résoudre. Le fait de trouver et corriger une faute sur des instances faciles permet de ne plus avoir cette faute sur les instances difficiles du problème.

Résultats de Test sous $conf_{bounds}$ et $conf_{best}$

Pour obtenir des résultats plus significatifs, nous avons fait croître la difficulté du problème en augmentant la longueur de la règle.

Le tableau 9.2 présente les résultats de CPTEST (phase de test) sous $conf_{bounds}$ et $conf_{best}$ des quatre premiers CPUT. Nous avons pris comme intervalle de coût de la fonction objectif $[l, u] = [50, 100]$ sachant que la règle minimale pour $m = 8$ est de taille 34.

CPTEST arrive à détecter les fautes de type ϕ^+ pour les différents cas. Prenons le cas de CPUT3, sans un intervalle $[l, u]$, CPTEST détecte la faute après $\simeq 7min$ de temps de calcul. Sous l'intervalle $[50, 100]$, CPTEST détecte la faute rapidement après $7s$. Ici, la fonction objectif est relâchée, ce qui donne des temps de résolution réduits.

Tester les programmes à contraintes sous $conf_{bounds}$ permet également d'attaquer des instances difficiles du problème. La figure 9.1 donne un comparatif de temps entre la résolution des instances (de 1 à 21) et le test de CPUT3 sous $conf_{bounds}$. Á partir de l'instance $m = 9$, résoudre le CPUT3 devient difficile alors que CPTEST sous $conf_{bounds}$ arrive à trouver des non-conformités pour des instances supérieures à 18.

9.3. n-reines

Le problème des n-reines est un problème classique en PPC.

La partie **(A)** de la figure 6.1 présente le Modèle-Oracle des n-reines. Nous avons appliqué une suite de raffinements $\mathcal{R}^n$ pour avoir un CPUT avec de nouvelles structures de

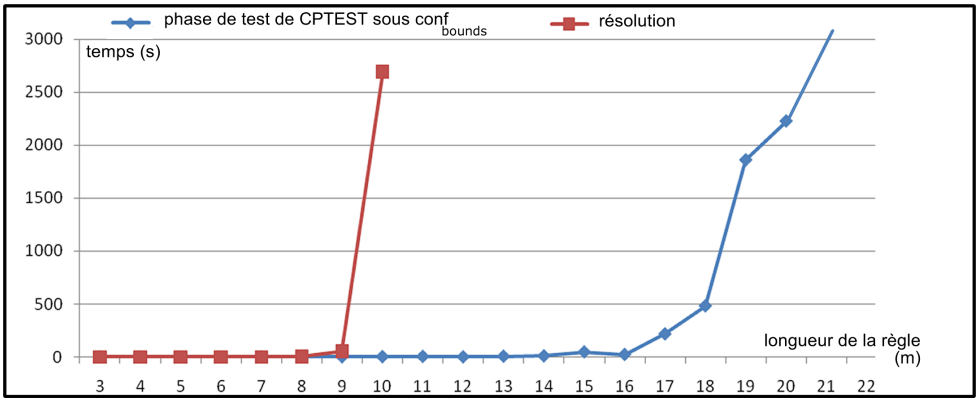


FIGURE 9.1.: Comparaison des temps de Test et de résolution de CPUT3.

données qui représentent les diagonales descendantes et ascendantes, des contraintes qui cassent la symétrie, des contraintes redondantes, une contrainte globale, etc.

En prenant cette suite de raffinements $\mathcal{R}^n$, nous avons introduit des fautes ϕ afin d'obtenir sept CPUT contenant des fautes (i.e., annexe).

Prenons le CPUT3 et CPUT5 du tableau 9.1 :

Test

Le CPUT3 est constitué de 12 ensembles différents de contraintes, une instance de 8-reines génère 48 variables et 236 contraintes. La faute ϕ injectée dans CPUT3 représente une mauvaise formulation de la contrainte `cc11` qui réduit son ensemble de solutions à vide. Soumettre CPUT3 à CPTEST pour une phase de test sous $conf_{one}$ permet de détecter une faute de type ϕ^* en moins de $8s$:

$$\texttt{nc}(\phi^*,\texttt{8},-)$$

Sous une conformité $conf_{all}$, CPTEST retourne en moins d'une seconde la non-conformité suivante :

$$\texttt{nc}(\phi^-,\texttt{8},\texttt{q4}) \quad \text{avec } \texttt{q4=[7 3 1 6 8 5 2 4]}$$

La solution `q4` représente une solution des 8-reines qui n'est pas une solution acceptable par CPUT3. Ceci dit, la faute est de type ϕ^-.

Le CPUT5 est constitué de six ensembles de contraintes, l'instance de 8-reines nous donne 24 variables et 50 contraintes. Nous avons introduit une faute ϕ dans la formulation de la contrainte `cc4`. CPTEST détecte une faute ϕ^+ sous $conf_{one}$ et une faute ϕ^- sous $conf_{all}$:

$$\texttt{nc}(\phi^+,\texttt{8},\texttt{q1}) \text{ avec } \texttt{q1=[8 7 6 5 4 3 2 1]}$$
$$\texttt{nc}(\phi^-,\texttt{8},\texttt{q5}) \text{ avec } \texttt{q5= [5 7 1 3 8 6 4 2]}$$

La solution `q1` n'est pas une solution valide des 8-reines où les reines sont positionnées sur une même diagonale descendante. Cette non-conformité affirme la présence de faute de type ϕ^+ dans CPUT5.

La solution `q5` affirme la présence de ϕ^- dans le CPUT5 où `q5` représente une solution du Modèle-Oracle qui n'est pas une solution de CPUT5.

Localisation et correction

La phase de localisation de CPTEST retourne `cc11` comme *SuspiciousSet* pour CPUT3 en moins de $9s$. En effet, la faute ϕ^* a été introduite dans `cc11`. Pour corriger cette faute, CPTEST propose d'enlever cette contrainte de CPUT3 pour rétablir la conformité avec le Modèle-Oracle.

Pour le CPUT5, la phase de localisation retourne également la contrainte fautive comme suspect `cc4`. La phase de correction propose la correction suivante :

$$CorrectionSet = \{(\texttt{cc4}, \texttt{c2} :: \texttt{28ct})\}$$

Ici, CPTEST propose de remplacer l'ensemble `cc4` par 28 contraintes élémentaires pour corriger l'instance $n = 8$ du CPUT5. Les 28 contraintes sont un sous-ensemble des 29 contraintes élémentaires qu'encapsule `c2` dans le Modèle-Oracle. Dès lors, remplacer `cc4` par `c2` corrige toute instance du CPUT5.

9.4. Golfeurs sociables

Le problème des golfeurs sociables est l'un des problèmes les plus difficiles de la CSPLib (voir `prob010`). Nous avons m joueurs de golf, n semaines et k groupes de joueurs de taille l. Les quatre paramètres (m, n, k, l) définissent l'ensemble des instances du problème. L'objectif est de trouver un calendrier ou un planning pour tous les joueurs durant les n semaines de façon à ne pas avoir deux joueurs dans le même groupe plus d'une fois. Ce problème présente un grand nombre de symétries, des symétries sur les groupes, les joueurs, les semaines, etc.

Nous avons altéré une suite de raffinements $\mathcal{R}^n$ avec de mauvais raffinements incluant des fautes ϕ. Ceci nous a permis d'avoir cinq différents CPUT qu'on retrouve en annexe.

Le tableau 9.1 présente des résultats de test et de mise-au-point des différents CPUT. Nous avons choisi une instance du problème, notée `sI`, de 4 semaines et 3 groupes de 3 joueurs.

Test

Prenons CPUT2 et CPUT4 qui sont constitués de 5 ensembles de contraintes. L'instance `sI` nous donne 36 variables et 2 738 contraintes pour chaque CPUT. Une faute

étant introduite, respectivement, en `cc2` et `cc4`. Nous avons soumis les deux CPUT à CPTEST pour une phase de test.

CPTEST retourne une non-conformité pour CPUT2 sous $conf_{one}$ et $conf_{all}$ en moins d'une seconde :

$$\texttt{nc}(\phi^+,\texttt{sI},\texttt{s1})$$
$$\texttt{nc}(\phi^-,\texttt{sI},\texttt{s2})$$

La solution `s1` représente une solution de CPUT2 qui n'est pas une solution acceptable par le Modèle-Oracle. Ceci révèle l'existence d'une faute de type ϕ^+ dans CPUT2. De plus, `s2` représente une solution correcte de l'instance des golfeurs sociables qui ne peut être retournée par CPUT2. Dès lors, la faute ϕ^- sous $conf_{all}$ est détectée. Notre CPUT2 inclut des contraintes qui cassent certaines symétries du problème, ce qui introduit la plupart du temps des fautes de type ϕ^-.

La faute introduite en `cc4` de CPUT4 a réduit l'ensemble de ses solutions à vide. CPTEST arrive à détecter cette faute ϕ^* en $10s$ de temps de calcul :

$$\texttt{nc}(\phi^*,\texttt{sI},-)$$

Localisation et correction

La phase de mise-au-point de CPUT2 retourne un *SuspiciousSet* vide pour dire que le CPUT2 à besoin de renforcement au niveau des contraintes pour ne plus accepter de solutions erronées. Pour cela, CPTEST propose l'ajout de 75 contraintes élémentaires au CPUT2 pour corriger la faute ϕ^+, le temps de calcul du *CorrectionSet* étant de $18.32s$.

La localisation de faute dans CPUT4 retourne en moins de $4s$:

$$SuspiciousSet = \{\ \texttt{cc4},\ \texttt{cc5}\}$$

Les deux contraintes `cc4` et `cc5` sont considérées comme suspectes par CPTEST. La contrainte `cc4` où la faute a été introduite est la suivante :

```
forall(g in 1..golfers)
   assign[g,1]==((g-1) div groupSize) + 1;
```

Cette contrainte casse les symétries sur les joueurs et sur les groupes : le premier joueur prend une place dans le premier groupe et ainsi de suite jusqu'à ce que tous les groupes soient complets.
La contrainte `cc5` casse les symétries sur les groupes et sur les semaines : pour la première semaine, le premier groupe joue en premier, le second en deuxième et ainsi de suite.

```
forall(g in 1..golfers : g <= groupSize)
   assign[g,2]==g;
```

La faute introduite en `cc4` concerne la symétrie sur les groupes qu'on retrouve également en `cc5` ce qui amène CPTEST à renvoyer `cc5` comme contrainte suspecte.

La phase de correction retourne le *CorrectionSet* suivant après $1min$ et $7s$ de calcul :

$$CorrectionSet = \{(\texttt{cc4, c1|c2 ::58ct}), (\texttt{cc5, c1|c2 ::58ct})\}$$

Deux corrections possibles sont proposées par CPTEST. La première est de remplacer `cc4` par 58 contraintes élémentaires tirées de `c1` et `c2` du Modèle-Oracle qui totalisent chacune 121 et 2 593 contraintes élémentaires. La deuxième correction proposée étant de remplacer `cc5` par un autre ensemble de 58 contraintes élémentaires.

9.5. Ordonnancement de véhicules

Le problème d'ordonnancement de véhicules (Parrello & Kabat, 1986) illustre plusieurs caractéristiques intéressantes de la PPC incluant le paramétrage étendu du programme, l'utilisation de contraintes redondantes, de contraintes globales et des contraintes qui cassent des symétries ainsi que l'utilisation de structures de données spécialisées.

Ce problème revient alors à trouver une position pour chaque véhicule dans une chaîne de montage. Cette chaîne de véhicules passe par des installateurs d'option qui ont des contraintes de capacité. Prenons la figure 9.2, par exemple l'installateur B peut traiter la demande en *option*2 de 3 véhicules sur 4.

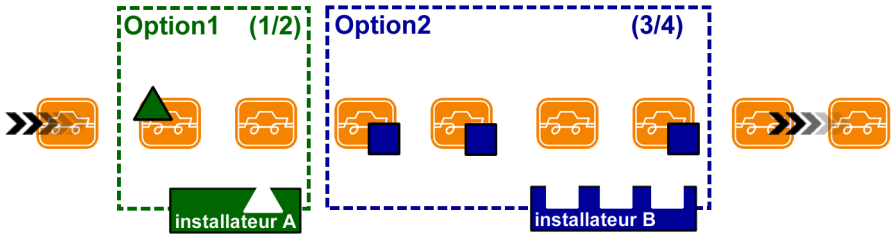


FIGURE 9.2.: Chaine de montage des véhicules avec deux installateurs d'options.

En annexe, on retrouve le Modèle-Oracle et les 7 CPUT après des injections de fautes ϕ.
Parmi les raffinements que nous avons appliqué s'y trouve l'ajout de contraintes redondantes : si l'installateur de l'option `o` est de capacité `l` véhicules sur une sous-chaîne de `u` véhicules, ceci implique que la dernière sous-chaîne de taille `u` ne contiendra que `l` véhicules demandant l'option `o`, la dernière sous-chaînes de taille $(2 \times \texttt{u})$ ne contiendra que $(2 \times \texttt{l})$ et ainsi de suite. Nous avons également pris comme raffinement l'ajout de la contrainte globale `pack` (voir exemple 13).

Nous avons pris une instance du problème, notée `cI`, avec 5 options, 6 classes et 10 véhicules. Chaque classe de véhicules représente l'ensemble des options requises, par exemple la classe 1 a besoin de l'option 1, 3 et 4.

Test

Prenons le CPUT2 qui est formé de 3 ensembles de contraintes, l'instance `cI` génère 60 variables et 139 contraintes. Nous avons injecté une faute ϕ dans `cc3`.

$$\texttt{nc}(\phi^+,\texttt{cI},\texttt{o2}) \text{ avec } \texttt{o2=[4 1 6 4 3 5 3 6 2 5]}$$

Cette non-conformité est retournée par CPTEST après $6s$ de calcul. Une faute ϕ^+ est détectée sous une conformité $conf_{one}$ où `o2` représente une chaîne de montage non valide : la sous-chaîne `[...1 6...]` comporte deux véhicules de classe 1 et 6 qui demande l'option 1 sachant que l'installateur de l'option 1 a une capacité de 1/2.
Le test sous $conf_{all}$ détecte une faute de type ϕ^+ :

$$\texttt{nc}(\phi^+,\texttt{cI},\texttt{o3}) \text{ avec } \texttt{o3=[4 6 2 5 3 6 1 3 5 4]}$$

La chaîne $o3$ est également une chaîne non valide qui peut être retournée par CPUT2. Le temps que prend CPTEST pour retourner cette non-conformité est de $17.95s$. Ici, CPTEST prend plus de temps, car il essaie en premier lieu de détecter des fautes de type ϕ^- , autrement il passe à ϕ^* puis ϕ^+.

Prenons maintenant CPUT7 qui est constitué de 6 ensembles de contraintes, l'instance `cI` nous donne 66 variables et 185 contraintes. Nous avons introduit une faute dans `cc5` qui représente une contrainte qui casse des symétries :

$$\texttt{nc}(\phi^*,\texttt{cI},-)$$

Cette non-conformité est retournée sous $conf_{one}$ pour dire que la faute ϕ^* qui a été injectée dans CPUT7 l'a rendu insatisfiable.

Le test sous $conf_{all}$ retourne la non-conformité suivante :

$$\texttt{nc}(\phi^-,\texttt{cI},\texttt{o6}) \text{ avec } \texttt{o6=[5 4 3 5 4 6 2 6 3 1]}$$

La chaîne `o6` représente ici une chaîne valide qui ne peut pas être retournée par CPUT7 (faute ϕ^-).

Localisation et correction

Concernant le CPUT2, CPTEST retourne un *SuspiciousSet* vide et propose comme *CorrectionSet* l'ajout de 32 contraintes élémentaires pour rétablir la conformité avec le Modèle-Oracle.

La phase de localisation pour CPUT7 retourne en $5s$:

$$SuspiciousSet = \{\ \texttt{cc5, cc6}\}$$

Les deux ensembles de contraintes `cc5` et `cc6` sont alors susceptibles de contenir la faute ϕ^*. La correction de CPTEST prend $15s$ pour retourner le *CorrectionSet* suivant :

$$CorrectionSet = \{\ \texttt{(cc5,}-\texttt{), (cc6,}-\texttt{)}\}$$

Ici, deux corrections sont possibles pour rétablir la conformité. La première est la suppression de `cc5`, la deuxième étant la suppression de `cc6`.

9.6. Conclusion

Dans ce chapitre, nous avons présenté une procédure d'expérimentation qui consiste à : sélectionner un problème PPC ; écrire son Modèle-Oracle sous OPL ; appliquer différentes suites de raffinements incluant des fautes significatives ; obtenir un ensemble de CPUT ; soumettre les CPUTs à CPTEST pour une phase de test, de localisation et de correction de fautes. Les problèmes PPC sont soigneusement choisis : les règles de Golomb, les n-reines, les golfeurs sociables et ordonnancement de véhicules, où chaque problème présente des caractéristiques intéressantes de modélisation PPC. Notre étude expérimentale nous a permis d'atteindre différents objectifs que nous nous sommes fixés : la validation de nos approches de test et de mise-au-point présentées dans la thèse ; la validation de notre implémentation de l'outil CPTEST ; une étude comparative du test sous $conf_{bounds}$ et $conf_{best}$; une étude du temps nécessaire pour une phase de localisation et/ou de correction automatique.

En somme, les résultats présentés sont encourageants. Les fautes injectées dans les 26 différents CPUT sont détectées à l'aide de non-conformités retournées par CPTEST dans des temps variant entre 11 ms et 35 s. La localisation des fautes par CPTEST a localisé la contrainte fautive sur 14 cas des 26 CPUT alors que le reste des cas sont des CPUT sous-contraints. Les temps pour la localisation sont intéressants qui ne dépassent pas les 14 s. Quant à la correction automatique, une correction est calculée pour chaque CPUT sachant que les temps peuvent atteindre les 2 min tout en soulignant un cas où le temps atteint les 30 min.

Cinquième partie .

Conclusions et Perspectives

10. Conclusions et perspectives

Dans ce dernier chapitre, nous rappelons d'abord les contributions de notre thèse, puis nous présentons les conclusions générales de notre travail. Enfin, nous proposons quelques perspectives.

10.1. Rappel des contributions

Dans cette thèse, nous présentons trois contributions principales.

Notre première contribution est la définition d'une méthodologie de test des programmes à contraintes (Lazaar *et al.*, 2009; Lazaar *et al.*, 2010b). Cette dernière s'appuie sur la définition de différentes relations de conformité selon la classe de problèmes abordés :

$conf_{one}$ cette relation est dédiée aux problèmes de satisfaction de contraintes qui cherchent une seule solution,

$conf_{all}$ pour les problèmes qui cherchent toutes les solutions,

$conf_{bounds}$ cette relation concerne les problèmes d'optimisation et définie une conformité dans un intervalle donné $[l, u]$,

$conf_{best}$ une autre relation pour les problèmes d'optimisation où la conformité est définie de façon globale sans intervalle donné.

Ces relations sont définies entre un premier modèle à contraintes qui représente la spécification du problème (Modèle-Oracle) et le programme à contraintes qui est sous Test (CPUT).

Notre méthodologie de test s'appuie également sur un modèle de fautes spécifiques à la PPC. Ce modèle est défini à travers différents raffinements $\mathcal{R}$ possibles apportés au CPUT par des spécialistes de la modélisation à contraintes avec des ajouts de contraintes globales et/ou de contraintes redondantes, des reformulations, etc. Ces raffinements peuvent introduire différents types de fautes :

faute positive ϕ^+ qui augmente l'ensemble des solutions du CPUT,

faute négative ϕ^- qui enlève des solutions valides de l'ensemble de solutions du CPUT,

faute zéro ϕ^* qui réduit l'ensemble de solutions du CPUT à vide.

Dans cette thèse, nous avons proposé différents processus de test selon les problèmes abordés. Au cœur de ces différents processus, on retrouve un même générateur de données de test basé sur la négation de contraintes (i.e., *one_ negated*). Un appel de ce générateur, par exemple *one_ negated(A,B)*, renvoie une solution du système A qui n'est pas

une solution de B (i.e., une non-conformité entre A et B). Si une non-conformité entre le Modèle-Oracle et le CPUT existe, ce générateur permet de la renvoyer (*correction*). Autrement, il ne retourne pas de fausses alarmes (*complétude*).

Notre seconde contribution concerne la mise-au-point des programmes à contraintes, à savoir la localisation de fautes (Lazaar *et al.*, 2010a) et la correction automatique (Lazaar *et al.*, 2011a).

Nous avons proposé une nouvelle approche de localisation de fautes des programmes à contraintes qui, contrairement aux approches existantes, ne fait pas appel à l'analyse de trace. Cette approche prend le cadre de test, défini dans cette thèse, comme support et se base sur la définition de contraintes suspectes. De ce fait, nous avons proposé un processus de localisation sous l'hypothèse d'avoir une seule contrainte fautive dans le CPUT. Ce processus, nommé *locate*, permet de retourner un ensemble de contraintes suspectes (i.e., *SuspiciousSet*) dont la contrainte fautive (preuve de *correction* à l'appui). Par ailleurs, nous avons levé l'hypothèse d'avoir une seule contrainte en proposant le processus ϕ-*locate*. Ce processus est une généralisation de *locate* avec une complexité exponentielle, dans le cas général, due au nombre de sous-ensembles possibles du CPUT.

La correction automatique est un sujet émergent. Notre travail peut être considéré comme une première approche de correction automatique des programmes à contraintes. Cette approche se base sur la localisation de fautes et le Modèle-Oracle pour le calcul des corrections possibles. Pour toutes les contraintes suspectes, le processus de correction ϕ-*correction* calcule une correction possible qui permet de rétablir la conformité entre le Modèle-Oracle et le CPUT.

Notre troisième contribution représente le développement de l'outil CPTEST. Cet outil permet de tester, localiser les fautes et corriger automatiquement des programmes écrits en OPL. CPTEST est écrit en Java avec 25 000 lignes de code. Il comprend un analyseur syntaxique complet du langage OPL et des modules implémentant *one_ negated*, les différents processus de test, ϕ-*locate* et ϕ-correction.

Á l'aide de cet outil, nous avons effectué une étude expérimentale sur différents problèmes de la PPC (i.e., règles de Golomb, n-reines, golfeurs sociables, ordonnancement de véhicules). Cet étude avait comme objectif : la détection des différentes fautes possibles ; une étude comparative entre $conf_{bounds}$ et $conf_{best}$ pour les problèmes d'optimisation ; une analyse des temps de localisation et de correction ; etc.

Nous avons soumis à CPTEST au total 26 différents CPUT qui contiennent des fautes injectées. CPTEST est parvenu à détecter la totalité des fautes dans des temps qui varient entre 11 ms et 35 s, des temps acceptables pour une phase de test.

La phase de localisation de CPTEST localise les fautes de façon exacte dans 14 cas sur 26. Dans 2 cas de figures, CPTEST retourne un *SuspiciousSet* de deux contraintes incluant la contrainte fautive. Le reste des cas (i.e., 10 cas) sont des cas où les CPUTs sont sous-contraints et CPTEST retourne un ensemble vide. Concernant les temps d'exécution pour la localisation des fautes, CPTEST ne dépasse pas les 14 s ce qui représente un temps raisonnable pour une localisation de faute.

La correction automatique assure quant à elle une correction pour chaque faute intro-

duite, les temps ici varient entre 4 s et 2 min. Nous soulignons également un temps qui atteint les 30 min pour que CPTEST arrive à corriger un des CPUTs.

10.2. Les conclusions

Les théories de test existantes sont dédiées aux programmes conventionnels et semblent être inopérantes pour capter les spécificités de test des programmes à contraintes et cela pour différentes raisons : pas de notion de séquentialité en PPC où le flot de contrôle est guidé par les contraintes ; un processus de développement complètement différent où en PPC on part d'un modèle déclaratif initial qui subit des raffinements successifs ; les modèles de fautes sont différents ; etc.

Notre premier travail consistait à poser les jalons d'une théorie de test pour les programmes à contraintes avec un oracle de test (i.e., Modèle-Oracle) ; un programme sous test (i.e., CPUT) ; des relations de conformités ; des données de test ; un processus de test ; un modèle de faute.
Cette théorie définit la référence de test comme étant le premier modèle à contraintes fidèle à la spécification du problème. Au chapitre des désavantages, il faut mentionner l'impérieuse nécessité de disposer de Modèle-Oracles de qualité, c'est-à-dire de modèles qui résolvent fidèlement le problème initial. De plus, le processus de test qu'on propose fait appel à de la réfutation avec la négation de contraintes.

Notre démarche de localisation répond en premier lieu, comme toute autre approche, à l'hypothèse d'avoir une seule contrainte fautive. Cette hypothèse se traduit en une seule instruction fautive dans les programmes conventionnels. Sans cette hypothèse, notre approche reste valide mais avec une généralisation qui passe à une complexité exponentielle.

Le travail de correction automatique s'inscrit comme étant une première approche de correction des programmes à contraintes. Ce travail est basé sur les résultats de la localisation des fautes et sur le Modèle-Oracle dans le calcul automatique des corrections.

La réalisation de l'outil CPTEST et son succès opérationnel nous indique que l'approche qui consiste à valider automatiquement des programmes à contraintes en OPL, lorsqu'un Modèle-Oracle est disponible, est viable et pourrait être généralisée sur d'autres langages PPC.

10.3. Les perspectives

Les travaux effectués durant cette thèse ouvrent le chemin à des perspectives dont chacune peut faire l'objet d'un programme de recherche. Ces perspectives s'inscrivent particulièrement dans le développement, la vérification et la validation en PPC.

Perspectives à court terme

Notre approche de localisation de fautes complexes ϕ est d'une complexité exponentielle dans le cas général (algorithme *ϕ-locate*). Une perspective à court terme serait d'atteindre une complexité inférieure. Pour cela, nous évoquons la piste des algorithmes dichotomiques. L'algorithme QuickXplain (Junker, 2004) permet de trouver une explication minimale à une faute de type ϕ^* sous une complexité $\mathcal{O}(2k * log(n/k) + 2k)$ avec n le nombre de contraintes et k la taille de l'explication (Junker, 2004). La piste à creuser serait de pouvoir étendre ces algorithmes pour une localisation de faute de type ϕ^- et ϕ^+.

Une autre perspective serait un traitement à posteriori des corrections proposées. Prenons par exemple une correction qui est formée de plusieurs contraintes de différences. Il est intéressant d'avoir une règle d'inférence qui prend la correction en question et génère automatiquement des contraintes globales comme un *allDifferent*.

L'expérience de CPTEST avec les programmes OPL semble être fructueuse. Il serait intéressant de revivre cette expérience avec d'autres langages comme Choco ou Gecode qui sont des solveurs boite-blanche. Ici, l'expérience serait plus ouverte avec la possibilité d'expérimenter différents critères de test.

Négation et test des contraintes globales

Notre approche de test fait appel à la réfutation où, dans le cas général, on cherche une solution d'un système $\mathcal{A}$ qui n'est pas une solution de $\mathcal{B}$. Le générateur *one_negated* est donc basé sur la négation de contraintes. Ce générateur est amené des fois à nier des contraintes globales. Dans ce cas, notre démarche reste naïve avec un traitement à posteriori qui permet de faire une réécriture de la contrainte globale, le résultat étant facile à nier. Cette négation est sémantiquement correcte car l'ensemble des solutions est conservé, mais d'un point de vue opérationnel, elle reste coûteuse.

Une piste intéressante serait la définition d'une contrainte générique Neg qui permet de prendre toute contrainte globale et de produire un filtrage dédié à sa version niée. Un tel travail demande une représentation générique des contraintes globales. Ceci dit, les représentations génériques des contraintes globales sont des sujets émergents et d'actualité en PPC où on note plus de 300 contraintes globales (Beldiceanu *et al.*, n.d.) présentées avec des automates d'états finis (DFA), ou encore des représentations MDD (*Multivalued Decision Diagram*) (Hoda *et al.*, 2010), des automates à piles, des grammaires (Kadioglu & Sellmann, 2010), etc.

Une contrainte générique comme *regular* (Pesant, n.d.) permet de faire un filtrage sur toute contrainte globale qui peut être représentée par un DFA. En prenant une telle représentation, on peut alors définir aisément Neg. Notre travail préliminaire sur la négation automatique de contraintes trouve un écho particulier dans les travaux de N. Beldiceanu sur la négation des contraintes globales (Beldiceanu *et al.*, n.d.). En effet, ces travaux montrent qu'une telle approche est viable et pourrait être généralisée à n'importe quelle

combinaison de contraintes. Notre perspective s'inscrit dans ce contexte, avec néanmoins quelques différences. Nous envisageons de fonder le calcul de la négation sur l'utilisation de la contrainte générique *regular* (Lazaar *et al.*, 2011b).

Définir une contrainte globale, avec le plus puissant filtrage possible, est un travail laborieux qui demande une bonne expertise dans la matière. Une autre perspective qui semble être intéressante serait le test des contraintes globales. De plus, notre cadre de test peut servir de support à un outil de test dédié aux contraintes globales où le Modèle-Oracle serait une spécification déclarative et le CPUT la contrainte globale. Ceci permet de tester le comportement d'un filtrage d'une contrainte globale vis-à-vis de sa spécification déclarative.

Critères de Test

Le cadre de test que nous proposons considère le solveur de contraintes comme une boite noire où la détection de fautes se fait sans aucune information concernant la résolution des contraintes. Un travail qui semble être intéressant serait de pouvoir mettre la main sur le solveur et expérimenter différents critères de test. L'utilisation de solveur boite blanche nous donne la possibilité d'instrumenter le code et de faire une analyse de traces. Ces analyses nous donnent la possibilité de proposer différents critères de test. Par exemple, un critère de couverture des réveils de contraintes dans le *store* ou bien, le degré de participation de telle ou telle contrainte dans une résolution donnée, etc.

Test de mutation

Notre procédure expérimentale fait appel à l'injection de faute. Ce procédé est connu dans le monde du test par *Test de Mutation* : on sélectionne une modification possible du code (i.e., un opérateur de mutation) qu'on applique sur le programme, le programme résultant est appelé *mutant*. Si une suite de test détecte le changement alors on dit que le *mutant* est *tué*. L'objectif du test est alors de tuer un maximum de *mutants*.

Cependant, le nombre de changements possibles est énorme ; il est impossible de produire des mutants qui représentent toute faute possible. De ce fait, la mutation vise seulement un sous-ensemble de ces fautes avec espoir d'avoir des mutants qui peuvent simuler toute faute possible. Cette théorie est basée sur deux hypothèses : l'Hypothèse du *Programmeur Compétent* qui va développer un programme très proche de la version correcte (DeMillo *et al.*, 1978; Acree *et al.*, 1979), et l'hypothèse sur l'*effet de couplage* (DeMillo *et al.*, 1978) pour dire qu'une faute peut être complexe et due, non pas à un seul changement, mais à plusieurs à la fois.

Une perspective qui s'inscrit dans cette voie est de construire un générateur automatique de mutants sous les deux hypothèses précédentes. Pour cela, il faudra définir des opérateurs de mutation dédiés aux programmes à contraintes. Par exemple, un opérateur de mutation possible serait une relaxation de contrainte.

Apprentissage par renforcement

Dans cette thèse, nous avons eu l'occasion de détailler les raffinements possibles en PPC. Une perspective voisine serait de transformer ce processus de raffinement en un problème d'apprentissage automatique, plus précisément, un problème d'apprentissage par renforcement (Mitchell, 1997). Le but est alors d'apprendre, à partir de différentes expériences, les raffinements à appliquer, de façon à optimiser le temps de résolution du programme. Il existe différents travaux qui, assemblés, peuvent être une bonne base pour cette perspective. On peut noter les travaux de Puget concernant la détection automatique des symétries (Puget, 2005) ; ou encore le travail récent de Beldiceanu et Simonis (Beldiceanu & Simonis, 2011) qui, à partir de solutions et/ou non-solutions, propose une liste ordonnée de contraintes globales à utiliser.

Sixième partie .

Annexes

Annexe A : Négation des contraintes globales

NEGATION **for free!**

Nadjib Lazaar
INRIA Rennes Bretagne Atlantique,
Campus Beaulieu, 35042 Rennes, France
Email: Nadjib.Lazaar@inria.fr

Nourredine Aribi
Université d'Oran Es-Senia, Lab. LITIO,
B.P. 1524 EL-M'Naouar, 31000 Oran, Algerie
Email: Aribi_Noureddine@yahoo.fr

Arnaud Gotlieb
INRIA Rennes Bretagne Atlantique,
Campus Beaulieu, 35042 Rennes, France
Email: Arnaud.Gotlieb@inria.fr

Yahia Lebbah
Université d'Oran Es-Senia, Lab. LITIO,
B.P. 1524 EL-M'Naouar, 31000 Oran, Algerie
Email: ylebbah@gmail.com

***Abstract*—**
[1]Global constraint design is a key success of CP for solving hard combinatorial problems. Many works suggest that automaton-based definitions and filtering make easier the design of new global constraints. In this paper, from such a design, we present an approach that gives an automaton-based definition of the NEGATION of a global constraint... for free! For a given global constraint C, the idea lies in giving operators for computing an automaton that recognizes only tuples that are not solution of C, and use the REGULAR global constraint to automatically reason on this automaton. We implemented this approach for automaton-based global constraints, including `global_contiguity` and $\leq_{\texttt{lex}}$ constraints, and got experimental results that show that their automatically computed negation is highly competitive with more syntactic transformations.

***Keywords*-Global Constraints; Negation; Deterministic Finite Automaton.**

I. INTRODUCTION

Modern constraint programming languages aim at making easy problems formulation and solving. One of the key success of CP is global constraints design. Since its introduction in [6], automaton-based definition of global constraint has grown and is now recognized as a mainstream technique. Carlsson and Beldiceanu proposed in [6], [3] to use automata representation and reformulation for designing new global constraints from constraint checkers. Pesant proposed in [14] a generic global constraint, the REGULAR global constraint which holds if a fixed-length sequence of finite-domain variables represents a word of a given regular language. In another context, Andersen et al. [1] proposed to use the multivalued decision diagram structure (MDD) to replace the domain store where constraints have an MDD-Based presentation.

As suggested by the above mentioned works, building new global constraints is often required to address challenging combinatorial problems. Obviously, having *logical negation* in the tool-box would be interesting to facilitate this process. In our previous works related to program verification [10], [11], [12], we faced the problem of negating existing global constraints. Our (naive) solution involved simple syntactic transformations of the original constraints to easily compute its negation. For example, the negation of: `inverse(all[R](i in R) g[i], all[S](j in S) f[j]);` in OPL was easily expressed by:
`or(i in S) g[f[i]]! = i; or(j in R) f[g[j]]! = j.`
As possible, the syntactic transformations can exploit also the existing global constraints to express the negation form of a given constraint. For example, the negation of an `atLeast` constraint can be expressed using the `atMost` and vice-versa, `GCC` by `atLeast` and `atMost`, `allDifferent` by a disjunction of `GCC`, etc.

However, those syntactic transformations did not capture the essence of *logical negation* and did not filter constraints in a sufficient and consistent way.

In (Constraint) Logic Programming, *negation–as–failure* has been the traditional approach to deal with negation in the general framework of the Clark completion. However, it is well known that *negation–as–failure* corresponds only to logical negation on ground instances. *Constructive negation*, as proposed by Stuckey in [18], presents a sound and complete operational model of negation in the Herbrand Universe. An interesting implementation of this operator in the constraint concurrency model of Oz has been proposed by Schulte in [16]. Constructive constraint negation is general as it can handle any constraint but is also ineffective in terms of filtering. Indeed, no dedicated filtering algorithms is available for the negation of the constraint and thus, these operators are usually not useful to prune the search space. More recently, constraint negation has been considered in the more general context of *logical connectives* [5], [2], [13]. However, in these works, negation is proposed for constraints defined in extension and cannot be applied to global constraints that capture complex relations among a set of variables.

[1]This work is supported by INRIA-DGRSDT (France, Algeria).

In this paper, we present an approach that takes the automaton-based design of a global constraint as input and automatically returns an automaton-based definition of the NEGATION of this global constraint. For a given global constraint C, the idea is first to give operators for computing a Deterministic Finite Automaton (DFA) that recognizes the tuples that are not solution of C ; and second to use the REGULAR global constraint [14] to automatically derive filtering rules for this new automaton.

One can choose an MDD-based design and just swap end-states to get the negation form. But this approach has two limitations: First, it is expensive, because for a given constraint, the generated MDD contains only the feasible paths. To do such negation, the infeasible portion has to be generated as well as the feasible one. Second, for efficiency reasons, MDD-based global constraints are usually represented by fixed-width MDDs [8] which are correct but produce imprecise relaxations. This representation may include assignments violating the constraint, thus, building the negation of a given global constraint by swapping the end-states between accepting and non-accepting states in a fixed-width MDD, may be unsound. On the contrary, we will see that using a folded DFA for building the negation is guaranteed to be sound.

This paper contains global constraint examples that were automatically negated through our approach, including the negation of the `global_contiguity` and $\leq_{\texttt{lex}}$ constraints. We implemented our approach in Gecode, where a good implementation of REGULAR is available, and got experimental results on these global constraints that show our negation is highly competitive with more syntactic transformations.

The paper is organized as follows. The next section describes the process of constructing the automaton of the negated constraint and using REGULAR. Section III illustrates the approach on two constraints: `global_contiguity`, and $\leq_{\texttt{lex}}$. The experimentations are described in section IV. Section V concludes the paper.

II. NEGATION ON DFA-BASED GLOBAL CONSTRAINTS

In this section, we present an efficient method to handle the negation of the automaton-based design global constraints. This kind of global constraint has behind a specific DFA (Deterministic Finite Automaton) as a checker of ground instances. We summarize the approach in two points:
- From the DFA of a given global constraint `C`, we generate, using an automatic process, the complement which is the DFA of the negation form ($\neg$`C`).
- We derive the filtering algorithm using the REGULAR constraint.

A. Notations

A Deterministic Finite Automata (DFA) $\mathcal{A}$ of a given constraint C is defined as a 7-tuple, $(X, E, \Psi, \Sigma, \delta, e_0, F)$, consisting of:

- a sequence of finite-domain variables X (i.e., signature of the constraint C),
- a finite set E of states e_i ,
- a finite set of labeled states Ψ s.t. $source(e_0)$: the starting state, $node(e_i)$: intermediate state, $sink(e_i)$: sink state[2], $final(e_i)$: final state,
- a finite alphabet Σ,
- a transition function δ, (e_i, s, e_j) is a transition where $e_i, e_j \in E, s \in \Sigma \cup \{\$\}$[3],
- the start state e_0 where $source(e_0) \in \Psi$,
- a set of final states $F \subseteq E$ where $\forall e_i \in F : final(e_i) \in \Psi$.

We stress the fact that any state e_i should have exclusively one of the four possible labels in Ψ. We note $\mathcal{L}$ the language recognized by $\mathcal{A}$ (i.e., $\mathcal{L}(\mathcal{A})$) where $\mathcal{L} \subseteq \Sigma^*$.

B. DFA Complement

To have the complement of a global constraint DFA's we use three operators, namely: *Complete*, *Swap-state* and *Clean-up* operators.

1) A Complete DFA : A deterministic automaton $\mathcal{A}$ of a given constraint C considers essentially the patterns where the constraint is evaluated to the accepted or satisfied states (i.e., *final* states). It does not consider all the possible patterns of the constraint. The patterns that violate the constraint are not considered, because they do not lead to satisfaction states. Thus, to complete an automaton we have to add all possible states in order to be able to consider all the possible patterns of the constraint instance (i.e., $\mathcal{C}(\mathcal{A})$). Formally speaking, the *complete* operator on $\mathcal{A}$ returns an extended automaton $\mathcal{C}(\mathcal{A})$ that takes in account all transitions as well as those leading to a *sink* state.

Definition 1 ($\mathcal{C}(\mathcal{A})$): The complete automaton of $\mathcal{A} = (X, E, \Psi, \Sigma, \delta, e_0, F)$ is $\mathcal{C}(\mathcal{A}) = (X, E', \Psi', \Sigma, \delta', e_0, F)$ s.t.:

- $E' = E \cup \{e_k : \exists s \in \Sigma, e_i \in E \ s.t. \ (e_i, s, e_k) \notin \delta\}$
- $\Psi' = \Psi \cup \{sink(e_k) : e_k \in E' \backslash E\}$
- $\delta' = \delta \cup \{(e_i, s, e_k) : \exists s \in \Sigma, e_i \in E', e_k \notin E\}$

For $|E| = n$ and $|\Sigma| = m$, the *complete* operator adds at most nm states and transitions, and is $\mathcal{O}(nm)$. It is correct where it preserves $\mathcal{L}(\mathcal{A})$ by adding only *sink* states. It is also complete where, for each state of the computed DFA's, there is as outgoing arcs as symbols in Σ.

2) A DFA Swap-state: The *swap-state* $\mathcal{S}$ swaps the *sink* states to *final* and vice-versa.

[2]State e_i is a *sink* state if and only if it is not a *final* state and there are no transitions leading from e_i to another state.

[3]$\$ represents the empty transition.

Definition 2 ($\mathcal{S}(\mathcal{A})$): Let us take the automaton $\mathcal{A} = (X, E, \Psi, \Sigma, \delta, e_0, F)$, a swap-state on $\mathcal{A}$ is $\mathcal{S}(\mathcal{A}) = (X, E, \psi', \Sigma, \delta, e_0, F')$ s.t.:

$$\forall e_i, e_j \in E : \ final(e_i), sink(e_j) \in \Psi \Rightarrow sink(e_i), final(e_j) \in \Psi'$$

It is obvious to say that the *swape-state* operator is correct and complete where all (and only) *sink* (resp. *final*) states are swapped.

3) A DFA *Clean-up:* The *Clean-up* operator on a DFA $\mathcal{A}$, noted $\mathcal{U}(\mathcal{A})$, is the inverse function of the *complete* operator (i.e., $\mathcal{U}(\mathcal{C}(\mathcal{A})) = \mathcal{C}(\mathcal{U}(\mathcal{A})) = \mathcal{A}$) , where the *clean-up* reduces the automaton by removing all transitions leading to a *sink* state.

Definition 3 ($\mathcal{U}(\mathcal{A})$): The clean-up operator of $\mathcal{A} = (X, E, \Psi, \Sigma, \delta, e_0, F)$ is $\mathcal{U}(\mathcal{A}) = (X, E', \Psi', \Sigma, \delta', e_0, F)$ s.t.:

- $E' = E \backslash \{e_k : sink(e_k) \in \Psi\}$
- $\Psi' = \Psi \backslash \{sink(e_k) : e_k \in E\}$
- $\delta' = \delta \setminus \{(e_i, s, e_k) \in E \times \Sigma \times E \ : \ sink(e_k) \in \Psi\}$

The *clean-up* operator preserves $\mathcal{L}(\mathcal{A})$ where it cannot remove a *final* or a *node* state (*correctness*). At the end, the computed DFA's removes all *sink* states and transitions leading to its (*completeness*).

Using the three operators seen before, we get the complement of a given DFA of a global constraint.

Theorem 1: Let $\mathcal{A}$ a DFA. $\bar{\mathcal{A}}$ is the complement s.t.:

$$\bar{\mathcal{A}} = \mathcal{U}(\mathcal{S}(\mathcal{C}(\mathcal{A})))$$

PROOF*:* Let $\mathcal{L}(\mathcal{A})$ (resp. $\mathcal{L}(\mathcal{B})$) a regular language for some DFA $\mathcal{A} = (X, E, \Psi, \Sigma, \delta, e_0, F)$ (resp. $\mathcal{B} = (X, E', \Psi', \Sigma, \delta', e_0, F')$) s.t. $\mathcal{B} = \mathcal{U}(\mathcal{S}(\mathcal{C}(\mathcal{A})))$. $\mathcal{B}$ is the complement of $\mathcal{A}$ iff $\overline{\mathcal{L}(\mathcal{A})} = \mathcal{L}(\mathcal{B})$ (i.e., $\overline{\mathcal{L}(\mathcal{A})} = \Sigma^* - \mathcal{L}(\mathcal{A})$ as stated by [9]).

- $w \in \mathcal{L}(\mathcal{A}) \Rightarrow w \notin \mathcal{L}(\mathcal{B})$: Let $w \in \mathcal{L}(\mathcal{A})$, so $\exists e_i$ s.t.$(e_0, w, e_i) \in \delta^*$ and $final(e_i) \in \Psi$. e_i is a *final* state in $\mathcal{C}(\mathcal{A})$ (**Def.1**) where it is swapped to *sink* state by $\mathcal{S}(\mathcal{C}(\mathcal{A}))$ (**Def.2**). By $\mathcal{U}(\mathcal{S}(\mathcal{C}(\mathcal{A})))$ e_i will be removed as it is *sink* state, so $final(e_i) \notin \Psi'$ (**Def.3**) and $w \notin \mathcal{L}(\mathcal{B})$.

- $w \in \mathcal{L}(\mathcal{B}) \Rightarrow w \notin \mathcal{L}(\mathcal{A})$: In the same way, the inverse is also true.

Property 1: The complement of a regular language is regular [9].
The property guarantees that the complement of a given DFA automaton $\mathcal{A}$ (i.e., the recognized regular language $\mathcal{L}$) is a DFA (i.e., regular language).

Property 2: The complement of a DFA of a given constraint C represents the DFA of the negated form $\neg$C.
A DFA of a given constraint C represents the solution set of the constraint, therefore all instantiations that do not belong to this solution set are recognized by the complement DFA. So, the complement DFA represents the solution set of $\neg$C.

C. Filtering the negation with the REGULAR constraint

Having the automaton is not enough to get a filtering algorithm of the negation of a given global constraint: rules associated to the regular expressions recognized by the automaton have to be considered [6]. While automatic construction of the automaton of the negation is easy, finding filtering rules is difficult, especially when generalized arc-consistency is required. In the general case, as the automaton for the negation is a DFA that can be augmented with counters, the generic global constraint GRAMMAR could be used to automatically derive filtering rules [17], [15]. However, this constraint has exponential cost w.r.t. the states of the automaton. When strings are of fixed length, [7] pointed out an approach where the GRAMMAR constraint is processed with the REGULAR global constraint [14] by transforming the push-down automaton associated to the constrained grammar to a finite-state automaton. Thus, in our approach, we selected the REGULAR global constraint to encode generic filtering rules for the negation of global constraints.

A *regular language membership constraint* is a constraint C on a sequence of finite-domain variables x associated with a DFA $\mathcal{A} = (X, E, \Psi, \Sigma, \delta, e_0, F)$ s.t:

$$regular(\mathtt{x}, \mathcal{A}) = \{\tau \ : \ \tau \ tuple \ of \ \mathtt{x} \ recognized \ by \ \mathcal{A}\}$$

The consistency algorithm of the REGULAR constraint has three main phases. The *forward*, the *backward* and the *maintaining* phases collect states from E that support the pair (x_i, v_i) (i.e., $v_i \in D_{x_i}$).

The *forward* phase unfolds the DFA $\mathcal{A}$ by constructing the corresponding Multivalued Decision Diagram (MDD) which is an acyclic graph by construction. The MDD contains different layers L_i $(L_1, L_2, \ldots, L_n)$. Each layer contains states from E where arcs appear between consecutive layers. The first layer L_1 contains only the start state e_0 $(source(e_0) \in \Psi)$. We unfold the DFA from L_1 to L_n according to the transition function σ.

The *backward* phase removes states and the corresponding incoming arcs from layer L_n to L_1. We start by removing from the last Layer L_n all no *final* states and their incoming arcs. For a layer L_i, we remove all states and their incoming arcs that have no outgoing arcs.

There is also a *maintaining* phase if a domain reduction is provoked by another constraint. Here the MDD needs

to be maintained by removing all arcs corresponding to the removed value. We remove also, for each layer, all unreachable states or those without outgoing arcs.

To show how REGULAR is used in our framework, we illustrate the automatic derivation for the negation of `global_contiguity` and `Lex` in the next section.

III. Case Studies

In this section we take two case studies to illustrate our approach, namely `global_contiguity` and $\leq_{\texttt{lex}}$ constraints. We construct DFA of the negated form and we get the filtering algorithm using the REGULAR constraint.

A. Case study: ¬global_contiguity

The `global_contiguity(var)` [3], [4] is defined on a vector of variables `var`. Each variable `var[i]` can take value in $\{0, 1\}$. The `global_contiguity` constraint holds since the valuation of the sequence of variables `var` contains no more than one group of contiguous 1. For example, if we take a sequence of 10 variables, the sequence 0011110000 is a correct sequence where 0011100110 is not.

The DFA of `global_contiguity` constraint is given in Fig. 1 **part(a)**. It corresponds to:
$\mathcal{A} = (var, \{e_0, e_1, e_2, e_3\}, \Psi, \{0, 1\}, \delta, e_0, \{e_3\})$,
$\Psi = \{source(e_0), node(e_1), node(e_2), final(e_3)\}$,
$\delta = \{(e_0, 0, e_0), (e_0, 1, e_1), (e_0, \$, e_3), (e_1, 1, e_1), (e_1, 0, e_2), (e_1, \$, e_3), (e_2, 0, e_2), (e_2, \$, e_3)\}$.

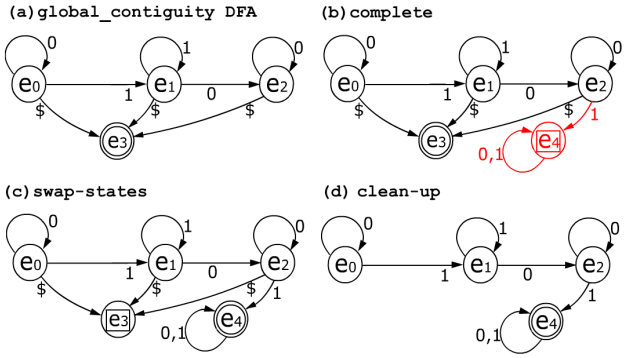


Figure 1. Complement DFA of the `global_contiguity` constraint.

To construct the complement of the DFA shown in Fig.1 **part(a)**, we first call the *complete* operator (Fig.1 **part(b)**) where the *sink* state e_4 is added to complete the automaton. Second, the *swap-states* operator swaps the added state e_4 to *final* and the *final* state e_3 is swapped to *sink* state (Fig.1 **part(c)**). The *clean-up* step removes the resulting *sink* state e_3 (Fig.1 **part(d)**).

Once the DFA of the negated form constructed, we exploit the filtering algorithm of the REGULAR constraint.

The regular expression of the consistent tuples of `global_contiguity` is given by:

$$0^*1^*0^*$$

If we consider the negation form, we have as regular expression of the consistent tuples of ¬`global_contiguity`:

$$0^*11^*00^*1\{0, 1\}^*$$

These two regular expressions can be easily modeled in any CP language containing the REGULAR constraint.

The fact that the automaton of `global_contiguity` constraint is defined on the variables values, enabled to exploit efficiently and directly the REGULAR constraint.

Let us take an example with four variables (x_1, x_2, x_3, x_4). Fig.2 shows the three phases of REGULAR consistency algorithm. The MDD is constructed with four layers corresponding to the variables. The *forward* phase unfolds the negated DFA of Fig.1 where the *backward* phase removes 9 arcs and 6 states. If another constraint reduces the domain of x_3 by removing the value 0, the *maintaining* phase removes 6 arcs and 3 states to have at the end only two solutions (1010 and 1011).

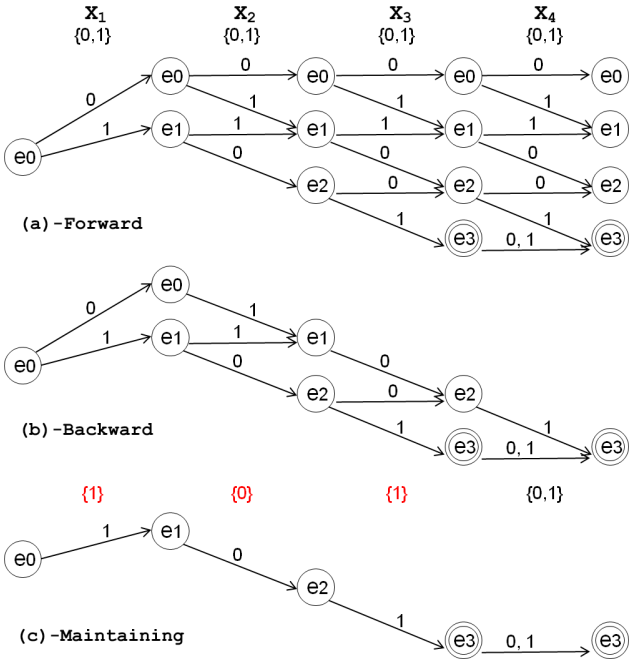


Figure 2. REGULAR constraint on ¬`global_contiguity` with four variables.

B. Case study: ¬Lex

The lexicographic ordering constraint[3], [4] $\vec{x} \leq_{lex} \vec{y}$ over two vectors of variables $\vec{x} =< x_0, x_1, ..., x_{n-1} >$ and $\vec{y} =< y_0, y_1, ..., y_{n-1} >$ holds iff $n = 0$, or $x_0 < y_0$, or

$x_0 = y_0$, and $\vec{x} =< x_1, ..., x_{n-1} > \leq_{lex} < y_1, ..., y_{n-1} >$. The automaton is defined on the relation between every two consecutive variables. In order to exploit the REGULAR constraint, we should transform the Lex constraint as following:

$$\vec{x} \leq_{lex} \vec{y} \equiv LexRel(r_0, r_1, ..., r_{n-1})$$

where $r_i \in \{<, =, >\}$, and $rel(r_i, x_i, y_i) \equiv x_i \ r_i \ y_i$. The automaton of $LexRel(r_0, r_1, ..., r_{n-1})$ is given in Figure 3 **part(a)** where it corresponds to:
$\mathcal{A} = ((\vec{x}, \vec{y}), \{e_0, e_1, e_2\}, \Psi, \{=, <, >\}, \delta, e_0, \{e_1, e_2\})$,
$\Psi = \{source(e_0), final(e_1), final(e_2)\}$,
$\delta = \{(e_0, =, e_0), (e_0, <, e_1), (e_0, \$, e_2), (e_1, =, e_1), (e_1, <, e_1), (e_1, >, e_1)\}$.

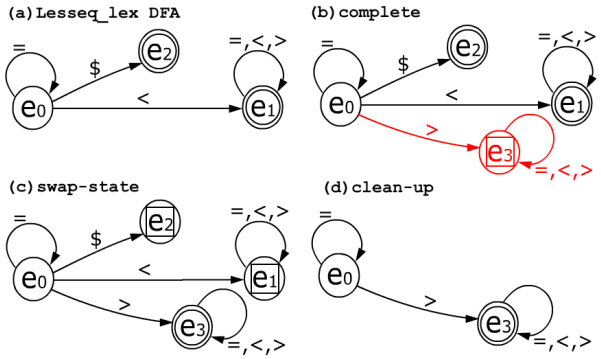


Figure 3. Complement DFA of the $\leq_{\texttt{lex}}$ constraint.

This constraint can be easily implemented with the REGULAR constraint, where the accepted regular expression is:

$$=^* \mid \ =^* < \{=, <, >\}^*$$

Negating form of the $\leq_{\texttt{lex}}$ constraint is shown in **part(d)** of the Figure 3 which represents the complement of the DFA of $\leq_{\texttt{lex}}$. **part (b,c,d)** shows respectively the *complete*, *swap-state* and *clean-up* steps to obtain the complement. From the complement of this constraint (i.e., DFA of $\neg\leq_{\texttt{lex}}$) we get the associated regular expression:

$$=^* > \{=, <, >\}^*$$

With the regular expression of the negated $\leq_{\texttt{lex}}$, we are able to exploit efficiently and directly the REGULAR constraint for filtering.

Let us take a simple example with $\vec{x} = [x_1, x_2, x_3, x_4]$ and $\vec{y} = [y_1, y_2, y_3, y_4]$ where each variable takes a value in $[0, 10]$. Fig.4 shows the three phases of REGULAR consistency algorithm on $\neg \leq_{lex}$. The MDD is constructed with four layers corresponding to the variables (x_i, y_i). The *forward* phase unfolds the negated DFA of Fig.3 and the

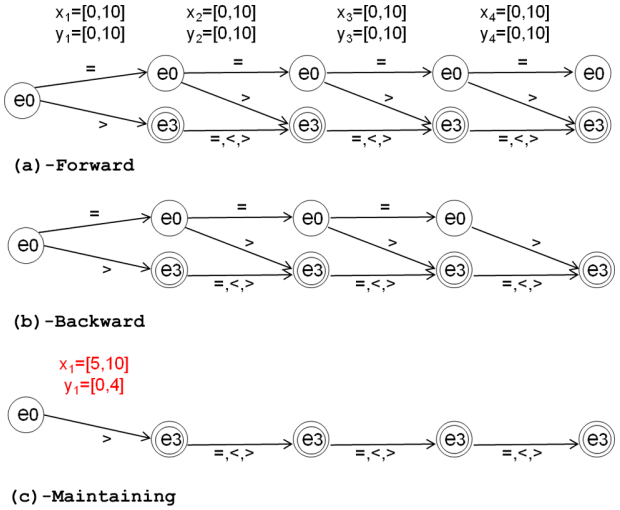


Figure 4. REGULAR constraint on $\neg \leq_{\texttt{lex}}$ with four variables.

backward phase removes the state e_0 form the last layer. In the case of a reduction by other constraints where the domain of x_1 is reduced to $[5, 10]$ and y_1 to $[0, 4]$, the *maintaining* phase removes 6 arcs and 3 states.

IV. Experimental validation

The goal of our experimental validation was to check that an automaton-based negated global constraint version (gotten for free through the presented framework) was more effective than a version where the negation is syntactically computed. For both the `global_contiguity` and $\leq_{\texttt{lex}}$ constraints, we built Gecode models and run our experiments on Intel Core2Quad CPU, Q6600 of 2.4 GHz, Linux machine with 3 Go of RAM.

A. `global_contiguity`

The declarative specification of `global_contiguity` can be given by:

$$\texttt{global\_contiguity(x)} \equiv \forall i, j \in 1..n : i < j \ s.t.$$

$$(x_i = 1) \wedge (x_j = 0) \Rightarrow (\forall k \in j + 1..n : x_k = 0)$$

Where, the negated form can be declaratively given by:

$$\neg\texttt{global\_contiguity(x)} \equiv \exists i, j \in 1..n : i < j \ s.t.$$

$$(x_i = 1) \wedge (x_j = 0) \wedge (\exists k \in j + 1..n : x_k = 1)$$

The Table I contains our experimental results on `global_contiguity`. We give a comparison between the implementation of the declarative specification of $\neg$`global_contiguity` and DFA-based implementation using our negation approach and the REGULAR constraint. The reported results are on different instances (from 200 to 11.10^3 variables) where a solving to get the first 100

solutions is launched. The results are on time/memory consumptions, number of propagations and the generated nodes. For each instance, from 200 to 10^3, the DFA-Based negation gives an interesting and impressive results comparing to the syntactic transformations based negation. For example, let us take the instance of 10^3 variables, the syntactic approach take more than five minutes and 2.5 Go of memory. With our DFA approach, the solving to get the first 100 solutions takes only 32 ms and 9 Mo of memory. For big instances (more than 10^3 variables), the syntactic approach reports an out-of-memory. Our approach stills giving interesting results also for the huge instance (11.10^3 variables) with 32 $sec.$. Fig. 5 shows the increase of time consumption of a solving to get the first 100 solutions with a syntactic negation and a DFA-based negation according to the grow-up of instances. The time consumption in the syntactic transformations increase following an exponential, where the increase time using our approach is in a linear way.

B. $\leq_{leq}$

The declarative specification of $\leq_{leq}$ on $\vec{x} =< x_0, x_1, ..., x_{n-1} >$ and $\vec{y} =< y_0, y_1, ..., y_{n-1} >$ can be given as follows:

$$\vec{x} \leq_{leq} \vec{y} \equiv (n = 0) \vee (x_0 < y_0) \vee$$

$$(x_0 = y_0 \wedge < x_1, ..., x_{n-1} > \ \leq_{\texttt{lex}} \ < y_1, ..., y_{n-1} >)$$

where the negation form is simply:

$$\neg(\vec{x} \leq_{lex} \vec{y}) \equiv ((n = 1) \wedge (x_0 > y_0)) \vee ((n > 1) \wedge ((x_0 > y_0) \vee$$

$$(< x_1, ..., x_{n-1} > \ \neg\leq_{\texttt{lex}} \ < y_1, ..., y_{n-1} >)))$$

This is a first transformation to get the negation of $\leq_{\texttt{lex}}$. One can also express the negation using the global constraint $>_{\texttt{lex}}$ which is available on Gecode:

$$\neg(\vec{x} \leq_{lex} \vec{y}) \equiv \vec{x} >_{leq} \vec{y}$$

The Table II contains results on the syntactic negation and the $>_{\texttt{lex}}$ global constraint. We compare the two results with our DFA-based negation approach. The reported results are on 200 to 8.10^3 variables instances. The DFA-based negation is better than the syntactic negation and the original constraint $>_{\texttt{lex}}$. Let us take the big instance with 8.10^3 variables, our generic approach to negate $\leq_{\texttt{lex}}$ have a time consumption three times less than the syntactic negation and two times less than $>_{\texttt{lex}}$ constraint. For memory consumption, the $>_{\texttt{lex}}$ have a consumption two times or more than the DFA-based negation. Through these comparaisons, we see that the DFA-based negation is widely better than the syntactic negation and remains very competitive with its equivalent well established global constraint $>_{lex}$.

V. Conclusion

In this paper, we have proposed an approach to get automatically a filtering algorithm for the negation of an automaton-based global constraint. Our approach is built over automata operations and exploits the REGULAR global constraints to automatically derive filtering rules for the negation. Through experiments, we evaluated this approach on two well-known global constraints, namely the negation of `global_contiguity` and $\leq_{\texttt{lex}}$, for which we automatically derived filtering algorithms. The Gecode models and results show that our versions are efficient. We forecast 1) to extend our approach to push-down automata by using the generic GRAMMAR constraint and 2) to extend it to logical connectives (i.e., conjunction, disjunction) between global constraints.

References

[1] H. R. Andersen, T. Hadzic, J. N. Hooker, and P. Tiedemann. A constraint store based on multivalued decision diagrams. In *Proceedings of the 13th international conference on Principles and practice of constraint programming*, CP'07, pages 118–132, Berlin, Heidelberg, 2007. Springer-Verlag.

[2] Fahiem Bacchus and Toby Walsh. Propagating logical combinations of constraints. In *IJCAI*, pages 35–40, 2005.

[3] Nicolas Beldiceanu, Mats Carlsson, Romuald Debruyne, and Thierry Petit. Reformulation of global constraints based on constraints checkers. *Constraints*, 10(4):339–362, 2005.

[4] Nicolas Beldiceanu, Mats Carlsson, Sophie Demassey, and Thierry Petit. Global constraint catalogue: Past, present and future. *Constraints*, 12:21–62, March 2007.

[5] Jefferson C., Moore N. C. A., Nightingale P., and Petrie K. E. Implementing logical connectives in constraint programming. *Artif. Intell.*, 174(16-17):1407–1429, 2010.

[6] Mats Carlsson and Nicolas Beldiceanu. From constraints to finite automata to filtering algorithms. In *ESOP*, pages 94–108, 2004.

[7] Katsirelos G., Narodytska N., and Walsh T. Reformulating global grammar constraints. In *CPAIOR*, volume 5547 of *LNCS*, pages 132–147. Springer, 2009.

[8] Samid Hoda, Willem-Jan Van Hoeve, and J. N. Hooker. A systematic approach to mdd-based constraint programming. In *Proceedings of the 16th international conference on Principles and practice of constraint programming*, CP'10, pages 266–280, Berlin, Heidelberg, 2010. Springer-Verlag.

[9] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley, 2006.

[10] Nadjib Lazaar, Arnaud Gotlieb, and Yahia Lebbah. Fault localization in constraint programs. In *ICTAI (1)*, pages 61–67. IEEE Computer Society, 2010.

Table I
EXPERIMENTAL RESULTS ON ¬global_contiguity.

var	syntactic transformations based negation				DFA − based negation			
	T	M	P	N	T	M	P	N
200	53.13	24.39	65 111	396	2.09	0.42	366	397
300	186.07	77.90	116 535	496	3.31	0.77	468	497
400	509.93	179.22	170 209	596	4.81	1.559	566	597
500	1 082.35	344.86	244 235	696	6.69	2.20	668	697
600	1 936.68	589.79	315 320	796	8.77	2.96	766	797
700	3 125.01	930.34	411 935	896	11.43	4.80	868	897
800	4 735.71	1 381.68	500 407	996	14.19	6.03	966	997
900	6 760.44	1 960.24	619 627	1 096	17.27	7.28	1 068	1 097
1 000	19 407.02	2 681.01	725 507	1196	22.88	8.53	1 166	1 197
1 100	—	OOM	—	—	24.17	9 925	1 268	1 297
1 200	—	OOM	—	—	28.67	14 214	1 366	1 397
1 300	—	OOM	—	—	32.78	16 330	1 468	1 497
1 400	—	OOM	—	—	37.58	18 766	1 566	1 597
1 500	—	OOM	—	—	42.69	21 266	1 668	1 697
1 600	—	OOM	—	—	47.83	23 702	1 766	1 797
1 700	—	OOM	—	—	53.34	26 276	1 868	1 897
1 800	—	OOM	—	—	59.32	28 712	1 966	1 997
1 900	—	OOM	—	—	65.09	31 212	2 068	2 097
11 000	—	OOM	—	—	1 923.53	896 958	11 166	11 197

T : time(ms), M : memory(MB), P : propagations, N : nodes, OOM : Out − Of − Memory

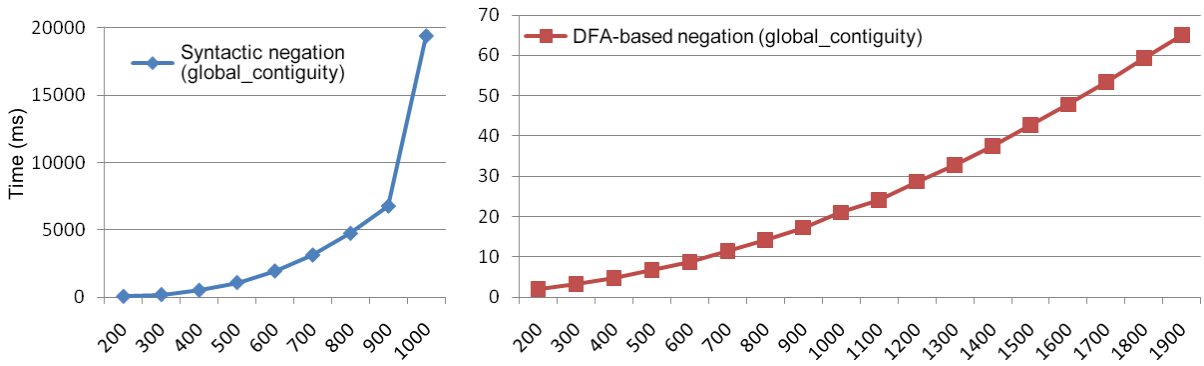


Figure 5. Time consumptions for ¬global_contiguity (syntactic and DFA-Based negation).

Table II
EXPERIMENTAL RESULTS ON ¬ $\leq_{\text{lex}}$.

var	syntactic transformations based negation				$>_{\text{lex}}$				DFA − based negation			
	T	M	P	N	T	M	P	N	T	M	P	N
200	7.00	1.67	1 944	400	6.60	2.03	2 341	400	4.27	0.82	302	400
300	13.24	3.50	2 708	500	12.07	4.69	3 311	500	7.07	2.10	402	500
400	21.45	6.42	3 544	600	19.09	7.64	4 341	600	11.16	3.00	502	600
500	30.86	9.56	4 308	700	27.65	11.87	5 311	700	15.28	4.32	602	700
600	43.32	13.35	5 144	800	38.13	16.03	6341	800	20.30	7.75	702	800
700	56.77	17.38	5 908	900	49.67	20.90	7 311	900	26.28	9.00	802	900
800	71.89	22.00	6 744	1 000	62.72	26.73	8 341	1 000	32.62	11.41	902	1 000
900	90.12	27.12	7 508	1 100	77.12	33.59	9 311	1 100	39.70	15.03	1 002	1 100
10^3	107.97	33.02	8 344	1 200	92.54	40.50	10 341	1200	47.38	16.57	1 102	1 200
2.10^3	402.09	123.06	16 344	2 200	334.72	153.91	20 341	2 200	161.96	65.46	2 102	2 200
3.10^3	889.68	270.51	24 344	3 200	731.38	352.69	30 341	3 200	344.10	147.19	3 102	3 200
4.10^3	1 591.25	475.89	32 344	4 200	1 300.39	625.27	40 341	4 200	597.54	255.35	4 102	4 200
5.10^3	2 527.08	738.53	40 344	5 200	2 059.30	970.92	50 341	5 200	915.20	395.56	5 102	5 200
6.10^3	3 758.49	1 059.63	48 344	6 200	3 010.67	1 388.53	60 341	6 200	1 310.84	567.67	6 102	6 200
7.10^3	5 194.56	1 440.67	56 344	7 200	4 115.96	1 879.97	70 341	7 200	1 772.84	770.78	7 102	7 200
8.10^3	6 951.67	1 880.27	64 344	8 200	5 681.13	2 446.10	80 341	8 200	2 309.79	1 004.37	8 102	8 200

T : time(ms), M : memory(MB), P : propagations, N : nodes, OOM : Out − Of − Memory

[11] Nadjib Lazaar, Arnaud Gotlieb, and Yahia Lebbah. On testing constraint programs. In *CP*, volume 6308 of *LNCS*, pages 330–344. Springer, 2010.

[12] Nadjib Lazaar, Arnaud Gotlieb, and Yahia Lebbah. A framework for the automatic correction of constraint programs. In *ICST*, page Forthcomming. IEEE Computer Society, 2011.

[13] Olivier Lhomme. Arc-consistency filtering algorithms for logical combinations of constraints. In *CPAIOR 2004*, volume 3011 of *LNCS*, pages 209–224, 2004.

[14] Gilles Pesant. A regular language membership constraint for finite sequences of variables. In Mark Wallace, editor, *CP*, volume 3258 of *LNCS*, pages 482–495. Springer, 2004.

[15] Claude-Guy Quimper and Toby Walsh. Global grammar constraints. In Frédéric Benhamou, editor, *CP*, volume 4204 of *LNCS*, pages 751–755. Springer, 2006.

[16] Christian Schulte. Programming deep concurrent constraint combinators. In *PADL 2000*, volume 1753 of *LNCS*, pages 215–229. Springer, 2000.

[17] Meinolf Sellmann. The theory of grammar constraints. In Frédéric Benhamou, editor, *CP*, volume 4204 of *LNCS*, pages 530–544. Springer, 2006.

[18] Peter J. Stuckey and Peter J. Stuckey. Negation and constraint logic programming, 1995.

Annexe B : Modèle-Oracle et CPUTs

Les règles de Golomb

CPUT1

```
/****************
* OPL 6.3 Model *
*****************/
using CP;

int m=...;

dvar int x[1..m] in 0..m*m;

minimize x[m];
subject to{

cc1: forall(i in 1..m-1)
      x[i] < x[i+1];

cc2: forall(ordered i, j in 2..m, k in 1..m)
      x[i] == x[i-1]+k => x[j]!=x[j-1]+k;
}                                                                    .
```

CPUT2

```
/****************
* OPL 6.3 Model *
*****************/
using CP;

int m=...;
tuple indexerTuple{ int i; int j;}
{indexerTuple} indexes = {<i,j> | i,j in 1..m : i<j};

dvar int x[1..m] in 0..m*m;
dvar int d[indexes];

minimize x[m];
subject to {

cc1: forall(i in 1..m-1)
       x[i] < x[i+1];

cc2: forall(ind in indexes)
      d[ind] == x[ind.i]-x[ind.j];

cc3: forall(ind1 in indexes, ind2 in indexes, ind3 in indexes : (ind1.i==ind2.i)&&
      (ind2.j==ind3.i)&&(ind1.i==ind3.j)&&(ind1.i<ind2.j<ind1.j))
        d[ind1] == d[ind2]+d[ind3];

cc4: forall(ordered i, j in 2..m,k in 1..m)
      x[i]==x[i-1]+k => x[j]!=x[j-1]+k;
}
```

CPUT3

```
/****************
* OPL 6.3 Model *
*****************/
using CP;

int m=...;
tuple indexerTuple{ int i; int j;}
{indexerTuple} indexes = {<i,j> | i,j in 1..m : i<j};

dvar int x[1..m] in 0..m*m;
dvar int d[indexes];

minimize x[m];
subject to {

cc1: forall (i in 1..m-1)
       x[i] < x[i+1];

cc2: forall(ind in indexes)
       d[ind] == x[ind.i]-x[ind.j];

cc3: forall(ind1 in indexes, ind2 in indexes, ind3 in indexes : (ind1.i==ind2.i)&&
      (ind2.j==ind3.i)&&(ind1.j==ind3.j)&&(ind1.i<ind2.j<ind1.j))
        d[ind1]==d[ind2]+d[ind3];

cc4: forall(ind1,ind2,ind3,ind4 in indexes :(ind1.i==ind2.i)&&(ind1.j==ind3.j)&&
      (ind2.j==ind4.j)&&(ind3.i==ind4.i)&&(ind1.i<m-1)&&(3<ind1.j<m+1)&&(2<ind2.j<m)
      &&(1<ind3.i<m-1)&&(ind1.i < ind3.i < ind2.j < ind1.j))
          d[ind1]==d[ind2]+d[ind3]-d[ind4];

cc5: forall(i in 2..m, j in 2..m, k in 1..m : i < j)
                x[i]==x[i-1]+k => x[j]!=x[j-1]+k;
}
```

CPUT4

```
/****************
* OPL 6.3 Model *
*****************/
using CP;

int m=...;
tuple indexerTuple{ int i; int j;}
{indexerTuple} indexes = {<i,j> | i,j in 1..m : i<j};

dvar int x[1..m] in 0..m*m;
dvar int d[indexes];

minimize x[m];
subject to {

cc1: forall(i in 1..m-1)
      x[i] < x[i+1];

cc2: forall(ind in indexes)
      d[ind] == x[ind.j]-x[ind.i];

cc3: x[m] >= (m*(m-1))/2;

cc4: x[2] >= x[m]-x[m-1];

cc5: forall(i in m..m*2)
      count(all(j in indexes) d[j],i)==1;

cc6: x[1] == 0;

cc7: forall(ind1 in indexes, ind2 in indexes, ind3 in indexes :(ind1.i==ind2.i)&&
      (ind2.j==ind3.i)&&(ind1.j==ind3.j)&&(ind1.i<ind2.j<ind1.j))
       d[ind1] == d[ind2]+d[ind3];


cc8: forall(ind1, ind2, ind3, ind4 in indexes :(ind1.i==ind2.i)&&(ind1.j==ind3.j)&&
      (ind2.j==ind4.j)&&(ind3.i==ind4.i)&&(ind1.i<m-1)&&(3<ind1.j<m+1)&&
      (2<ind2.j<m)&&(1<ind3.i<m-1)&&(ind1.i<ind3.i<ind2.j<ind1.j))
         d[ind1] == d[ind2]+d[ind3]-d[ind4];

cc9: forall(i in 2..m, j in 2..m, k in 1..m : i < j)
      x[i] == x[i-1]+k => x[j]!=x[j-1]+k;
}
```

CPUT5

```
/****************
* OPL 6.3 Model *
*****************/
using CP;

int m=...;
tuple indexerTuple{ int i; int j;}
{indexerTuple} indexes = {<i,j> | i,j in 1..m : i<j};

dvar int x[1..m] in 0..m*m;
dvar int d[indexes];

minimize x[m];
subject to {

cc1: forall (i in 1..m-1)
      x[i] > x[i+1];

cc2: forall(ind in indexes)
      d[ind] == x[ind.i]-x[ind.j];

cc3: x[1] == 0;

cc4: x[m] >= (m*(m-1))/2;

cc5: allDifferent(d);

cc6: x[2] <= x[m]-x[m-1];

cc7: forall(ind1 in indexes, ind2 in indexes,ind3 in indexes: (ind1.i==ind2.i)&&
      (ind2.j==ind3.i) &&(ind1.j==ind3.j)&&(ind1.i<ind2.j < ind1.j))
        d[ind1]==d[ind2]+d[ind3];

cc8: forall(ind1,ind2,ind3,ind4 in indexes: (ind1.i==ind2.i)&&(ind1.j==ind3.j)&&
      (ind2.j==ind4.j)&&(ind3.i==ind4.i)&&(ind1.i<m-1)&&(3<ind1.j<m+1)&&
       (2<ind2.j<m)&&(1<ind3.i<m-1)&&(ind1.i < ind3.i < ind2.j < ind1.j))
        d[ind1]==d[ind2]+d[ind3]-d[ind4];

cc9: forall(i in 2..m, j in 2..m, k in 1..m : i < j)
      x[i]==x[i-1]+k => x[j] != x[j-1]+k;
}
```

CPUT6

```
/****************
* OPL 6.3 Model *
*****************/
using CP;

int m=...;
tuple indexerTuple{ int i; int j;}
{indexerTuple} indexes = {<i,j> | i,j in 1..m : i<j};

dvar int x[1..m] in 0..m*m;
dvar int d[indexes];

minimize x[m];
subject to {

cc1: forall (i in 1..m-1)
      x[i] < x[i+1];

cc2: forall(ind in indexes)
      d[ind] == x[ind.i]+x[ind.j];

cc3: x[1] == 0;

cc4: x[m]>=(m*(m-1))/2;

cc5: allDifferent(d);

cc6: x[2] <= x[m]-x[m-1];

cc7: forall(ind1 in indexes, ind2 in indexes,ind3 in indexes: (ind1.i==ind2.i)&&
      (ind2.j==ind3.i) &&(ind1.j==ind3.j)&&(ind1.i<ind2.j < ind1.j))
        d[ind1]==d[ind2]+d[ind3];

cc8: forall(ind1,ind2,ind3,ind4 in indexes: (ind1.i==ind2.i)&&(ind1.j==ind3.j)&&
      (ind2.j==ind4.j)&&(ind3.i==ind4.i)&&(ind1.i<m-1)&&(3<ind1.j<m+1)&&(2<ind2.j<m)
      &&(1<ind3.i<m-1)&&(ind1.i < ind3.i < ind2.j < ind1.j))
        d[ind1]==d[ind2]+d[ind3]-d[ind4];

cc9: forall(i in 2..m, j in 2..m, k in 1..m : i < j)
      x[i]==x[i-1]+k => x[j] != x[j-1]+k;
}
```

CPUT7

```
/****************
* OPL 6.3 Model *
*****************/
using CP;

int m=...;
tuple indexerTuple{ int i; int j;}
{indexerTuple} indexes = {<i,j> | i,j in 1..m : i<j};

dvar int x[1..m] in 0..m*m;
dvar int d[indexes];

minimize x[m];
subject to {

cc1: forall (i in 1..m-1)
      x[i] < x[i+1];

cc2: forall(ind in indexes)
      d[ind] == x[ind.i]-x[ind.j];

cc3: x[1] == 0;

cc4: x[m] >= (m*(m*1))/2;

cc5: allDifferent(d);

cc6: x[2] <= x[m]-x[m-1];

cc7: forall(ind1 in indexes, ind2 in indexes,ind3 in indexes: (ind1.i==ind2.i)&&
      (ind2.j==ind3.i) &&(ind1.j==ind3.j)&&(ind1.i<ind2.j < ind1.j))
         d[ind1]==d[ind2]+d[ind3];

cc8: forall(ind1,ind2,ind3,ind4 in indexes: (ind1.i==ind2.i)&&(ind1.j==ind3.j)&&
      (ind2.j==ind4.j)&&(ind3.i==ind4.i)&&(ind1.i<m-1)&&(3<ind1.j<m+1)&&(2<ind2.j<m)
       &&(1<ind3.i<m-1)&&(ind1.i < ind3.i < ind2.j < ind1.j))
           d[ind1]==d[ind2]+d[ind3]-d[ind4];

cc9: forall(ordered i,j in 2..m, k in 1..m*m : i < j)
      x[i]==x[i-1]+k => x[j] == x[j-1]+k;
}
```

Les n-reines

CPUT1

```
/****************
* OPL 6.3 Model *
*****************/
using CP;

int n=...;
range Domain=1..n;

dvar int queens[Domain] in 1..n;
dvar int col[Domain] in 1..n;
dvar int diag1[Domain];
dvar int diag2[Domain];
dvar int diag3[Domain];
dvar int diag4[Domain];

subject to{

cc1: allDifferent(queens);

cc2: forall(i in Domain)
      diag1[i]==queens[i]+i;

cc3: forall(i in Domain)
      diag2[i]==queens[i]-i;

cc4: forall(i in Domain)
      diag3[i]==col[i]+i;

cc5: forall(i in Domain)
      diag4[i]==col[i]-i;

cc6: allDifferent(diag1);

cc7: allDifferent(diag2);

cc8: allDifferent(diag3);

cc9: allDifferent(diag4);

cc10: forall(r,c in Domain)
       (queens[c]==r) == (col[r]==c);

cc11: sum(i in Domain)
        queens[i]==(n*(n-1)/2);

cc12: sum(i in Domain)
        col[i]==(n*(n+1)/2);
}
```

CPUT2

```
/****************
* OPL 6.3 Model *
*****************/
using CP;

int n=...;
range Domain=1..n;

dvar int queens[Domain] in 1..n;
dvar int col[Domain] in 1..n;
dvar int diag1[Domain];
dvar int diag2[Domain];
dvar int diag3[Domain];
dvar int diag4[Domain];

subject to{

cc1: allDifferent(queens);

cc2: forall(i in Domain)
      diag1[i]==queens[i]+i;

cc3: forall(i in Domain)
      diag2[i]==queens[i]-i;

cc4: forall(i in Domain)
      diag3[i]==col[i]+i;

cc5: forall(i in Domain)
      diag4[i]==col[i]-i;

cc6: allDifferent(diag1);

cc7: allDifferent(diag2);

cc8: allDifferent(diag3);

cc9: allDifferent(diag4);

cc10: forall(r,c in Domain)
       (queens[c]==r) == (col[r]==c);

cc11: sum(i in Domain)
        queens[i]==(n*(n+1)/2);

cc12: sum(i in Domain)
        col[i]==(n*(n-1)/2);
}
```

CPUT3

```
/****************
* OPL 6.3 Model *
*****************/
using CP;

int n=...;
range Domain=1..n;

dvar int queens[Domain] in 1..n;
dvar int col[Domain] in 1..n;
dvar int diag1[Domain];
dvar int diag2[Domain];
dvar int diag3[Domain];
dvar int diag4[Domain];

subject to{

cc1: allDifferent(queens);

cc2: forall(i in Domain)
      diag1[i]==queens[i]+i;

cc3: forall(i in Domain)
      diag2[i]==queens[i]-i;

cc4: forall(i in Domain)
      diag3[i]==col[i]+i;

cc5: forall(i in Domain)
      diag4[i]==col[i]-i;

cc6: allDifferent(diag1);

cc7: allDifferent(diag2);

cc8: allDifferent(diag3);

cc9: allDifferent(diag4);

cc10: forall(r,c in Domain)
       (queens[c]==r) == (col[r]==c);

cc11: sum(i in Domain)
        queens[i]==(n+(n+1)/2);

cc12: sum(i in Domain)
        col[i]==(n*(n+1)/2);
}
```

CPUT4

```
/****************
* OPL 6.3 Model *
*****************/
using CP;

int n=...;
range Domain=1..n;

dvar int queens[Domain] in 1..n;
dvar int col[Domain] in 1..n;
dvar int diag1[Domain];
dvar int diag2[Domain];
dvar int diag3[Domain];
dvar int diag4[Domain];

subject to{

cc1: allDifferent(queens);

cc2: forall(i in Domain)
      diag1[i]==queens[i]+i;

cc3: forall(i in Domain)
      diag2[i]==queens[i]-i;

cc4: forall(i in Domain)
      diag3[i]==col[i]+i;

cc5: forall(i in Domain)
      diag4[i]==col[i]-i;

cc6: allDifferent(diag1);

cc7: allDifferent(diag2);

cc8: allDifferent(diag3);

cc9: allDifferent(diag4);

cc10: forall(r,c in Domain)
       (queens[c]==r) == (col[r]==c);

cc11: sum(i in Domain)
        queens[i]==(n*(n+1)/2);

cc12: sum(i in Domain)
        col[i]==(n+(n+1)/2);
}
```

CPUT5

```
/****************
* OPL 6.3 Model *
*****************/
using CP;

int n=...;
range Domain=1..n;

dvar int queens[1..n] in 1..n;
dvar int diag1[Domain];
dvar int diag2[Domain];


subject to{

cc1: forall( i in Domain)
      diag1[i]==queens[i]+i;

cc2: forall( i in Domain)
      diag2[i]==queens[i]-i;

cc3: sum(i in Domain)
       queens[i]==(n*(n+1) div 2);


cc4: forall(ordered i,j in Domain)
      diag1[i] <= diag1[j];

cc5: allDifferent(queens);

cc6: allDifferent(diag2);
}
```

CPUT6

```
/****************
* OPL 6.3 Model *
*****************/
using CP;

int n=...;
range Domain=1..n;

dvar int queens[Domain] in Domain;
dvar int b[Domain,Domain] in 0..1;

subject to{

cc1: forall(i in Domain)
      sum(j in Domain)b[i][j]==1;

cc2: forall(j in Domain)
      sum(i in Domain)b[i][j]==1;

cc3: forall(i,j in Domain)
      (b[i][j]==1)=>(queens[i]==j);
}
```

CPUT7

```
/****************
* OPL 6.3 Model *
*****************/
using CP;

int n=...;
range Domain=1..n;

dvar int x[0..1,Domain] in Domain;
dvar int queens[Domain] in Domain;

subject to{

cc1: forall(i,v in Domain)
      (x[0][i]==v) == (x[1][v]==i);

cc2: forall(i in Domain)
      x[0][i]==queens[i];
}
```

Golfeurs sociables

Modèle-Oracle

```
/****************
* OPL 6.3 Model *
*****************/
using CP;

int weeks = ...;
int groups = ...;
int groupSize = ...;
int golfers = groups * groupSize;

range Golfer = 1..golfers;
range Week = 1..weeks;
range Group = 1..groups;

dvar int assign[Golfer,Week] in Group;

subject to {

c1: forall(gr in 1..3, w in 1..weeks)
  sum(g in 1..golfers)
     (assign[g,w]==gr) == groupSize;

c2: forall(g1,g2 in 1..golfers:g1 != g2
      forall(w1,w2 in 1..weeks:w1 != w2)
    (assign[g1,w1]==assign[g2,w1])
   +(assign[g1,w2]==assign[g2,w2]) <= 1;
}
```

CPUT1

```
/****************
* OPL 6.3 Model *
*****************/
using CP;

int weeks = ...;
int groups = ...;
int groupSize = ...;
int golfers = groups * groupSize;

range Golfer = 1..golfers;
range Week = 1..weeks;
range Group = 1..groups;

dvar int assign[Golfer,Week] in Group;

subject to {

cc1: forall(gr in 1..groups,w in 1..weeks)
      sum(g in 1..golfers)
       (assign[g,w]!=gr) == groupSize;

cc2: forall(g1,g2 in 1..golfers:g1 != g2)
      forall(w1,w2 in 1..weeks:w1 != w2)
     (assign[g1,w1]==assign[g2,w1])
     +(assign[g1,w2]==assign[g2,w2]) <= 1;

cc3: forall(g in 1..golfers)
      assign[g,1]==((g-1) div groupSize)+1;

cc4: forall(g in 1..golfers:g <= groupSize)
      assign[g,2]==g;

cc5: forall (g in 1..golfers,w in 1..weeks:
                  w >= 2 && g <= groupSize)
      assign[g,w] == g;
}
```

CPUT2

```
/****************
* OPL 6.3 Model *
*****************/
using CP;

int weeks = ...;
int groups = ...;
int groupSize = ...;
int golfers = groups * groupSize;

range Golfer = 1..golfers;
range Week = 1..weeks;
range Group = 1..groups;

dvar int assign[Golfer,Week] in Group;

subject to {

cc1: forall(gr in 1..groups,w in 1..weeks)
      sum(g in 1..golfers)
       (assign[g,w]==gr) == groupSize;

cc2: forall(g1,g2 in 1..golfers:g1 != g2)
      forall(w1,w2 in 1..weeks:w1 != w2)
     (assign[g1,w1]==assign[g2,w1])
     +(assign[g1,w2]==assign[g2,w2]) <= 10;

cc3: forall(g in 1..golfers)
      assign[g,1]==((g-1) div groupSize)+1;

cc4: forall(g in 1..golfers:g <= groupSize)
      assign[g,2]==g;

cc5: forall (g in 1..golfers,w in 1..weeks:
                 w >= 2 && g <= groupSize)
      assign[g,w] == g;
}
```

CPUT3

```
/****************
* OPL 6.3 Model *
*****************/
using CP;

int weeks = ...;
int groups = ...;
int groupSize = ...;
int golfers = groups * groupSize;

range Golfer = 1..golfers;
range Week = 1..weeks;
range Group = 1..groups;

dvar int assign[Golfer,Week] in Group;

subject to {

cc1: forall(gr in 1..groups,w in 1..weeks)
      sum(g in 1..golfers)
       (assign[g,w]==gr) == groupSize;

cc2: forall(g1,g2 in 1..golfers:g1 != g2)
      forall(w1,w2 in 1..weeks:w1 != w2)
     (assign[g1,w1]==assign[g2,w1])
     +(assign[g1,w2]==assign[g2,w2]) <= 1;

cc3: forall(g in 1..golfers)
      assign[g,1]!=((g-1) div groupSize)+1;

cc4: forall(g in 1..golfers:g <= groupSize)
      assign[g,2]==g;

cc5: forall (g in 1..golfers,w in 1..weeks:
                 w >= 2 && g <= groupSize)
      assign[g,w] == g;
}
```

CPUT4

```
/****************
* OPL 6.3 Model *
*****************/
using CP;

int weeks = ...;
int groups = ...;
int groupSize = ...;
int golfers = groups * groupSize;

range Golfer = 1..golfers;
range Week = 1..weeks;
range Group = 1..groups;

dvar int assign[Golfer,Week] in Group;

subject to {

cc1: forall(gr in 1..groups,w in 1..weeks)
      sum(g in 1..golfers)
       (assign[g,w]==gr) == groupSize;

cc2: forall(g1,g2 in 1..golfers:g1 != g2)
      forall(w1,w2 in 1..weeks:w1 != w2)
     (assign[g1,w1]==assign[g2,w1])
     +(assign[g1,w2]==assign[g2,w2]) <= 1;

cc3: forall(g in 1..golfers)
      assign[g,1]==((g-1) div groupSize)+1;

cc4: forall(g in 1..golfers:g <= groupSize)
      assign[2,g]==g;

cc5: forall (g in 1..golfers,w in 1..weeks:
                 w >= 2 && g <= groupSize)
      assign[g,w] == g;
}
```

CPUT5

```
/****************
* OPL 6.3 Model *
*****************/
using CP;

int weeks = ...;
int groups = ...;
int groupSize = ...;
int golfers = groups * groupSize;

range Golfer = 1..golfers;
range Week = 1..weeks;
range Group = 1..groups;

dvar int assign[Golfer,Week] in Group;

subject to {

cc1: forall(gr in 1..groups,w in 1..weeks)
      sum(g in 1..golfers)
       (assign[g,w]==gr) == groupSize;

cc2: forall(g1,g2 in 1..golfers:g1 != g2)
      forall(w1,w2 in 1..weeks:w1 != w2)
     (assign[g1,w1]==assign[g2,w1])
     +(assign[g1,w2]==assign[g2,w2]) <= 1;

cc3: forall(g in 1..golfers)
      assign[g,1]==((g-1) div groupSize)+1;

cc4: forall(g in 1..golfers:g <= groupSize)
      assign[g,2]==g;

cc5: forall (g in 1..golfers,w in 1..weeks:
                 w >= 2 && g <= groupSize)
      assign[g,g] == w;
}
```

Ordonnancement de véhicules

Modèle-Oracle

```
/****************
* OPL 6.3 Model *
****************/
using CP;

int  nbCars    = ...;
int   nbOptions = ...;
int   nbSlots   = ...;
range   Cars    = 1..nbCars;
range   Options = 1..nbOptions;
range   Slots   = 1..nbSlots;
int demand[Cars] = ...;
int option[Options,Cars] = ...;
int capacity[Options,0..1] = ...;
int optionDemand[i in Options] = ...;

dvar int slot[Slots] in Cars;

subject to {

c1: forall(c in 1..nbCars)
      sum(s in 1..nbSlots)
        (slot[s] == c) == demand[c];

c2: forall(o in 1..nbOptions)
  forall(s in 1..nbSlots-capacity[o][1]+1)
      sum(j in s..(s + capacity[o][1]-1))
      option[o][slot[j]] <= capacity[o][0];
}
```

CPUT1

```
/****************
* OPL 6.3 Model *
****************/
using CP;

int  nbCars    = ...;
int   nbOptions = ...;
int   nbSlots   = ...;
range   Cars    = 1..nbCars;
range   Options = 1..nbOptions;
range   Slots   = 1..nbSlots;
int demand[Cars] = ...;
int option[Options,Cars] = ...;
int capacity[Options,0..1] = ...;
int optionDemand[i in Options] = ...;

dvar int slot[Slots] in Cars;

subject to {

cc1: forall(c in 1..nbCars)
      sum(s in 1..nbSlots)
       (slot[s] == c) == demand[c];

cc2: forall(o in 1..nbOptions)
      forall( i in 1..optionDemand[o])
     sum(s in 1 .. (nbSlots-i*capacity[o][1]))
      option[o][slot[s]] >=
      optionDemand[o]-i*capacity[o][0];
}
```

CPUT2

```
/****************
* OPL 6.3 Model *
*****************/
using CP;

int  nbCars    = ...;
int   nbOptions = ...;
int   nbSlots   = ...;
range   Cars    = 1..nbCars;
range   Options = 1..nbOptions;
range   Slots   = 1..nbSlots;
int demand[Cars] = ...;
int option[Options,Cars] = ...;
int capacity[Options,0..1] = ...;
int optionDemand[i in Options] = ...;

dvar int slot[Slots] in Cars;
dvar int setup[Options,Slots] in 0..1;

subject to {

cc1: forall(c in 1..nbCars)
      sum(s in 1..nbSlots)
      (slot[s] == c) == demand[c];

cc2: forall(o in 1..nbOptions)
      forall(s in 1..nbSlots)
      setup[o,s] == option[o][slot[s]];

cc3: forall(o in 1..nbOptions)
       forall(i in 1..optionDemand[o])
 sum(s in 1..(nbSlots-i*capacity[o][1]))
   setup[o,s] >=
     optionDemand[o]-i*capacity[o][0];
}
```

CPUT3

```
/****************
* OPL 6.3 Model *
*****************/
using CP;

int  nbCars    = ...;
int   nbOptions = ...;
int   nbSlots   = ...;
range   Cars    = 1..nbCars;
range   Options = 1..nbOptions;
range   Slots   = 1..nbSlots;
int demand[Cars] = ...;
int option[Options,Cars] = ...;
int capacity[Options,0..1] = ...;
int optionDemand[i in Options] = ...;

dvar int slot[Slots] in Cars;
dvar int setup[Options,Slots] in 0..1;

subject to {

cc1: forall(c in 1..nbCars)
      sum(s in Slots)
      (slot[s]==c) == demand[c];

cc2: forall(o in 1..nbOptions,s in Slots)
      setup[o,s] ==
    option[o,maxl(1,slot[s])]*(slot[s]>0);

cc3: forall(o in 1..nbOptions)
      forall(i in 1..optionDemand[o])
   sum(s in 1..(nbSlots-i*capacity[o][1]))
        setup[o,s] >=
         optionDemand[o]-i*capacity[o][0];
}
```

CPUT4

```
/****************
* OPL 6.3 Model *
*****************/
using CP;

int  nbCars    = ...;
int   nbOptions = ...;
int   nbSlots   = ...;
range   Cars    = 1..nbCars;
range   Options = 1..nbOptions;
range   Slots   = 1..nbSlots;
int demand[Cars] = ...;
int option[Options,Cars] = ...;
int one[i in 1..nbCars] = 1;
int capacity[Options,0..1] = ...;
int optionDemand[i in Options] = ...;

dvar int slot[Slots] in Cars;
dvar int setup[Options,Slots] in 0..1;
dvar int demandV[1..nbCars] in 1..nbSlots;

subject to{

cc1: forall(c in 1..nbCars)
      sum(s in 1..nbSlots)
       (slot[s]==c) == demand[c];

cc2: pack(slot,demandV, one);

cc3: forall(c in 1..nbCars)
      demandV[c] == demand[c];

cc4: forall(o in 1..nbOptions)
 forall(s in 1..(nbSlots-capacity[o][1]+1))
   sum(j in s..(s+capacity[o][1]-1))
    setup[o,j] <= capacity[o][0];
}
```

CPUT5

```
/****************
* OPL 6.3 Model *
*****************/
using CP;

int  nbCars    = ...;
int   nbOptions = ...;
int   nbSlots   = ...;
range   Cars    = 1..nbCars;
range   Options = 1..nbOptions;
range   Slots   = 1..nbSlots;
int demand[Cars] = ...;
int option[Options,Cars] = ...;
int one[i in 1..nbCars] = 1;
int capacity[Options,0..1] = ...;
int optionDemand[i in Options] = ...;

dvar int slot[Slots] in Cars;
dvar int setup[Options,Slots] in 0..1;
dvar int demandV[1..nbCars] in 1..nbSlots;

subject to{

cc1: pack(slot, slot, one);

cc2: forall(c in 1..nbCars)
      demandV[c] == demand[c];

cc3: forall(c in 1..nbCars )
      sum(s in Slots )
       (slot[s]==c) == demand[c];

ct4: forall(o in 1..nbOptions)
 forall(s in 1..(nbSlots-capacity[o][1]+1))
      sum(j in s..(s+capacity[o][1]-1))
      option[o][slot[j]] <= capacity[o][0];

cc5: forall(o in 1..nbOptions,s in Slots)
      setup[o,s] == option[o][slot[s]];

cc6: forall(o in 1..nbOptions)
     forall(i in 1..optionDemand[o])
      sum(s in 1..(nbSlots-i*capacity[o][1]))
      setup[o,s] >=
        optionDemand[o]-i*capacity[o][0];
}
```

CPUT6

```
/****************
* OPL 6.3 Model *
*****************/
using CP;

int  nbCars    = ...;
int   nbOptions = ...;
int   nbSlots   = ...;
range   Cars    = 1..nbCars;
range   Options = 1..nbOptions;
range   Slots   = 1..nbSlots;
int demand[Cars] = ...;
int option[Options,Cars] = ...;
int one[i in 1..nbCars] = 1;
int capacity[Options,0..1] = ...;
int optionDemand[i in Options] = ...;

dvar int slot[Slots] in Cars;
dvar int setup[Options,Slots] in 0..1;
dvar int demandV[1..nbCars] in 1..nbSlots;

subject to{

cc1: pack(demandV, slot, one);

cc2: forall(c in 1..nbCars)
      demandV[c] == demand[c];

ct3: forall(c in 1..nbCars )
      sum(s in Slots)
     (slot[s]==c) == demand[c];

cc4: forall(o in 1..nbOptions)
 forall(s in 1..(nbSlots-capacity[o][1]+1))
     sum(j in s..(s+capacity[o][1]-1))
     option[o][slot[j]] <= capacity[o][0];

cc5: forall(o in 1..nbOptions)
      forall(s in Slots)
       setup[o,s] == option[o][slot[s]];

ct6: forall(o in 1..nbOptions)
   forall( i in 1..optionDemand[o])
   sum(s in 1..(nbSlots-i*capacity[o][1]))
      setup[o,s] >
      optionDemand[o]-i*capacity[o][0];
}
```

CPUT7

```
/****************
* OPL 6.3 Model *
*****************/
using CP;

int  nbCars    = ...;
int   nbOptions = ...;
int   nbSlots   = ...;
range   Cars    = 1..nbCars;
range   Options = 1..nbOptions;
range   Slots   = 1..nbSlots;
int demand[Cars] = ...;
int option[Options,Cars] = ...;
int one[i in 1..nbCars] = 1;
int capacity[Options,0..1] = ...;
int optionDemand[i in Options] = ...;

dvar int slot[Slots] in Cars;
dvar int setup[Options,Slots] in 0..1;
dvar int demandV[1..nbCars] in 1..nbSlots;

subject to{

cc1: pack(demandV, slot, one);

cc2: forall(c in 1..nbCars)
      demandV[c] == demand[c];

ct3: forall(c in 1..nbCars )
     sum(s in Slots )
      (slot[s]==c) == demand[c];

cc4: forall(o in 1..nbOptions)
 forall(s in 1..(nbSlots-capacity[o][1]+1))
     sum(j in s..(s+capacity[o][1]-1))
     option[o][slot[j]] <= capacity[o][0];

cc5: forall(o in 1..nbOptions,s in Slots)
      setup[o,s] != option[o][slot[s]];

cc6: forall(o in 1..nbOptions)
    forall(i in 1..optionDemand[o])
    sum(s in 1..(nbSlots-i*capacity[o][1]))
      setup[o,s] >=
        optionDemand[o]-i*capacity[o][0];
}
```

Glossaire

(k_i, x_i)	donnée de test
$\mathcal{K}$	Ensemble des instances faisables
$\mathcal{R}$	Raffinement unitaire
ϕ^-	Faute négative
ϕ^*	Faute zéro
ϕ^+	Faute positive
$\triangleq$	Définition
$bounds_{f,l,u}(Q)$	Solutions faisables de Q dans un intervalle $[l, u]$
$conf_{all}$	Relation de conformité : toutes les solutions recherchées
$conf_{best}$	Relation de conformité : une solution optimale recherchée
$conf_{bounds}$	Relation de conformité : une solution faisable dans un intervalle $[l, u]$ recherchée
$conf_{one}$	Relation de conformité : une solution recherchée
$f,\ f'$	Fonctions objectif
$nc(\phi, k_i, x_i)$	non-conformité
$optim(Q)$	Solutions optimales de Q
$Proj_X(Q)$	Projection des solutions de Q sur les variables X
$sol(Q)$	Solutions faisables de Q

References

Acree, Allen Troy, Budd, Timothy Alan, DeMillo, Richard A., Lipton, Richard J., & Sayward, Frederick Gerald. 1979. *Mutation analysis.* tech-report GIT-ICS-79/08. Georgia Institute of Technology, Atlanta, Georgia.

Andersen, Henrik Reif, Hadzic, Tarik, Hooker, John N., & Tiedemann, Peter. 2007. A constraint store based on multivalued decision diagrams. *Pages 118–132 of : Proceedings of the 13th international conference on principles and practice of constraint programming - CP07.* Springer-Verlag.

Backofen, Rolf, & Will, Sebastian. 1998. Excluding symmetries in concurrent constraint programming. *Pages 73–87 of : Proceedings of the 4th international conference on principles and practice of constraint programming - CP98.* Springer-Verlag.

Bailey, James, & Stuckey, Peter J. 2005. Discovery of minimal unsatisfiable subsets of constraints using hitting set dualization. *Pages 174–186 of : Practical aspects of declarative languages, 7th international symposium - PADL05.*

Beldiceanu, Nicolas, & Simonis, Helmut. 2011. A constraint seeker : finding and ranking global constraints from examples. *Pages 12–26 of : Proceedings of the 17th international conference on principles and practice of constraint programming - CP11.*

Beldiceanu, Nicolas, Carlsson, Mats, & Rampon, Jean-Xavier. Global constraint catalog 2nd edition. Septembre 2011.

Beldiceanu, Nicolas, Carlsson, Mats, Debruyne, Romuald, & Petit, Thierry. 2005. Reformulation of global constraints based on constraints checkers. *Constraints journal*, **10**(4), 339–362.

Bernot, Gilles, Gaudel, Marie Claude, & Marre, Bruno. 1991. Software testing based on formal specifications : a theory and a tool. *Software engineering journal*, **6**(Novembre), 387–405.

Bessière, Christian, Meseguer, Pedro, Freuder, Eugene C., & Larrosa, Javier. 2002. On forward checking for non-binary constraint satisfaction. *Artificial intelligence*, **141**(October), 205–224.

Carlsson, Mats, & Beldiceanu, Nicolas. 2004. From constraints to finite automata to filtering algorithms. *Pages 94–108 of : Programming languages and systems, 13th european symposium on programming - ESOP04.*

Cellier, Peggy, Ducassé, Mireille, Ferré, Sébastien, & Ridoux, Olivier. 2009. DeLLIS : A data mining process for fault localization. *Pages 432–437 of : International conference software engineering (SEKE).* Knowledge Systems Institute Graduate School.

Charnley, John, Colton, Simon, & Miguel, Ian. 2006. Automatic generation of implied constraints. *Pages 73–77 of : Proceeding of the 17th european conference on artificial intelligence - ECAI06.* Riva del Garda, Italy : IOS Press.

Cheng, B. M. W., Lee, J. H. M., & Wu, J.C.K. 1996. Speeding up constraint propagation by redundant modeling. *Pages 91–103 of : Proceedings of the 2nd international conference on principles and practice of constraint programming - CP96.* Springer-Verlag.

Chiu, C. K., Chou, C. M., Lee, J. H. M., Leung, Ho-fung, & Leung, Y. W. 2002. A constraint-based interactive train rescheduling tool. *Constraints journal*, **7**(April), 167–198.

CHOCO-Team. 2010. *choco : an open source java constraint programming library.* Research report 10-02-INFO. École des Mines de Nantes.

Cleve, Holger, & Zeller, Andreas. 2005. Locating causes of program failures. *Pages 342–351 of : Proceedings of the 27th international conference on software engineering - ICSE05.* New York, NY, USA : ACM.

Cohen, David A., Jeavons, Peter, Jefferson, Christopher, Petrie, Karen E., & Smith, Barbara M. 2006. Constraint symmetry and solution symmetry. *In : Proceedings AAAI06.*

Collavizza, Hélène, Vinh, Nguyen Le, Rueher, Michel, Devulder, Samuel, & Gueguen, Thierry. 2011. A dynamic constraint-based bmc strategy for generating counterexamples. *Pages 1633–1638 of : Proceedings of the 2011 acm symposium on applied computing.* SAC11. New York, NY, USA : ACM.

Collavizza, Hélène, Rueher, Michel, & Hentenryck, Pascal Van. 2008. Cpbpv : A constraint-programming framework for bounded program verification. *Pages 327–341 of : Proceedings of the 14th international conference on principles and practice of constraint programming - CP08.* LNCS 5202. Springer-Verlag.

Cotta, Carlos, Dotú, Iván, Fernández, Antonio J., & Hentenryck, Pascal. 2007. Local search-based hybrid algorithms for finding golomb rulers. *Constraints journal*, **12**(Septembre), 263–291.

Cousot, Patrick, & Cousot, Radhia. 1977. Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. *Pages 238–252 of : Proceedings of the 4th acm sigact-sigplan symposium on principles of programming languages.* POPL77. New York, NY, USA : ACM.

Dahmen, Michael. 1991. A debugger for constraints in prolog. *Pages 118–133 of : Technical report ECRC-91-11, ECRC.*

Debroy, Vidroha, & Wong, W. Eric. 2010 (April). Using mutation to automatically suggest fixes for faulty programs. *Pages 65–74 of : 3rd international conference on software testing, verification and validation - ICST10.*

Debruyne, Romuald, & Bessiére, Christian. 2001. Domain filtering consistencies. *Journal of artificial intelligence research*, **14**(May), 205–230.

Dechter, Rina. 1990. Enhancement schemes for constraint processing : backjumping, learning, and cutset decomposition. *Artificial intelligence*, **41**(Janvier), 273–312.

DeMillo, Richard A., Lipton, Richard J., & Sayward, Frederick Gerald. 1978. Hints on test data selection : Help for the practicing programmer. *Computer*, **11**(4), 34–41.

Deransart, Pierre, Hermenegildo, Manuel V., & Maluszynski, Jan (eds). 2000a. *Analysis and visualization tools for constraint programming, constrain debugging (DiSCiPl project)*. Lecture Notes in Computer Science, vol. 1870. Springer.

Deransart, Pierre, Hermenegildo, Manuel V., & Maluszynski, Jan (eds). 2000b. *Analysis and visualization tools for constraint programming, constrain debugging (DiSCiPl project)*. Lecture Notes in Computer Science, vol. 1870. Springer.

Deransart, Pierre, Hermenegildo, Manuel V., & Maluszynski, Jan (eds). 2000c. *Analysis and visualization tools for constraint programming, constraint debugging (DiSCiPl project)*. Lecture Notes in Computer Science, vol. 1870. Springer.

Dijkstra, Edsger Wybe. 1997. *A discipline of programming*. 1st edn. Upper Saddle River, NJ, USA : Prentice Hall PTR.

Dincbas, Mehmet, Hentenryck, Pascal Van, Simonis, Helmut, Aggoun, Abderrahmane, & Herold, Alexander. 1988. The chip system : Constraint handling in prolog. *Pages 774–775 of :* Lusk, Ewing, & Overbeek, Ross (eds), *9th international conference on automated deduction*. Lecture Notes in Computer Science, vol. 310. Springer Berlin / Heidelberg. 10.1007/BFb0012892.

Dooms, Grégoire, Van Hentenryck, Pascal, & Michel, Laurent. 2007. Model-driven visualizations of constraint-based local search. *Pages 271–285 of : Proceedings of the 13th international conference on principles and practice of constraint programming - CP07*. Providence, RI, USA : Springer-Verlag.

Ducassé, Mireille. 1986. OPIUM : un outil de trace sophistiqué pour prolog. *Pages 281–292 of : Séminaire programmation en logique - SPLT86*.

Duran, Joe W., & Ntafos, Simeon C. 1984. An evaluation of random testing. *Ieee trans. software engineering*, **10**(4), 438–444.

Fages, François, Soliman, Sylvain, & Coolen, Rémi. 2004. Clpgui : A generic graphical user interface for constraint logic programming. *Constraints journal*, **9**(4), 241–262.

Felfernig, Alexander, & Burke, Robin D. 2008. Constraint-based recommender systems : technologies and research issues. *Pages 3 :1–3 :10 of : Proceedings of the 10th international conference on electronic commerce*. ICEC '08. New York, NY, USA : ACM.

Ferrand, Gérard, Lesaint, Willy, & Tessier, Alexandre. 2003. Towards declarative diagnosis of constraint programs over finite domains. *Corr*, **cs.SE/0309032**.

Flener, Pierre, Pearson, Justin, Agren, Magnus, Carlos, Garcia-Avello, Celiktin, Mete, & Dissing, Søren. 2007a. Air-traffic complexity resolution in multi-sector planning. *Journal of air transport management*, **13**(6), 323 – 328.

Flener, Pierre, Pearson, Justin, Reyna, Luis G., & Sivertsson, Olof. 2007b. Design of financial cdo squared transactions using constraint programming. *Constraints journal*, **12**(June), 179–205.

Flener, Pierre, Pearson, Justin, & Sellmann, Meinolf. 2009. Static and dynamic structural symmetry breaking. *Annals of mathematics and artificial intelligence*, **57**(1), 37–57.

Fourer, Robert, Gay, David M., & Kernighan, Brian W. 2003. *AMPL : A modeling language for mathematical programming.* Second edn. Pacific Grove, California : Thomson/Brooks/Cole.

Gent, Ian P., & Smith, Barbara M. 2000. Symmetry breaking in constraint programming. *Pages 599–603 of : Proceeding of the 11th european conference on artificial intelligence - ECAI00.* IOS Press.

Ginsberg, Matthew L. 1993. Dynamic backtracking. *Journal of artificial intelligence research*, **1**(Aout), 25–46.

Goodenough, John B., & Gerhart, Susan L. 1975. Toward a theory of test data selection. *Pages 493–510 of : Proceedings of the international conference on reliable software.* New York, NY, USA : ACM.

Gotlieb, Arnaud. 2009. Tcas software verification using constraint programming. *The knowledge engineering review.* Accepted for publication.

Gourlay, John S. 1983. A mathematical framework for the investigation of testing. *Ieee trans. software engineering*, **9**(Novembre), 686–709.

Hentenryck, Pascal Van, & Carillon, Jean-Philippe. 1988. Generality versus specificity : An experience with ai and or techniques. *Pages 660–664 of : Proceedings AAAI88.*

Hnich, Brahim, Richardson, Julian, & Flener, Pierre. 2003 (Mar.). *Towards automatic generation and evaluation of implied constraints.* Tech. rept. 2003-014. Department of Information Technology, Uppsala University.

Hoda, Samid, Van Hoeve, Willem-Jan, & Hooker, J. N. 2010. A systematic approach to mdd-based constraint programming. *Pages 266–280 of : Proceedings of the 16th international conference on principles and practice of constraint programming - CP10.* Berlin, Heidelberg : Springer-Verlag.

Holland, Alan, & O'Sullivan, Barry. 2005. Robust solutions for combinatorial auctions. *Pages 183–192 of : Acm conference on electronic commerce (ec-2005).*

Howden, William E. 1976. Reliability of the path analysis testing strategy. *Software engineering journal*, **2**(May), 208–215.

IBM, Corp. 2009. *Ibm ilog opl migration to version 6.x.* Technical report.

ILOG. 2003. *Ilog solver 6.0 debugger user's manual.* Technical report.

Jeffrey, Dennis, Feng, Min, Gupta, Neelam, & Gupta, Rajiv. 2009. Bugfix : A learning-based tool to assist developers in fixing bugs. *Pages 70–79 of : Icpc.*

Jiang, Yuejun, Richards, Thomas, & Richards, Barry. 1994. Nogood backmarking with min-conflict repair in constraint satisfaction and optimization. *Pages 21–39 of : Proceedings of the second international workshop on principles and practice of constraint programming.* London, UK : Springer-Verlag.

Jones, James A., & Harrold, Mary Jean. 2005. Empirical evaluation of the tarantula automatic fault-localization technique. *Pages 273–282 of : Proceedings of the 20th ieee/acm international conference on automated software engineering.* ASE '05. New York, NY, USA : ACM.

Jones, James A., Harrold, Mary Jean, & Stasko, John T. 2002. Visualization of test information to assist fault localization. *Pages 467–477 of : Proceedings of the 24th international conference on software engineering - ICSE02.*

Junker, Ulrich. 2004. Quickxplain : Preferred explanations and relaxations for over-constrained problems. *Pages 167–172 of : Proceedings AAAI04.*

Junker, Ulrich, & Vidal, David. 2008. Air traffic flow management with ilog cp optimizer. *In : International workshop on constraint programming for air traffic control and management.* 7th EuroControl Innovative Research Workshop and Exhibition (INO'08).

Jussien, Narendra, & Barichard, Vincent. 2000. The palm system : explanation-based constraint programming. *Pages 118–133 of : In proceedings of trics : Techniques for implementing constraint programming systems, a post-conference workshop of cp 2000.*

Kadioglu, Serdar, & Sellmann, Meinolf. 2010. Grammar constraints. *Constraints journal*, **15**(1), 117–144.

Langevine, Ludovic, Deransart, Pierre, Ducassé, Mireille, & Jahier, Erwan. 2001. Prototyping clp(fd) tracers : a trace model and an experimental validation environment. *In : Wlpe.*

Laurière, Jean-Louis. 1976. *Un langage et un programme pour énoncer et résoudre des problèmes combinatoires.* Ph.D. thesis.

Lazaar, Nadjib. 2011. CPTEST : A framework for the automatic fault detection, localization and correction of constraint programs. *In : In proceedings of cstva : Constraints in software testing, verification and analysis, a post-conference workshop of ICST11.*

Lazaar, Nadjib, Gotlieb, Arnaud, & Lebbah, Yahia. 2009 (Juin). Vers une théorie du test des programmes à contraintes. *In : Proceedings of the 5ème journées francophones de programmation par contraintes - JFPC09.*

Lazaar, Nadjib, Gotlieb, Arnaud, & Lebbah, Yahia. 2010a (Oct.). Fault localization in constraint programs. *In : Proc. of the 2010 ieee international conference on tools with artificial intelligence, ictai'10.*

Lazaar, Nadjib, Gotlieb, Arnaud, & Lebbah, Yahia. 2010b (Septembre). On testing constraint programs. *In : Proceedings of the 16th international conference on principles and practice of constraint programming - CP10.*

Lazaar, Nadjib, Gotlieb, Arnaud, & Lebbah, Yahia. 2011a. A framework for the automatic correction of constraint programs. *Page Forthcomming of : 4th international conference on software testing, verification and validation - ICST11.* Berlin, Germany : IEEE Computer Society.

Lazaar, Nadjib, Aribi, Nourredine, Gotlieb, Arnaud, & Lebbah, Yahia. 2011b (Oct.). *Negation for Free !* Rapport de recherche.

Marriott, Kim, & Stuckey, Peter J. 1998. *Programming with constraints : an introduction.* MIT Press.

Marriott, Kim, Nethercote, Nicholas, Rafeh, Reza, Stuckey, Peter J., Garcia De La Banda, Maria, & Wallace, Mark. 2008. The design of the zinc modelling language. *Constraints journal*, **13**(3), 229–267.

McSherry, David, & Aha, David W. 2007. The ins and outs of critiquing. *Pages 962–967 of : Proceedings of the 20th international joint conference on artifical intelligence.* San Francisco, CA, USA : Morgan Kaufmann Publishers Inc.

Meier, Micha. 1995. Debugging constraint programs. *Pages 204–221 of : Proceedings of the first international conference on principles and practice of constraint programming.* London, UK : Springer-Verlag.

Mitchell, Tom. 1997. *Machine learning, chapter 13 : Reinforcement learning.* Cambridge, MA, USA : McGraw Hill.

Nadel, Bernard A. 1990. Representation selection for constraint satisfaction : A case study using n-queens. *Ieee expert : Intelligent systems and their applications*, **5**(June), 16–23.

Parrello, Bruce D., & Kabat, Waldo C. 1986. Job-shop scheduling using automated reasoning : a case study of the car-sequencing problem. *J. autom. reason.*, **2**(1), 1–42.

Pesant, Gilles. A regular language membership constraint for finite sequences of variables. *In : Proceedings of the 10th international conference on principles and practice of constraint programming - CP04.*

Puget, Jean-François. 2006. An efficient way of breaking value symmetries. *In : Proceedings AAAI06.*

Puget, Jean-Francois. 2005. Automatic detection of variable and value symmetries. *Pages 475–489 of : Cp.*

Quimper, Claude-Guy, & Walsh, Toby. Global grammar constraints. *In : Proceedings of the 12th international conference on principles and practice of constraint programming - CP06.* Springer-Verlag.

Rankin, William T. 1993. *Optimal golomb rulers : An exhaustive parallel search implementation.* M.Phil. thesis, Duke University, Durham.

Régin, Jean-Charles. 1994. A filtering algorithm for constraints of difference in csps. *Pages 362–367 of : Proceedings of the twelfth national conference on artificial intelligence (vol. 1).* Proceedings AAAI94. Menlo Park, CA, USA : American Association for Artificial Intelligence.

Régin, Jean-Charles. 2011. *Hybrid optimization. book chapter : Global constraints : a survey.* New York, NY, USA : Springer.

Rocquencourt, Inria, des Mines de Nantes, Ecole, de Rennes, Insa, d'Orléans, Université, Cosytec, & ILOG (eds). 2001. *Outils d'analyse dynamique pour la programmation par contraintes (oadymppac).* Vol. 1870. Projet RNTL.

Rodriguez, Joaquin. 2000. Empirical study of a railway traffic management constraint programming model. WIT Press, Southampton, ROYAUME-UNI (2000) (Monographie).

Rodriguez, Joaquin, & Kermad, L. 1998. Constraint programming for real-time train circulation management problem in railway nodes. *Comprail'98.*

Rossi, Francesca, Beek, Peter van, & Walsh, Toby. 2006. *Handbook of constraint programming (foundations of artificial intelligence). chapter 10 : Symmetry in constraint programming.* New York, NY, USA : Elsevier Science Inc.

Sahinidis, Nikolaos V. 2002. *Convexification and global optimization in continuous and mixed-integer nonlinear programming.* Kluwer Academic Publishers Group.

Sanlaville, Eric. 2005. Bulletin de la société française de recherche opérationnelle et d'aide à la decision. ROADEF'05. ACM.

Schrijver, Alexander. 1986. *Theory of linear and integer programming.* New York, NY, USA : John Wiley & Sons, Inc.

Schulte, Christian. 2002. *Programming constraint services.* Lecture Notes in Artificial Intelligence, vol. 2302. Springer-Verlag.

Shapiro, Ehud Y. 1983. *Algorithmic program debugging.* Cambridge, MA, USA : MIT Press.

Sifakis, Joseph. 1982. A unified approach for studying the properties of transition systems. *Theor. comput. sci.*, **18**, 227–258.

Simonis, Helmut, Davern, Paul, Feldman, Jacob, Mehta, Deepak, Quesada, Luis, & Carlsson, Mats. 2010. A generic visualization platform for cp. *Pages 460–474 of : Proceedings of the 16th international conference on principles and practice of constraint programming - CP10.* Springer-Verlag.

Smith, Barbara M., Stergiou, Kostas, & Walsh, Toby. 1999. Modelling the golomb ruler problem.

Van Hentenryck, P. 1999. *The opl optimization programming language.* Cambridge, MA, USA : MIT Press.

Van Hentenryck, Pascal. 1989a. *Constraint satisfaction in logic programming.* Cambridge, MA, USA : MIT Press.

Van Hentenryck, Pascal. 1989b. *Constraint satisfaction in logic programming.* Cambridge, MA, USA : MIT Press.

Van Hentenryck, Pascal, & Michel, Laurent. 2005. *Constraint-based local search.* The MIT Press.

Verfaillie, Gerard, Lemaître, Michel, & Schiex, Thomas. 1996. Russian doll search for solving constraint optimization problems. *Pages 181–187 of : Proceedings AAAI96.*

Vessey, Iris. 1985. Expertise in debugging computer programs : A process analysis. *International journal of man-machine studies*, **23**(5), 459–494.

Wei, Yi, Pei, Yu, Furia, Carlo A., Silva, Lucas S., Buchholz, Stefan, Meyer, Bertrand, & Zeller, Andreas. 2010. Automated fixing of programs with contracts. *Pages 61–72 of : Proceedings of the 19th international symposium on software testing and analysis.* ISSTA '10. New York, NY, USA : ACM.

Weimer, Westley, Forrest, Stephanie, Le Goues, Claire, & Nguyen, ThanhVu. 2010. Automatic program repair with evolutionary computation. *Commun. acm*, **53**(5), 109–116.

Weyuker, Elaine J., & Ostrand, Thomas J. 1980. Theories of program testing and the application of revealing subdomains. *Ieee trans. software engineering*, **6**(3), 236–246.

Winston, W. L. 2004. *Operations research ; applications and algorithms.* Brooks/Cole.

Zeller, Andreas. 2002. Isolating cause-effect chains from computer programs. *Pages 1–10 of : Proceedings of the 10th acm sigsoft symposium on foundations of software engineering.* SIGSOFT '02/FSE-10. New York, NY, USA : ACM.

Table des figures

List of Algorithms

Liste des tableaux

Résumé

Le développement des langages de modélisation des programmes à contraintes a eu un grand impact dans le monde industriel. Des langages comme OPL (Optimization Programming Language) de IBM Ilog, Comet de Dynadec, Sicstus Prolog, Gecode, ainsi que d'autres, proposent des solutions robustes aux problèmes du monde réel. De plus, ces langages commencent à être utilisés dans des applications critiques comme la gestion et le contrôle du trafic aérien (Flener *et al.*, 2007a; Junker & Vidal, 2008), le e-commerce (Holland & O'Sullivan, 2005) et le développement de programmes critiques (Collavizza *et al.*, 2008; Gotlieb, 2009). D'autre part, il est connu que tout processus de développement logiciel effectué dans un cadre industriel inclut impérativement une phase de test, de vérification formelle et/ou de validation. Par ailleurs, les langages de programmation par contraintes (PPC) ne connaissent pas d'innovations majeures en termes de vérification et de mise au point. Ceci ouvre la voie à des recherches orientées vers les aspects génie logiciel dédiés à la PPC. Notre travail vise à poser les jalons d'une théorie du test des programmes à contraintes pour fournir des outils conceptuels et des outils pratiques d'aide à cette vérification. Notre approche repose sur des hypothèses quant au développement et au raffinement dans un langage de PPC. Il est usuel de démarrer à partir d'un modèle simple et très déclaratif, une traduction fidèle de la spécification du problème, sans accorder un intérêt à ses performances. Par la suite, ce modèle est raffiné par l'introduction de contraintes redondantes ou reformulées, l'utilisation de structures de données optimisées, de contraintes globales, de contraintes qui cassent les symétries, etc. Nous pensons que l'essentiel des fautes introduites est compris dans ce processus de raffinement. Le travail majeur présenté dans la présente thèse est la définition d'un cadre de test qui établit des règles de conformité entre le modèle initial déclaratif et le programme optimisé dédié à la résolution d'instances de grande taille. Par la suite, nous proposons un cadre conceptuel pour la mise-au-point de ce type de programmes avec une méthodologie pour la localisation et la correction automatique des fautes. Nous avons développé un environnement de test, nommé CPTEST, pour valider les solutions proposées, sur des problèmes académiques du monde de la PPC ainsi qu'un problème du monde réel qui est le problème d'ordonnancement des véhicules.

Abstract

The success of several constraint-based modelling languages such as OPL (Optimization programming Language) of IBM Ilog, COMET of Dynadec, Sicstus Prolog, Gecode, appeals for better software engineering practices, particularly in the testing phase. These languages aim at solving industrial combinatorial problems that arise in optimization, planning, or scheduling. Recently, a new trend has emerged that propose also to use CP programs to address critical applications in e-Commerce (Holland & O'Sullivan, 2005), air-traffic control and management (Flener *et al.*, 2007a; Junker & Vidal, 2008), or critical software development (Collavizza *et al.*, 2008; Gotlieb, 2009) that must be thoroughly verified before being used in applications. While constraint program debugging drew the attention of many researchers, few supports in terms of software engineering and testing have been proposed to help verify this kind of programs. In the present thesis, we define a testing theory for constraint programming. For that, we propose a general framework of constraint program development which supposes that a first declarative and simple constraint model is available from the problem specifications analysis. Then, this model is refined using classical techniques such as constraint reformulation, surrogate, redundant, implied and global constraint addition, or symmetry-breaking to form an improved constraint model that must be tested before being used to address real-sized problems. We think that most of the faults are introduced in this refinement step and propose a process which takes the first declarative model as an oracle for detecting non-conformities and derive practical test purposes from this process. Therefore, we enhance the proposed testing framework to introduce a methodology for an automatic tuning with fault localization and correction in constraint programs. We implemented these approaches in a new tool called CPTEST that was used to automatically detect, localize and correct faults on classical benchmark programs and real-world CP problem : the car-sequencing problem.