# A Framework for the Automatic Correction of Constraint Programs

Nadjib Lazaar, Arnaud Gotlieb
INRIA Rennes Bretagne Atlantique
Campus Beaulieu
35042 Rennes, France
{nadjib.lazaar, arnaud.gotlieb}@inria.fr

Yahia Lebbah
University of Oran Es-Senia
Lab. LITIO,B.P. 1524 EL-M'Naouar
31000 Oran, Algeria
ylebbah@gmail.com

***Abstract*—Constraint programs, such as those written in high-level constraint modelling languages, e.g., OPL (Optimization Programming Language), COMET, ZINC or ESSENCE, are more and more used in business-critical programs. As any other critical programs, they require to be thoroughly tested and corrected to prevent catastrophic loss of money. This paper presents a framework for the automatic correction of constraint programs that takes into account the specificity of the software development process of these programs as well as their typical faults. We implemented this framework in our testing platform CPTEST for OPL programs. Using mutation testing, our experimental results show that well-known constraint programs written in OPL can be automatically corrected using our framework.**

## I. Introduction

Constraint Programming emerged since 1960's for solving difficult combinatorial problems [RBW06] and evolved through the development of high-level modelling languages such as OPL (Optimization Programming Language) [VH99], COMET [VHM05], ZINC [MNR+08], or ESSENCE [FGJ+07]. In these languages, statements are replaced by constraints and any constraint program execution yields solutions to a constraint system instead of returning values. All these languages adopt an imperative-like syntax, but differ from traditional languages (e.g., C, C++ and Java) by their semantics which usually relies on fixpoint computations. A few years ago, constraint programs started to be used in critical programs. In particular, constraint programs were developed for e-commerce in business-critical program to solve combinatorial auctions [HO05]. Other critical software sectors also started to be concerned such as air-traffic control and management [FPA+07], [JV08] and software verification and certification [CRVH08], [Got09].

As any other critical programs, constraint programs must be thoroughly tested and corrected before being used on real-size instances of problems. Typical faults in critical constraint programs include bad formulation or refinement of a specific constraint and addition of erroneous constraints. Unfortunately, these faults cannot easily be handled by existing software testing approaches because of the following reasons: firstly, constraint programs are intrinsically non-deterministic as they represent unordered sets of solutions and traditional testing approaches usually consider only deterministic programs ; secondly, the refinement process of constraint programs development is specific to Constraint Programming. Indeed, constraint program developers usually start with an initial declarative model of the problem, which faithfully translates the problem specification, without granting interest to its performances. As this model cannot handle large-sized instances of the problem, they exploit several refinement techniques to build an improved model. For example, usual refinement techniques include the use of dedicated data structures, constraint reformulation, *global constraints* addition, redundant and surrogate constraint addition, as well as constraints which break symmetries (these constraints usually improve considerably the effectiveness of the solving process). The refinement process, carried out by the developers, is an error-prone and most of the faults are introduced during this step. Manual correction of faults in constraint programs is costly as it requires both the knowledge of the program being developed but also knowledge of how constraints are handled by the underlying constraint solver. In fact, the user often resorts exploring manually one by one each of the constraints of its constraint model, which can be cumbersome even for small programs. In this context, finding ways to correct automatically constraint programs becomes therefore essential.

Automatic correction (also called bug-fixing or program repair) of programs is an emerging trend in software testing, as witnessed by several recent and persuading contributions. Given a buggy program, a failing test run and several passing test runs, the goal is to find automatically a fix for the program [JFGG09], [WFLGN10] that pass the failing test run. Two main approaches can be distinguished: approaches using mutation testing operators to suggest automatically fixes to faulty programs [DW10], [WFLGN10] and approaches using formal specifications to repair programs. In particular, Wei et al.[WPF+10] proposed to exploit program's contracts (pre/post conditions) to find a fix. Unfortunately, these approaches cannot be directly applied to constraint programs for the reasons mentioned above. Moreover, automatic correction of constraint programs should take into account the peculiarities of the software development process of these programs.

In [LGL10b] and [LGL10a], we introduced CPTEST, a Testing framework for constraint programs written in OPL.

In this framework, given a combinatorial problem to solve, a first highly declarative constraint model (noted M for *Model-Oracle*) is used as a reference to detect non-conformities within a refined and optimized constraint program solving the same problem (noted CPUT for *Constraint Program Under Test*). Unlike the declarative Model-Oracle, the CPUT is intended to solve hard instances of the problem. The non-conformities between both M and CPUT result from faults introduced during the refinement process and can be found by using M as a testing oracle. In [LGL10a], we extended this approach by localizing a subset of constraints in the CPUT that may contain the fault. By solving auxiliary constraint problems, CPTEST can identify constraints that cannot be part of a selected non-conformity and can thus be discarded from the set of potential faulty constraints. In this paper, we propose to find automatically fixes for incorrect constraint programs. As constraint programs implicitly represents sets of solutions and a Model-Oracle is available, adding or withdrawing constraints within the CPUT allows us to find automatically fixes for potential faulty constraints. We implemented this approach on top of CPTEST and got experimental results on a set of four classical OPL constraint programs, namely the Golomb rulers, N-queens, Social Golfer and Car sequencing. These results show that OPL constraint programs can be automatically corrected using our framework CPTEST.

The rest of the paper is organized as follows. The next section presents our automatic correction process on an illustrative example, namely the Golomb Rulers. Section 3 and 4 recall how faults in constraint programs can be detected and localized by solving auxiliary constraint problems. Section 5 contains the algorithm we propose to correct automatically constraint programs while Section 6 presents our experimental results. Finally, Section 7 concludes the paper.

## II. An Illustrative Example

In this section we illustrate automatic correction of a classical constraint program example: the Golomb rulers problem. Solving the Golomb rulers problem has various applications in Radio communications and X-Ray crystallography. Although it has a simple formulation, solving hard instances of the problem is considered as a very challenging problem [Ran93]. A Golomb ruler is a set of $m$ marks $0 = x_1 < x_2 < ... < x_m$ such as $m(m-1)/2$ 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 is to find a ruler of order $m$ with minimal length ($minimize\ x_m$).

A first declarative model of this problem is given in Fig.1 which is a simple constraint model available from the problem specification analysis. In our testing framework, we called this model the $Model - Oracle$ (M). Then, this model is refined using techniques such as constraint reformulation, surrogate and global constraint addition, or symmetry-breaking to build an improved constraint model called the *Constraint Program Under Test* (CPUT) that must be thoroughly tested and corrected before using it on hard instances. Figure 2 shows a CPUT of Golomb rulers written in OPL. One can easily get

```
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]);
}
```

Fig. 1. A Model-Oracle for Golomb rulers in OPL.

```
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)        Faulty constraint
  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 in indexes, ind2 in indexes,ind3 in indexes:
       (ind1.i==ind2.i)&&(ind2.j==ind3.i)&&(ind1.j==ind3.j)&&
       (ind1.i<ind2.j < ind1.j))
          d[ind1]==d[ind2]+d[ind3];

cc8: forall(ind1,ind2,ind3,ind4 in indexes: (ind1.i==ind2.i)&&
       (ind1.j==ind3.j)&&(ind2.j==ind4.j)&&(ind3.i==ind4.i)&&
       (ind1.i<m-1)&&(3<ind1.j<m+1)&&(2<ind2.j<m)&&
       (1<ind3.i<m-1)&&(ind1.i < ind3.i < ind2.j < ind1.j))
          d[ind1]==d[ind2]+d[ind3]-d[ind4];

cc9: forall(i in 2..m, j in 2..m, k in 1..m : i < j)
          x[i]==x[i-1]+k => x[j] != x[j-1]+k;
}
```

Fig. 2. A CPUT for Golomb rulers in OPL.

confidence in the fact that the M of Figure 1 actually solves the Golomb rulers, but this is even more difficult for the CPUT of Figure 2.

When $m = 6$, our CPTEST framework reports that the CPUT of Figure 2 is unsatisfiable which witnesses the existence of a fault. Then, CPTEST automatically localizes suspicious constraints and reports `cc1` as containing a fault. By using the algorithms presented in this paper, CPTEST additionally proposes to remove `cc1` and to add three other constraints shown in Tab.I. Note that the (unknown) fault in the CPUT resulted from a small modification in `cc1` where the operator $<$ was incorrectly changed to $>$.

## III. Fault Detection

### A. Notations and Hypothesis

In this section, we introduce the notations required to understand the rest of the paper and the main hypothesis on which our approach is based.

TABLE I
AN AUTOMATIC $CPUT$ CORRECTION

| Removed Constraints | Added Constraints |
|---|---|
| cc1: forall (i in 1..m-1)<br>x[i] > x[i+1]; | x[4] < x[5];<br>x[3] < x[4];<br>x[2] < x[3]; |

As mentioned previously, we consider the initial declarative constraint model to be a testing oracle, called the `Model-Oracle` noted $M$. $M$ represents all the solutions of the problem and strictly conforms the problem specification. We suppose that $M$ is satisfiable and then possesses at least one solution. Considering unsatisfiable Model-Oracles may be interesting for some applications (such as software verification [Got09]) but we excluded these cases in order to avoid having to consider equivalence of unsatisfiable models. The *Constraint Program Under Test* is a constraint program noted $P$ (possibly unsatisfiable) which is expected to conform the solutions of $M$. $sol(M)$ (respectively $sol(P)$) denotes the set of solutions of $M$ (respectively P).The general conformity relation between $M$ and $P$ is given by the following definition:

*Definition 1 (conf):*

$$P \ conf \ M \ \Leftrightarrow sol(P) \neq \emptyset \ \wedge \ sol(P) \subseteq sol(M)$$

### B. Non-conformity Detection

As proving conformity on all the instances of a problem is undecidable in the general case, we proposed in [LGL10b] a testing process aiming at detecting non-conformities. For a given instance, a non-conformity is either a solution of $P$ that is not a solution of the model-oracle $M$, or $P$ is unsatisfiable (i.e., $sol(P) = \emptyset$). Systematic non-conformities detection can be sought by combining the negation of a constraint of $M$ with the constraints of $P$ (i.e., $P \wedge \neg C_i$ where $C_i \in M$). Badly formulating a constraint in $P$ can remove and/or add solutions to $M$. In [LGL10b], we proposed an algorithm called $one\_negated$ shown below, for systematically searching non-conformities in $P$. Note that, in our framework, a non-conformity is materialized by a solution of $P$ that is not a solution of $M$.

**Algorithm 1:** one_negated($P$, $M$)

```
nc ← ∅
foreach C_i ∈ M do
    nc ← solve(P ∧ ¬C_i)
    if nc then
        return ¬conf(nc)
return conf
```

In Algorithm 1, $solve()$ denotes a call to a constraint solver that provides only a single solution.

## IV. FAULT LOCALIZATION

In [LGL10a], we proposed another algorithm to localize faulty constraints, under the hypothesis that only a single constraint is faulty in $P$. In this section, we extend this approach by proposing an algorithm that cans localize several faults in $P$. By fixing a parameter $k$, the algorithm called $k\_locate$ and given below, is able to localize at most $k$ faulty constraints in $P$. The fault localization process is based on the following definition and property. Given a constraint program $P = \{C_1, C_2...C_n\}$ that does not conform to its model-oracle $M$, we introduce the notion of $suspicious\ set$:

*Definition 2 (Suspicious set):*
$T_i$ is suspicious in $P$ w.r.t. $M \equiv M \wedge P \backslash T_i$ is satisfiable.

Roughly speaking, a suspicious set in $P$ is a subset of $P$ that gives an explanation to the fault in $P$. Algorithm 2

**Algorithm 2:** k_locate(M, P, k)

```
1 set ← ∅
2 T ← powset(P, k)
3 for i ∈ 1..k do
4     foreach T_j ∈ T : |T_j| = i do
5         if sol(M ∧ P\{T_j}) ≠ ∅ then
6             set ← set ∪ {T_j}
7             foreach T_l ∈ T : T_j ⊂ T_l do
8                 T ← T/{T_l}
9 return set
```

takes as inputs the Model-Oracle $M$, the CPUT $P$ and $k$ the parameter that specifies the maximum number of considered faulty constraints. $powset(P, k)$ is the set of all subsets of P that have a cardinality less or equal to $k$ and not including the empty set. Our algorithm is complete as any complex fault involving $i$ constraints where $i \leq k$ will be reported by Algorithm 2. It is worth noticing that $k\_locate$ reports only sets of minimal size (line 7 and 8). Formally speaking,

$$T_i \in k\_locate(P, M, k) \Rightarrow$$

$$\begin{cases} \forall T_j \in powset(P \backslash T_i, k - card(T_i)) : \\ T_i \cup T_j \notin k\_locate(P, M, k) \end{cases}$$

The idea behind Algorithm 2 is to localize faulty constraints by iterating on all the possible subsets of $P$, with cardinality less than $k$, and by keeping only the subsets $T_i$ for which $sol(M \wedge P \backslash \{T_i\})$ has solutions. Thus, Algorithm 2 is also correct as it provides only subsets of $P$ that are suspicious. Note however that each proposed subset necessarily contains the faulty constraints, but may also includes non-faulty constraints. When $M \wedge P \backslash \{T_i\}$ fails, it means that $T_i$ does not contain all the faulty constraints.

The size of set $T$ computed by $powset(P, k)$ is $\sum_{i=1..k}(C_m^i) = C_{m+1}^{k+1}$, which is exponential in $k$ and then the algorithm may become intractable when $k$ increases. However, we believe that usually, only a few constraints are involved within a faulty CPUT, keeping the value of $k$ as small as possible. The proposed $k\_locate$ algorithm can be enhanced by using dichotomic schemes among the faulty constraints, such

as the QuickExplain algorithm of [Jun04], but we considered this possibility to be outside the scope of the paper and will not go into further details on it.

## V. Automatic Fault Correction in Constraint Programs

Once a fault is localized in $P$, our approach tries to reformulate the constraints of the subset found to be responsible of the fault. In order to restore the conformity with the model-oracle. We propose an automatic correction based on the computation of a set of constraints in $M$ that should be incorporated to $P$ to correct it.

Figure 3 shows a possible correction of a given $P$ w.r.t. its model-oracle $M$, where $P \equiv \bigwedge_{i \in 1..3} T_i$ and $T_i$ is a subset of constraints. The detection step returns a non-conformity point saying that $P$ does not conform to $M$ and contains faults (part(a) Figure 3). The localization step reports that $T_3$ is a suspicious set (part(b) Figure 3) where $sol(M \wedge P \backslash T_3) \neq \emptyset$). The automatic correction process first removes the suspicious set (part(c) Figure 3) and then proposes to add $C'$, where $C' \in M$, to restore the conformity of $P$ w.r.t $M$ (part(d) Figure 3).

Algorithm $correction$ aims at finding the set of constraints to revise non-conforming constraints within $P$. It returns a set of pairs where each pair is composed of two sets: a set of suspicious constraints $R$ and a set $A$ of constraints from the model-oracle. For each pair $(R, A)$, removing or revising $R$ and adding or reformulating $A$ enable to correct automatically $P$. The algorithm begins by finding the set of suspicious constraints, under the hypothesis that there are at most $k$ faulty constraints. For each suspicious set $T_i$, the $calculate$ function computes the correcting constraints $C_i$ of the model-oracle $M$ for which $(P \backslash T_i) \wedge \neg C_i$ does not fail. More precisely, any constraint $C_i$ where $sol((P \backslash T_i) \wedge \neg C_i) \neq \emptyset$ should be in the correction set, as there exist solutions of $(P \backslash T_i)$ which do not conform to $C_i$. Thus, adding $C_i$ to the program will withdraw these points.

**Algorithm 3:** correction(M, P, k)

```
set ← k_locate(M, P, k)
if card(P) = card(⋃_{T_i ∈ set} T_i) then
    return (∅, calculate(M, P))
else
    corr ← {(∅, ∅)}
    foreach T_i ∈ set do
        corr ← corr ∪ {(T_i, calculate(M, P\T_i))}
    return corr

calculate(M, P):
set ← ∅
foreach C_i ∈ M do
    if sol(P ∧ ¬C_i) ≠ ∅ then
        set ← set ∪ {C_i}
return set
```

*Proposition 1:* Provided that there are at most $k$ faulty constraints, we have the following property: the results of $calculate(M, P \backslash T_i)$ is conforming the Model-Oracle $M$, according to definition 1.

*Proof.*

Any constraint $C'$ computed by $calculate(M, P \backslash T_i)$ satisfies

$$\forall C_i \in M, sol(((P \backslash T_i) \wedge C') \wedge \neg C_i) = \emptyset$$

which means that

$$sol((P \backslash T_i) \wedge C') \subseteq sol(M)$$

Therefore, $(P \backslash T_i) \wedge C'$ is conforming $M$.

## VI. Experimental Validation

All our experiments were performed on Intel Core2 Duo CPU 2.40Ghz machine with 2.00 GB of RAM and all the programs and results are available online at (www.irisa.fr/celtique/lazaar/CPTEST).

### A. CPTEST

In this section, we give a brief overview of CPTEST, the testing framework we built for testing OPL constraint programs. It includes a complete OPL parser and a backend process that produces dedicated OPL programs as output that must be solved to detect, localize and correct faults. CPTEST includes implementations of Algorithms 1,2 and 3 given above. The underlying constraint solver of CPTEST is based on ILOG CP Optimizer 2.1. Figure 4 shows a snapshot of CPTEST, acting on a classical constraint program that solves the well-known N-queens problem.

### B. Faults Injection

The purpose of our experiments was to check that CPTEST can automatically detect, localize and correct faults in well-known constraint programs, namely Golomb rulers, N-queens, social golfer and car sequencing problems. For that, we manually injected significant faults in these CP programs in order to generate faulty OPL programs, called mutants. We fed CPTEST with the resulting faulty constraint programs and let CPTEST to correct automatically them.

Table III shows the main results on the four problems. It contains four columns, named mutation, detection, localization and correction. The columns related to mutation show the various mutants while the columns named detection give the non-conformity points that reveal the existence of faults in the mutants and the time consumption. The columns localization give the suspicious constraints and the columns correction give two sets of constraints: a set of constraints to be removed and a set of constraints to be added in order to correct $P$.

### C. CP Problems

*Golomb Rulers:* As presented in Sec.II, Golomb rulers aims at finding a ruler where the distance between any two marks is different than any other two marks. It is also an optimization problem where one wants to find an optimal ruler of minimal length. For our experiments, we selected $m = 6$

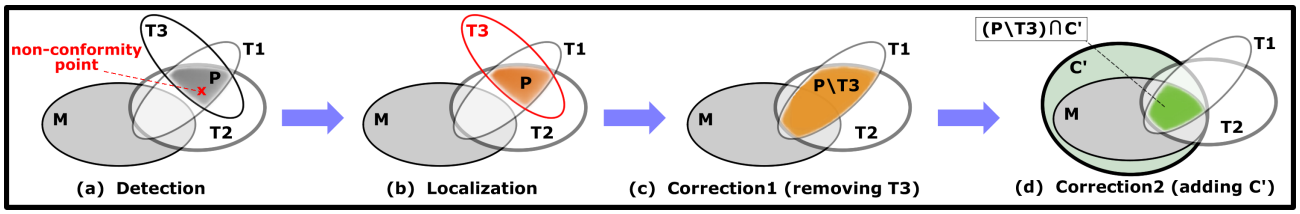


Fig. 3. A possible $P$ correction.

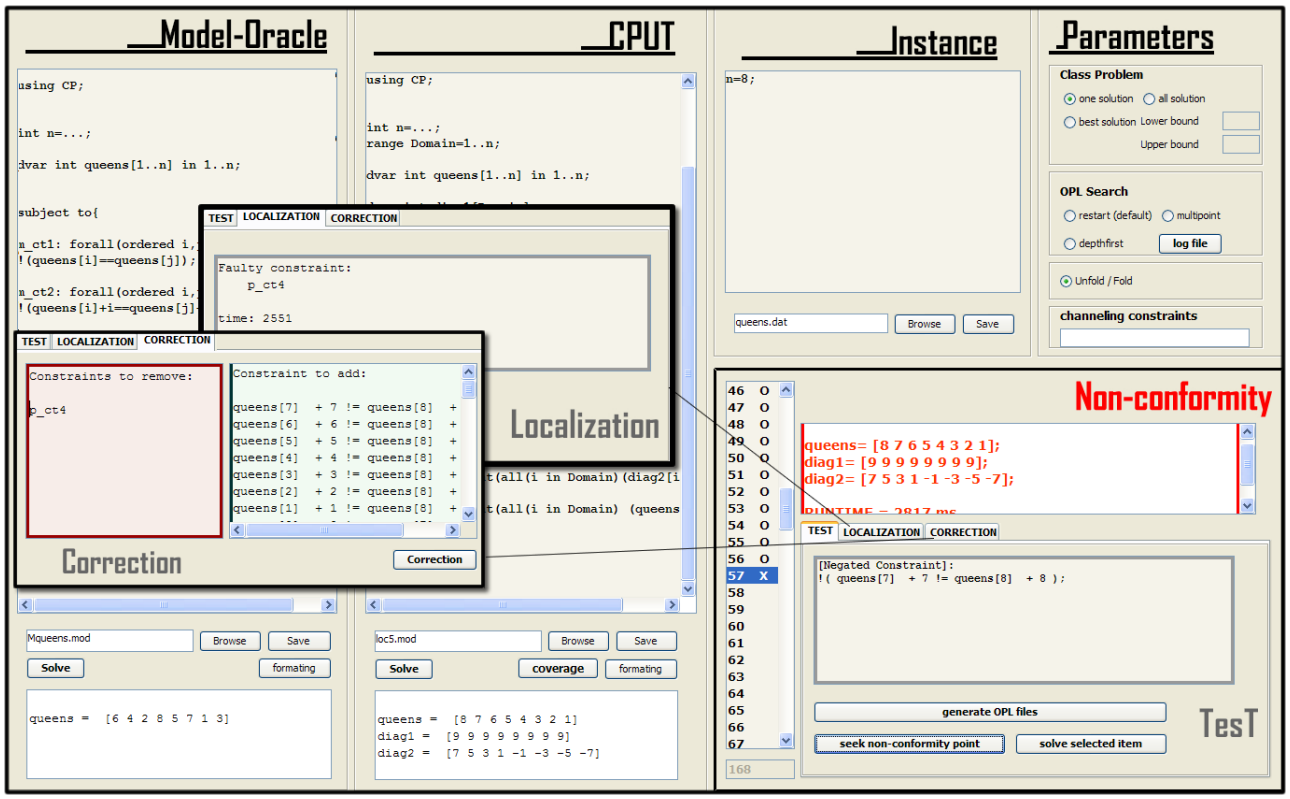


Fig. 4. CPTEST acting on n-queens problem.

as an instance that covers all constraints of the programs (i.e., all the constraints of $P$ are used within the solving process). We built 7 mutants from an improved Golomb rulers CP program by systematically injecting faults. For example, the fault injected in `Mut7` consists in replacing the *allDifferent* global constraint as follows:

```
allDifferent(all(ind in indexes ) d[ind]);
```

by

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

Let us look in more details at `Mut2` and `Mut5` in Table III. CPTEST returns a non-conformity for `Mut2` (i.e., `x=[0 1 3 10 13 20]` where `x` is not a Golomb ruler as `3-0=13-10`). This non-conformity is a solution of the `Mut2` and not of the model-oracle. In the localization step, CPTEST reports an empty set saying that `Mut2` contains a relaxed constraint and does not handle a large part of the problem specification. Finally, the correction step gives the set of constraints to be added to `Mut2` for correction: it returns 49 elementary constraints to be added to `Mut2`.

`Mut5` is an over-constrained program where the fault injected in `ct1` reduces it to fail (i.e., $sol(Mut5) = \emptyset$), so it does not conform to the model-oracle). The localization step returns `ct1` as a suspicious constraint in less than $4sec$. The correction step proposes to remove `ct1` and replace it by 3 elementary constraints ( `x[4] < x[5] ; x[3] < x[4] ; x[2] < x[3] ;`) in $\backsimeq 30min$. This CPU time may appear as being enormous for a simple constraint program, but recall that a large search space must be explored and that even simple specification problem may lead to constraint problems that require hours to be solved.

*N-queens:* The N-queens problem requires to place $N$ queens on an $N \times N$ chessboard. Its model-oracle can be

given with only three constraints, the first one ensures that all queens are placed on distinct column, while the second and the third constraint ensure that two queens cannot be placed on the same upper or lower-diagonal on the chessboard. We have an improved N-queens program with new data structures, redundant, surrogate and global constraints. We produced 7 faulty mutants (from `Mut1` to `Mut7`) by injecting faults in the correct improved CP program. We took $N = 8$ as an instance that covers all the constraints. Let us look at `Mut3`, `Mut5` and `Mut6` from tab.III.

`Mut3` does not conform the model-oracle as the fault injected in `ct11` makes $P$ unsatisfiable (i.e., $sol(Mut3) = \emptyset$). The localization step returns `ct11`, as the only suspicious constraint. CPTEST reports that `ct11` must be removed to conform the model-oracle.

`Mut5` is also non-conform to the model-oracle where CPTEST returns a non-conformity `q1=[8 7 6 5 4 3 2 1]`. `q1` is not an 8-queens solution where all queens are placed on the second diagonal. This non-conformity is due to the fault injected in `ct4`. CPTEST reports that `ct4` is suspicious constraint and suggests, in correction step, to replace it by 28 elementary constraints.

CPTEST returns a non-conformity point for `Mut6` which is a solution of the mutant and not of the model-oracle (i.e., `q2=[7 2 3 6 8 1 5 4]`). Indeed, `q2` is not an 8-queens solution as `queen[2]` and `queen[3]` are on the same diagonal. This non-conformity point reveals a fault in `Mut6`, which was injected in `ct3`. The localization step reports that `Mut6` needs additional constraints where the fault on `ct3` is a relaxation. For correction, we add 56 elementary constraints[1] in order to restore the conformity between `Mut6` and the model-oracle.

*Social Golfer:* Social golfer is one of the hardest problem of the CSPLib[HMGW] (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.

We take an instance of the problem with 4 weeks, 3 groups of 3 golfers. We injected faults in an improved social golfer program to build 5 mutants (from `Mut1` to `Mut5`). Let us look at `Mut2` and `Mut4` from Table III. CPTEST returns a non-conformity for `Mut2` as a bad schedule (i.e., `sg1`) due to the fault in `ct2` constraint. CPTEST reports an empty set saying that `Mut2` needs some constraints to be added. After that, the correction step of CPTEST returns 75 elementary constraints to add to `Mut2`.

The fault in `ct4` of `Mut4` reduces the set of solutions to empty. So `Mut4` does not conform to the model-oracle. CPTEST returns two suspicious constraints (`ct4, ct5`) in less than $4sec$. In the correction step, CPTEST gives two possible corrections, the first one being to replace `ct4` by 58 elementary constraints ; the second one being to replace `ct5` by another set of 58 elementary constraints.

[1]All results and the set of constraints can be consulted online at url www.irisa.fr/celtique/lazaar/CPTEST

TABLE II
NUMBER OF SOLUTIONS OF CAR SEQUENCING MUTANTS BEFORE AND AFTER CORRECTION.

| **Mutant** | `Mut1` | `Mut2` | `Mut3` | `Mut4` | `Mut5,Mut6,Mut7` |
|---|---|---|---|---|---|
| before | 26 466 | 43 259 | 35 754 | 42 271 | 0 |
| after | 6 | 6 | 6 | 6 | 6 |

*Car Sequencing:* Car sequencing is a real-world CP problem that amounts to find an assignment of cars to the slots of a car-production company, where cars are grouped by classes. Each class represents cars with some specific options. The assembly line must satisfy some option capacity constraints. The model-oracle of car sequencing is taken from the OPL book [VH99]. We have an improved car sequencing program that includes interesting features of CP with wide parameter settings, redundant, surrogate and global constraints addition, and specialized data structures definition. We inject fault in this CP program to get 7 mutants.

Let us take `Mut2`, `Mut4` and `Mut7` of car sequencing from Table III. `Mut4` does not conform the M as the fault injected in `ct2` reduced it to fail. CPTEST returns the faulty constraint `ct2` which can be advantageously corrected by using 37 elementary constraints returned by the correction step.

The fault injected in `Mut7` is localized in `ct5` ($sol(Mut7) = \emptyset$). (`ct5, ct6` ) are returned as suspicious constraints by the localization step. To correct the programs, CPTEST suggests two possible corrections, either remove `ct5` or add `ct6` .

A non-conformity is reported for `Mut2` (i.e., `[4 1 6 4 3 5 3 6 2 5]`) which is an incorrect assembly line. `Mut2` does not conform the model-oracle and no suspicious constraints are reported in localization step as `Mut2` needs constraint addition. To restore the conformity status of `Mut2`, CPTEST returns 32 elementary constraints to add.

$card(sol(Mut2)) = 43\ 259$ where the model-oracle have only 6 possible solutions, so the mutant can return 43 253 faulty solutions. To get a conformity status, the set of solutions of `Mut2` must be a subset of solutions of the model-oracle (i.e., $sol(Mut2) \subseteq sol(M)$). Figure 5 gives the number of solutions of the mutant after the addition of the 32 elementary constraints one by one in an incremental way. The set of solutions is then reduced after each constraint addition until it reaches 6 solutions. Table II shows the number of solutions of the different car sequencing mutants before and after correction. We note that the model-oracle of car sequencing have only 6 possible solutions. `Mut1` to `Mut4`, the correction removes an important number of bad solutions. The set of solutions of `Mut5`, `Mut6` and `Mut7` are reduced to empty due to the fault injected. The correction on these mutants keep the 6 possible solutions as correct solutions.

### *D. Threats to Validity*

External threats to validity lay on the source of constraint problems that we used for our experiments. We have selected

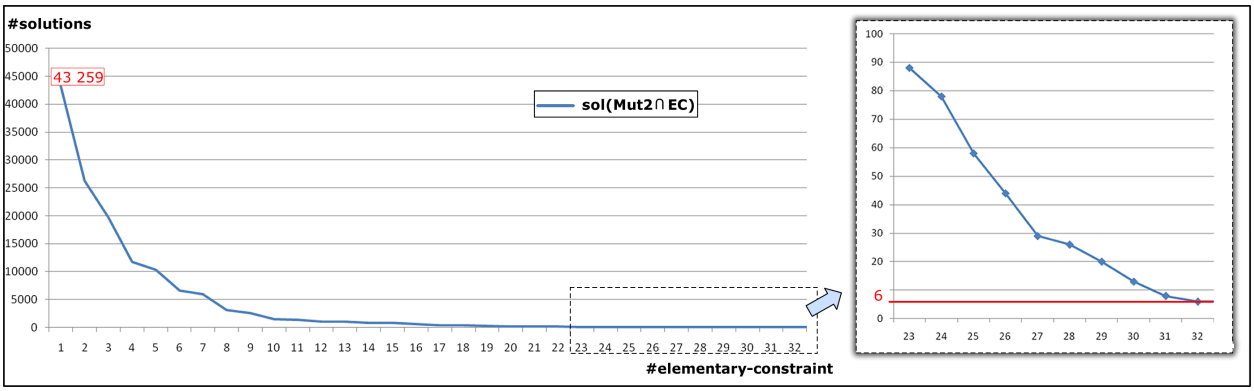


Fig. 5. `Mut2` correction.

TABLE III
FAULT DETECTION, LOCALIZATION AND CORRECTION ON CLASSICAL BENCHMARK PROBLEMS

| | Mutants | Fault injected | Detection | | Localization | | Correction | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | non-conformity | time | susp. ct | time | removed ct | added ct | time |
| Golomb rulers (m=6) | Mut1 | ct2 | [0 1 3 6 13 20] | 0.86s | ∅ | 0.96s | ∅ | 50 EC | 17.43s |
| | Mut2 | ct3 | [0 1 3 10 13 20] | 1.32s | ∅ | 2.65s | ∅ | 49 EC | 27.03s |
| | Mut3 | ct3 | [0 1 3 6 13 20] | 0.67s | ∅ | 4.96s | ∅ | 50 EC | 46.70s |
| | Mut4 | ct5 | [0 9 11 12 15 19] | 1.01s | ∅ | 10.54s | ∅ | 5 EC | 29.63s |
| | Mut5 | ct1 | sol(Mut5)=∅ | 10.46s | ct1 | 9.57s | ct1 | 3 EC | 1 698.04s |
| | Mut6 | ct7 | sol(Mut6)= ∅ | 9.37s | ct7 | 9.36s | ct7 | 22 EC | 107.81s |
| | Mut7 | ct9 | sol(Mut7)= ∅ | 35.10s | ct9 | 13.50s | ct9 | ∅ | 108.45s |
| n-queens (n=8) | Mut1 | ct11 | sol(Mut1)=∅ | 8.32s | ct11 | 7.53s | ct11 | ∅ | 18.21s |
| | Mut2 | ct12 | sol(Mut2)=∅ | 8.21s | ct12 | 8.10s | ct12 | ∅ | 18.10s |
| | Mut3 | ct11 | sol(Mut3)=∅ | 7.85s | ct11 | 8.14s | ct11 | ∅ | 18.35s |
| | Mut4 | ct12 | sol(Mut2)=∅ | 8.06s | ct12 | 8.16s | ct12 | ∅ | 18.23s |
| | Mut5 | ct4 | [8 7 6 5 4 3 2 1] | 2.78s | ct4 | 2.32s | ct4 | 28 EC | 7.21s |
| | Mut6 | ct3 | [7 2 3 6 8 1 5 4] | 3.34s | ∅ | 0.98s | ∅ | 56 EC | 9.28s |
| | Mut7 | ct1 | [8 4 3 6 5 7 2 1] | 3.07s | ∅ | 0.53s | ∅ | 56 EC | 9.12s |
| s. golfer (sg_I) | Mut1 | ct1 | sol(Mut1)=∅ | 10.28s | ct1 | 3.65s | ct1 | 58 EC | 32.95s |
| | Mut2 | ct2 | sg1 | 0.31s | ∅ | 3.51s | ∅ | 75 EC | 18.32s |
| | Mut3 | ct3 | sol(Mut3)=∅ | 9.85s | ct3 | 3.71s | ct3 | 58 EC | 36.28s |
| | Mut4 | ct4 | sol(Mut4)=∅ | 9.37s | ct4<br>ct5 | 3.64s | ct4<br>ct5 | 58 EC<br>58 EC | 67.39s |
| | Mut5 | ct5 | sol(Mut5)=∅ | 9.21s | ct5 | 3.75s | ct5 | 58 EC | 34.18s |
| sg_I: weeks = 4; groups = 3; groupSize = 3;<br>sg1=[[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]] | | | | | | | | | |
| car seq. (cSeq_I) | Mut1 | ct2 | [4 5 4 6 3 6 5 1 3 2] | 5.68s | ∅ | 1.34s | ∅ | 32 EC | 6.09s |
| | Mut2 | ct3 | [4 1 6 4 3 5 3 6 2 5] | 5.82s | ∅ | 1.54s | ∅ | 32 EC | 6.28s |
| | Mut3 | ct2 | [4 6 2 5 3 6 1 3 5 4] | 7.18 | ∅ | 1.51s | ∅ | 31 EC | 3.28s |
| | Mut4 | ct2 | sol(Mut4)=∅ | 2.78s | ct2 | 3.07s | ct2 | 37 EC | 34.95s |
| | Mut5 | ct1 | sol(Mut5)=∅ | 2.62s | ct1 | 4.96 | ct1 | ∅ | 9.08s |
| | Mut6 | ct6 | sol(Mut6)=∅ | 2.64s | ct6 | 5.09s | ct6 | ∅ | 9.23s |
| | Mut7 | ct5 | sol(Mut6)=∅ | 2.40s | ct5<br>ct6 | 5.09s | ct5<br>ct6 | ∅<br>∅ | 14.98s |
| cSeq_I: nbSlots = 10; nbCars = 6; nbOptions = 5; demand = [1, 1, 2, 2, 2, 2];<br>capacity = [[1,2],[2,3],[1,3],[2,5],[1,5]]; optionDemand = [5 ,6 ,3 ,4 ,2];<br>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]]; | | | | | | | | | |

**susp. ct:** suspicious constraints, EC: elementary constraint

well-known and difficult problems from the Constraint Community, which allowed us to validate our approach on pertinent empirical data. However, these problems come from academia and might not perfectly reflect industrial usage of Constraint Programming in critical applications. In addition, we manually injected faults of our own in the constraint programs under test in our experiments. Although these faults were selected to show the capabilities of an automatic correction approach and thus, are not easy to spot and fix, 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 correction of constraint programs is based on the availability of a Model-Oracle for a given 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 necessarily kept for analysis in industrial development process.

## VII. CONCLUSION

This paper has introduced a new framework for the correction of constraint programs. The proposed framework is built on the hypothesis that a first declarative and simple constraint

model-oracle is available from the problem specification analysis. This model-oracle is then refined using various techniques to form an improved constraint program. We have defined a conformity relation forcing the solutions of the constraint program to be included in those of the model-oracle. The proposed correction algorithm is preceded by a localization step to detect suspicious constraints violating the conformity relation. After that, the correction process finds the constraints of the model-oracle that should be removed or added to the constraint program in order to correct it. The final result of the correction process is a set of pairs, where each pair is composed of suspicious constraints and correction constraints.

Experiments with our tool are promising and provide a first validation of the proposed approach. We believe that tools that can automatically test and correct constraint programs will help to facilitate the adoption of constraint programs in critical applications. Further works include the improvement of the constraint solving process that we introduced here by considering improved versions of the computation of negated constraints, dichotomic algorithms to explore suspicious set of constraints and the exploitation of open constraint solvers to investigate coverage criteria for constraint programs.

## REFERENCES


[CRVH08] H. Collavizza, M. Rueher, and P. Van Hentenryck. Cpbpv: A constraint-programming framework for bounded program verification. In *Proc. of CP2008*, LNCS 5202, pages 327–341, 2008.

[DW10] Vidroha Debroy and W. Eric Wong. Using mutation to automatically suggest fixes for faulty programs. In *Proc. of ICST'10*, pages 65–74, April 2010.

[FGJ+07] A.M. Frisch, M. Grum, C. Jefferson, B. Martnez, and H.I. Miguel. The design of essence: a constraint language for specifying combinatorial problems. In *Proceedings of IJCAI'07*, pages 80–87, 2007.

[FPA+07] P. Flener, J. Pearson, M. Agren, Garcia-Avello C., M. Celiktin, and S. Dissing. Air-traffic complexity resolution in multi-sector planning. *Journal of Air Transport Management*, 13(6):323 – 328, 2007.

[Got09] A. Gotlieb. Tcas software verification using constraint programming. *The Knowledge Engineering Review*, 2009. Accepted for publication.

[HMGW] Brahim Hnich, Ian Miguel, Ian P. Gent, and Toby Walsh. A problem library for constraints. www.csplib.org.

[HO05] Alan Holland and Barry O'Sullivan. Robust solutions for combinatorial auctions. In *ACM Conference on Electronic Commerce (EC-2005)*, pages 183–192, 2005.

[JFGG09] Dennis Jeffrey, Min Feng, Neelam Gupta, and Rajiv Gupta. Bugfix: A learning-based tool to assist developers in fixing bugs. In *ICPC*, pages 70–79, 2009.

[Jun04] Ulrich Junker. Quickxplain: Preferred explanations and relaxations for over-constrained problems. In *AAAI*, pages 167–172, 2004.

[JV08] U. Junker and D. Vidal. Air traffic flow management with ilog cp optimizer. In *International Workshop on Constraint Programming for Air Traffic Control and Management*, 2008. 7th EuroControl Innovative Research Workshop and Exhibition (INO'08).

[LGL10a] N. Lazaar, A. Gotlieb, and Y. Lebbah. Fault localization in constraint programs. In *Proc. of the 2010 IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2010*, Oct. 2010.

[LGL10b] N. Lazaar, A. Gotlieb, and Y. Lebbah. On testing constraint programs. In *Proc. of Principles of Constraint Programming, CP'2010*, Sept. 2010.

[MNR+08] 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.

[Ran93] W. T. Rankin. Optimal golomb rulers: An exhaustive parallel search implementation. Master's thesis, Duke University, Durham, 1993.

[RBW06] Francesca Rossi, Peter van Beek, and Toby Walsh. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science Inc., New York, NY, USA, 2006.

[VH99] P. Van Hentenryck. *The OPL optimization programming language*. MIT Press, Cambridge, MA, USA, 1999.

[VHM05] Pascal Van Hentenryck and Laurent Michel. *Constraint-Based Local Search*. The MIT Press, 2005.

[WFLGN10] Westley Weimer, Stephanie Forrest, Claire Le Goues, and ThanhVu Nguyen. Automatic program repair with evolutionary computation. *Commun. ACM*, 53(5):109–116, 2010.

[WPF+10] Yi Wei, Yu Pei, Carlo A. Furia, Lucas S. Silva, Stefan Buchholz, Bertrand Meyer, and Andreas Zeller. Automated fixing of programs with contracts. In *Proc. of 19th ISSTA, Trento, Italy*, pages 61–72, July 2010.