Vers une Théorie du Test des programmes à contraintes

Nadjib Lazaar*, Arnaud Gotlieb*, Yahia Lebbah**

* INRIA Rennes Bretagne Atlantique, Campus Beaulieu, 35042 Rennes Cedex, France
** Université d'Oran Es-Senia, B.P. 1524 EL-M'Naouar, 31000 Oran, Algerie
Nadjib.Lazaar@irisa.fr, Arnaud.Gotlieb@irisa.fr, ylebbah@gmail.com

Abstract

Tout processus de développement logiciel effectué dans un cadre industriel inclut désormais une phase de test ou de vérification formelle, y compris pour le développement des programmes à contraintes. Notre travail vise à poser les jalons d'une Théorie du test des programmes à contraintes qui puisse servir de socle à cette vérification. Cette nouvelle théorie est également motivée par le développement récent de plusieurs langages de modélisation de haut-niveau tels que OPL, ZINC, COMET, ou ESSENCE, qui ouvre la voie à des recherches orientées vers les aspects génie logiciel autour de la programmation par contraintes. Notre approche repose sur certaines hypothèses quant au développement et 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 d'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, et de contraintes globales. Nous pensons que l'essentiel des fautes introduites est compris dans cette deuxième étape. Dans cet article nous bâtissons les prémisses d'une Théorie du test des programmes à contraintes qui établit les règles de conformité entre le modèle initial déclaratif et le programme optimisé dédié à la résolution d'instances de grande taille. Nous illustrons cette Théorie à l'aide du problème des règles de golomb en OPL et détaillons une première validation expérimentale sur le problème d'ordonnancement des véhicules (POV).

1 Introduction

La Programmation par Contrainte a connu un véritable succès en résolution de problèmes combinatoires dans l'industrie et dans divers domaines (optimisation, transport, emploi du temps, ordonnancement, etc.). D'autres domaines d'application se développent désormais, comme la bio-informatique (Will et Backofen [1]), ou encore la biochimie (Fages et al. [4]). On trouve de nombreux langages et plate-formes PPC commercialisés comme COMET[1], OPL[2], SICStus[3] et académiques tels que CHOCO[4], ESSENCE [5], ZINC [9]. Or, pour les programmes développés dans ces langages, le processus de développement logiciel devrait également inclure une phase de vérification et/ou de validation qui inclut par exemple de la preuve de programme, de la vérification de modèles ("model-checking"), de l'analyse statique ou du test. Pour cette dernière approche dans le cas du test des programmes conventionnels, plusieurs théories ont été proposées telles que la théorie du test de Goodenough et Gerhart [6], la théorie mathématique du Test de Gourlay [7], la Théorie du Test de Weyuker et Ostrand [14].

Pourtant, le test des programmes à contraintes ne peut-être envisagé comme celui des programmes traditionnels. Cela pour deux raisons principales : premièrement, le modèle de faute des programmes à contraintes est différent de celui des programmes traditionnels car l'objectif est ici de calculer des solutions d'un système de contraintes ; deuxièmement, le processus de développement et de raffinement n'est pas le même pour les programmes à contraintes et pour les programmes traditionnels. En effet, il est usuel de démarrer à partir d'un modèle simple et très déclaratif du problème que l'on cherche à résoudre, une traduction fidèle de la spécification du problème, sans accorder d'intérêt à ses performances. Puis, des techniques puissantes de raffinement de modèle sont mises en oeuvre. Par exemple

[1] http ://www.dynadec.com/support/downloads/
[2] http ://www.ilog.com/products/oplstudio/
[3] http ://www.sics.se/isl/sicstuswww/site/
[4] http ://choco.sourceforge.net

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 des contraintes qui visent à casser des symétries du problème (ces contraintes améliorent considérablement l'efficacité de la résolution). Illustrons ce raffinement sur le problème classique des règles de Golomb. Ces règles trouvent leurs champs 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).

Une règle de Golomb [11] peut être définie comme un ensemble de m entiers $0 = x_1 < x_2 < ... < x_m$ 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 si elle contient m marques, et qu'elle est de longueur x_m. L'objectif est de trouver une règle de longueur minimale ($minimize\ x[m]$). Une modélisation simple et très déclarative de ce problème est donnée dans la partie (A) de la Fig. 1. La partie (B) de la Fig. 1 présente quant à elle un modèle raffiné optimisé[5] où une structure de donnée a été introduite (matrice), des symétries ont été cassées statiquement, des contraintes redondantes ont été introduites et une contrainte globale posée (alldifferent). Arrivé à ce point, comment être sûr que le modèle de la partie (B) de la Fig. 1 est conforme à celui de la partie (A) ? En effet, ces raffinements de modèle, réalisés par le développeur, peuvent introduire des fautes. Par exemple, mal formuler une contrainte peut conduire à un modèle surcontraint (supprimer des solutions) ou à un modèle dans lequel des solutions ont été ajoutées.

Dans cet article, nous proposons les prémisses d'une Théorie de la conformité des programmes à contraintes. Notre Théorie s'appuie sur la définition de relations de conformité entre les modèles, sur la proposition de définitions concernant la génération de test (donnée de test, jeu de test idéal, critères de test, faute et test de non-conformité). Nous évoquons une méthode de génération automatique de données de test pour les programmes à contraintes que nous validons sur un exemple réaliste : celui de l'ordonnancement de véhicules (POV).

Le reste de cet article est organisé comme suit. La section 2 est un passage en revue de quelques langages à contraintes et de leurs outils de mise au point. La section 3 décrit notre théorie du test des programmes à contraintes avec les notions de conformité. En section 4, un schéma de validation expérimentale de la théorie est présenté sur l'exemple de l'ordonnancement de véhicules. Enfin la section 5 conclut l'article avec des perspectives d'extension de cette Théorie.

[5]On se place dans un contexte où la contrainte globale $Golomb(Var)$ n'est pas disponible.

2 Etat de l'art

Au cours des dernières années, beaucoup de travaux ont été menés autour des langages de modélisation de programmes à contraintes. Ces langages, conçus pour un haut niveau de modélisation, permettent une expérimentation facile avec différents solveurs pour un même problème mais ne présentent pas d'innovations majeures en termes de mise au point des programmes à contraintes. Le langage ESSENCE [5] est un langage formel de spécification de problèmes combinatoires qui s'appuie sur la langue naturelle. Le projet G12 développe le langage ZINC[6] et ses variantes (MiniZinc, FlatZinc) dans le but de créer une plateforme logicielle pour résoudre des problèmes d'optimisation combinatoire réels [9]. ZINC définit des prédicats permettant la réutilisation de code, ainsi que des méthodes de traduction vers des modèles de plus bas-niveau. Néanmoins, ces deux plateformes n'offrent pas en standard d'outils pour aider au test ou à la mise au point des programmes. Le langage OPL (*Optimization Programming Language*)[12] est un langage de modélisation utilisant la programmation mathématique et la programmation par contraintes. Il offre à l'utilisateur un outil de mise au point dans OPL Studio qui détecte des erreurs de syntaxe (oubli du ';' après une instruction par exemple), des erreurs dites "sémantiques" (erreur de nom ou de type par exemple) et des erreurs d'exécution (initialisation d'un tableau de longueur N avec $N+1$ données par exemple). COMET [13] est lui un langage orienté objet implanté en C++ avec un certain nombre d'abstractions innovatrices de modélisation et de contrôle pour la recherche locale. Son outil de mise au point est simplement une interface à `gdb`, le "debugger" standard de la suite `gcc`. On peut citer aussi CHOCO [7] qui est une librairie JAVA pour la modélisation des contraintes qui offre, comme COMET, une interface à un debugger du langage sous-jacent (`jdb`). Mais, les fautes les plus difficiles à trouver, celles liées à la correction des modèles à contraintes, ne sont pas visées et détectées par ces outils.

La mise au point des programmes à contraintes a fait l'objet de nombreux travaux de Recherche, en particulier à l'occasion du projet OADymPPaC[8]. Ceux-ci ont abouti à la définition de modèles de trace génériques

[6]http ://www.g12.cs.mu.oz.au/
[7]http ://www.emn.fr/x-info/choco-solver/doku.php
[8]http ://contraintes.inria.fr/OADymPPaC/

```
int m=...;

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

minimize x[m];
subject to {
    forall (i in 1..m-1)
        x[i] < x[i+1];
    forall (i in 1..m, j in 1..m,
           k in 1..m, l in 1..m: (i < j, k < l))
       x[j] - x[i] != x[l] - x[k];
}
```

- A -

```
int m=...;

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

tuple indexerTuple {
int i;
int j;
}

{indexerTuple} indexes = {<i, j> | i,j in 1..m : i < j};
dvar int d[indexes];

 minimize x[m];
 subject to {

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

(2)      forall(ind in indexes)
            d[ind] == x[ind.j]-x[ind.i];
(3)     x[1]=0;
(4)     x[m] >= (m * (m - 1)) / 2;
(5)     allDifferent(all(ind in indexes ) d[ind]);
(6)     x[2] <= x[m]-x[m-1];
(7)     forall(ind1 in indexes, ind2 in indexes, ind3 in indexes :
          (ind1.i==ind2.i)&&(ind2.j==ind3.j)&&(ind1.j==ind3.j)&&
           ( ind1.i < ind2.j < ind1.j))
            d[i,j]=d[i,k]+d[k,j];
(8)      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];
(9)     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;

        }
```

- B -

FIG. 1 – $M_x(k)$ et $P_x(k)$ des règles de Golomb en OPL.

[3, 8] et à la réalisation d'outils d'observation et d'analyse de traces post-mortem, tels que Codeine pour Prolog, Morphine [8] pour Mercury, ILOG Gentra4CP[9], 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 de domaines effectués. On peut citer également l'outil CLPGUI [4] qui offre une interface graphique et intuitive à la visualisation des traces. Ces outils aident à comprendre les comportements d'un programme à contraintes et aident à leur optimisation, mais ne sont pas dédiés à la détection de fautes. En effet, celle-ci exige la donnée d'une référence (une spécification ou un oracle) afin de déterminer la divergence entre une implantation et cette référence. Tab. 1 récapitule les fonctionnalités des langages que nous venons de citer. Il est à noter que aucun d'eux ne présentent, à notre connaissance, de Théorie et/ou d'outil de test des programmes à contraintes a proprement parler.

[9] http ://www2.ilog.com/preview/Discovery/gentra4cp/

3 Vers une théorie du test des programmes à contraintes

3.1 Notations et définitions

Dans ce qui suit, nous adoptons les notations suivantes : x dénote un vecteur de variable, x_i dénote une valuation de ce vecteur, $(x\backslash x_i)$ représente la substitution des variables x par x_i, la différence de deux ensembles E et F est notée $E\backslash F \triangleq \{x/(x \in E) \wedge (x \notin F)\}$, la différence symétrique de E et F, se note $E\Delta F \triangleq \{x/\,((x \in E) \wedge (x \notin F)) \vee ((x \in F) \wedge (x \notin E))\} = (E \cup F)\backslash(E \cap F)$.

Un programme à contraintes comprend d'une part un modèle $M_x(k)$, qui est une conjonction de contraintes $C_i(x)$, où k représente l'ensemble des paramètres du modèle (pour les règles de Golomb, il s'agit de l'ordre de la règle tandis que le vecteur de marques représentent les variables) ; et d'autre part une procédure de recherche ou d'optimisation notée *Solve*.

TAB. 1 – les différents langages de modélisation des programmes à contraintes. (PL : prog. linéaire, PPC : prog. par contraintes, SAT : satisfiabilité, RL : recherche locale, MAP : mise au point)

//////////	CHOCO	COMET	ESSENCE	OPL	Sicstus	ZINC
Indépendant des solveurs	Oui	Non	Oui	Oui	Non	Oui
Solveur PL	Non	Oui	Oui	Oui	Oui	Oui
Solveur PPC	Oui	Oui	Oui	Oui	Oui	Oui
Solveur SAT	Non	Oui	Non	Non	Non	Oui
Solveur RL	Oui	Oui	Non	Non	Non	Non
Outil de MAP	jdb	gdb	Non	OPL Studio	fdbg	Non
Plateforme	JAVA	C++	HASKELL	C++	Prolog	Mercury
Dernière version	CHOCO-V2	Comet 1.1	Essence 1.1.0	OPL 6.1.1	Sics Prolog 4.0.5	MiniZinc 0.9

Model $M_x(k)$
$\begin{cases} C_1(x) \\ . \\ . \\ . \\ C_n(x) \end{cases}$
Solve($M_x(k)$)

On considère que $k \in \mathcal{K}$ où $\mathcal{K}$ représente l'ensemble des valeurs possible des paramètres pour lequel le modèle $M_x(k)$ possède des solutions. Notons que le vecteur de variables x peut dépendre de l'instance du modèle considérée. Par exemple, si $k = 3$ dans Golomb, l'instance du modèle cherche une règle avec 3 marques (x=(0, 1, 3)) tandis que pour $k = 4$ elle cherche une règle de 4 marques (x=(0, 1, 4, 6)).

On considère une fonction f qui est une fonction de coût à optimiser sur l'Espace de Recherche. Pour fixer le discours, nous considérons uniquement les problèmes de minimisation ($Minimize(f)$), sachant que les problèmes de maximisation s'obtiennent par symétrie. La Fig. 2 donne un exemple de fonction objectif sur $\mathbb{R}$ où le point x_1 représente un minimum global (une solution optimale) avec $f(x_1) = b$. Cette valeur b est pratiquement difficile à atteindre dans les instances réalistes des problèmes, c'est pourquoi nous nous intéressons dans la Théorie, à la notion de *quasi-optimaux*. En effet un développeur est prêt à concéder, à titre de perte maximale autorisée sur la fonction objectif, un coût strictement positif noté l. Dans la Fig. 2, les points x_0, x_3 représentent des solutions quasi-optimales car les valeurs $f(x_0)$ et $f(x_3)$ ne diffèrent de la valeur optimale que d'une quantité inférieure ou égale à l, tandis que le point x_2 n'est pas une solution quasi-optimale car $f(x_2) > b + l$. Cette perte l est relative au problème traité. Soit M un modèle à contraintes, $sol(M)$ dénote l'ensemble des solutions de M, $quasi_f(M, l)$ dénote l'ensemble des solutions quasi-optimales de M par rapport à une fonction objectif f et une perte l, et $best_f(M)$ dénote l'ensemble des minimums globaux.

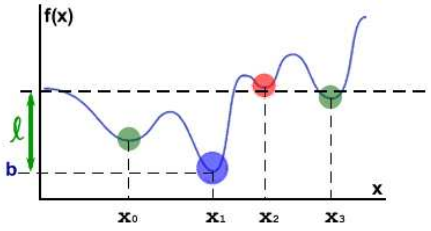

FIG. 2 – Fonction objectif dans le cas univarié.

La vérification de la correction d'un programme implique généralement la présence d'une référence, nommée oracle de test. Cette référence, dans notre approche, est le modèle original et déclaratif du problème. Ainsi, il devient possible d'utiliser ce modèle de plusieurs manières pour générer des données de test, nous y reviendrons dans la suite. A ce point, il est important de noter que le modèle déclaratif devient pour la théorie que nous présentons, une référence incontestée et idéalisée du programme à contraintes sous test. Nous considérons donc ce modèle comme étant correct par rapport aux exigences de l'utilisateur, ce qui est une hypothèse forte mais habituellement faite dans les théories classiques du test des programmes [6]. Nous considérons aussi que ce modèle possède des solutions et caractérise, pour une instanciation des paramètres, toutes les solutions du problème. Si, pour certaines instanciations des paramètres k, le modèle n'a pas de solution, ces valeurs de k sont simplement exclues de l'ensemble de définition $\mathcal{K}$.

Le Modèle–oracle de test est un modèle à contraintes $M_x(k)$ qui caractérise l'ensemble des solutions du problème et qui est conforme aux exigences de l'utilisateur. *Le Programme à Contraintes Sous Test (CPUT)* est un modèle à contraintes $P_x(k)$ qui est l'objet du

test. Le programme $P_x(k)$ vise à résoudre des instances difficiles du problème et il est légitime de le tester avant de l'utiliser pour ces instances car elles peuvent nécessiter un temps de résolution très long ou bien mobiliser des ressources importantes. Il est important de noter qu'étant donné k_0 une instance de k et x_0 un point de l'espace de recherche, vérifier que $M_{(x\backslash x_0)}(k\backslash k_0)$ est vrai est généralement peu coûteux, tandis que trouver x_0 qui satisfait aux contraintes du modèle M est un problème difficile.

Definition 1 (Ensemble $\mathcal{S}$) *On appelle $\mathcal{S}$ l'ensemble des solutions des instances $P_x(k)$:*

$$\mathcal{S} = \bigcup_{k\in\mathcal{K}} sol(P_x(k)).$$

3.2 Relations de conformité

On note $conf$ la relation de conformité entre le CPUT $P_x(k)$ et l'oracle de test $M_x(k)$. Nous définissons la relation de conformité $conf$ selon le type de problème abordé. En effet, cette définition présente des différences selon que l'on cherche une, toutes ou une meilleure solution d'une instance.

3.2.1 Une seule solution

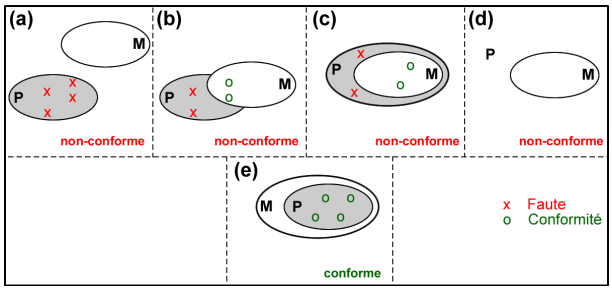

FIG. 3 – $conf_{one}$ Conformité entre $P_x(k)$ et $M_x(k)$ pour une seule solution.

La Fig. 3 présente l'ensemble $sol(M_x(k))$ noté M, et l'ensemble $sol(P_x(k))$ noté P. Les points étiquetés **x** représentent des non-conformités (fautes) tandis que les points étiquetés **o** sont conformes. Les parties (a), (b) et (c) de la Fig. 3 montrent que $P_x(k)$ est non-conforme au Modèle–Oracle $M_x(k)$. En effet, résoudre $P_x(k)$ peut conduire à des solutions qui ne satisfont pas le modèle $M_x(k)$. La partie (d) est aussi une non-conformité car P ne contiens aucune solution. En revanche, la partie (e) montre que $P_x(k)$ est conforme à $M_x(k)$ car elle ne contient pas de points de non-conformité. En observant que P doit être inclus dans M, ce schéma suggère la définition suivante pour la relation de conformité $conf_{one}$:

Definition 2 ***(Relation de conformité : $conf_{one}$)***

$$P\ conf_{one}\ M \Leftrightarrow$$
$$\forall k \in \mathcal{K} :\ sol(P_x(k)) \neq \emptyset \wedge sol(P_x(k)) \subseteq sol(M_x(k))$$

Cette relation de conformité impose que l'ensemble des solutions du programme à contraintes sous test soit non vide. En effet, sans cela, on risque de considérer comme conforme un modèle inconsistant (sans solution). L'ensemble des non-conformités est alors : $sol(P_x(k))\backslash sol(M_x(k))$.

3.2.2 Toutes les solutions

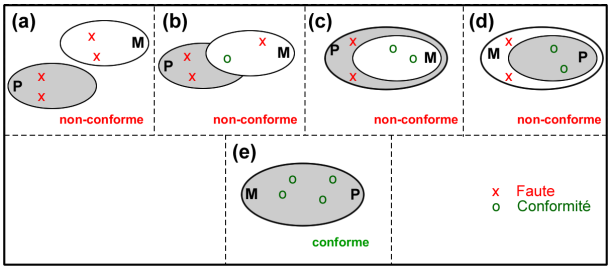

FIG. 4 – $conf_{all}$ Conformité entre $P_x(k)$ et $M_x(k)$ pour toutes les solutions.

On reprend sur la Fig. 4 les ensembles P et M. Dans le cas de recherche de toutes les solutions, les parties (a), (b), (c) et (d) montrent des points de non-conformité. Dans (d), une solution du modèle qui ne serait pas solution du CPUT est une faute. En effet, $P_x(k)$ est conforme à $M_x(k)$ ssi les deux ensembles P et M sont égaux ce qui induit la définition suivante :

Definition 3 ***(Relation de conformité : $conf_{all}$)***

$$P\ conf_{all}\ M \Leftrightarrow \forall k \in \mathcal{K} :\ sol(P_x(k)) = sol(M_x(k))$$

Note : $sol(M_x(k))$ ne peut être vide car k prend ses valeurs dans $\mathcal{K}$.
L'ensemble des non-conformités est alors : $sol(P_x(k))\Delta sol(M_x(k))$.

3.2.3 Une meilleure solution

La Fig. 5 présente la relation de conformité dans le cas où une meilleure solution (quasi-optimale) du CPUT est recherchée. P représente cette fois l'ensemble $quasi_f(P_x(k), l)$, B l'ensemble $best_f(M_x(k))$ qui est l'ensemble des minimums globaux de $M_x(k)$. L'ensemble Q représente $quasi_f(M_x(k), l)$ où les solutions quasi-optimales sont incluses. On choisit quatre solutions quasi-optimales de P. La partie (a) est un cas

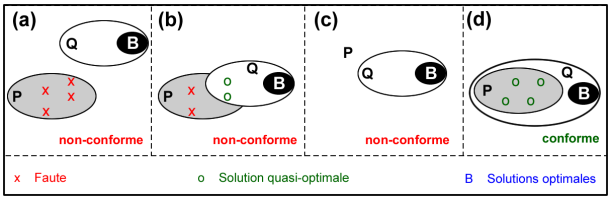

FIG. 5 – $conf_{best}$ Conformité entre $P_x(k)$ et $M_x(k)$ pour une meilleure solution.

de non-conformité car les quasi-optimaux du CPUT ne sont pas inclus dans les quasi-optimaux du modèle-oracle. La partie (b) est un autre cas de non-conformité car au moins deux solutions quasi-optimales ne sont pas dans Q. Le cas (c) présente aussi une non-conformité car P ne contiens aucune solution quasi-optimale ; cela signifie que la minimisation n'a pas pu atteindre une solution avec un coût $\leq l + b$. En revanche, la partie (d) montre un cas de conformité car les quasi-optimaux de P sont dans Q. Ceci suggère une définition fondée sur l'appartenance des solutions quasi-optimales de $P_x(k)$ aux solutions quasi-optimales du modèle-oracle $M_x(k)$:

Definition 4 ***(Relation de conformité :*** $conf_{best}$***)***

$P\ conf_{best}\ M \Leftrightarrow$

$\forall k \in \mathcal{K}$:
$quasi_f(P_x(k), l) \neq \emptyset \wedge$
$quasi_f(P_x(k), l) \subseteq quasi_f(M_x(k), l)$

où l est la perte maximale autorisée sur la fonction objectif f.

L'ensemble des non-conformités est alors : $quasi_f(P_x(k), l) \backslash quasi_f(M_x(k), l)$.

La question naturelle qui se pose est celle de la preuve de conformité : peut-on prouver la conformité du CPUT par rapport à son modèle-oracle ? Si on se place dans le cadre de la résolution des contraintes à domaines finis, prouver la conformité d'une instance est un problème NP_difficile car cela nécessite au moins de trouver toutes les solutions du CPUT. Il nous faut donc réduire cette ambition à la recherche de non-conformité par le test, ce qui justifie l'introduction d'une Théorie du test des programmes à contraintes.

3.3 Test de non-conformité

Nous introduisons ici les définitions permettant d'exprimer la non-conformité.

Definition 5 (Donnée de Test) *Etant donné un modèle-oracle $M_k(x)$, une donnée de test est définie comme une paire (k_0, x_0).*

Definition 6 (Jeu de Test) *Un jeu de test, noté T est un ensemble fini non nul de données de test, tq :*

$$T \subseteq \mathcal{K} \times \mathcal{S}$$

où $\mathcal{S}$ est défini dans Def.1.

Soit $T_d \subseteq \mathcal{K}$, où $T_d = \{k_1, k_2, ...k_l\}$, et $S_i = sol(P_x(k_i))$, on note alors :

$$T \subseteq \mathcal{D} = \{(k_i, s_{ij})\ /k_i \in \mathcal{K}, s_{ij} \in S_i\}$$

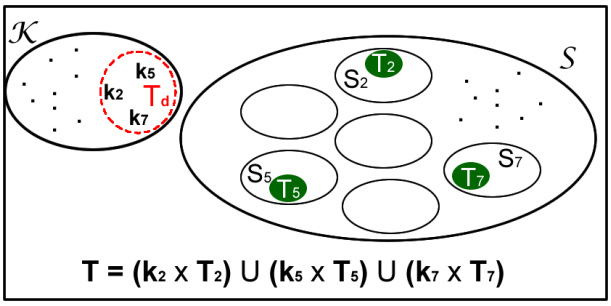

FIG. 6 – Exemple de jeu de test " T ".

La Fig. 6 montre un exemple de jeu de test, où $\mathcal{K} = \{..., k_2, ..., k_5, ..., k_7, ...\}$ donne un ensemble d'instances de $P_x(k)$. L'ensemble $\mathcal{S}$ contient les ensembles de solutions, un pour chaque instance. Si $T_d = \{k_2, k_5, k_7\}$, alors on a trois ensembles de solutions $sol(P_x(k_2))$, $sol(P_x(k_5))$, $sol(P_x(k_7))$, desquels on extrait trois sous-ensembles T_2, T_5, T_7 pour former le jeu de test :

$$T = (k_2 \times T_2) \cup (k_5 \times T_5) \cup (k_7 \times T_7).$$

La qualité d'un jeu de test peut être évaluée par sa force à détecter des non-conformités. Si ces dernières existent, le jeu de test idéal les fera ressortir.

Definition 7 (Jeu de test idéal) *T est un jeu de test idéal ssi :*
$\forall (k_i, x_i) \in T, P_{(k\backslash k_i)}(x\backslash x_i)\ conf_\alpha\ M_{(k\backslash k_i)}(x\backslash x_i) \Rightarrow$
$\forall (k_j, x_j) \in \mathcal{D}, P_{(k\backslash k_j)}(x\backslash x_j)\ conf_\alpha\ M_{(k\backslash k_j)}(x\backslash x_j)$
où α vaut one, all, ou best

L'intérêt de cette définition provient de la remarque suivante : l'ensemble T est un jeu de test pratique, il est donc fini, tandis que l'ensemble $\mathcal{D}$ est en général infini. C'est tout l'intérêt du Test que de pratiquer ce genre de réduction, au prix de la complétude. En pratique, trouver un jeu de test idéal semble être une tâche ardue aussi nous nous contenterons de chercher des jeux de test ayant des propriétés intéressantes que nous détaillons dans la suite. Il est important de noter que le jeu de test devra être de taille modeste afin de garder un processus de test prenant un temps fini et raisonnable.

3.4 Critères de Test

Les propriétés portant sur les jeux de test s'expriment généralement en termes de *critères de test*, qui sont simplement des procédés de sélection des jeux de test. Nous proposons ici un premier critère très simple :

Definition 8 (Toutes_les_contraintes) *Le critère Toutes_les_contraintes exige que chaque contrainte d'un modèle M ait contribué au moins une fois à la résolution du système de contraintes.*

Autrement dit, satisfaire le critère *Toutes_les_contraintes* demande à ce que chaque contrainte du modèle ait été postée au moins une fois dans le système de contraintes. Un second critère, plus difficile à satisfaire, peut être défini comme suit :

Definition 9 (Tous_les_reveils) *Le critère Tous_les_reveils exige que chaque contrainte d'un modèle M ait contribué au moins deux fois à la résolution du système de contraintes.*

Autrement dit, satisfaire le critère *Tous_les_reveils* demande à ce que chaque contrainte du modèle ait été postée au moins une fois dans le système de contraintes et que chaque contrainte ait été réveillée au moins une fois lors de la résolution.

Pour illustrer cette notion de critère de test, nous appliquons le critère *toutes_les_contraintes* pour générer un jeu de test dans le cas des règles de Golomb. On rappelle que $P_x(k)$ est représenté sur la partie droite de la Fig. 1. Nous recherchons des instances k pour lesquelles toute contrainte du CPUT $P_x(k)$ est postée au moins une fois. Il y a 9 contraintes dans le CPUT $P_x(k)$ et le paramètre k qui est l'ordre de la règle, prend des valeurs dans $\mathcal{K} = 1..+\infty$.

Pour $k = 1$, seule la contrainte numérotée 3 est postée :

```
(3)    x[1]=0 ;
```

Ainsi, le critère *toutes_les_contraintes* n'est pas satisfait. Pour $k = 2$, les contraintes 1, 2, 3, 4, 6, sont postées :

```
(1)    x[1] < x[2];
(2)    d[ind12] = x[ind12.j] - x[ind12.i];
(3)    x[1] = 0;
(4)    x[2] >= 1;
(6)    x[2] <= x[2] - x[1];
```

Mais les contraintes 5, 7, 8 et 9 n'ont toujours pas été postées. Pour $k = 3$, les contraintes suivante sont postées :

```
(1)    x[1] < x[2];
       x[2] < x[3];
(2)    d[ind12] = x[ind12.j] - x[ind12.i];
       d[ind13] = x[ind13.j] - x[ind13.i];
       d[ind23] = x[ind23.j] - x[ind23.i];
(3)    x[1] = 0;
(4)    x[3] >= 3;
(5)    allDifferent(d[ind12],d[ind13],d[ind23]);
(6)    x[2] <= x[3] - x[2];
(7)    d[ind13] = d[ind12] + d[ind23];
(9)    x[2] = x[1] + 1 => x[3] != x[2] + 1;
       x[2] = x[1] + 2 => x[3] != x[2] + 2;
       x[2] = x[1] + 3 => x[3] != x[2] + 3;
```

Mais, la huitième contrainte n'a toujours pas été postée. Enfin, avec $k = 4$, on a :

```
...
(8)    d[ind14] = d[ind13] + d[ind24] - d[ind23];
...
```

Ainsi, le critère *toutes_les_contraintes* est satisfait. Il est à noter que le jeu de test avec les instances $\{1, 2, 3, 4\}$ couvre bien le critère choisi mais que d'autres jeux de test auraient pu être choisis, comme par exemple $\{4\}$ ou $\{5, 6, 7\}$. Nous ne cherchons d'ailleurs pas à minimiser la taille du jeu de test puisque cela pourrait conduire à diminuer nos chances de détecter des non-conformités. Un sujet intéressant serait de générer automatiquement les instances du problème qui garantissent la couverture des critères de test proposées. Il faut aussi noter que les langages de modélisation intègrent des instructions qui rendent le système de contraintes conditionnel. Par exemple, OPL propose une contrainte *if_then_else* qui peut être paramétrée par les valeurs des paramètres k.

Nous proposons ici d'autres critères de test pour les problèmes qui recherchent une seule solution (Fig.3). L'idée sous-jacente est de regarder la structure d'un modèle à contraintes dérivé du problème de conformité. En effet, la recherche de non-conformités peut être adressée avec la remarque suivante : Si $\exists k, sol(P_x(k) \wedge \neg M_x(k)) \neq \emptyset$ alors $P_x(k)$ est non-conforme à $M_x(k)$. Sachant que $M_x(k) \equiv (C_1 \wedge C_2 ... \wedge C_n)$, on a :

$P_x(k) \wedge \neg M_x(k) \equiv$
$(P_x(k) \wedge \neg C_1) \vee (P_x(k) \wedge \neg C_2) \vee ..(P_x(k) \wedge \neg C_n)$
Rechercher des non-conformités peut être fait en résolvant un ou plusieurs de ces disjonctions. On a donc les critères suivants :

Definition 10 (Une_contrainte_niée) *Le critère Une_contrainte_niée exige que $\exists i$ tel que $sol(P_x(k) \wedge \neg C_i) \neq \emptyset$.*

Definition 11 (Une_paire_de_contraintes_niées) *Le critère Une_paire_de_contraintes_niées exige que $\exists i$ et j tels que $sol(P_x(k) \wedge \neg C_i) \neq \emptyset \wedge sol(P_x(k) \wedge \neg C_j) \neq \emptyset$.*

et ainsi de suite.

Illustrons le premier critère sur un exemple contenant une non-conformité.

Exemple : Soit X un ensemble de variables de type entier, N est un entier positif et val une valeur entière. Nous avons le modèle de départ $M_x(k)$ et le CPUT $P_x(k)$ et nous nous intéressons à la recherche d'une seule solution :

$M_x(k)$:	$P_x(k)$:
$\exists R \subseteq X,\ \|R\| = N$: $(\forall x \in R :\ x = val) \wedge (\forall y \in X \backslash R :\ y \neq val)$	$atMost(N, X, val)$

La contrainte globale $atMost(N, X, val)$ est vraie ssi au plus N variables de X valent val. En fait, nous faisons l'hypothèse qu'un développeur a raffiné le modèle $M_x(k)$ en utilisant cette contrainte globale.

Prenons maintenant une instance du modèle où k_i vaut $(N = 2, val = 3)$:

$M_x(k \backslash k_i)$:
$(x = 3 \vee y = 3) \wedge (x = 3 \vee z = 3) \wedge (y = 3 \vee z = 3) \wedge$
$(x = 3 \vee y = 3 \vee z = 3) \wedge (x \neq 3 \vee y = 3 \vee z = 3) \wedge$
$(x = 3 \vee y \neq 3 \vee z = 3) \wedge (x = 3 \vee y = 3 \vee z \neq 3) \wedge$
$(x \neq 3 \vee y \neq 3 \vee z \neq 3)$
$P_x(k \backslash k_i) : atMost(2, \{x, y, z\}, 3)$

En observant que $M_x(k \backslash k_i)$ est une conjonction de 8 contraintes, notées $C_1, .., C_8$ et si on cherche à générer un jeu de test pour le critère *Une_contrainte_niée* avec le disjonct C_2, alors :

on recherche les solutions de $atMost(2, \{x, y, z\}, 3) \wedge \neg C_2$ où $\neg C_2 \equiv (y \neq 3 \wedge z \neq 3)$. Ici, on trouve par exemple le point $(x, y, z) = (3, 9, 84)$ qui témoigne d'une non-conformité. Ainsi, cette génération de test démontre la non-conformité du CPUT par rapport à son modèle-oracle. Mais, un autre disjonct aurait pu être sélectionné et la non-conformité aurait pu rester non detectée (par exemple le disjonct $atMost(2, \{x, y, z\}, 3) \wedge \neg C_8$).

En effet, le CPUT qui utilise la contrainte globale $atMost$ est non-conforme au modèle-oracle initial car il implémente en fait un autre modèle, le modèle $M'_x(k)$ suivant :

$$\exists R \subseteq X,\ |R| = N\ :$$
$$(\forall x \in R :\ x = val) \Rightarrow (\forall y \in X \backslash R :\ y \neq val).$$

Il est important de noter que pour ces derniers critères de test, nous utilisons la négation des contraintes. Or, cette négation n'est pas toujours simple à déterminer ou calculer. Considérez par exemple, la négation d'une contrainte globale telle que $golomb(Var)$. Ce point est limitatif et nécessite de plus ample développements dans notre approche.

4 Validation

Dans cette section, nous proposons une première validation expérimentale de la Théorie proposée. Nous cherchons à vérifier que les notions introduites sont compatibles avec le développement de problèmes issus de l'Industrie. Nous utilisons un exemple dérivé d'un problème réel, celui de l'ordonnancement des véhicules (POV) de la distribution OPL. Nous utilisons les techniques de mutation ("mutation testing" [2]) qui consistent à injecter des fautes dans le CPUT afin de valider la qualité d'un jeu de test. Ces insertions systématiques de fautes sont souvent utilisées pour la vérification expérimentale de techniques de test.

Le problème d'ordonnancement des véhicules [10] illustre plusieurs caractéristiques intéressantes de la PPC incluant le paramétrage étendu du modèle, l'utilisation de contraintes redondantes et globales, l'utilisation de structures de données spécialisées.

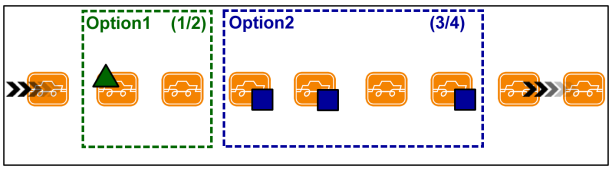

FIG. 7 – Chaine de montage.

Les contraintes de capacité qu'on trouve dans POV, sont formalisées en utilisant des contraintes de la forme " pas plus de r sur s " (r *outof* s), indiquant que chaque sous-séquence de s voitures, l'unité peut produire au maximum r voitures avec l'option en question (Fig. 7). Le problème d'ordonnancement des véhicules revient alors à trouver une affectation des voitures aux positions qui satisfait les contraintes de capacité. L'espace de recherche dans ce problème est composé des valeurs possibles des positions dans la chaîne de montage. Les voitures exigeant les mêmes options sont groupées dans une même classe. Dans ce qui suit, nous abuserons de la terminologie et nous référons simplement une voiture à une classe de voitures.

La partie gauche de la Fig. 8 donne le modèle–oracle de POV en OPL. Les variables *slot* sont des variables de décision (chaine de montage).

Le jeu de contraintes suivant représente les contraintes de capacité, qui sont exprimées en termes des variables d'installation des options($option[o][slot[s]]$). Si une contrainte de capacité pour une option o est de la forme " pas plus de l sur u ", on considère tous les sous-séquences de taille u de la chaîne de montage où il est nécessaire d'avoir au plus l éléments de la sous-séquence qui exigent l'option o. Ceci est exprimé dans OPL comme suit :

```
forall(o in Options & s in [1..nbSlots-capacity[o].u+1])
     sum(j in [s..s+capacity[o].u-1])
        option[o][slot[j]] <= capacity[o].l;
```

```
using CP;

int  nbCars    = ...; // # of cars
int   nbOptions = ...;// # of options
int   nbSlots   = ...;// # of slots

range   Cars    = 1..nbCars;
range   Options = 1..nbOptions;
range   Slots   = 1..nbSlots;

int demand[Cars] = ...;
int option[Options,Cars] = ...;

tuple Tcapacity {
    int l;
    int u;
};
Tcapacity capacity[Options] = ...;
int optionDemand[i in Options] = sum(j in Cars) demand[j] * option[i,j];
```

```
dvar int slot[Slots] in Cars;


subject to {
   // # of cars = demand
   forall(c in Cars )
      sum(s in Slots ) (slot[s] == c) == demand[c];

   forall(o in Options, s in 1..(nbSlots - capacity[o].u + 1) )
      sum(j in s..(s + capacity[o].u - 1))
            option[o][slot[j]] <= capacity[o].l;









};
```

- A -

```
dvar int slot[Slots] in Cars;
dvar int setup[Options,Slots] in 0..1;

subject to {
   // # of cars = demand
   forall(c in Cars )
      sum(s in Slots ) (slot[s] == c) == demand[c];

   forall(o in Options, s in 1..(nbSlots - capacity[o].u + 1) )
      sum(j in s..(s + capacity[o].u - 1))
            setup[o,j] <= capacity[o].l;

   forall(o in Options, s in Slots )
      setup[o,s] == option[o][slot[s]];

   forall(o in Options, i in 1..optionDemand[o])
      sum(s in 1 .. (nbSlots - i * capacity[o].u)) setup[o,s] >=
         optionDemand[o] - i * capacity[o].l;


};
```

- B -

FIG. 8 – $M_x(k)$ et $P_x(k)$ du POV en OPL.

L'efficacité de la résolution peut être améliorée en ajoutant une nouvelle structure de donnée `setup[o,s]`qui vaut 1 si l'option o est installée à la voiture qui occupe la position s, 0 sinon. A ce niveau, les variables `setup`et `slot`doivent être connectées :

```
forall(o in Options, s in Slots )
  setup[o,s] == option[o][slot[s]];
```

Il faut maintenant remplacer les `option[o,slot[s]]`par `setup[o,s]`. Ceci améliore nettement l'exécution du modèle. Puisque la résolution de contrainte sur des tableaux indexés de variable est couteuse. Il est ainsi préférable de factoriser ces expressions autant que possible.

Les contraintes redondantes n'enlèvent pas de solutions : elles expriment plutôt des propriétés sur les solutions qui peuvent aider la résolution à explorer l'espace de recherche plus efficacement. Un CPUT intermédiaire $P_x(k)$ peut être défini en ajoutant une contrainte redondante. Si l'option o a une contrainte de capacité "pas plus de l sur u", il s'ensuit que les dernières positions u peuvent contenir seulement l voitures demandant l'option o, donc les autres positions doivent contenir toutes les voitures restantes exigeant l'option o :

```
install[o,1] + ... +
install[o,nbPositions - u] >= demandeOption[o] - 1.
```

En règle générale, les dernières $k \times u$ positions peuvent seulement contenir $k \times l$ voitures exigeant l'option o.

```
install[o,1] + ... +
install[o,nbPositions - k*u] >= demandeOption[o] - k*1.
```

Ces contraintes sont définies dans OPL comme suit :

```
forall(o in Options & i in [1..DemandeOption[o]])
     sum(s in [1..nbPositions-i * capacite[o].u])
        install[o,s] >= DemandeOption[o] -i*capacite[o].l;
```

Ces raffinements nous amènent à au CPUT de la partie droite de la fig.8.

Si par erreur, dans cette contrainte redondante on a mis un " $<$ " au lieu du "$\geq$", le modèle intermediaire $P_x(k)$ na pas de solutions dans ce cas, on pourra dire à ce moment que $P_x(k)$ est non-conforme.
Nous avons créé 2 mutants du CPUT de POV, le tableau suivant montre les points de mutation dans le programme :

	Mutation
M1	3ème contrainte : s in Slots en s in Cars
M2	4ème contrainte : capacity[o].l en capacity[o].u

Le critère de couverture impose :

```
nbCars * nbOptions * nbSlots >= 1 ;
forall( o in Options) capacity[o].l <= capacity[o].u <= nbSlots ;
```

Nous prenons l'instance k_i qui satisfait le critère de couverture :

```
nbCars = 6;
nbOptions = 5;
```

```
nbSlots = 10;
demand = [1, 1, 2, 2, 2, 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]
       ];
capacity = [<1,2>,<2,3>,<1,3>,<2,5>,<1,5>];
```

Nous appliquons par la suite le critère Une_contrainte_niée en choisissant pour chaque mutant une contrainte à nier. Nous répétons ce processus encore une fois pour chaque mutant. Le tableau suivant donne l'ensemble des points de non-conformités atteints. **CC op.1** signifie : contrainte de capacité de l'option 1. On note s et non-s respectivement pour satisfaite et non-satisfaite :

	M1	**M1**
non-conformité	4 5 2 6 3 1 5 4 3 6	pas de solution
CC op.1	non-s	—
CC op.2	non-s	—
CC op.3	non-s	—
CC op.4	s	—
CC op.5	non-s	—

Le mutant 1 a été tué et on peut conclure qu'il n'est pas conforme au modèle original. Le mutant 2 n'a pas de solution ce qui nous permet de dire qu'il n'est pas conforme au modèle original.

5 Conclusion

Nous avons présenté les bases d'une première théorie du test pour les programmes à contraintes. Nous avons vu que la preuve de conformité entre le modèle original et le modèle raffiné est un problème difficile. Justement, nous avons adopté une démarche par réfutation qui consiste à aborder la problématique de la génération des cas de test de non-conformité. Ces cas de non conformité permettent d'exhiber les solutions du programme qui ne respectent pas sa spécification. Nous avons illustré cette stratégie sur des exemples, notamment le problème significatif de l'ordonnancement des véhicules. Le schéma de validation de la théorie donne une idée sur la difficulté du choix du critère de test. Les perspectives de ce travail sont multiples : la définition d'autres critères de test, la proposition d'un processus de génération des données de test pertinentes, le traitement de la négation des contraintes, et finalement l'automatisation du processus de test.

Références

[1] R. Backofen and S. Will. A constraint-based approach to fast and exact structure prediction in three-dimensional protein models. *Constraints*, 11(1) :5–30, 2006.

[2] R.A. DeMillo, R.J. Lipton, and F.G. Sayward. Hints on Test Data Generation : Help for the Practicing Programmer. *IEEE Computer Magazine*, 11(4) :34–41, Apr. 1978.

[3] P. Deransart, M. V. Hermenegildo, and J. Maluszynski, editors. *Analysis and Visualization Tools for Constraint Programming, Constrain Debugging (DiSCiPl project)*, volume 1870 of *Lecture Notes in Computer Science*. Springer, 2000.

[4] F. Fages, S. Soliman, and R. Coolen. Clpgui : A generic graphical user interface for constraint logic programming. *Constraints*, 9(4) :241–262, 2004.

[5] A. M. Frisch, W. Harvey, C. Jefferson, B. Martínez-Hernández, and I. Miguel. Essence : A constraint language for specifying combinatorial problems. *Constraints*, 13(3) :268–306, 2008.

[6] J. B. Goodenough and Susan L. Gerhart. Toward a theory of test data selection. In *Proceedings of the international conference on Reliable software*, pages 493–510, New York, NY, USA, 1975. ACM.

[7] J. S. Gourlay. *Theory of testing computer programs*. PhD thesis, Ann Arbor, MI, USA, 1981.

[8] L. Langevine, P. Deransart, M. Ducassé, and E. Jahier. Prototyping clp(fd) tracers : a trace model and an experimental validation environment. In *WLPE*, 2001.

[9] 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.

[10] B. D. Parrello, Waldo C. Kabat, and L. Wos. Job-shop scheduling using automated reasoning : a case study of the car-sequencing problem. *J. Autom. Reason.*, 2(1) :1–42, 1986.

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

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

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

[14] E. J. Weyuker and T. J. Ostrand. Theories of program testing and the application of revealing subdomains. *IEEE Trans. Softw. Eng.*, 6(3) :236–246, 1980.