cptest4choco : test et mise au point des modèles à contraintes

Nadjib Lazaar[1] **Arnaud Gotlieb** [2] **Yahia Lebbah** [3]

[1] LIRMM- Université de Montpellier 2, France
[2] Certus Software V&V Center, Simula Research Laboratory, Oslo, Norway
[3] Laboratoire LITIO, Université d'Oran, ,B.P. 1524 EL-M'Naouar, 31000 Oran, Algérie

lazaar@lirmm.fr arnaud@simula.no ylebbah@gmail.com

Résumé

Les programmes à contraintes écrits dans un langage de haut-niveau (e.g., OPL, COMET, ZINC ou CHOCO) sont de plus en plus utilisés dans des domaines critiques qui nécessitent une phase de test et de validation. Dans ce papier, nous décrivons un environnement logiciel de test et de mise-au-point, nommé *cptest4choco*, dédié aux programmes écrits en CHOCO. Cet environnement implémente des approches de détection, de localisation et de correction automatique de fautes présentes dans les modèles CHOCO. Les résultats obtenus montrent l'efficacité des approches de test pour la mise-au-point des modèles à contraintes écrits en CHOCO.

Abstract

Constraint programs, such as those written in high-level constraint modelling languages (e.g., OPL, COMET, ZINC or CHOCO), are more and more used in critical programs which require to be thoroughly tested and corrected to prevent catastrophic outcomes. This paper presents a software environment for constraint programs written in CHOCO, called *cptest4choco*, and different approaches for an automatic CP tuning (including fault detection, localization and correction). This environment takes into account the specificities of the software development process of CP programs as well as their typical faults. We have obtained experimental results on well-known constraint programs written in CHOCO, showing that their tuning is effective and illustrate the efficiency of *cptest4choco* functionalities.

1 Introduction

Ces dernières années, les langages de modélisation en programmation par contraintes (PPC) [14] ont connu un réel développement. Ils ont été conçus avec un haut niveau d'abstraction permettant une grande expressivité. Des langages comme OPL (Optimization Programming Language)[16],COMET [17], ZINC [12] ainsi que CHOCO [1] proposent des solutions robustes à des problèmes réels. De plus, ces langages commencent à être utilisés dans des applications critiques comme la gestion et le contrôle du trafic aérien [3, 7], le e-commerce [6] et le développement de programmes critiques [1, 5].

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. A 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 :

- En PPC, un problème est modélisé avec un ensemble de contraintes non ordonné. Il n'y a donc pas de notion de séquence, comme on en trouve dans les programmes conventionnels.
- Le flot de contrôle dans un programme conventionnel est guidé par les données alors qu'en PPC il est guidé par les contraintes.
- 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

1. www.emn.fr/z-info/choco-solver/

contraintes.

Dans un processus de développement des programmes à contraintes, tel qu'il semble être pratiqué dans les milieux industriels, 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 test et la révision de contraintes dans un but de correction sont des tâches manuelles qui nécessitent une grande expertise.

Dans cet article, nous présentons le développement d'une bibliothèque de test et de mise-au-point basé sur un cadre formel de test des programmes à contraintes [9, 11]. Cette bibliothèque, nommée *cptest4choco*, permet d'effectuer de la détection, de la localisation et de la correction automatique des fautes des programmes à contraintes écrit en CHOCO. Étant donnée une spécification d'un problème, la traduction de cette spécification en un premier modèle à contraintes, noté MO pour *Modèle-Oracle*, représente notre référence de test qui permettra de détecter des fautes introduites dans un *programme à contraintes sous test* (i.e., noté CPUT pour *Constraint Program Under Test*). A l'instar du MO, le CPUT est dédié à résoudre des instances difficiles du problème. La distance et la divergence (i.e., ensemble de non-conformités) entre le MO et le CPUT sont dues aux fautes introduites durant le processus de raffinement.

Dans [8], nous avons augmenté le cadre de test avec une approche de localisation de faute qui retourne un ensemble de contraintes susceptibles de contenir des fautes. Dans [10], nous avons proposé une approche qui permet de calculer automatiquement des corrections possibles.

La bibliothèque *cptest4choco* nous a permis de faire une étude expérimentale sur des problèmes académiques : règles de golomb et les n-reines). L'objectif de cette étude est de montrer qu'il est possible de tester et/ou corriger automatiquement des programmes à contraintes.

Le reste de cet article est organisé comme suit. La section 2 présente le contexte et illustre nos approches sur l'exemple des règles de Golomb. La section 3 passe en revue la partie théorique du cadre de test et de mise-au-point en PPC. La section 4 présente la bibliothèque de test *cptest4choco*. En section 5, nous présentons quelques résultats expérimentaux, enfin, la section 6 conclut l'article.

2 Exemple Illustratif

Dans cette section, nous illustrons nos approches de test et de mise-au-point en PPC sur les règles de Golomb. 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) [15, 2]. La formulation de ce problème est assez simple, en revanche, la résolution est très couteuse sur des instances qui semblent être accessibles (e.g., règles de 15 marques) [13]. Une règle de Golomb est définie comme un ensemble de m entiers $0 = x_1 < x_2 < ... < x_m$ tel que les $m(m-1)/2$ distances $\{x_j - x_i | 1 \leq i < j \leq m\}$, sont différentes. Une telle règle d'ordre m est dite de longueur x_m. L'objectif est de trouver une règle de longueur minimale.

```
/******************
 *   Model-Oracle  *
 ******************/

int m = size;
MO = new CPModel();
// variables
IntegerVariable[] marks = Choco.makeIntVarArray("mark", m, 0, m*m);
IntegerVariable length = Choco.makeIntVar("length", 1, m*m, Options.V_OBJECTIVE);
// Constraints
MO.addConstraint(Choco.eq(length, marks[m-1]));                          //c1

for (int i = 0; i < m-1; i++)
    MO.addConstraint(Choco.lt(marks[i], marks[i+1]));                    //c2


for (int i = 0; i < m-1; i++)                                            //c3
    for (int j = i+1; j < m; j++)
        for (int k = 0; k < m-1; k++)
            for (int l = k+1; l < m; l++)
                if(i!=k || j!=l)
    MO.addConstraint(Choco.neq(Choco.minus(marks[j],marks[i]),
                     Choco.minus(marks[l],marks[k])));
```

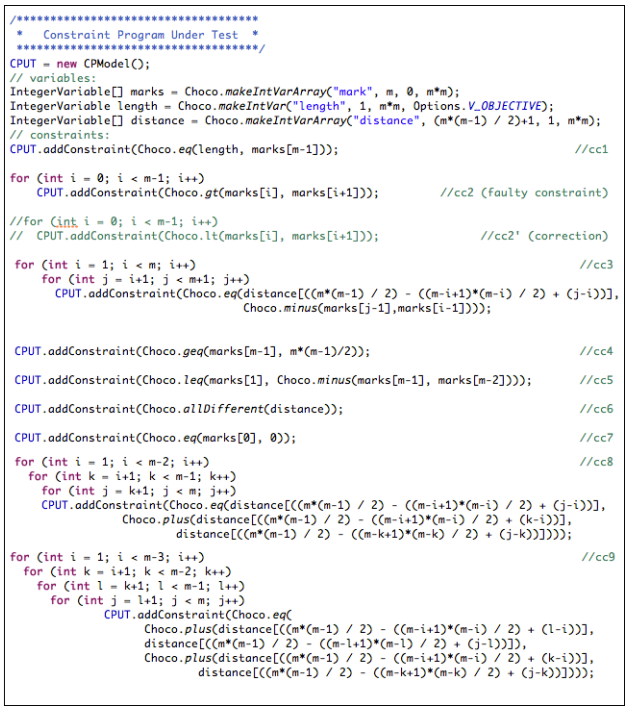
FIGURE 1 – MO for Golomb rulers written in CHOCO.

Un premier Modèle déclaratif du problème en CHOCO est donné dans la Figure 1. Dans notre cadre de test, ce modèle représente le MO. Nous obtenons par la suite un modèle optimisé suite à un processus de raffinement incluant l'ajout de contraintes globales, reformulation de contraintes, cassure de symétrie, ajout de contraintes redondantes, etc. Ce modèle représente le CPUT. Un des CPUT possibles, écrit en CHOCO, est donné dans la Figure 2. Il est incontestable que le modèle de la Figure 1 résolve bien le problème des règles de Golomb. Ce qui n'est pas aussi évident pour le modèle de la Figure 2.

Prenons l'instance $m = 6$. *cptest4choco* retourne l'état *unsat* pour le CPUT de la Figure 2. Ce qui révèle la présence d'une faute dans le CPUT en question. En phase de localisation, *cptest4choco* retourne `cc2` comme étant une contrainte suspecte. En effet, une mauvaise formulation de cette contrainte réduit l'ensemble des solutions du CPUT à vide. *cptest4choco* propose une correction qui consiste à remplacer `cc2` par `cc2'`.

3 Contexte

Comme nous l'avons mentionné, nous considérons le premier modèle déclaratif comme oracle de test (i.e., MO). Le MO représente l'ensemble des solutions (au moins une) du problème et est strictement conforme à la spécification de départ.



```
/***********************************
 *   Constraint Program Under Test  *
 ***********************************/
CPUT = new CPModel();
// variables:
IntegerVariable[] marks = Choco.makeIntVarArray("mark", m, 0, m*m);
IntegerVariable length = Choco.makeIntVar("length", 1, m*m, Options.V_OBJECTIVE);
IntegerVariable[] distance = Choco.makeIntVarArray("distance", (m*(m-1) / 2)+1, 1, m*m);
// constraints:
CPUT.addConstraint(Choco.eq(length, marks[m-1]));                                  //cc1

for (int i = 0; i < m-1; i++)
    CPUT.addConstraint(Choco.gt(marks[i], marks[i+1]));          //cc2 (faulty constraint)

//for (int i = 0; i < m-1; i++)
//  CPUT.addConstraint(Choco.lt(marks[i], marks[i+1]));               //cc2' (correction)

 for (int i = 1; i < m; i++)                                                       //cc3
     for (int j = i+1; j < m+1; j++)
       CPUT.addConstraint(Choco.eq(distance[((m*(m-1) / 2) - ((m-i+1)*(m-i) / 2) + (j-i))],
                                   Choco.minus(marks[j-1],marks[i-1])));


 CPUT.addConstraint(Choco.geq(marks[m-1], m*(m-1)/2));                             //cc4

 CPUT.addConstraint(Choco.leq(marks[1], Choco.minus(marks[m-1], marks[m-2])));     //cc5

 CPUT.addConstraint(Choco.allDifferent(distance));                                 //cc6

 CPUT.addConstraint(Choco.eq(marks[0], 0));                                        //cc7

 for (int i = 1; i < m-2; i++)                                                     //cc8
   for (int k = i+1; k < m-1; k++)
     for (int j = k+1; j < m; j++)
     CPUT.addConstraint(Choco.eq(distance[((m*(m-1) / 2) - ((m-i+1)*(m-i) / 2) + (j-i))],
                Choco.plus(distance[((m*(m-1) / 2) - ((m-i+1)*(m-i) / 2) + (k-i))],
                           distance[((m*(m-1) / 2) - ((m-k+1)*(m-k) / 2) + (j-k))])));

for (int i = 1; i < m-3; i++)                                                      //cc9
  for (int k = i+1; k < m-2; k++)
    for (int l = k+1; l < m-1; l++)
      for (int j = l+1; j < m; j++)
            CPUT.addConstraint(Choco.eq(
                  Choco.plus(distance[((m*(m-1) / 2) - ((m-i+1)*(m-i) / 2) + (l-i))],
                  distance[((m*(m-1) / 2) - ((m-l+1)*(m-l) / 2) + (j-l))]),
                  Choco.plus(distance[((m*(m-1) / 2) - ((m-i+1)*(m-i) / 2) + (k-i))],
                          distance[((m*(m-1) / 2) - ((m-k+1)*(m-k) / 2) + (j-k))]))));
```

FIGURE 2 – CPUT pour le problème des règles de Golomb écrit en CHOCO.

Le programme à contraintes sous test, le CPUT, est sensé être conforme aux solutions du MO. La notation $sol(\text{MO})$ (respectivement $sol(\text{CPUT})$) représente l'ensemble des solutions du MO (respectivement CPUT).

Definition 1 (*relation de conformité)*

$$\text{CPUT } conf \text{ MO} \Leftrightarrow sol(\text{CPUT}) \neq \emptyset \wedge sol(\text{CPUT}) \subseteq sol(\text{MO})$$

3.1 Modèle de faute

Dans notre cadre de test, nous nous intéressons aux fautes qui correspondent à des états de non-conformité. Les autres fautes de type syntaxique sur la formulation des contraintes et/ou les fautes présentent au niveau du solveur de contraintes, ne sont pas prises en compte par notre approche.

Notre modèle de fautes est défini à l'aide de trois types de fautes :

Definition 2 (Faute positive ϕ^+) *Une faute positive sur un* CPUT *par rapport à son Modèle-Oracle* MO*, notée ϕ^+, est une faute qui ajoute des solutions au* CPUT *:*

$$\phi^+ \triangleq sol(\text{CPUT}) \backslash sol(\text{MO}) \neq \emptyset$$

Definition 3 (Faute négative ϕ^-) *Une faute négative sur un* CPUT *par rapport à son Modèle-Oracle* MO*, notée ϕ^-, est une faute qui supprime des solutions du* CPUT *:*

$$\phi^- \triangleq sol(\text{MO}) \backslash sol(\text{CPUT}) \neq \emptyset$$

Definition 4 (faute zéro ϕ^*) *Une faute zéro sur un* CPUT *par rapport à son Modèle-Oracle* MO*, notée ϕ^*, est une faute qui réduit l'ensemble des solutions du* CPUT *à vide :*

$$\phi^* \triangleq sol(\text{CPUT}) = \emptyset$$

En utilisant ces trois définitions, une non-conformité entre le CPUT et son MO est tirée d'une faute positive, négative ou de type zéro.

La définition 1 est une définition générique de la relation de conformité. Il est à noter que cette notion de conformité entre ces deux entités peut être plus spécifique en prenant en compte 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 d'une solution optimale).

En conséquence, une faute négative (ϕ^-) n'affecte pas la conformité si nous abordons la classe de problèmes qui cherche une solution, puisque l'objectif est d'atteindre au moins une solution acceptable via le Modèle-Oracle. Ainsi, tout raffinement qui peut supprimer des solutions (mais non pas toutes), comme la cassure de symétrie, reste acceptable.

3.2 Détection de faute

La preuve de conformité sur l'ensemble des instances du problème est un problème indécidable dans le cas général. Ainsi, nous avons proposé dans [9] un processus de test qui permet de détecter les non-conformités. Etant donnée une instance, une non-conformité est soit une solution du CPUT qui n'est pas une solution du MO, soit un état *unsat* du CPUT (i.e., $sol(\text{CPUT}) = \emptyset$). La détection systématique des non-conformités peut être atteinte en combinant la négation des contraintes du MO avec les contraintes du CPUT (i.e., $\text{CPUT} \wedge \neg C_i$ avec $C_i \in$ MO). Une mauvaise formulation d'une contrainte du CPUT peut ajouter/supprimer des solutions. Dans [9], nous avons proposé un algorithme, nommé $one\_negated$ donné ci-dessous, qui retourne des non-conformités si ces dernières existent. Il est à noter qu'une non-conformité entre un CPUT et son MO est une solution x retournée par un processus de test (du fait d'une faute ϕ^+ ou ϕ^-)), ou un état $unsat$ (du fait d'une faute ϕ^*).

Algorithm 1: one_negated(CPUT, MO)

foreach $C_i \in$ MO **do**
  $x \leftarrow solve(\text{CPUT} \wedge \neg C_i)$
  if x **then return** x
return $\emptyset$

3.3 Localisation de faute

Dans [8], nous avons proposé une approche qui permet d'expliquer une faute dans un CPUT. Cette approche de localisation retourne une explication formée de contraintes suspectes. L'algorithme $locate$, donné dans cette section, permet de calculer cet ensemble de contraintes suspectes. L'approche est basée sur la définition suivante. Etant donné un CPUT $= \{C_1, C_2...C_n\}$ qui n'est pas conforme à son MO avec la présence d'une faute ϕ, un *ensemble suspect* est un sous ensemble de contraintes suspectes qui explique la faute ϕ :

Definition 5 (contrainte suspecte) $C_i \in$ CPUT *est suspecte ssi :*
$\phi^+, \phi^* : sol(\text{MO}) \cap sol(\text{CPUT} \backslash C_i) \neq \emptyset$
$\phi^- \quad : sol(\text{MO} \wedge \neg C_i) \neq \emptyset$

Algorithm 2: locate(CPUT, MO, nc)

```
1  SuspiciousSet ← ∅
2  if (φ+ ∨ φ*) then
3      foreach C_i ∈ CPUT do
4          if sol(MO ∧ CPUT\C_i) ≠ ∅ then
5              SuspiciousSet ← SuspiciousSet ∪ {C_i}
6  else
7      foreach C_i ∈ CPUT do
8          if sol(MO ∧ ¬C_i) ≠ ∅ then
9              SuspiciousSet ← {C_i}
10             break
11 if CPUT ≡ SuspisiousSet then SuspisiousSet ← ∅
12 return SuspiciousSet
```

L'algorithme 2 prend en entrée le Modèle-Oracle MO, le CPUT et la non-conformité nc. L'idée de base développée dans cet algorithme est de localiser les contraintes fautives en parcourant l'ensemble des contraintes du CPUT et en retenant que celles qui répondent à la définition 5.

3.4 Correction automatique des fautes

Une fois une faute localisée dans un CPUT, notre approche de correction tente de remplacer la ou les contraintes fautives par une correction possible afin de rétablir un état de conformité avec le Modèle-Oracle. L'approche proposée est basé sur le calcul d'un sous-ensemble de contraintes à partir de MO qui remplace la partie fautive des contraintes. L'algorithme 3 permet de calculer l'ensemble des contraintes $CorrectionSet$ qui corrige le CPUT. Corriger le CPUT consiste à enlever les solutions du CPUT qui ne sont pas des solutions de MO. En d'autres termes, réduire l'ensemble des solutions de MO $\wedge \neg$CPUT à vide ($sol(\text{MO} \wedge \neg\text{CPUT})$). L'algorithme 3 prend en entrée le $SuspiciousSet$ retourné par $locate$. Si cet ensemble est vide (i.e., CPUT sous-contraint), $correction$ calcule l'ensemble des contraintes du Modèle-Oracle à ajouter au CPUT. Autrement, l'algorithme calcule une correction possible qui doit remplacer le $SuspiciousSet$ pour rétablir la conformité. En d'autres termes, l'ensemble des contraintes correctrices C_i représentent des contraintes du MO qui satisfont : $sol((\text{CPUT} \backslash suspiciousSet) \wedge \neg C_i) \neq \emptyset$ où il existe des solutions de $(\text{CPUT} \backslash suspiciousSet)$ qui ne satisfont pas C_i.

Algorithm 3: correction(MO, CPUT, $SuspiciousSet$)

```
1  correctionSet ← ∅
2  P ← CPUT\SuspiciousSet
3  foreach C_i ∈ MO do
4      if sol(P ∧ ¬C_i) ≠ ∅ then
5          correctionSet ← correctionSet ∪ {C_i}
6  return correctionSet
```

4 *cptest4choco* library

Dans cette section, nous donnons un survol de notre cadre de test *cptest4choco* pour tester les programmes à contraintes écrits en CHOCO. *cptest4choco* est une librairie de test écrite en Java pour l'environnement CHOCO, en produisant en sortie des programme CHOCO dédiés. Ces programmes à contraintes en sortie sont résolus pour détecter, localiser et corriger les fautes. *cptest4choco* implémente les algorithmes 1,2 et 3 donnés dans les sections précédentes. La spécification de la librairie *cptest4choco* et tous les modèles utilisés pour l'expérimenter, sont accessibles en ligne [2]. La distribution courante contient quatre paquetages, `de détection`, `de localisation`, `de correction` et `de négation`, décrits ci-dessous.

4.1 *cptest4choco* : le paquetage de `détection`

En premier, ce paquetage contient une implémentation des modèles de faute et les relations de non-conformité (i.e., ϕ^+, ϕ^- et ϕ^*). Deuxièmement, il implémente les relations de conformité décrites dans [11]. Ce dernier travail (i.e., [11]) contient seulement les définitions de base ((e.g., définition 1). Ce paquetage contient aussi une implémentation complète de l'Algorithme 1 utilisé dans le processus de détection des fautes. Cet algorithm prend entrée deux modèles CHOCO et retourne, si possible, un(des) cas de non-conformité, c'est-à-dire une instanciation des variables satisfaisant un modèle et non-satisfaisant le deuxième. D'un

2. www.lirmm.fr/∼lazaar/cptest4choco

point de vue utilisateur, plusieurs fonctionnalités de *cptest4choco* sont disponibles en invoquant cette bibliothèque dans un projet Choco.

4.2 *cptest4choco* : le paquetage de `localisation`

Ce paquetage implémente notre approche de localisation des fautes que nous avons introduite dans la section 3.3. Le processus de localisation prend en entrée le cas de non-conformité retourné par le paquetage de `detection`, et retourne l'ensemble des contraintes *suspectes*. Cet ensemble permet de repérer la contrainte à l'origine de l'erreur dans le modèle à contrainte.

4.3 *cptest4choco* : le paquetage de `correction`

Ce paquetage implémente l'algorithme 3. Le processus de correction automatique prend en entrée l'ensemble des contraintes suspectes, et propose en sortie un ensemble de corrections possibles pour rétablir la conformité entre CPUT et son MO. Nous verrons dans la partie expérimentale que ce paquetage est le plus élaboré dans notre environnement *cptest4choco*, et nécessite un effort considérable de compréhension de la part de l'utilisateur.

4.4 *cptest4choco* : négation des contraintes

Notre approche de détection des fautes et leur correction automatique est basée sur la réfutation des contraintes, et l'exploitation d'une négation automatique des contraintes. La librairie *cptest4choco* dispose de la négation de plusieurs contraintes globales sur les domaines discrets, soit en les réécrivant automatiquement en utilisant les opérateurs arithmétiques et relationnels, soit en faisant appel à des contraintes globales. Par exemple, la négation de la contrainte globale *atLeast* peut être obtenue en utilisant la contrainte globale *atMost*. Cependant, les contraintes globales telles que la *cumulative* ou la contrainte *circuit*, nécessitent, pour leur négation, de lui développer une contrainte dédiée. La négation des contraintes globales complexes fait partie de nos travaux futures.

5 Validation expérimentale

Dans cette section, nous présentons les premiers résultats expérimentaux de *cptest4choco* sur des problèmes académiques. Nous avons sélectionné les règles de golomb ainsi que les n-reines. Les résultats que nous donnons sur la détection des fautes ainsi que la mise-au-point des programmes à contraintes ont été produits sur une machine Intel Core i7 CPU 2.9Ghz Mac OS, 8 GO de RAMM DDR3.

5.1 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 *cptest4choco* à détecter des fautes dans des modèles CHOCO ; 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 MO en CHOCO.
- Appliquer les différents raffinements possibles pour obtenir un programme à contraintes $\mathcal{P}$ dédié à résoudre des instances difficiles du problème.
- Injecter manuellement des fautes significatives dans $\mathcal{P}$ et obtenir des CPUT incluant ces fautes.
- Soumettre chaque CPUT à *cptest4choco* pour une phase de test.
- Dans le cas où une faute ϕ est détectée, lancer *cptest4choco* pour une phase de localisation.
- Lancer la correction automatique de *cptest4choco*.

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 1 présente les résultats de *cptest4choco* sur les règles de Golomb et les n-reines. Le tableau est constitué de quatre colonnes : CPUTs : cette colonne décrit les différents CPUT et l'injection de fautes qui indique dans quelle contrainte du CPUT la faute ϕ a été introduite ; Test : Cette colonne est dédiée à la phase de test de *cptest4choco* avec des temps en seconde ; Localisation : Cette partie du tableau présente les $SuspiciousSet$ retournés par $locate$; Correction : Les résultats de la phase de correction de *cptest4choco* avec la taille des $CorrectionSet$ et des temps en secondes.

5.2 Règles de Golomb

Comme nous l'avons présenté dans la section 2, le problème des règles de Golomb consiste à trouver des règles où les distances entre les paires de marques sont de valeurs différentes. C'est aussi un problème d'optimisation où on cherche une règle de longueur minimale. Pour nos expérimentations, nous avons choisi l'instance $m = 8$ qui couvre l'ensemble des contraintes. En alternant la suite de raffinements, nous avons introduit des fautes ϕ, ce qui nous a permis de produire sept CPUT différents. Par exemple, la faute injectée dans CPUT5 consiste à remplacer la contrainte globale `allDifferent` par :

TABLE 1 – Détection des fautes, localisation et correction sur des problèmes classiques

	Mutants	Fault injected	Detection		Localization		Correction		
			non-conformity	time	susp. cstr	time	removed cstr	added cstr	time
Golomb rulers (m=8)	Mut1	cc2	ϕ^+ : [0 1 2 3 4 5 6 7]	0.87	∅	2.48	∅	368 EC	42.13
	Mut2	cc3	ϕ^+ : [0 1 2 3 4 5 6 7]	0.11	∅	1.69	∅	431 EC	43.53
	Mut3	cc3	ϕ^- : [0 7 15 17 18 31 37 58]	0.08	∅	1.72	∅	466 EC	46.59
	Mut4	cc5	ϕ^+ : [0 4 8 12 16 20 24 28]	0.16	∅	1.46	∅	340 EC	14.25
	Mut5	cc1	ϕ^* :sol(Mut5)=∅	0.35	cc1	315.85	cc1	763 EC	30.49
	Mut6	cc7	ϕ^* :sol(Mut6)= ∅	0.37	cc7	137.07	cc7	763 EC	34.81
	Mut7	cc9	ϕ^* :sol(Mut7)= ∅	35.10	cc9	13.50	cc9	∅	108.45
n-queens (n=8)	Mut1	cc12	ϕ^* : sol(Mut1)=∅	0.34	cc12	0.87	cc12	∅	0.17
	Mut2	cc11	ϕ^* :sol(Mut2)=∅	0.39	cc11	0.99	cc11	∅	0.44
	Mut3	cc12	ϕ^* :sol(Mut3)=∅	0.20	cc12	0.40	cc12	∅	0.36
	Mut4	cc11	ϕ^* :sol(Mut2)=∅	0.01	cc11	0.18s	cc11	∅	0.30
	Mut5	cc6	ϕ^+ :[8 7 6 5 4 3 2 1]	0.52	cc6	0.23	cc6	28 EC	0.67
	Mut6	cc7	ϕ^- [1 5 8 6 3 7 2 4]	0.24	∅	0.31	∅	28 EC	0.44

susp. cstr : suspicious constraints EC : elementary constraint

$$\forall i, j, k \quad s.t., i \neq j : \\ mark_{i+1} = mark_i + k \Rightarrow mark_{j+1} = mark_j + k$$

Prenons le CPUT2, CPUT3 et CPUT5 de la parte règles de Golomb du tableau 1. *cptest4choco* retourne une non-conformité pour le CPUT2 (i.e., x=[0 1 2 3 4 5 6 7] où x n'est pas une règle de Golomb avec 1−0=2−1). Cette non-conformité représente une solution du CPUT2 qui n'est pas une solution du Modèle-Oracle MO (i.e., faute de type ϕ^+). La phase de localisation de *cptest4choco* retourne l'état *sous-contraint* avec un ensemble vide pour dire que le CPUT en question contient une contrainte relâchée et ne prend pas en compte la totalité de la spécification du problème. Comme phase de correction, *cptest4choco* nous retourne un ensemble de 431 contraintes élémentaire à ajouter au CPUT2 pour corriger la faute et rétablir la conformité avec le MO.

En ce qui concerne le CPUT3, *cptest4choco* détecte une faute de type ϕ^-. En effet, la solution retournée x=[0 7 15 17 18 31 37 58] représente une non-conformité où x ne peut être retournée par le CPUT3. En phase de localisation, *cptest4choco* pointe la contrainte cc3 du CPUT3 comme contrainte suspecte. En effet, la faute a été introduite dans la contrainte cc7 du CPUT3. La phase de correction retourne, en moins de 1min, 466 contraintes élémentaires qui peuvent remplacer cc3 dans l'instance $m = 8$ du CPUT3 pour rétablir la conformité avec le Modèle-Oracle.

cptest4choco arrive à détecter également les fautes de type ϕ^*. Prenons maintenant le CPUT5 dont la faute introduite dans cc1 le rend insatisfiable (i.e., $\phi^* \Rightarrow sol(Mut5) = \emptyset$). La phase de localisation retourne toutes les contraintes de type cc1 comme étant des contraintes suspectes. *cptest4choco* propose par la suite de remplacer les contraintes de type cc1 par un ensemble de 763 contraintes élémentaires.

```
/*******************
 *   Model-Oracle   *
 *******************/

int n = size;
int sum= n*(n+1)/2;
MO = new CPModel();
// variables
IntegerVariable[] queens = Choco.makeIntVarArray("queen", n, 1, n);
// constraints
for (int i = 0; i < n-1; i++)                                    //c1
    for(int j=i+1; j<n; j++)
        MO.addConstraint(Choco.neq(queens[i], queens[j]));

for (int i = 0; i < n-1; i++)                                    //c2
    for(int j=i+1; j<n; j++)
        MO.addConstraint(Choco.neq(Choco.plus(queens[i], i),
                Choco.plus(queens[j], j)));


for (int i = 0; i < n-1; i++)                                    //c3
    for(int j=i+1; j<n; j++)
        MO.addConstraint(Choco.neq(Choco.minus(queens[i], i),
                Choco.minus(queens[j], j)));
```

FIGURE 3 – MO for n-queens written in CHOCO.

5.3 N-reines

Le problème des n-reines est un problème classique en PPC qui consiste à placer n reines d'un jeu d'échecs sur un échiquier de $n \times n$ cases sans que les reines ne puissent se menacer mutuellement. Par conséquent, deux dames ne devraient jamais partager la même ligne, colonne, ou diagonale. La Figure 3 présente le Modèle-Oracle des n-reines. Nous avons appliqué une suite de raffinements pour avoir un modèle optimisé avec de nouvelles structures de données qui représentent les diagonales descendantes et ascendantes, des contraintes qui cassent la symétrie, des contraintes redondantes, des contraintes globales, etc. Durant cette suite de raffinements, nous avons introduit des fautes ϕ afin d'obtenir sept CPUT contenant des fautes. Prenons le CPUT3 et CPUT5 du tableau 1 :

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 repré-

sente une mauvaise formulation de la contrainte `cc12` qui réduit son ensemble de solutions à vide. Soumettre CPUT3 à *cptest4choco* pour une phase de test permet de détecter une faute de type ϕ^* en moins de 1s. La phase de localisation de *cptest4choco* retourne `cc12` comme contrainte suspecte pour CPUT3. En effet, la faute ϕ^* a été introduite dans `cc12`. Pour corriger cette faute et étant donnée la contrainte `cc12` une contrainte redondante, *cptest4choco* propose d'enlever cette contrainte de CPUT3 pour rétablir la conformité avec le Modèle-Oracle.

Pour le CPUT5, nous avons introduit une faute ϕ dans la formulation de la contrainte `cc6`. CPTEST détecte une faute ϕ^+ avec la solution `q1=[8 7 6 5 4 3 2 1]`. Cette solution 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 des faute de type ϕ^+ dans CPUT5. La phase de localisation retourne également la contrainte fautive comme suspecte `cc6`. Ici, *cptest4choco* propose de remplacer la `cc4` par 28 contraintes élémentaires pour corriger l'instance $n = 8$ du CPUT5. Les 28 contraintes sont un sous-ensemble des contraintes élémentaires qu'encapsule `c2` dans le Modèle-Oracle. Par conséquent, remplacer `cc6` par c2 corrige toute instance du CPUT5.

cptest4choco retourne une non-conformité pour le CPUT6. Cette non-conformité est une solution du Modèle-Oracle qui n'est pas une solution du CPUT6 (i.e., `q2=[1 5 8 6 3 7 2 4]`). Ainsi, `q2` révèle la présence d'une faute de type ϕ^-. Cette faute a été injectée dans `cc7`. La phase de localisation retourne toutes les contraintes de type `cc7` comme suspectes dans CPUT6. Une des corrections possibles est celle proposée par *cptest4choco* qui propose un ajout de 28 contraintes élémentaires pour rétablir une conformité entre CPUT6 et le MO.

5.4 Discussion

Une des reproches que nous pouvons émettre sur la validité de notre approche concerne les problèmes que nous avons utilisés dans nos expérimentations. Nous avons choisi les règles de Golomb et les n-reines comme problèmes bien connus dans la communauté PPC. Simples mais non triviaux, ces problèmes servent souvent comme des exemples pour illustrer des techniques en PPC. Ceci nous a permis de valider notre approche sur des données empiriques pertinentes. Cependant, ces problèmes viennent du milieu académique et peuvent ne pas refléter pleinement l'utilisation industrielle de la programmation par contraintes dans des applications critiques. En outre, nous avons injecté manuellement les fautes dans nos propres modèles à contraintes pour des phase de test et de mise-au-point. Bien que ces fautes ont été bien choisies pour montrer les capacités des approches de test et de mise-au-point, elles ne sont pas faciles à repérer et à réparer.

```
/*********************************
 *   Constraint Program Under Test *
 *********************************/
int n = size;
int sum= n*(n+1)/2;
CPUT = new CPModel();
// variables:
IntegerVariable[] queens = Choco.makeIntVarArray("queen", n, 1, n);
IntegerVariable[] rows = Choco.makeIntVarArray("row", n, 1, n);
IntegerVariable[] diag1 = Choco.makeIntVarArray("diag1", n);
IntegerVariable[] diag2 = Choco.makeIntVarArray("diag2", n);
IntegerVariable[] diag3 = Choco.makeIntVarArray("diag3", n);
IntegerVariable[] diag4 = Choco.makeIntVarArray("diag4", n);
// constraints:
CPUT.addConstraint(Choco.allDifferent(queens));                  //cc1


for(int i=0; i<n; i++)                                           //cc2
    CPUT.addConstraint(Choco.eq(diag1[i], Choco.plus(queens[i], i)));

for(int i=0; i<n; i++)                                           //cc3
    CPUT.addConstraint(Choco.eq(diag2[i], Choco.minus(queens[i], i)));

for(int i=0; i<n; i++)                                           //cc4
    CPUT.addConstraint(Choco.eq(diag3[i], Choco.plus(rows[i], i)));

for(int i=0; i<n; i++)                                           //cc5
    CPUT.addConstraint(Choco.eq(diag4[i], Choco.minus(rows[i], i)));

CPUT.addConstraint(Choco.allDifferent(diag1));                   //cc6
CPUT.addConstraint(Choco.allDifferent(diag2));                   //cc7
CPUT.addConstraint(Choco.allDifferent(diag3));                   //cc8
CPUT.addConstraint(Choco.allDifferent(diag4));                   //cc9

CPUT.addConstraint(Choco.inverseChanneling(queens, rows));       //cc10

CPUT.addConstraint(Choco.eq(Choco.sum(queens),sum));             //cc11
CPUT.addConstraint(Choco.eq(Choco.sum(rows),sum));               //cc12
```

FIGURE 4 – CPUT for n-queens written in CHOCO.

Nous ne savons pas si elles sont réalistes ou non. Contrairement à d'autres langages où plusieurs programmes et bugs sont disponibles (par exemple, Java, C et C ++), les programmes écrits dans des langages PPC ne sont pas disponibles à partir des référentiels sur le Web.

Etant donné que CHOCO est une bibliothèque Java, nous pouvons utiliser la notation JML pour exprimer des post-conditions du modèle en question. Des environnements tels que [1] permettent de trouver des contre-exemples où certaines données de test satisfont le programme Java et ne répondent pas à la post-condition écrite en JML. Ces environnements sont dédiés à la vérification sémantique des programmes et ils sont souvent confrontés à des difficultés pour faire face à la nature exponentielle de l'espace de recherche des variables Java. En outre, ces environnements ne sont pas capables de capturer la sémantique des contraintes et de remédier aux mauvais usages des contraintes. Dans les approches que nous avons proposées, nous travaillons à un niveau d'abstraction élevé où l'espace de recherche est beaucoup plus raisonnable et est défini uniquement par les variables de décision. Ici, la post-condition est exprimée comme un modèle CHOCO, ce qui est plus concis et plus compact qu'une post-condition JML écrite en termes de variables Java. En fait, notre approche est étroitement dédié aux développeurs CHOCO, en facilitant la validation d'un processus de raffinement et d'opti-

misation d'un modèle CHOCO.

Au final, nos approches ont été bâties sur l'hypothèse d'avoir une référence de test (modèle-oracle), un modèle de contraintes initial extrait de la spécification du problème. Cependant, cette hypothèse pourrait être difficile à satisfaire. Les premiers modèles ne sont pas nécessairement conservés pour analyse dans le processus de développement industriel.

6 Conclusion

Dans cette article, nous avons présenté une nouvelle plateforme de test et de mise-au-point dédiée aux modèles écrits en CHOCO, nommée *cptest4choco*. Cette plateforme est présentée sous forme d'une bibliothèque Java avec une implémentation des fonctionnalités de test, de localisation de faute et de correction automatique. Ces fonctionnalités et ces approches sont basées sur la notion de Modèle-Oracle de test représenté par un premier modèle déclaratif, simple et fidèle à la spécification de départ du problème en question. La réalisation de *cptest4choco* et son expérimentation nous indiquent que l'approche qui consiste à valider automatiquement des programmes à contraintes en CHOCO, lorsqu'un Modèle-Oracle est disponible, est viable et pourrait être généralisée à d'autres langages PPC.

Références

[1] Hélène Collavizza, Michel Rueher, and Pascal Van Hentenryck. Cpbpv : a constraint-programming framework for bounded program verification. *Constraints*, 15(2) :238–264, 2010.

[2] Carlos Cotta, Iván Dotú, Antonio J. Fernández, and Pascal Hentenryck. Local search-based hybrid algorithms for finding golomb rulers. *Constraints Journal*, 12 :263–291, Septembre 2007.

[3] Pierre Flener, Justin Pearson, and Marc Bourgois. Constraint programming for air traffic management : preface. *Knowledge Eng. Review*, 27(3) :287–289, 2012.

[4] Pierre Flener, Justin Pearson, Luis G. Reyna, and Olof Sivertsson. Design of financial cdo squared transactions using constraint programming. *Constraints*, 12(2) :179–205, 2007.

[5] Arnaud Gotlieb. Tcas software verification using constraint programming. *Knowledge Eng. Review*, 27(3) :343–360, 2012.

[6] Alan Holland and Barry O'Sullivan. Robust solutions for combinatorial auctions. In *ACM Conference on Electronic Commerce (EC-2005)*, pages 183–192, 2005.

[7] U. Junker and D. Vidal. Air traffic flow management with ilog cp optimizer. In *International Workshop on Constraint Programming for Air Traffic Control and Management*, 2008. 7th EuroControl Innovative Research Workshop and Exhibition (INO'08).

[8] Nadjib Lazaar, Arnaud Gotlieb, and Yahia Lebbah. Fault localization in constraint programs. In *ICTAI (1)*, pages 61–67, 2010.

[9] Nadjib Lazaar, Arnaud Gotlieb, and Yahia Lebbah. On testing constraint programs. In *CP*, pages 330–344, 2010.

[10] Nadjib Lazaar, Arnaud Gotlieb, and Yahia Lebbah. A framework for the automatic correction of constraint programs. In *ICST*, pages 319–326, 2011.

[11] Nadjib Lazaar, Arnaud Gotlieb, and Yahia Lebbah. A cp framework for testing cp. *Constraints*, 17(2) :123–147, 2012.

[12] K. Marriott, N. Nethercote, R. Rafeh, P. J. Stuckey, M. Garcia De La Banda, and M. Wallace. The design of the zinc modelling language. *Constraints*, 13(3) :229–267, 2008.

[13] W. T. Rankin. Optimal golomb rulers : An exhaustive parallel search implementation. Master's thesis, Duke University, Durham, 1993.

[14] Francesca Rossi, Peter van Beek, and Toby Walsh. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science Inc., New York, USA, 2006.

[15] Barbara M. Smith, Kostas Stergiou, and Toby Walsh. Modelling the golomb ruler problem. 1999.

[16] P. Van Hentenryck. *The OPL optimization programming language*. MIT Press, Cambridge, MA, USA, 1999.

[17] Pascal Van Hentenryck and Laurent Michel. *Constraint-Based Local Search*. The MIT Press, 2005.