Constraints (2012) 17:123–147
DOI 10.1007/s10601-012-9116-0

A CP framework for testing CP

Nadjib Lazaar · Arnaud Gotlieb · Yahia Lebbah

Published online: 9 March 2012

Abstract The success of several constraint-based modeling languages such as OPL, ZINC or COMET, appeals for better software engineering practices, particularly in the testing phase. This paper introduces a testing framework enabling automated test case generation for constraint programming. 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, global constraint and symmetry-breaking to form an improved constraint model that must be thoroughly 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. We implemented this approach in a new tool called CPTEST that was used to automatically detect non-conformities on classical benchmark programs, such as the Golomb rulers, n-queens, social golfer and the car-sequencing problems.

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

N. Lazaar (✉)
INRIA Rennes Bretagne Atlantique, Campus Beaulieu, Rennes, France
e-mail: nadjib.lazaar@inria.fr

A. Gotlieb
Certus Software V&V Center, Simula Research Laboratory, Oslo, Norway
e-mail: arnaud@simula.no

Y. Lebbah
Université d'Oran Es-Senia, Lab. LITIO, B.P. 1524 EL-M'Naouar, Oran, Algérie
e-mail: ylebbah@gmail.com

1 Introduction

Constraint programs such as those written in modern Constraint Programming languages and platforms (e.g. OPL[1], COMET[2], ZINC [3], CHOCO[4], GECODE[5], ...), aim at solving industrial combinatorial problems that arise in optimization, planning or scheduling. Recently, a new trend has emerged that proposes also to use CP programs to address critical applications in e-Commerce [11], air-traffic control and management [8, 13] or critical software development [5, 10]. While constraint program debugging drew the attention of many researchers, little support in terms of software engineering and testing has been proposed to help verify critical constraint programs. Automatic debugging of constraint programs has been an important topic of the OADymPPaC[6] project, that resulted in the definition of generic trace models [6, 14], the development of post-mortem trace analyzers, such as Codeine for Prolog, Morphine [14] for Mercury, ILOG Gentra4CP, or JPalm/JChoco. These models and tools help understand constraint programs and contribute to their optimization and correctness, but they are not dedicated to systematic fault detection. Indeed, functional fault detection requires the definition of a reference (called an oracle in software testing) in order to check the conformity between an implementation and its reference [23]. More recently, Castañeda et al. [15] propose to use a constraint model for testing a deployed financial trading system of continuous double auctions. In this work, the constraint model is used as a testing oracle for a conventional program. Automatic fault detection also requires the definition of test purpose to decide when to stop testing [24]. However, conventional software development benefits from research advances in software verification (including static analysis, model checking or automated test data generation), developers of constraint programs are still confined to perform systematic verification by hand.

Automatic constraint program testing cannot be easily handled by existing testing approaches because of the two following reasons: firstly, constraint programs are intrinsically non-deterministic as they represent sets of solutions and conventional definitions of program conformity do not apply; secondly, the refinement process of constraint programs is specific to CP. Indeed, developers usually start with an initial declarative constraint model of the problem, which faithfully captures the problem specification, without considering its performance. As this model cannot handle large-sized instances of the problem, several refinement techniques are used to build an improved model. For example, usual refinement techniques include the use of dedicated data structures, constraint reformulation, global constraints, redundant and surrogate constraint as well as constraints which break symmetries and usually improve considerably the effectiveness of the solving process. The refinement process, carried out by the developer, is an error-prone process and we

[1] www.ilog.com/products/oplstudio/

[2] www.dynadec.com/support/downloads/

[3] www.g12.cs.mu.oz.au/

[4] choco.sourceforge.net

[5] www.gecode.org

[6] pauillac.inria.fr/∼contraintes/OADymPPaC/

believe that most of the faults are introduced during this step. Although writing a constraint model solving a given problem is sometimes challenging, refining a constraint model using model transformations is even harder. In fact, this process requires both knowledge of the problem domain in order to identify optimization opportunities, and also a deep understanding of the constraint solver capabilities.

In this article, we propose a testing framework for checking the correctness of a constraint program implementation. The oracle for the constraint program under test is an initial declarative model considered to be valid w.r.t. the user requirements. Our framework is based on the definition of four distinct conformity relations to handle both constraint solving problems and optimization problems. A practical consequence of these definitions is the proposal of test purposes for evaluating the conformance of constraint programs. Note that this paper does not address another essential topic of CP verification which is the correctness of solvers or optimizers. This problem is also essential to CP but it can be addressed with conventional testing approaches as most of CP implementations are done in classical languages such as C, C++ or Java. We propose an algorithm for checking the correctness of a constraint program under test by solving a set of derived constraint problems that try to raise non-conformities. We implemented our approach in a tool called CPTEST[7] that seeks non-conformities in Optimization Programming Language programs. CPTEST was used to find non-conformities in various OPL constraint programs such as the Golomb rulers, n-queens, social golfer and car-sequencing problems. By using fault injection, we were able to show that CPTEST can find almost all the non-conformities between a high-declarative model and its optimized implementation.

The rest of the paper is organized as follows: Section 2 illustrates our testing framework on a simple example in order to show a typical non-conformity case. Section 3 introduces our notations and hypothesis. Section 4 presents our testing framework and gives the definition of conformity relations. In Section 5, our algorithms for finding non-conformities are introduced and illustrated. Section 6 presents the CPTEST tool and Section 7 details our experimental evaluation. Finally, Section 8 concludes the paper and draws some perspectives to this work.

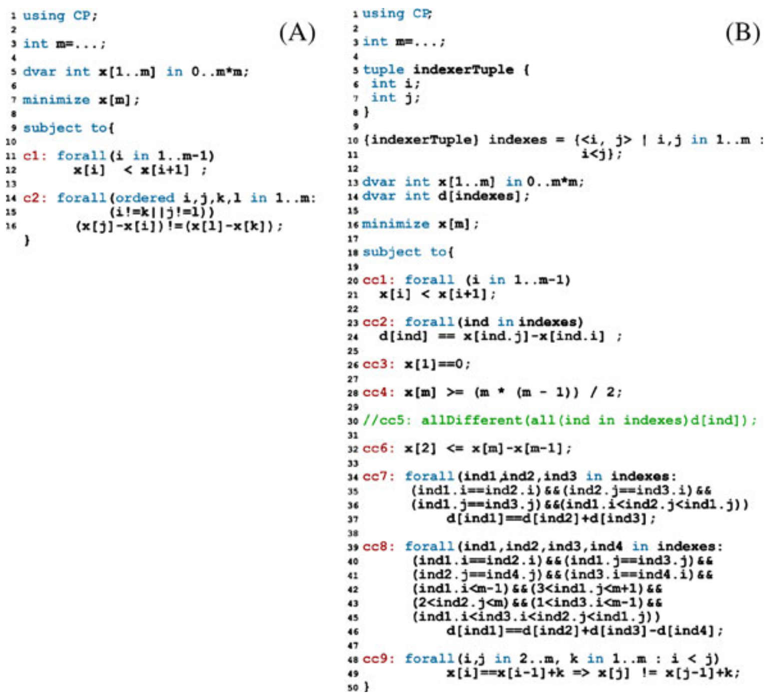
2 An illustrative example

The Golomb rulers is a classical CP problem which has various applications in fields such as Radio communications or X-Ray crystallography. A Golomb ruler [17] is a set of m marks $0 = x_1 < x_2 < \ldots < x_m$, where $m(m-1)/2$ distances $\{x_j - x_i | \ 1 \leq i < j \leq m\}$ are distinct. A ruler is of *order* m if it contains m marks, and it is of *length* x_m. The goal of Golomb rulers problem is to find a ruler of order m with minimal length (*minimize* x_m) (see prob006 in CSPLib[8]).

A declarative model of this problem is given in part `A` of Fig. 1 while part `B` presents a refined and improved model. It easy to convince a CP developer that

[7]www.irisa.fr/celtique/lazaar/CPTEST/

[8]www.csplib.org



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

```
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[1]==0;

cc4: x[m] >= (m * (m - 1)) / 2;

//cc5: allDifferent(all(ind in indexes)d[ind]);

cc6: x[2] <= x[m]-x[m-1];

cc7: forall(ind1,ind2,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,j in 2..m, k in 1..m : i < j)
            x[i]==x[i-1]+k => x[j] != x[j-1]+k;
}
```

Fig. 1 $M_x(k)$ and $P_x(k)$ of Golomb rulers problem in OPL

model `A` actually solves the Golomb rulers problem, but it is more difficult for model `B`. Indeed, model `B` uses a matrix (i.e., `d[indexes]`) as new decision variables to represent distances between marks . The auxiliary variables are systematically connected to the basic variables (i.e., `x`) using a channeling constraint (i.e., `cc2`). The model `B` statically breaks symmetries (i.e., `cc6`), it contains redundant, surrogate, implied and global constraint (i.e., `cc5,cc7,cc8,cc9,`). We can find also as a refinement, a new objective function on `d`. In this paper, we address the fundamental question of revealing non-conformities in between the constraint program under test `B` and the model-oracle `A`. Testing `B` before using it on large instances of the problem (when $m > 15$) is highly desirable as computing a global minimum of the problem for these instances may require computation time greater than a week. Note that `B` is syntactically correct and provides correct Golomb rulers for small values of m. Our testing framework tries to find an instantiation of the variables that satisfies the constraints of `B` and violates at least one constraint of `A`. This testing process is detailed in Section 5. With $m = 8$, our CPTEST framework returns $x = [0\ 1\ 3\ 6\ 10\ 26\ 27\ 28]$ in less than 6 *sec* on a standard machine[9], indicating that `B` contains a non-conformity w.r.t. `A` and therefore contains a fault. Indeed, x is not

[9]Intel Core2 Duo CPU 2.40Ghz with 2 GB of RAM

a Golomb ruler as $27 - 26 = 1 - 0 = 1$. In fact, a careful analysis of model `B` shows that this non-conformity can be tackled by adding constraint `cc5` (which is written as comment in part `B`). If `cc5` is added, CPTEST provides a *conformity certificate* showing that the CP program actually computes a global minimum in 10 034.69 *sec.* (about 3 hours).

3 Notations and hypothesis

3.1 Notations

A constraint program includes a constraint model $\mathcal{M}$, which is a conjunction of constraints C_i over a set of decision variables noted x parameterized by a set of parameters noted k. For the Golomb rulers, k is the order of the ruler while x represents the vector of marks. (e.g., If $k = 3$ then a possible ruler is `x =[0 1 3]` while if $k = 4$, it is `x =[0 1 4 6]`). A constraint program includes also a generic procedure $solve()$ representing either the call to a constraint solver or the call to an optimization procedure. In this latter case, we note f the cost function (for the sake of clarity, we will consider only minimization problems). We consider that k belongs to $\mathcal{K}$ the set of possible values of the parameters. $sol(\mathcal{Q})$ denotes the set of solutions of $\mathcal{Q}$ and $Proj_y(sol(\mathcal{Q}))$ expresses the projection of $sol(\mathcal{Q})$ on the set of variables y where y is a subset of the decision variables x (i.e., $y \subseteq x$). For instance, let us take the following constraint program:

$$Q \equiv x, y, z \in 1..3 : allDifferent(x, y, z)$$

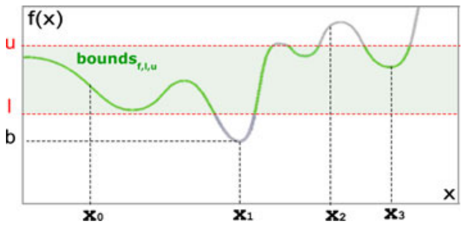
- $sol(Q) = \{(1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1)\}$
- $Proj_{\{x,y\}}(sol(Q)) = \{(1, 2), (1, 3), (2, 1), (2, 3), (3, 1), (3, 2)\}$

In optimization problems, one usually starts with feasible solutions ranging in a cost interval $[l, u]$. Therefore, we introduce the set

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

To illustrate these notations, Fig. 2 shows an example of a real valued objective function where point x_1 is a global minimum with a cost b, points x_0, x_3 belong to

Fig. 2 Objective solutions



$Bounds_{f,l,u}(\mathcal{Q})$ and x_1, x_2 do not belong to $Bounds_{f,l,u}(\mathcal{Q})$. This example stresses that the (unknown) global minimum might not be included in $Bounds_{f,l,u}(\mathcal{Q})$.

3.2 Hypothesis

Our testing framework, described in the next section, is based on two working hypothesis. In our work, we focus on faults introduced in constraint models while making the assumption that the underlying constraint solver is correct, called the *solver correctness hypothesis*. Of course, this is a strong assumption as optimized solvers usually result from long-term and error-prone software development process. However, faults in constraint solvers that are developed in programming languages such as C++, C or Java, can be tackled by classical software testing techniques. Another working hypothesis of our framework is usually called the *competent developper hypothesis* [7]. It states that, even some faults are introduced, the resulting program of a development process would certainly fulfill almost all of its specifications. In other words, the resulting program may contain faults, but it will certainly solve the problem it has been designed for. This is a classical hypothesis we need in our testing framework to focus on practical defects found in constraint programs development processes.

4 A framework for testing constraint programs

4.1 Model–Oracle and CPUT

In our framework, we consider the initial declarative constraint model as a testing oracle, called the *Model–Oracle* , noted $\mathcal{M}_x(k)$. $\mathcal{M}_x(k)$ represents all the solutions of the problem and strictly conforms to the problem specifications; this is why we call it "Oracle". We suppose that, for any parameter instantiation, $\mathcal{M}_x(k)$ possesses at least one solution. Considering unsatisfiable Model–Oracle could be interesting for some applications (e.g., Software Verification [10]) but we excluded them from the framework, because we did not want to deal with equivalence of unsatisfiable models.

The *Constraint Program Under Test* (*CPUT*) is a model noted $\mathcal{P}_z(k)$, that refines the Model–Oracle. Possible refinements include the use of auxiliary variables to build channeling constraints [3], the use of redundant, surrogate, or implied constraints [21], the usage of global constraints [1, 19], objective function reformulation [20], symmetry breaking techniques [16, 18] [2]. Figure 3 shows the possible model refinements that can be performed to get an optimized model.

In our framework, we are interested in the testing of the CPUT against the Model–Oracle. Our ultimate goal is to find non-conformities between those two models, that solve the same problem. The non-conformities we are looking for are those that correspond to defects introduced during the refinement process. Note that, unlike $\mathcal{M}_x(k)$, $\mathcal{P}_z(k)$ is intended to solve difficult instances of the problem. We built our framework on the hypothesis that checking whether $M_{(x\backslash x_0)}(k_0)$ is true where x_0

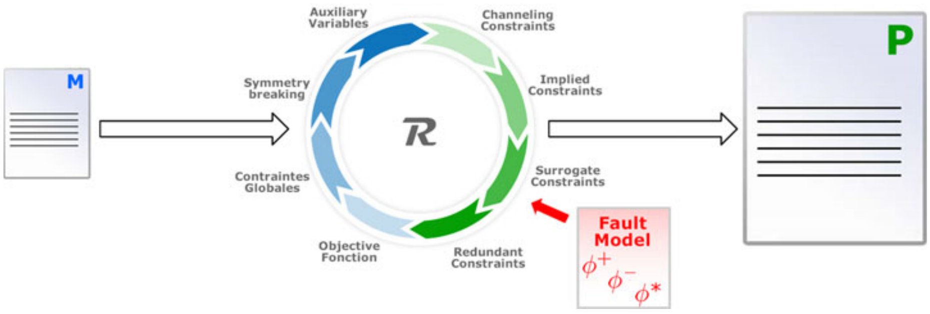


Fig. 3 Refinements and fault model

is a point of the search space is not hard, while finding such an x_0 satisfying the constraints may be hard. Note also that, using the *competent developper hypothesis*, we assume that $\mathcal{P}_z(k)$ and $\mathcal{M}_x(k)$ are strongly related, share about the same set of variables (i.e., $x \subseteq z$) even if $\mathcal{P}_z(k)$ can have more variables, and more importantly they are intended to solve the same problem. The variables of z that are distinct of x (called *auxiliary variables* in our framework), become instantiated as soon as variables in x are instantiated. In other words, the values of those variables depend on the values of x.

4.2 Fault model

In our framework, we are interested in finding faults that correspond to non-conformities. Other types of faults such as syntactical faults, mostly due to ill-formed constraints (e.g., $z > x +$ or $x + *y$), can be tackled by syntax checkers, while faults in the constraint solvers are not considered here due to the *solver correctness hypothesis*.

In our context, a fault is defined with one of the following definitions:

Definition 1 (Positive fault ϕ^+) Given a CPUT $\mathcal{P}_z(k)$ and a Model–Oracle $\mathcal{M}_x(k)$, a *positive fault*, noted ϕ^+, is defined as:

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

Definition 2 (Negative fault ϕ^-) A *negative fault*, noted ϕ^-, is defined as

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

Definition 3 (Unsat fault ϕ^*) An *unsat fault*, noted ϕ^*, is defined as:

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

Note that the unsat fault definition makes implicit reference to the satisfiability of $\mathcal{M}_x(k)$. Recall that this is an assumption of our framework. Using these definitions, any fault is defined as a non-conformity between the CPUT $\mathcal{P}_z(k)$ and its Model–Oracle $\mathcal{M}_x(k)$.

4.3 Conformity relations

The correctness of a CPUT w.r.t. its Model–Oracle is approached through the notion of conformity relation. Such a relation aims at assessing the correctness of the CPUT, a notion that can be expressed with various levels of depth. We propose four set-based definitions of conformity divided in two groups: conformity relations adapted for constraint solving problems and conformity relations for optimization problems.

Conformity relations for constraint solving problems The simplest definition of correctness, well-adapted for problems where a single solution is sought, is given by the following conformity relation:

Definition 4 $(conf_{\text{one}})$

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

Roughly speaking, for a given instance k, $conf^k_{\text{one}}$ asks the solutions of the CPUT to be included in the solutions of the Model–Oracle. As an example, Fig. 4 presents both the sets $sol(\mathcal{M}_x(k))$ noted M and $Proj_x(sol(\mathcal{P}_z(k)))$ noted P, where red crosses raise non-conformities (i.e., faults in the CPUT) while green points are conforming w.r.t. the Model–Oracle. Parts (a)(b) of Fig. 4 exhibit non-conformities as solving $\mathcal{P}_z(k)$ can lead to solutions which do not satisfy $\mathcal{M}_x(k)$. Part (c) does not exhibit non-conformity but, as P does not contain any solution, it does not conform the Model–Oracle for $conf_{\text{one}}$. This example also shows that unsatisfiable models must be considered as non-conforming w.r.t. Model–Oracles, in order to tackle unsatisfiable CPUTs. On the contrary, part (d) of Fig. 4 shows that $\mathcal{P}_z(k)$ conforms $\mathcal{M}_x(k)$ for $conf_{\text{one}}$, as P cannot contain any non-conformity points.

Whenever all the solutions are sought, another definition of conformity is useful:

Definition 5 $(conf_{\text{all}})$

$$\begin{aligned} P\ conf^k_{\text{all}}\ M &\triangleq Proj_x(sol(\mathcal{P}_z(k))) = sol(\mathcal{M}_x(k))\ (\neq \emptyset) \\ P\ conf_{\text{all}}\ M &\triangleq (\forall k \in \mathcal{K}, P\ conf^k_{\text{all}}\ M) \end{aligned}$$

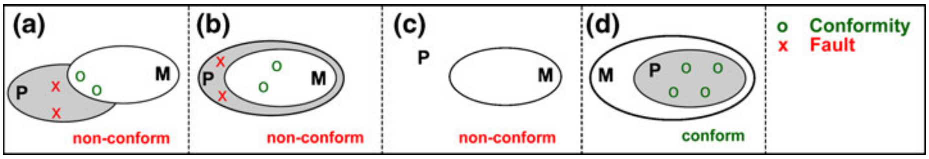


Fig. 4 $conf_{\text{one}}$ on $\mathcal{P}_z(k)$ and $\mathcal{M}_x(k)$

Satisfying this conformity relation is very demanding and is not always required or desired. For instance, the CPUT in part `B` of Fig. 1 includes constraints that break symmetries of the problem (e.g., `cc6`), which yields to lose solutions from the Model–Oracle. As a result, those two models cannot be conform w.r.t. $conf_{\text{all}}$.

In Fig. 5, parts (a)(b) and (c) exhibit non-conformities. Part (c) shows a solution of the Model–Oracle which is not solution of the CPUT; therefore, the CPUT is a faulty over-constrained model. Part (b) exhibits the opposite case where the CPUT is a faulty under-constrained model. Proving that $\mathcal{P}_z(k)$ conforms $\mathcal{M}_x(k)$ for one of these two conformity relations is highly desirable. Unfortunately, such a proof would require to find all the solutions of the CPUT, which is an NP-hard problem for many constraint languages (e.g., the finite domains constraint language).

Conformity relations for optimization problems Conformity relations for optimization problems are harder to define, as CP developers usually start their refinement process with the definition of bounds for the optimal case [20]. Note also that non-conformities may arise in the cost function and our conformity relations should be able to tackle these cases.

Figure 6 presents the conformity relation where feasible solutions of the CPUT are sought in $[l, u]$ (i.e., l: lower bound, u: upper bound). B_P denotes the set $bounds_{f',l,u}(P_x(k))$, B_M denotes the set $bounds_{f,l,u}(\mathcal{M}_x(k))$ while B is the set of global minima of $\mathcal{M}_x(k)$. Part (a) exhibits four non-conformities as these points are not feasible solutions of the Model–Oracle $\mathcal{M}_x(k)$ in $[l, u]$. For the same reason, Part (b) exhibits two non-conformities as two feasible solutions of B_P with cost in $[l, u]$ do not belong to B_M. Part (c) presents also a non-conformity as B_P does not contain any feasible point meaning that the minimization problem cannot find a feasible solution with cost in $[l, u]$. On the contrary, part (d) shows conformity because solutions of B_P belong to B_M. Formally speaking,

Definition 6 ($conf_{\text{bounds}}$)

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

Note that the definition of $conf_{\text{bounds}}$ does not require that $f = f'$ and then cases where the objective function has been refined can also be handled. However, the

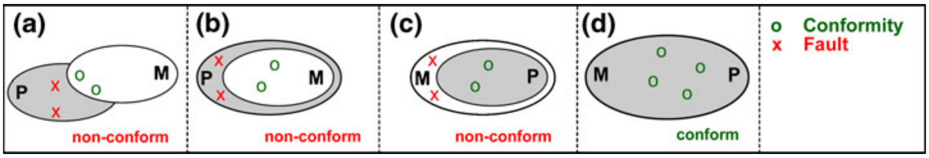


Fig. 5 $conf_{\text{all}}$ on $\mathcal{P}_z(k)$ and $\mathcal{M}_x(k)$

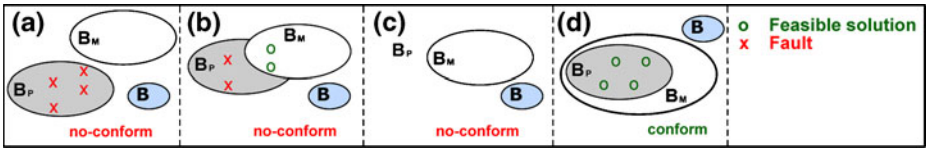


Fig. 6 $conf_{\text{bounds}}$ on $P_x(k)$ and $\mathcal{M}_x(k)$

refinement on the objective function has to preserve the values of the objective function where the bounds $[l, u]$ are defined, for both f and f'. The conformity relation $conf_{\text{bounds}}$ is useful for addressing hard optimization problems as it does not require the computation of global minima. As a result, it can be used to assess the correctness of models on relaxed instances of the global optimization problems. We will come back to this advantage in the experimental validation section.

Note that, for some problems, it may be useful to assess not only the correctness but also to guarantee the computation of optimal solutions. This can be performed by using the conformity relation $conf_{\text{best}}$ which ensures that a global optimum belongs to $[l, u]$. A conformity $conf_{\text{best}}$ is equivalent to $conf_{\text{one}}$ where, all global minima of a given CPUT have to be global minima of the Model–Oracle.

5 Algorithms for finding non-conformities

Testing a CPUT w.r.t. its Model–Oracle requires to select test data. In our framework, a test data defines an instance of the CPUT and a single point of the search space.

Definition 7 (Test data) Given a CPUT $\mathcal{P}_z(k)$ and its Model–Oracle $\mathcal{M}_x(k)$, a *test data* is a pair (k_0, x_0) where k_0 is a parameter value and x_0 is a vector of variable values.

Evaluating $M_k(x)$ on the test data (k_0, x_0) returns *true* when x_0 is a solution of the model, *false* otherwise. Test execution is performed by evaluating both $P_{z\backslash z_0}(k_0)$ and $M_{x\backslash x_0}(k_0)$, checking whether the results are the same, and issuing a test verdict that can be either *pass* or *fail*. It is worth noting that test verdicts depend on the selected conformity relation, as shown in Table 1.

Any *fail* in the testing process corresponds to a non-conformity:

Table 1 Possible test verdicts on (k_0, x_0) for the various conformity relations

Model–Oracle	CPUT	Test verdict ($conf_{\text{one}}, conf_{\text{bounds}}, conf_{\text{best}}$)	Test verdict ($conf_{\text{all}}$)
true	*true*	*pass*	*pass*
true	*false*	*pass*	*fail*
false	*true*	*fail*	*fail*
false	*false*	*pass*	*pass*

Definition 8 (Non-conformity) For a given parameter value k, a non-conformity between a CPUT and its Model–Oracle, due to a fault ϕ (either ϕ^+,ϕ^- or ϕ^*), is a test data x on which the test execution produces a verdict $fail$. It is noted $nc(\phi, k, x)$.

Finding non-conformities is the ultimate goal of our testing framework. The following sections detail algorithms to find non-conformities, depending the selected conformity relation.

5.1 An algorithm based on $conf_{\text{one}}$

For the $conf_{\text{one}}$ conformity relation, only faults of type ϕ^* and ϕ^+ are relevant. In fact, faults of type ϕ^- are accepted because the CPUT can "loose" solution in this case. Recall that $conf_{\text{one}}$ corresponds to situations where the CPUT finds only one solution, while the Model–Oracle computes all the solutions.

Given an instance $k_0 \in K$, Algorithm 1 checks first the satisfiability of the CPUT. If it has no solution, then it returns a non-conformity $nc(\phi^*, k_0, -)$. If it has solutions, then a generator of non-conformities called *one_negated*, is called. This generator is detailed below in Section 5.4. Roughly speaking, calling *one_negated(A,B)* returns, if possible, a solution of A which is not a solution of B. In Algorithm $algo_{\text{one}}$, calling *one_negated* enables finding a solution of the CPUT which is not a solution of the Model–Oracle. As a result, a non-conformity $nc(\phi^+, k_0, x_0)$ is generated. Whenever Algorithm $algo_{\text{one}}$ cannot return a non-conformity for all instances in K, then the algorithm returns a conformity certificate (noted *conf-certificate*(K)). It is worth noticing that the delivered certificate is only valid for the instances of K. It cannot be used to assess the conformity of the CPUT w.r.t. its Model–Oracle for all the possible instances of the problem.

Algorithm 1:

```
Input  : Model–Oracle $\mathcal{M}_x$, CPUT $\mathcal{P}_z$, a set of instances $K$
Output: $nc(\phi, k_0, x_0)$ or $conf\text{-}certificate(K)$
while $K \neq \emptyset$ do
    $k_0 \leftarrow one\_of(K)$; $K \leftarrow K \backslash \{k_0\}$
    if $sol(\mathcal{P}_z(k_0)) = \emptyset$ then return $nc(\phi^*, k_0, -)$
    $V \leftarrow one\_negated(\mathcal{P}_z(k_0), \mathcal{M}_x(k_0))$
    if $V \neq \emptyset$ then
        $x_0 \leftarrow one\_of(V)$
        return $nc(\phi^+, k_0, x_0)$
    end
end
return $conf\text{-}certificate(K)$
```

The $one\_of(A)$ denotes a function that returns (non-deterministically) one of the elements of A.

5.2 An algorithm based on $conf_{\text{all}}$

Algorithm 2 is similar to Algorithm 1 except that it is also able to detect non-conformities related to ϕ^- fault. The only difference lies in the additional call to *one_negated*$(\mathcal{M}_x(k_0), \mathcal{P}_z(k_0))$.

When those algorithms do not return any non-conformity for the set K of instances, it means that either $\mathcal{M}$ $conf_{\text{one}}$ $\mathcal{P}$ or $\mathcal{M}$ $conf_{\text{all}}$ $\mathcal{P}$ for thoses instances. Hence, we also get correctness proofs restricted to the considered set of instances. This is a strong result, but in practice it comes at a high cost, because the whole search space for each value of K usually needs to be explored in those cases.

Algorithm 2:

```
Input   : Model–Oracle $\mathcal{M}_x$, CPUT $\mathcal{P}_z$, a set of instances $K$
Output: $nc(\phi, k_0, x_0)$ or $conf\text{-}certificate(K)$
while $K \neq \emptyset$ do
    $k_0 \leftarrow one\_of(K)$; $K \leftarrow K \setminus \{k_0\}$
    if $sol(\mathcal{P}_z(k_0)) = \emptyset$ then  return $nc(\phi^*, k_0, -)$
    $V \leftarrow one\_negated(\mathcal{P}_z(k_0), \mathcal{M}_x(k_0))$
    if $V \neq \emptyset$ then
        $x_0 \leftarrow one\_of(V)$
        return $nc(\phi^+, k_0, x_0)$
    end
    $V \leftarrow one\_negated(\mathcal{M}_x(k_0), \mathcal{P}_z(k_0))$
    if $V \neq \emptyset$ then
        $x_0 \leftarrow one\_of(V)$
        return $nc(\phi^-, k_0, x_0)$
    end
end
return $conf\text{-}certificate(K)$
```

5.3 An algorithm based on $conf_{\text{bounds}}$

A feasible solution is a solution with cost in $[l, u]$. A fault ϕ^* or ϕ^+ in the CPUT yields to introduce a non-conformity within $[l, u]$. Algorithm 3 checks first the satisfiability of the CPUT in $[l, u]$. If it has no solution, it returns $nc(\phi^*, k_0, -)$. In the other case, Algorithm 3 tries to find non-conformities with *one_negated*, of the form $nc(\phi^+, k_0, x_0)$. Finally, a conformity certificate (i.e., *conf-certificate*(K)) is provided in case if the algorithm cannot return such a non-conformity in a given interval $[l, u]$. As for the two other algorithms, this one provides a certificate that is valid only for the considered instances of K.

In the case of a global minimization problem where $conf_{\text{best}}$ can be used, the algorithm is similar to the one that has been presented. It seeks non-conformities revealing faults of type ϕ^+ or ϕ^*, by asking the *one_negated(A,B)* generator to look for a global minimum of A which is not a global minimum of B.

Interestingly, any non-conformity found by one of these algorithms can be stored for further investigations. Indeed, it can be used to debug the CPUT by looking at the violated constraint and the kind of detected fault (i.e., ϕ^+, ϕ^- or ϕ^*). The reported non-conformities can also complete a test set that serves to assess the correctness of the CPUT.

```
Algorithm 3:
  Input  : Model–Oracle ($\mathcal{M}_x(k) \equiv \mathcal{C}_x(k) \wedge minimize(f(x))$),
           CPUT ($\mathcal{P}_z(k) \equiv \mathcal{C}'_z(k) \wedge minimize(f'(z))$),
           instance set $K$, bounds $[l, u]$
  Output: $nc(\phi, k_0, x_0)$ or $conf\text{-}certificate(K)$
  while $K \neq \emptyset$ do
      $k_0 \leftarrow one\_of(K)$; $K \leftarrow K \backslash \{k_0\}$
      $A \leftarrow \mathcal{C}_x(k_0) \wedge (l \leq f(x) \leq u)$
      $B \leftarrow \mathcal{C}'_z(k_0) \wedge (l \leq f'(z) \leq u)$
      if $sol(B) = \emptyset$ then return $nc(\phi^*, k_0, -)$
      $V \leftarrow one\_negated(B, A)$
      if $V \neq \emptyset$ then
          $x_0 \leftarrow one\_of(V)$
          return $nc(\phi^+, k_0, x_0)$
      end
  end
  return $conf\text{-}certificate(K)$
```

5.4 The generator of non-conformities

The algorithms proposed above are all based on a generic generator of non-conformities, called *one_negated*:

The goal of Algorithm 4, that takes two constraint sets A and B as inputs, is to find a solution of:

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

This system correspond to any solution of A that is not solution of B. The generator selects a constraint C_i to negate and tries to find a solution of $(A \wedge \neg C_i)$. Provided that the underlying constraint solver is correct (*solver correctness hypothesis*), Algorithm *one_negated* is sound and complete. Soundness and completeness both come from the computation of $sol(A \wedge Proj_{\mathrm{vars(A)}}(\neg C_i))$ which is performed by the underlying constraint solver.

```
Algorithm 4: one_negated($A$, $B$)
  Input  : $A$ and $B = \{C_1, \ldots, C_n\}$, two sets of constraints
  Output: A singleton non-conformity $\{x_i\}$ if there is one, or $\emptyset$
  foreach $C_i \in B$ do
      $V \leftarrow vars(C_i)/vars(A)$
      if $V = \emptyset$ then $set \leftarrow sol(A \wedge \neg C_i)$
      else $set \leftarrow sol(A \wedge Proj_{vars(A)}(\neg C_i))$
      if $set \neq \emptyset$ then return $singleton(one\_of(set))$
  end
  return $\emptyset$
```

The $singleton(x)$ denotes a function that returns a singleton made from x.

When $vars(A) \subset vars(B)$, constraint projection is required. Such cases arise when channeling constraints through auxiliary variables have been used. Another unrelated problem concerns the computation of negation. In the rest of the section, we elaborate on those questions:

Projection Using auxiliary variables as a possible refinement adds new dimensions to the CPUT w.r.t the Model–Oracle. For example, consider a Model–Oracle $\mathcal{M}$ with: $x - y \neq x - z$; $x - y \neq y - z$; $x - z \neq y - z$; and a CPUT $\mathcal{P}$ with $c_1 : x - y = d_1$; $c_2 : x - z = d_2$; $c_3 : y - z = d_3$; $c_4 : allDifferrent(d_1, d_2, d_3)$;. Here, the CPUT defines 3 auxiliary variables (d_1, d_2 and d_3) within 3 channeling constraints (c_1, c_2 and c_3). Both constraint systems are semantically equivalent and we have $\mathcal{P} conf \mathcal{M}$ for any of the conformity relations related to constraint satisfaction. However, if we select c_2 for negation, the resulting system $\mathcal{M} \wedge \neg c_2$ has solutions. These solutions correspond to false alarms because instantiations of d_2 are unrelevant if we are interested in the conformity between the CPUT and its Model–Oracle. To bypass this problem of false alarms, we introduced constraint projection (i.e., $Proj_X$) in our framework. However, in practice, implementing constraint projection is challenging. A first option is to consider Fourier elimination when the system is composed only of linear inequalities [9, 12]. A second option is to let the user annotate channeling constraints. In the above example, only the *allDifferrent* constraint should be selected for negation. So, if the user annotates constraints c_1, c_2 and c_3 as channeling constraints, then our framework selects only the *allDifferrent* constraint. Note that these channeling constraints have to be present in the resulting system:

$$\big(\mathcal{M} \wedge (c_1 \wedge c_2 \wedge c_3)\big) \wedge \neg alldifferrent(d_1, d_2, d_3)$$

A third option is to automatically identify channeling constraints. We will discuss this option in the Implementation section.

Negation The *one_negated* algorithm is based on refutation to reach a solution of A which is not a solution of B. In general, primitive constraints (i.e., $(\{\wedge, \vee, =, \neq, <, \geq\})$ are closed under negation. The difficulty lies is the negation of global constraints. Of course, a simple syntactic negation is sometimes possible, by rewriting the global constraint into a set of simpler constraints, by using its declarative specification. For example,

$$allDifferent(x) \equiv \forall i, j : (i \neq j \Rightarrow x_i \neq x_j)$$

and:

$$\neg allDifferent(x) \equiv \exists i, j : (i \neq j \wedge x_i = x_j)$$

From an operational point of view, this transformation, although it is sound, is not satisfying as it introduces disjunction operators. Another option would be to express the negation by using other existing global constraints. For instance, $\neg atLeast(n, X, val) \equiv atMost(n - 1, X, val)$ and vice-versa. Looking at the most recent version of the global constraints catalog [1], we found that more than 20 global constraints may be negated this way.

A keypoint of our testing framework is that test data can be automatically generated using the same constraint solver as the one used for solving the CPUT. Using the *solver correctness hypothesis*, our framework only search for non-conformities in models. A difficult point however is to determine whether the approach can be used in practice. In particular, as we are interested in testing the CPUT, we have to examine whether finding non-conformities is cheaper than solving the CPUT. Another keypoint of our framework is that we can select small instances of the problem to check conformity. We will discuss this point in the experimental section.

6 Implementation

We implemented the testing framework shown above in a tool called CPTEST for OPL (Optimization Programming Language [22]). We chose OPL because it is one of the main programming environments for developing constraint programs and also critical constraint programs [8]. CPTEST is based on ILOG CP Optimizer 2.1 from ILOG OPL 6.1.1 Development Studio. All our experiments were performed on a Quadcore IntelXeon 3.16Ghz machine with 16GB of RAM. CPTEST, its requirements and all the models we used to perform these experiments are available online[10].

CPTEST includes a complete OPL parser and a backend process that produces dedicated OPL programs as output. These OPL programs must be solved in order to find non-conformities. If a solution is found, then CPTEST stops and reports the non-conformity to the user. Whenever all these OPL programs are shown to be inconsistent, then a conformity certificate is issued. The tool is parameterized by several options, including the selected conformity relation, the instance of the problem (value of k), etc.

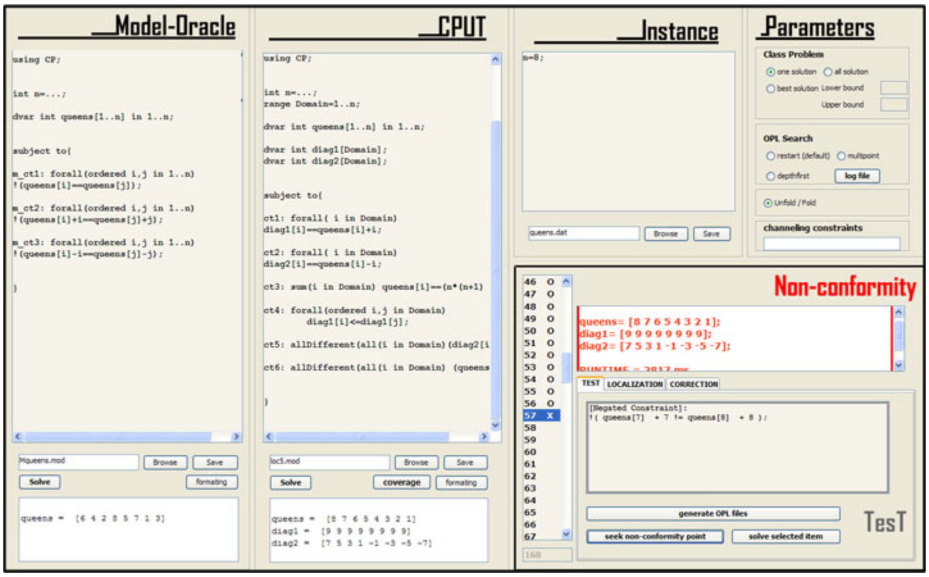
Figure 7 shows a snapshot of CPTEST, acting on the classical N-queens constraint program.

6.1 Constraint negation

CPTEST can negate most of the constraints that can be expressed in OPL. However, it cannot negate all the global constraints available, such as the `cumulative` or `circuit` global constraint. Table 2 summarizes the syntax of OPL constraints handled by CPTEST. OPL includes two aggregators, namely `forall` and `or`. The universal qualifier `forall` is used to declare a collection of closely related constraints and to build global constraints. Interestingly, the `or` aggregator can be used to negate `forall`, as `or` implements existential quantifier. The OPL `If-then-else` statement is less general than it may appear as its condition cannot contain decision variables. Its negation can be computed by negating the Then-part and Else-part without any loss of generality, as our goal is only to find non-conformities instead of computing the negation of a general model.

Our CPTEST tool handles several global constraints over discrete values, namely `allDifferent`, `allMinDistance`, `inverse`, `pack`, `forbiddenAssignments` and

[10]www.irisa.fr/celtique/lazaar/CPTEST



```
using CP;

int n=...;

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

subject to{

m_ct1: forall(ordered i,j in 1..n)
!(queens[i]==queens[j]);

m_ct2: forall(ordered i,j in 1..n)
!(queens[i]+i==queens[j]+j);

m_ct3: forall(ordered i,j in 1..n)
!(queens[i]-i==queens[j]-j);

}
```

Mqueens.mod

```
queens = [6 4 2 8 5 7 1 3]
```

```
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{

ct1: forall( i in Domain)
diag1[i]==queens[i]+i;

ct2: forall( i in Domain)
diag2[i]==queens[i]-i;

ct3: sum(i in Domain) queens[i]==(n*(n+1)

ct4: forall(ordered i,j in Domain)
        diag1[i]<=diag1[j];

ct5: allDifferent(all(i in Domain)(diag2[i

ct6: allDifferent(all(i in Domain) (queens

}
```

loc5.mod

```
queens = [8 7 6 5 4 3 2 1]
diag1 = [9 9 9 9 9 9 9 9]
diag2 = [7 5 3 1 -1 -3 -5 -7]
```

```
n=8;
```

queens.dat

```
46 O
47 O
48 O
49 O
50 O
51 O
52 O
53 O
54 O
55 O
56 O
57 X
58
59
60
61
62
63
64
65
66
67
168
```

```
queens= [8 7 6 5 4 3 2 1];
diag1= [9 9 9 9 9 9 9 9];
diag2= [7 5 3 1 -1 -3 -5 -7];

RUNTIME = 2817 ms
```

```
[Negated Constraint]:
!( queens[7]  + 7 != queens[8]  + 8 );
```

Fig. 7 CPTEST acting on n-queens problem

`allowedAssignments`. As seen before in Section 5.4, these constraints can be rewritten using an aggregation of constraints where computing their negation becomes trivial with the rules presented above and using the other global constraints.

– `allDifferent(all (i in R) x[i])` constraint is rewritten into:

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

  and its negation becomes trivial:

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

– `allMinDistance(all (i in R) x[i], val)` constraint takes two arguments, an array of variables `x` and an integer value `val`. The constraint implies that all the integer variables `x` differ, when taken one-to-one, by at least the value `val`. It can be rewritten into:

  `forall(ordered i,j in R)  abs(x[i]-x[j])>= val;`

  where its negation gives:

  `or(ordered i,j in R)  abs(x[i]-x[j])< val;`

Table 2 Syntax of OPL expressions handled by CPTEST

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*)

- `inverse(all[R](i in R) g[i], all[S](j in S) f[j])` constraint takes two arrays of integer variables and implies that for any value `i` of the firts array `g`, we have `g[f[i]] = i`. This constraint can be rewritten into:

  ```
  forall(i in S) g[f[i]]== i
  ```

  and its negation is:

  ```
  or(i in S) g[f[i]]!= i
  ```

- `pack(l, p, w)` constraint maintains the load of a set of bins, given a set of weighted items and an assignment of items to bins. In this example, consider that we have `n` items and `m` bins. Each item `i` has an integer weight `w[i]` and a constrained integer variable `p[i]` associated with it, indicating in which bin item `i` is to be placed. No item can be split up, and so an item can go in only one bin. Associated with each bin `j` is an integer variable `l[j]` representing the load in that bin; that is, the sum of the weights of the items that have been assigned to that bin. The constraint also ensures that the total sum of the loads of the bins is equal to the sum of the weights of the items being placed. This constraint can be rewritten into:

  ```
  forall(i in 1..m) sum(j in 1..n) ((p[j]==i)*(w[j]))==l[i]
  ```

  and its negation is:

  ```
  or(i in 1..m) sum(j in 1..n) ((p[j]==i)*(w[j]))!=l[i]
  ```

- `forbiddenAssignments` (resp. `allowedAssignments`) constraint defines the forbidden (resp. allowed) combinations of values given by a tuple with an arity equal to the number of considered variables. Each tuple defines a forbidden (resp. allowed) combination. Where, the negation of `forbiddenAssignments` constraint is simply the `allowedAssignments` constraint.

6.2 Constraint projection

We have seen in Section 5.4 the problem of auxiliary variables where an auxiliary variable, in a given CPUT, is out of the scope of the corresponding Model–Oracle during the negation and can lead us to false alarms. For that, we use constraint projection.

Practical solutions for the computation of constraint projection in CPTEST consist of annotating the CPUT with channeling constraints. Here, a channeling constraint C is a constraint relating the basic variables x of the Model–Oracle and the added auxiliary variables y (e.g., in Fig. 3 part(B), `cc2` is a channeling constraint). First, CPTEST asks the user which constraint is a channeling one. Otherwise, CPTEST splits the set of variables on basics and auxiliaries to check after which constraint is a channeling one.

6.3 Implemented optimizations

We implemented Algorithm 4 in CPTEST with several improvements. In particular, by noticing that it is unnecessary to search for non-conformities on constraints that are included in both the CPUT and the Model–Oracle, we implemented a simple rewriting system to check equality modulo Associativity-Commutativity ($\equiv_{AC}$). The system implements the following rules:

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

where $\circ \in \{+, *, \wedge, \vee\}$, $\times \in \{*, \wedge, \vee\}$ and $\bullet \in \{+, \wedge, \vee\}$. In Algorithm 4, the constraint C_i is discarded whenever there exists C'_i in D such as $C'_i \equiv_{AC} (C_i)$.

7 Experimental validation

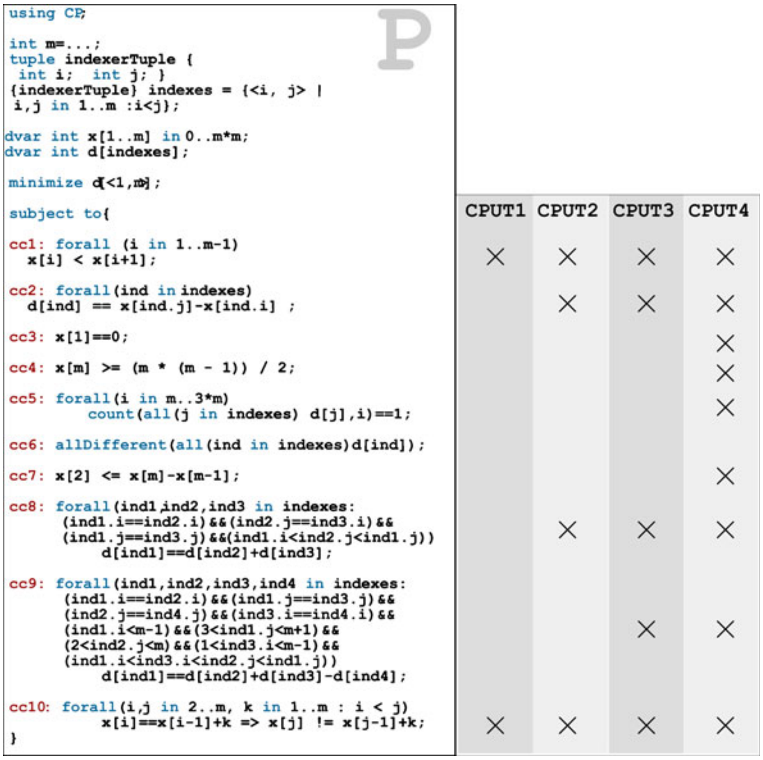
The goal of our experimental evaluation was to check that CPTEST is able to detect faults in OPL programs. We fed CPTEST with faulty programs using fault injection. Indeed, we developed optimized programs of well-known CP problems (Golomb rulers, n-queens, social golfer and the car sequencing problem) and we inject fault to have faulty versions of these programs.

7.1 The Golomb rulers problem

The four intermediate versions of the Golomb rulers we kept from our initial program development contain realistic faults. Figure 8 shows the four faulty versions expressed with the constraints of an optimized version of Golomb Rulers noted `P`. Note that constraints `cc3`, `cc4` and `cc7` break problem symmetries and remove solutions (valid Golomb rulers) w.r.t. the model-oracle.

For each CPUT, we studied its conformity w.r.t. the model-oracle (part `A` Fig. 1) using the four conformity relations. The results we got for an instance $m = 8$ are given in Table 3. For the $conf_{\text{bounds}}$ relation, the interval $[l, u] =][50, 100]$ was used to feed the relation, knowing that the global minimum is $x_m = 34$ when $m = 8$. Each time CPTEST returns a non-conformity that detect a given fault ϕ, these non-conformities were reported with the required CPU time in seconds. Firstly, the four faulty CPUT were reported as being non-conforming and the time required for finding these non-conformities is acceptable (less than a few minutes in the worst case). Secondly, this experiment shows that the most practical conformance relations (i.e., $conf_{\text{one}}$ and $conf_{\text{bounds}}$) are preferable to the other ones for efficiency reason. Indeed, for the first three CPUT, these relations gave results less than 10 *sec*. Note that non-conformities are represented either by invalid Golomb rulers that detect a ϕ^+ faults (e.g., $g1 = [0\ 7\ 8\ 18\ 24\ 26\ 35\ 44]$ and $44 - 35 = 35 - 26 = 9$ in the CPUT1/$conf_{\text{one}}$ case) or by valid Golomb rulers for ϕ^- faults (e.g., $g2$ for CPUT1/$conf_{\text{all}}$ case). These non-conformities correspond to cases where the CPUT misses solutions of the problem.

Interestingly, P is shown to be non-conforming with the $conf_{\text{all}}$ relation and the non-conformity that is found represent a valid Golomb ruler (i.e., ϕ^- is detected by $g13 = [0\ 7\ 9\ 12\ 37\ 54\ 58\ 64]$). In fact, recalling that P includes constraints that break the symmetries, this result was expected. Finally, note that conformity of P when $conf_{\text{best}}$ is selected was impossible to assess within the allocated time ($timeout = 5\,400 sec$). In fact, computing a global minimum of the Golomb ruler rapidly becomes hard even for small values of m (e.g., CPUT3/$conf_{\text{best}}$).



```
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 d[<1,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[1]==0;

cc4: x[m] >= (m * (m - 1)) / 2;

cc5: forall(i in m..3*m)
         count(all(j in indexes) d[j],i)==1;

cc6: allDifferent(all(ind in indexes)d[ind]);

cc7: x[2] <= x[m]-x[m-1];

cc8: forall(ind1,ind2,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];

cc9: 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];

cc10: forall(i,j in 2..m, k in 1..m : i < j)
          x[i]==x[i-1]+k => x[j] != x[j-1]+k;
}
```

	CPUT1	CPUT2	CPUT3	CPUT4
cc1	×	×	×	×
cc2		×	×	×
cc3				×
cc4				×
cc5				×
cc6				
cc7				×
cc8		×	×	×
cc9			×	×
cc10	×	×	×	×

Fig. 8 Faulty versions of the Golomb rulers

Our experimental evaluation also had the goal to check that computing non-conformities with CPTEST was less difficult than computing solutions. For that, Fig. 9 shows: A) the CPU time in seconds required to find a global optimum for instances of the Golomb rulers (square points) and B) the CPU time in seconds required to find non-conformities with CPTEST with the $conf_{\text{bounds}}$ conformity

Table 3 Non-conformities found by CPTEST in various CPUTs of the Golomb rulers problem (timeout = 5 400s)

—		$conf_{\text{one}}$	$conf_{\text{all}}$	$conf_{\text{bounds}}$	$conf_{\text{best}}$
Non-conformity		nc(ϕ^+,8,g1)	nc(ϕ^-,8,g2)	nc(ϕ^+,8,g3)	nc(ϕ^+,8,g4)
CPUT1	T(s)	4.29	21.45	5.64	7.31
Non-conformity		nc(ϕ^+,8,g6)	nc(ϕ^-,8,g2)	nc(ϕ^+,8,g7)	nc(ϕ^+,8,g8)
CPUT2	T(s)	5.62	40.78	4.64	174.43
Non-conformity		nc(ϕ^+,8,g6)	nc(ϕ^+,8,g6)	nc(ϕ^+,8,g7)	nc(ϕ^+,g8)
CPUT3	T(s)	9.53	45.78	7.15	389.04
Non-conformity		nc(ϕ^+,8,g9)	nc(ϕ^-,8,g10)	nc(ϕ^+,8,g11)	nc(ϕ^+,8,g12)
CPUT4	T(s)	12.60	0.15	9.01	12.53
Non-conformity		conf-certificate	nc(ϕ^-,8,g13)	conf-certificate	--
P	T(s)	3 448.46	0.18	3 658.13	timeout

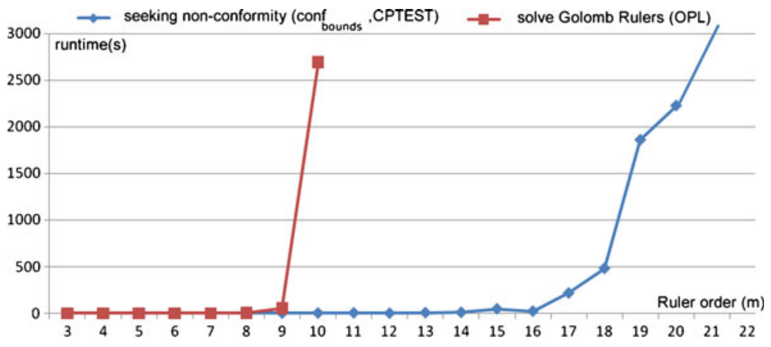


Fig. 9 Testing and solving time comparison on the Golomb rulers

relation (lozenge points). The search heuristic used in both cases is the default heuristic of OPL, i.e. depth-first search with restarts and branch-and-bound for the global optimization problem. CPTEST can find non-conformities when $m < 22$ in a reasonable amount of time because the hard global optimization problem has been relaxed in a simpler satisfaction problem, in order to deal with larger instances. This is the essence of the $conf_{\text{bounds}}$ conformity relation.

7.2 N-queens problem

The problem of placing n queens on an $n \times n$ chessboard is a classical problem where the model-oracle can be given by three constraints, the first one insures that all queens are placed in different columns, the second and the third constraint insure that no two queens are placed on the same upper or lower-diagonal on the chessboard. We improved this model by adding auxiliary variables and channeling constraints, redundant, surrogate and global constraints. We produced four CPUTs (from CPUT1 to CPUT4) by fault injection in the correct improved CP program [4]. We took two instances that cover all the constraints, a small instance with $N = 8$ and a big instance with $N = 100$.

We fed CPTEST with the four faulty CPUTs and the correct one noted `P`. Table 4 shows the results reported by CPTEST. Let us take CPUT1, CPUT2 and `P`.

CPTEST reports that CPUT1 is over-constrained for the two instances (i.e., 8 and 100) in less than one second, where the fault injected-in reduces the set of solution to empty (i.e., ϕ^* detected). CPUT2 is non-conforming to the Model–Oracle where CPTEST reports a non-conformity where one solution and/or all solutions are sought (i.e., ϕ^+ and ϕ^- detected).

For $conf_{\text{one}}$, CPTEST reports a non-conformity for $N = 8$ and $N = 100$ (resp. in less than a second and in $\simeq 8$ *min*) which is a solution of CPUT2 and not of the Model–Oracle (i.e., ϕ^+). If we analyze the non-conformities (i.e., `nc(`ϕ^+`,8,q1)` and `nc(`ϕ^+`,100,q2)` where `q1= [8 7 6 5 4 3 2 1]` and `q1=[100 99...2 1]`) we can see that all queens are on the second diagonal. Interestingly enough, the injected fault ϕ^+ is on the second diagonal constraint.

For $conf_{\text{all}}$, CPTEST reports a non-conformity for each instance that detects a ϕ^- fault (resp. in less than a second and in less than 1 *min*). `q3` and `q4` are good solutions that cannot be returned by CPUT2 because of the symmetry breaking constraint.

Table 4 Non-conformities found by CPTEST in various CPUTs of the n-queens problem (timeout = 5 400s)

		$Conf_{\text{one}}$		$Conf_{\text{all}}$	
	nc	nc(ϕ^*,8,-)	nc(ϕ^*,100,-)	nc(ϕ^*,8,-)	nc(ϕ^*,100,-)
CPUT1	T(s)	0.30	0.36	0.42	0.53
	nc	nc(ϕ^+,8,q1)	nc(ϕ^+,100,q2)	nc(ϕ^-,8,q3)	nc(ϕ^-,100,q4)
CPUT2	T(s)	0.23	446.77	0.25	44.48
	nc	nc(ϕ^+,8,q5)	nc(ϕ^+,100,q6)	nc(ϕ^-,8,q3)	nc(ϕ^-,100,q7)
CPUT3	T(s)	5.64	752.06	0.15	1.77
	nc	nc(ϕ^+,8,q8)	nc(ϕ^+,8,q9)	nc(ϕ^-,8,q10)	--
CPUT4	T(s)	5.17	52.98	0.20	timeout
	nc	conf	--	conf-certificate	--
P	T(s)	0.59	timeout	1.03	timeout

P is a correct version of an optimized n-queens program where no fault are injected-in. CPTEST reports a conformity certificate in a second for the small instance where $conf_{\text{one}}$ and/or $conf_{\text{all}}$ is experimented. For the large instance (i.e., $N = 100$) the timeout ($= 5\,400s$) is reached without providing any answer.

7.3 The social golfer problem

Social golfer is one of the hardest problem of the CSPLib (see prob010). We have m social golfers, n weeks and k groups of l size. Each golfer plays once a week in groups of l golfers. The problem asks to build a schedule of play for all golfers over the n weeks such that no golfer plays in the same group as any other golfer more than one time. The model-oracle can be built by two constraints, the first one to specify that each group has exactly l golfers (group size). The second one to say that each pair of golfers meets only at most once.

An improved program can be given by using SBSA (Symmetry-breaking by selective assignment) where on the first week, the first l golfers play in group 1, the second l golfers play in group 2, etc. On the second week, golfer 1 plays in group 1, golfer 2 plays in group 2, etc. At the end, the improved program contains five constraints; we inject-in faults to get four faulty CPUTs (CPUT1 to CPUTP4). We launch CPTEST on these faulty CPUTs to detect the injected fault. We take an instance of the problem with 9 golfers (3 groups of 3 golfers) playing during 4 weeks.

Let us look at CPUT1, CPUT4 and P from Table 5. The fault injected in the second constraint of CPUT1 leads CPUT1 to accept new solutions. In this case, CPUT1 can return bad solutions (i.e., ϕ^+ detected).

For $conf_{\text{one}}$, CPTEST detects ϕ^+ with a non-conformity s1 in less than 1 *sec*:

```
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 ]]
```

s1 is a bad schedule of social golfer where golfer 3 and golfer 4 play in the same group (group3) over 2 weeks (3rd and 4th week). For $conf_{\text{all}}$, CPTEST reports a good schedule for social golfer in less than 1 *sec*. This schedule cannot be reported by CPUT1 due to the symmetry breaking and ϕ^- is detected.

Table 5 Non-conformities found by CPTEST in various CPUTs of the golfer social problem (timeout = 5 400s)

—		$Conf_{\text{one}}$	$Conf_{\text{all}}$
Non-conformity		nc(ϕ^+,sg,s1)	nc(ϕ^-,sg,s2)
CPUT1	T(s)	0.23	0.62
Non-conformity		nc(ϕ^+,sg,s3)	nc(ϕ^-,sg,s2)
CPUT2	T(s)	0.37	1.31
Non-conformity		nc(ϕ^*,sg,-)	nc(ϕ^*,sg,-)
CPUT3	T(s)	0.21	0.34
Non-conformity		nc(ϕ^*,sg,-)	nc(ϕ^*,sg,-)
CPUT4	T(s)	0.11	0.27
Non-conformity		conf-certificate	nc(ϕ^-,sg,s2)
P	T(s)	0.26	0.79

`sg: weeks=4; groups=3; groupSize=3`

CPTEST reports that CPUT4 is over-constrained where the injected fault ϕ^* reduces its set of solution to empty. CPUT4 includes two symmetry breaking constraint: The first one, where the fault is injected, breaks symmetries on golfers and groups: *The first golfer plays in the first place in group 1, the second golfer in the second place and so on until we fulfil all groups.* The second constraint breaks symmetry on weeks and groups: *The first week, the first group size of golfers play in group 1, the second group size of golfers play in group 2, etc. On the second week, golfer 1 plays in group 1, golfer 2 plays in group 2, etc.* A fault injected in the first constraint creates an inconsistency between symmetries on groups in the two constraints. CPUT3 is, in this case, non-conforming to the Model–Oracle.

`P` is a correct version of social golfer improved program. It is conform to the Model–Oracle where CPTEST reports a conformity certificate for $conf_{\text{one}}$ in less than 1 *sec*. CPTEST reports a non-conformity where the $conf_{\text{all}}$ relation is experimented. The non-conformity here detect ϕ^- fault with a good schedule of social golfer that cannot be returned by `P`.

7.4 The car sequencing problem

The car sequencing problem (CSeq) illustrates interesting features of CP using wide parameter settings, redundant, surrogate, global constraints and specialized data structures definition (see prob001 in CSPLib). This is a constraint satisfaction problem that amounts to find an assignment of cars to the slots of a car-production company, which satisfies capacity constraints.

As a Model–Oracle of this problem, we took the model given in the OPL book [22]. In this model, capacity constraints are formalized by using constraints `r out of s`, saying that from each sub-sequence of `s` cars, a unit can produce at most `r` cars with a given option. Starting from this model, we built an optimized model by introducing several refinements: using a new auxiliary variable `setup[o,s]` which takes value 1 if option `o` is installed on slot s, redundant and global constraint (e.g., `pack` constraint). When building our improved model of car sequencing, we recorded four faulty constraint models that are used for experiments. Here again, the idea was to keep models that represent realistic faults instead of a posteriori injected faults. These four models are available online on the site mentioned above.

Table 6 gives the results of CPTEST on two instances of the problem: an assembly line of 10 cars, 6 classes and 5 options (i.e., `c1`); an assembly line with 55 cars, 7 classes and 5 options (i.e., `c2`). Using $conf_{\text{one}}$, CPTEST reports non-conformities for the three first CPUT in less than 1 *sec* for both instances. CPUT4 has no solution as the introduced fault ϕ^* on the `pack` constraint prunes dramatically the search space. This case is interesting as detecting this fault is really difficult. With the $conf_{\text{all}}$ relation, the results are balanced as three instances were not detected as non-conformant within the allocated time slot. For example, in CPUT2, the capacity constraint of the first option is violated (`1 out of 2`). This fault results from a bad formulation but it is quickly detected as ϕ^+ with $conf_{\text{one}}$. When $conf_{\text{all}}$ is selected, more constraints have to be negated and then our algorithm has to backtrack a lot, which explains the failure. The non-conformity reached in this case detects ϕ^- fault with a solution that satisfies the Model–Oracle and violates CPUT2, so it represents a correct assembly line that CPUT2 excludes from its solutions. Therefore, we can conclude that CPUT2 adds and removes solutions which make it difficult to detect as non-conform.

7.5 Threats to validity

External threats to validity lay on the source of constraint problems we used for our experiments. We have selected well-known and difficult problems from the Constraint Community, which allowed us to validate our approach on pertinent empirical data. However, these academic problems might not perfectly reflect industrial usage of Constraint Programming in critical applications. In addition, we manually injected faults in the constraint programs under test of our experiments. Although these faults were selected to show the capabilities of an automatic testing approach and fault detection where thus faults, we do not know whether they are realistic or not. Unlike other languages where tons of programs and bugs are available (e.g., Java, C and C++), programs written in constraint modeling languages are not available from repositories on the web. Finally, our overall approach of automatic test generation for constraint programs is based on the availability of a Model–Oracle for a given

Table 6 Non-conformities found by CPTEST in various CPUTs of the car sequencing problem (timeout = 5 400s)

		$Conf_{\text{one}}$		$Conf_{\text{all}}$	
Non-conformity		nc(ϕ^+,10,p1)	nc(ϕ^+,55,p2)	nc(ϕ^+,10,p1)	--
CPUT1	T(s)	0.30	1.23	2.49	timeout
Non-conformity		nc(ϕ^+,10,p3)	nc(ϕ^+,55,p4)	nc(ϕ^-,10,p5)	--
CPUT2	T(s)	0.85	1.65	1.20	timeout
Non-conformity		nc(ϕ^+,10,p6)	nc(ϕ^+,55,p7)	nc(ϕ^-,10,p5)	--
CPUT3	T(s)	0.24	0.70	90.73	timeout
Non-conformity		conf-certificate	conf-certificate	nc(ϕ^-,10,p8)	nc(ϕ^-,55,p9)
CPUT4	T(s)	0.96	1.06	1.26	100.22
Non-conformity		conf-certificate	--	nc(ϕ^-,10,p10)	--
P	T(s)	3.01	timeout	0.17	timeout

c1: NbCars=10; NbClasses=6; NbOptions=5; c2: NbCars=55; NbClasses=7; NbOptions=5 problem, i.e., an initial constraint model extracted from the problem specification. However, this hypothesis might be difficult to satisfy in all the cases, as initial models are not necessarily kept for analysis in industrial development processes.

8 Conclusion

In this paper, we introduced a testing framework that is adapted to a standard CP development process. The framework is built on solid testing-related notions such as conformity relations, oracle, fault, and it provides algorithms for finding non-conformities between a constraint program under test and a test oracle. We also introduced CPTEST, an implementation of our framework dedicated to the testing of OPL programs, and evaluated it on hard instances of well-known constraint problems such as Golomb ruler, n-queens, social golfer and car-sequencing. Our experimental evaluation shows that CPTEST can efficiently detect non-trivial faults in versions of these programs were manual faults were injected.

A desirable research perspective of our framework concerns its development towards more open CP plateforms. In particular, we would like to extend our framework to the open-source constraint solver GECODE. Looking at an open-source platform presents the advantage of allowing external users to exploit internal components to instrument the solver and tune the solver to their own needs. For example, measuring testing criteria related to constraint coverage, similarly to what can be done for conventional programs, requires instrumenting the solver source code, something impossible in black-box platforms such as OPL. Another perspective of our framework concerns the optimization of constraint negation computation. For global constraints, our framework is currently restricted to a syntactical transformation of global constraints while we envision several better and more generic ways to handle negation. Note also that as the design of global constraints is a work-intensive and error-prone activity, building procedures to test and find non-conformities within the design of global constraints is a very interesting research perspective.

Acknowledgements We are very grateful to Olivier Lhomme who pointed us the problem of auxiliary variables. Many thanks also to Michel Rueher, Claude Michel, Laurent Granvilliers and Nicolas Beldiceanu for helpful comments on early presentations of this work.

References

1. Beldiceanu, N., & Simonis, H. (2011). A constraint seeker: Finding and ranking global constraints from examples. In *Proceedings of the 17th international conference on Principles and practice of constraint programming—CP11* (pp. 12–26). Perugia, Italy.
2. Cohen, D. A., Jeavons, P., Jefferson, C., Petrie, K. E., & Smith, B. M. (2006). Constraint symmetry and solution symmetry. In *Proceedings of the 21st national conference on Artificial intelligence—AAAI06*. Boston, Massachusetts: AAAI Press.
3. Cheng, B. M. W., Lee, J. H. M., & Wu, J. C. K. (1996). Speeding up constraint propagation by redundant modeling. In *In 2nd International Conference on Principles and Practice of Constraint Programming—CP96* (pp. 91–103). Springer Verlag.
4. Clark, J. A., & Pradhan, D. K. (1995). Fault injection: A method for validating computer-system dependability. *IEEE Computer, 28*(6), 47–56.

5. Collavizza, H., Rueher, M., & Van Hentenryck, P. (2008). CPBPV: A constraint-programming framework for bounded program verification. In *Proceedings of the 14th international conference on Principles and practice of constraint programming—CP08* (pp. 327–341). Sydney, Australia.
6. Deransart, P., Hermenegildo, M. V., & Maluszynski, J., (Eds.) (2000). *Analysis and Visualization Tools for Constraint Programming, Constrain Debugging (DiSCiPl project)*, vol. 1870 of *LNCS*. Springer.
7. DeMillo, R. A., Lipton, R. J., & Sayward, F. G. (1978). Hints on test data generation: Help for the practicing programmer. *IEEE Computer Magazine, 11*(4), 34–41.
8. Flener, P., Pearson, J., Agren, M., Garcia-Avello, C., Celiktin, M., & Dissing, S. (2007). Air-traffic complexity resolution in multi-sector planning. *Journal of Air Transport Management, 13*(6), 323–328.
9. Fordan, A., & Yap, R. H. C. (1998). Early projection in CLP($\mathcal{R}$). In M. J. Maher, & J.-F. Puget (Eds.), *Principles and Practice of Constraint Programming—CP98, 4th International Conference* (pp. 177–191). Pisa, Italy.
10. Gotlieb, A. (2009). Tcas software verification using constraint programming. *The Knowledge Engineering Review*. Accepted in 2009, to be published in 2012.
11. Holland, A., & O'Sullivan, B. (2005). Robust solutions for combinatorial auctions. In *Proceedings of the 6th ACM Conference on Electronic Commerce—ACM-EC05* (pp. 183–192). Vancouver, British Columbia, Canada.
12. Imbert, J.-L. (1993). Fourier's elimination: Which to choose? In *Principles and Practice of Constraint Programming, first International Workshop—PPCP93* (pp. 117–129). Newport, Rhode Island.
13. Junker, U., & Vidal, D. (2008). Air traffic flow management with ilog cp optimizer. In *International Workshop on Constraint Programming for Air Traffic Control and Management*. Bretigny sur Orge, Paris, France.
14. Langevine, L., Deransart, P., Ducassé, M., & Jahier, E. (2001). Prototyping CLP(FD) tracers: A trace model and an experimental validation environment. In T. Kusalik (Ed.), *Proceedings of the 11th Workshop on Logic Programming Environments*. Computer Research Repository, CS.PL/0111043.
15. Lozano, R. C., Schulte, C., & Wahlberg, L. (2010). Testing continuous double auctions with a constraint-based oracle. In D. Cohen (Ed.), *16th International Conference on Principles and Practice of Constraint Programming—CP10*, vol. 6308 of *LNCS* (pp. 613–627). St Andrews, UK: Springer-Verlag.
16. Puget, J.-F. (2006). An efficient way of breaking value symmetries. In *Proceedings of the 21st national conference on Artificial intelligence—AAAI06* (pp. 117–122). Boston, Massachusetts: AAAI Press.
17. Rankin, W. T. (1993). Optimal golomb rulers: An exhaustive parallel search implementation. Master's thesis, Duke University, Durham.
18. Rossi, F., van Beek, P., & Walsh, T. (2006). *Handbook of Constraint Programming (Foundations of Artificial Intelligence). Chapter 10: Symmetry in Constraint Programming*. New York, NY, USA: Elsevier Science Inc.
19. Régin, J.-C. (2011). *Hybrid Optimization. Book Chapter: Global Constraints: A survey*. New York, NY, USA: Springer.
20. Sahinidis, N. V., & Twarmalani, M. (2002). *Convexification and Global Optimization in Continuous and Mixed-Integer Nonlinear Programming*. Kluwer Academic Publishers Group.
21. Van Hentenryck, P. (1989). *Constraint satisfaction in logic programming*. Cambridge, MA, USA: MIT Press.
22. Van Hentenryck, P. (1999). *The OPL optimization programming language*. MIT Press.
23. Weyuker, E. J. (1982). On testing non-testable programs. *Computer Journal, 25*(4), 465–470.
24. Zhu, H., Hall, P. A. V., & May, J. H. R. (1997). Software unit test coverage and adequacy. *ACM Computing Surveys, 29*(4), 366–427.