# Improving Constraint-Based Testing with Dynamic Linear Relaxations

Tristan Denmat, Arnaud Gotlieb, Mireille Ducassé

# Improving Constraint-Based Testing with Dynamic Linear Relaxations

Tristan Denmat Arnaud Gotlieb Mireille Ducassé
*IRISA / INSA / INRIA*
*35042 Rennes Cedex, France*
{*Tristan.Denmat,Arnaud.Gotlieb,Mireille.Ducasse*}*@irisa.fr*

## Abstract

*Constraint-Based Testing (CBT) is the process of generating test cases against a testing objective by using constraint solving techniques. In CBT, testing objectives are given under the form of properties to be satisfied by program's input/output. Whenever the program or the properties contain disjunctions or multiplications between variables, CBT faces the problem of solving non-linear constraint systems. Currently, existing CBT tools tackle this problem by exploiting a finite-domains constraint solver. But, solving a non-linear constraint system over finite domains is NP_hard and CBT tools fail to handle properly most properties to be tested. In this paper, we present a CBT approach where a finite domain constraint solver is enhanced by Dynamic Linear Relaxations (DLRs). DLRs are based on linear abstractions derived during the constraint solving process. They dramatically increase the solving capabilities of the solver in the presence of non-linear constraints without compromising the completeness or soundness of the overall CBT process. We implemented DLRs within the CBT tool TAUPO that generates test data for programs written in C. The approach has been validated on difficult non-linear properties over a few (academic) C programs.*

## 1 Introduction

A trend in program testing is to combine static and dynamic analysis through the usage of the constraints technology. *Constraint-Based Testing (CBT)* was introduced fifteen years ago, in the context of mutation testing [9], to generate test cases by using constraint solving techniques. Since then it has been continuously developed to cover several application areas including hardware verification [16], test data generation [13, 10, 19, 22, 21], counter-example generation [15, 3], or software verification [6]. In many works, the need for improving constraint propagation capabilities of current constraint solvers was early recognized. Indeed, it appeared that most interesting properties could not be easily checked without enumerating a large portion of the search space which was unreasonable in the presence of large-sized integers or floats. Typically, one cannot enumerate all triplets of 32-bit integers in a reasonable amount of time. Among the tools that implement the CBT approach, InKa [10], ATGen [18] and PathCrawler [19] are automated test data generators based on constraint propagation over finite domains. These tools extract a constraint program from the source code to be tested and then exploit constraint propagation and backtracking to find test data that cover a selected element (path, branch or statement) within the program. CBT faces a number of technical difficulties, including iterative constructs [10], pointers and dynamically allocated structures [24, 19], floating-point computations [2], and so on. Among them, dealing efficiently with nonlinear expressions (disjunctions, multiplication over variables, ...) is particularly challenging as these constructs are ubiquitous in programs and properties.

In this paper we focus on the problem of nonlinear expressions over integers in CBT. Though the general theory of nonlinear integer constraint solving is undecideable, CBT tools based on finite domains constraint solvers handle nonlinear computations by bounding the variation domain of program variables. Note that in imperative languages such as C or Java, variables of primitive types are intrinsically bounded. When they are used to refute properties, CBT tools fall very easily into the worst case and computation times become prohibitive. Indeed, in that case, CBT tools try to show that a constraint system formed of the negation of a property and a set of constraints extracted from the program (called *verification constraints*) is unsatisfiable, which is a NP-hard problem over finite domains.

The main contribution of the article is to gather existing techniques from various research fields (Constraint-Based Testing, Linear Programming and Abstract Interpretation)

to design a hybrid constraint solver for solving *verification constraints*. On-demand abstraction, or relaxation, is used during the constraint solving process. We call this on-the-fly relaxation a *Dynamic Linear Relaxation* (DLR). The relaxations are linear and then the simplex algorithm can be used to reason on the abstraction. In this sense, the linear relaxations we use have much to do with the Abstract Interpretation on polyhedra [7]. Using DLRs does not compromise the completeness or soundness of the overall CBT process. Information computed thanks to the abstraction is seen as extra-information communicated to the initial non-linear constraint system, which remains the problem to solve. DLRs have been implemented within TAUPO, a constraint-based automatic test data generation for C programs and our first experimental results of checking difficult non-linear properties are presented.

Section 2 illustrates our approach over a small-sized (but non trivial) C program and three non-linear properties to be checked. Section 3 gives some background on Verification Constraints generation and constraint solving. Section 4 explains how to relax non-linear constructs that can be found in imperative programs and details the notion of *Dynamic Linear Relaxations*. Section 5 describes our implementation called TAUPO and first experimental results. We conclude in Section 6.

## 2 Motivating example

```
int divide(long n) {
  int r,i = 0;  int j = 0;
  int A1, S1 = 0; int A2, S2 = 0;
  while(i++ < n && 2*S1 < n*n) {
      A1 = S1;
      S1 = A1 + 2*n - 2*i + 1; }
  r = i - 1;
  while(j++ < n && 2*S2 < n*n) {
      A2 = S2;
      S2 = A2 + 2*j - 1;   }
  printf(''r: %d, A1: %d, A2: %d'',r,A1,A2);
  return r; }
```

**Figure 1. The program `divide`**

Nonlinear properties of interest include numerical properties extracted from the specifications of the program under test. As an example, consider the C program of Fig.1 that solve the following ancient mathematical problem:
*A farmer owns a field that is a rectangular isosceles triangle. He wants to divide it into two fields of equal area, following a line that is parallel to a side of the triangle. The division can only be performed by whole numbers of feet and there must be a free strip of width at least one between the two new fields to plant some trees.*

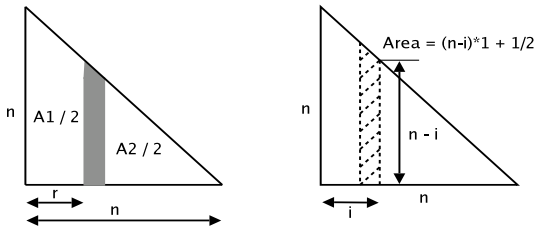


**Figure 2. Dividing the farmer's field**

$$\begin{array}{ll} \text{P1} & 2 * A_1 \leq n^2 \wedge 2 * A_2 \leq n^2 \\ \text{P2} & n^2 - A_1 - A_2 \leq 2 * n - 2 * r - 1 \\ \text{P3} & 2 * (n - (r+1))^2 \leq n^2 \leq 2 * (n - r)^2 \end{array}$$

**Figure 3. Three properties for `divide`**

Fig.2 explains the notations used in the program. Note that areas A1 and A2 correspond to twice the actual areas, in order to avoid non-integer variables.

Three properties (given in Fig.3) describe quite naturally the intended behaviour of the program: (P1) each area must be less than or equal to half the area of the original triangle ; (P2) the splitting error (the overall area minus the sum of the two field areas) is not bigger than a strip of width 1 ; (P3) the theoretical line of splitting is located in the free area. This latter property is stated as follows: the area on the right is less than or equal to half the whole area and this area plus the free zone is bigger than half the whole area.

Such properties and programs, although simple, are in fact representative of more industrial C code and specifications, which typically employ multiplication, division, modulo, polynomial expressions, disjunction, negation, and bit-to-bit operations to form non-linear expressions.

Constraint-Based Testing of the `divide` program may consist in showing that the constraint systems built upon the negation of the properties and any path of the program, is unsatisfiable. Our CBT method first inserts assertions corresponding to the three properties at the end of the `divide` program. Then, the program and the assertions are translated into a set of constraints by using the verification constraints generation process (see Sec.3). Finally, the resulting constraint system is solved with the hybrid constraint solver based on Dynamic Linear Relaxations (see Sec.4.2). On this example, our system automatically shows that no solution exist to the verification constraints[1] in 10.2s of CPU time on an Intel Pentium M 2Ghz with 1Gb of RAM. Whilst by using classical finite domain constraint solving, as implemented within the TAUPO test data generator, checking these nonlinear properties requires 53.3s of CPU time. This corresponds to a speedup of more than 5.

[1] in the example, each loop is unwound at most 4 times

## 3 Background on *Verification Constraints*

For the sake of clarity, in the rest of the paper, we will assume programs only contain basic statements: integer declarations ($T\,x$), integer expressions and assignments ($x := e$), statement sequences ($e_1; e_2$), and conditional statements (*if(b)* $e_1$ *else* $e_2$). We suppose that loops have been eliminated by replacing them with a given number of conditionals. Dealing with general loops in CBT is possible and has been described elsewhere [10]. We will also assume that properties to check appear in the source code as assertions (*assert e*). This facilitates our presentation without compromising its generality. Programs satisfying these restrictions, although quite simple, suffice to encounter difficulties related to non-linear expressions. Note however that our CBT tool also handles complex C programs that include pointer variables and dynamic structures [12, 5], function calls and floating-point variables [2] but this is outside the scope of the paper.

### 3.1 Verification Constraints generation

This section briefly describes how to generate *Verification Constraints* (VCs) that are constraints to be solved in order to get a test case that violates an assertion. When one shows that VCs have no solution then the corresponding assertion is satisfied. Note that this process is inspired from the *Verification Conditions* generation of the weakest precondition calculus of E. Diksjtra.

The first step of VCs generation consists in building the static single assignment form [8] of a program. In this equivalent form, each variable is assigned exactly once in the source code. For example, the piece of code $x = y; x = x + 1$ is translated into $x_0 = y; x_1 = x_0 + 1;$. A program under static single assignment is free of destructive assignments.

The second step consists in generating two sets: $E(S)$ and $V(S)$. $E(S)$ is a constraints set whose solutions correspond to program inputs on which statement $S$ either terminates normally or violates an assertion. In other words, solutions of $E(S)$ correspond to possible executions (normal and erroneous) of the program. $V(S)$ is a constraints set whose solutions correspond to program inputs on which execution of $S$ violates an assertion. Some of these solutions might also not satisfy constraints of $E(S)$ and, thus, do not correspond to possible executions of the program. Hence, erroneous executions of a program $S$ are those described by both $E(S)$ and $V(S)$. Consequently, a program will be free of assertion violations iff

$$sol(E(S) \wedge V(S)) = \emptyset$$

where *sol* denotes the set of integer solutions of a constraint set. On the contrary, any integer solution of $E(S) \wedge V(S)$ will be interpreted as a counter-example of an assertion of the program.

| S | E(S) | V(S) |
|---|---|---|
| $T\,x$ | $X \in min(T)..max(T)$ | false |
| $x := expr$ | $X = E(expr)$ | false |
| $e_1; e_2$ | $E(e_1) \wedge E(e_2)$ | $V(e_1) \vee V(e_2)$ |
| $If\ cond\ e_1\ Else\ e_2$ | $(R \Leftrightarrow E(cond)) \wedge$ $(R = 1 \wedge E(e_1) \vee$ $R = 0 \wedge E(e_2))$ | $V(e_1) \vee V(e_2)$ |
| $assert\ cond$ | $R \Leftrightarrow E(cond)$ | $R = 0$ |

**Figure 4. Verification Constraint Generation**

Fig.4 shows how to generate both sets $E(S)$ and $V(S)$ for the statements of our language. Program variables under static single assignment form are translated into mathematical variables. To distinguish them, we capitalize the mathematical variables on which constraints can be added. A variable declaration is translated into a domain constraint by using its type. For example, the declaration of a signed short integer $x$ leads to the constraint $E(short\ x) \stackrel{\text{def}}{=} X \in -2^{15}..2^{15} - 1$, where $a..b$ denotes all the integers in between $a$ and $b$. The generation of $E(S)$ and $V(S)$ for expressions ($expr$) and conditions ($cond$) is trivial so it is left apart to save space. Note that variable declaration and assignment statements cannot violate any assertion. Hence, $V(T\ x) \stackrel{\text{def}}{=} false$ and $V(x := expr) \stackrel{\text{def}}{=} false$. A sequence of statements leads to generate a conjunction of constraints in $E(S)$ while it leads to generate a disjunction in $V(S)$. Indeed, an assertion is violated in $e_1; e_2$ iff it is violated in $e_1$ OR in $e_2$. For generating $E(S)$ and $V(S)$ on conditionals and assertions, we use the notion of constraint reification. This process consists in associating a boolean variable $R$ to the truth value of a constraint. This is noted $R \Leftrightarrow C$ where $C$ denotes any basic constraint. This process is interesting as it avoids computing explicitly the negation of constraint $C$ which can be complex in the presence of non-elementary constraints. Reification deduces information on both sides: if the reification variable $R$ is equal to 1 (resp. 0) then $C$ is true (resp. false) and then $C$ is added to the constraint set while if constraint $C$ is shown to be true (resp. false) then $R = 1$ (resp. $R = 0$) is added to the constraint set. This process helps reasoning on the control flow of the program and on the violation of assertions.

In our approach, we build a hybrid solver based on Finite Domains constraint solving together with Dynamic Linear Relaxations that exploits Linear Programming techniques. The following gives a brief overview of these two paradigms.

## 3.2 Finite domains constraint solving

### 3.2.1 Local filtering

The fundamental idea in finite domains constraint solving is to prune the domain of the variables by using each constraint in turn as a filter. By considering a single constraint and its associated variables, one can infer that some values of the domains of these variables cannot be part of a solution. For example, consider the constraint $Z = X * Y$ with $D_X = 1..2$, $D_Y = 1..3$ and $D_Z = 4..8$, where $D_I$ denotes the domain of $I$. As all variables are positive, the following inference rules apply :

$$min\, D_X * min\, D_Y \leq Z \leq max\, D_X * max\, D_Y \quad (1)$$
$$min\, D_Z / max\, D_Y \leq X \leq max\, D_Z / min\, D_Y \quad (2)$$
$$min\, D_Z / max\, D_X \leq Y \leq max\, D_Z / min\, D_X \quad (3)$$

Note that inference rules depend on constraint and domains. If variables were not positive, inference rules would be more complicated. Using rule 1, a finite domains solver infers that $1 * 1 \leq Z \leq 2 * 3$ and, thus, remove values 7 and 8 from $D_Z$. With rule 2 the solver removes 1 from $D_X$. Rule 3 leads to $2 \leq Y$ and thus domains $D_X$ is pruned to 2..2, $D_Y$ is pruned to 2..3, while $D_Z$ is pruned to 4..6. At this point, the process has reached a fixpoint which is a state where no more value can be removed. Note that local filtering is incomplete in the sense that some values that are not part of a solution are not necessarily removed. In the example, there are no values for $X$ and $Y$ such that $Z = 5$.

### 3.2.2 Constraint propagation

Constraint propagation is the process that permits to deal with conjunction of constraints by propagating domain pruning information through the constraint store. Fig.5 shows a simplified version of the constraint propagation algorithm that is used in many finite domains solvers. The algorithm takes as inputs a constraint store $C$ and an initial set of domains $D$. Then, it iteratively applies the inference rules associated to each constraint of $C$ until a fixpoint is reached. As a result, the algorithm produces a new set of tighter domains $D'$ such that no more pruning is possible. Note that $D'$ contains all the solutions of the initial constraint problem. By adding a constraint to the previous example, we get the constraint store $\{Z = X * Y, Z = X + Y\}$. Domains are unchanged: $D_X = 1..2, D_Y = 1..3, D_Z = 4..8$. As previously seen, the first constraint leads to $D_X = 2..2, D_Y = 2..3, D_Z = 4..6$. Then, second constraint leads to $D_Z = 4..5$. Applying the first constraint again leads to $D_X = 2..2, D_Y = 2..2, D_Z = 4..4$ which corresponds to the single solution $(X = 2, Y = 2, Z = 4)$.

Actually, the set of domains $D'$ that is returned by the propagation algorithm is not necessarily reduced to a single point. During the propagation, if a domain becomes empty, the constraint store is shown to be unsatisfiable: there is no solution. If the domains are neither empty nor reduced to a single point, one must resort to enumerate all the remaining possible values of the domains to find a solution. This process, called labelling, is used to guide the search toward a solution. A variable $X$ is arbitrarily labelled to a value $a$ from its domain and the constraint $X = a$ is added to the constraint store. Constraint propagation is then restarted. If the constraint store is shown to be unsatisfiable under the hypothesis $X = a$ then $a$ is removed from the domain of $X$ and another value is tried. This process is repeated until either all the variables are labelled (a solution is found) or the domain of one variable becomes empty which shows the unsatisfiability of the constraint store.

**Input:** $C$, a constraint store and $D$ a set of domains
**Output:** $D'$ such that $D' \subseteq D$ and $sol(C, D) = sol(C, D')$

propagation(C,D)
  **repeat** $W := D$
    **forall** $c_i \in C$ **do**
      $D := local\_filtering(c_i, D)$
  **until** $D = W$
  **return** $D$

**Figure 5. Constraint propagation algorithm**

## 3.3 Linear Programming

Linear Programming encompasses a set of techniques aiming at finding the optimal value of a linear expression involving variables constrained by linear inequations [23]. A linear program is as follows : $maximize\ C$ under the constraints $A.X \leq B$ where $A$ is a (m,n)-matrix of values over $\mathbb{Q}$. $X$ is a vector of $n$ (rational) variables and $B$ is a vector of $m$ rational values. $C$ is a linear expression with coefficients in $\mathbb{Q}$. Though it has an exponential worst case complexity, the simplex algorithm is commonly used to solve linear programs as it performs well in practice. Note that *satisfiability* of a set of linear constraints can also be decided using the simplex as its first step aims at finding a solution to the $m$ linear inequations [23].

# 4 Solving the Verification Constraints

In this section, we focus on how to efficiently solve verification constraints generated from a program and a property to be checked. It is worth noticing that even if finite domains constraint propagation handles non-linear constraints, it behaves poorly on some trivial linear problems.

We start by giving such an example. Consider the following program:

$$P : int\, x, y, z;\ \ x := y + z;\ \ assert(x \leq y + z);$$

VC generation for program $P$ gives:

$$\begin{array}{rl} E(P) := & X, Y, Z \in -2^{31}..2^{31} - 1 \wedge \\ & X = Y + Z \wedge A \Leftrightarrow (X \leq Y + Z) \\ V(P) := & false \vee A = 0 \end{array}$$

Solving $E(P) \wedge V(P)$ leads to solve $X = Y + Z, X > Y + Z$ with $X, Y, Z \in -2^{31}..2^{31} - 1$. Unfortunately, in this example, constraint propagation without labelling does not remove any value from the domains. Thus, one must resort to enumerate all values from the domain of one variable before concluding that the constraint system is unsatisfiable and the assertion is verified.

In CBT problems, variable domains are potentially very large and constraint systems often have a few solutions only. This means that labelling has to be avoided as much as possible. Consequently, we suggest enriching constraint propagation with more precise deduction rules, by propagating also linear relationships among the variables.

## 4.1 Linear Relaxation of constraints

The principle of Linear Relaxation is to generate a linear program over rationals that encapsulates all the solutions of a nonlinear problem over integers [17]. In this technique, each nonlinear constraint is decomposed in binary or ternary constraints by introducing additional temporary variables. Then, each basic constraint over integers is over-approximated with linear inequalities over rationals by taking the variation domain of variables into account. As a result, the unsatisfiability of the linear program over rationals implies the unsatisfiability of the nonlinear constraints over integers.

In our framework, we tuned the Linear Relaxation technique to deal with the Verification Constraints. Fig.6 describes the Linear relaxation of some basic constraints that are encountered in the VCs. In the Figure, $\underline{X}$ denotes the lowest integer value of $D_X$ while $\overline{X}$ denotes its greatest integer value. Constraints on the left column of the array hold over integers while constraints on the right column hold over $\mathbb{Q}$.

| Constraint | Linear-Relaxation reformulation |
|---|---|
| $X \in a..b$ | $X \leq b, X \geq a$ |
| $Z = F_{lin}(X, Y, ...)$ | $Z = F_{lin}(X, Y, ...)$ |
| $Z = X * Y$ | $\left\{ \begin{array}{ll} Z - X.\underline{Y} - \underline{X}.Y + \underline{XY} & \geq 0, \\ X.\overline{Y} - Z - \underline{X}.\overline{Y} + \underline{X}.Y & \geq 0, \\ \overline{X}.Y - \overline{X}.\underline{Y} - Z + X.\underline{Y} & \geq 0, \\ \overline{XY} - \overline{X}.Y - X.\overline{Y} + Z & \geq 0 \end{array} \right\}$ |
| $R \Leftrightarrow F(X, Y, ...) \leq 0$ | $\left\{ \begin{array}{l} F(X, Y, ...) \leq \overline{F}.(1 - R)), \\ (1 - F(X, Y, ...) \leq (1 - \underline{F}).R) \end{array} \right\}$ |
| $F1_{lin} \vee F2_{lin}$ | $join(F1_{lin}, F2_{lin})$ |

**Figure 6. Linear Relaxation reformulation**

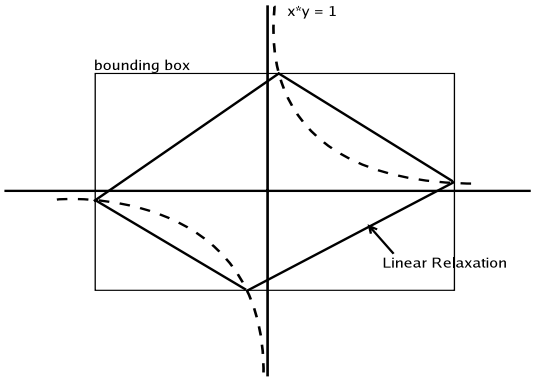


**Figure 7. Relaxation of multiplication**

### 4.1.1 Handling multiplication

The formula of Fig.6 for multiplication directly follows from the four following trivial inequalities:

$$\begin{array}{l} (X - \underline{X})(Y - \underline{Y}) \geq 0 \\ (X - \underline{X})(\overline{Y} - Y) \geq 0 \\ (\overline{X} - X)(Y - \underline{Y}) \geq 0 \\ (\overline{X} - X)(\overline{Y} - Y) \geq 0 \end{array}$$

In [1], it is proved that these inequalities are the smallest linear relaxation of $Z = X * Y$. Fig.7 shows a slice of the relaxation where $Z = 1$. Here, the rectangle corresponds to the bounding box of variables $X$ and $Y$, the dashed curve represents exactly $X * Y = 1$, while the four solid lines correspond to the inequalities of the relaxation.

### 4.1.2 Handling division, modulo and logical operators

Expressions built on division and modulo that are found in the VCs can be treated with the Linear Relaxation reformulation of the multiplication. The constraint $Q = A\ div\ B$ where $div$ denotes the Euclidian division rewrites to

$$(B * Q \leq A) \wedge (A < B * (Q + 1)) \wedge (B \neq 0)$$

Similarly, $R = A \, mod \, B$ where $mod$ denotes the Euclidian remainder rewrites to

$$(R = A - B * Q) \wedge (0 \leq R < B)$$

We applied the same principle for logical operators by studying the semantics of operators of the C programming language. The constraint $Z = X \; \&\& \; Y$ where $\&\&$ denotes the "logical and" operator rewrites to

$$Z = X * Y \; mod \; 2$$

The constraint $Z = X \; || \; Y$ where $||$ denotes "logical or" rewrites to

$$(Z = (X + Y - X * Y) \; mod \; 2) \wedge$$
$$(Z \geq X \; mod \; 2) \wedge (Z \geq Y \; mod \; 2)$$

while $Y = \tilde{} X$ where $\tilde{}$denotes "logical not" rewrites[2] to

$$(X_0 \in 0..1) \wedge (X * X0 = X) \wedge (Y = 1 - X_0)$$

#### 4.1.3 Handling reification

As said previously, reified constraint $R \Leftrightarrow C$ associates a boolean variable $R$ to the truth value of a constraint $C$. Without any loss of generality, let suppose that $C$ is of the form $F(X) \leq 0$ and the function $F$ is bounded[3], meaning that there exist $\underline{F}$ and $\overline{F}$ such that $\forall X \in D_X \; \underline{F} \leq F(X) \leq \overline{F}$. Then $R \Leftrightarrow F(X) \leq 0$ rewrites to the conjunction

$$(F(X) \leq \overline{F}.(1 - R)) \wedge (1 - F(X) \leq (1 - \underline{F}).R)$$

For example, consider the constraint $R \Leftrightarrow X \leq Y$, then $F(X,Y) = X - Y$, $\underline{F} = \underline{X} - \overline{Y}$, $\overline{F} = \overline{X} - \underline{Y}$, and the reified constraint rewrites to

$$(X - Y - (\overline{X} - \underline{Y}) * (1 - R) \leq 0) \wedge$$
$$(Y - X + 1 - (\overline{Y} - \underline{X} + 1) * R \leq 0)$$

Note that these inequations are interpreted over $\mathbb{Q}$. As a consequence, the boolean variable $R$ is interpreted as a rational over the continuous set $[0, 1]$. Hence, as already said, the set of solutions of these constraints (over-) approximates only the solutions over integers.

#### 4.1.4 Handling disjunctions

We use a special constraint for disjunctions that was first introduced in [10]. The principle of this constraint is to prove that one of the two disjuncts is unsatisfiable with the rest of the constraints and, thus, replace the overall disjunction by the other disjunct. When this case-based reasoning fails, the union of domains is computed. For example, from the disjunctive constraint $X = Y \vee X = 5$ with domains $D_X = -1000..1000, D_Y = 0..1$, if one cannot prove that one of the disjunct is implied by current information in the constraint store then it is easy to deduce that $D_X = 0..5, D_Y = 0..1$.

In our framework, we propose to extend the union principle with linear relations. For example, considering $X = Y + 10 \vee X = Y - 10$ with domains $D_X = D_Y = 0..20$ we deduce that $-10 \leq X - Y \leq 10$ while the above reasoning over domains would not have deduce anything new on the domains. The smallest linear relaxation of the disjunction of two linear inequations subsets is the convex hull of the corresponding polyhedra. Unfortunately, computing the convex hull of two polyhedra when they are given under the form of inequations is exponential in the number of dimensions of the polydra. Consequently, as originally proposed in the Abstract Interpretation community [20], we compute only an (over-)approximation of the convex hull by using a special operator called the *weak-join* operator. Roughly speaking, the weak-join of two sets of inequations consists in 1) enlarging the first polyhedron without changing the slope of the lines until it encloses the second polyhedron ; 2) enlarging the second polyhedron in the same way ; 3) returning the intersection of these two new sets of inequations. Fig.8 shows the difference between the convex hull and the weak join of two polyhedra $E$ and $F$. Formally, let $S = E \cup F$ be the set of inequations that appear in $E$ or in $F$. Let suppose that each inequation in $S$ is of the form $A_i.X \leq b_i$ where $A_i$ is a vector of $n$ coefficients, $X$ is a vector of $n$ variables and $b_i$ is a rational number. For each inequation in $S$ do

$$\begin{aligned} e &= maximize(A_i.X, E) \\ f &= maximize(A_i.X, F) \\ c &= max(e, f) \end{aligned}$$

$maximize(A_i.X, E)$ denotes a call to the simplex algorithm that computes the maximum value of expression $A_i.X$ under the linear constraints $E$. Then, $join(E, F) = \{A_i.X \leq c\}_{i \in 1..|S|}$. Based on the simplex, this algorithm performs well in practice.

### 4.2 Dynamic Linear Relaxations

Previous section presented the notion of linear relaxation within a bounding box. Here, we describe how to integrate these linear relaxations within the constraint propagation process through *Dynamic* Linear Relaxations (DLRs). The term "dynamic" is justified by the fact that linear relaxations depend on domains that are refined during the constraint solving process.

[2] In C, any non-null integer value is understood as true

[3] We consider finite domains and continuous functions only

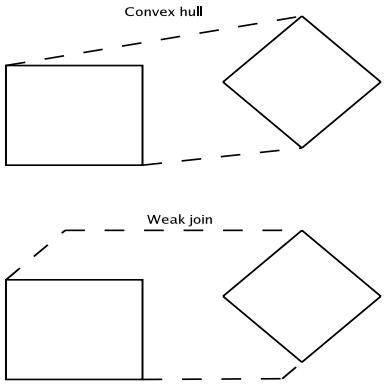


**Figure 8. Weak Join vs Convex Hull**

#### 4.2.1 Constraint propagation with DLRs

As shown previously, the linear relaxations we consider are over-approximations of binary or ternary constraints computed with the current domains of variables. These linear relaxations are then considered by a dedicated linear solver based on the simplex. As a first result, the linear solver can sometimes report on the detection of inconsistency of the original nonlinear problem. Unlike constraint propagation, the linear solver is complete with respect to inconsistency detection, meaning that if the relaxed linear program is inconsistent then this will be discovered. In this case, the nonlinear constraint system is shown to have no solution meaning that the corresponding assertion is verified. Recall that our primary goal is to avoid as much as possible the costly enumeration step to prove inconsistency.

Fig.9 shows the fixpoint algorithm that combines constraint propagation and DLRs. Function *DLR(C,D')* calls the dedicated linear solver and tests the satisfiability of the relaxed linear problem. When the relaxed linear problem contains solutions, one can still benefit from the simplex-based linear solver to prune the variation domains of variables. This process, called *enclose*, projects the current polyhedron over each variable and compares the resulting domain with the current domain of variable, as explained below.

#### 4.2.2 The *enclose* process

The *enclose* process computes an integer bounding box that includes all the integer solutions of the linear relaxed problem. The box can also include integer points that are no part of the polyhedron, as shown in Fig.10.

Fig.11 details the algorithm of the *enclose* process. This algorithm performs two calls to the simplex algorithm per variable appearing in the linear relaxation, one call for each bound. Then, it prunes the current domain of variable by

**Input:** $C$ a constraint store and $D$ a set of domains
**Output:** $D'$ such that $D' \subseteq D$ and $sol(C, D) = sol(C, D')$

propagation_DLR(C,D)
**repeat**
$D' := propagation(C, D)$
$D := DLR(C, D')$
**until** $D = D'$
**return** $D$

**Figure 9. Constraint propagation with DLRs**

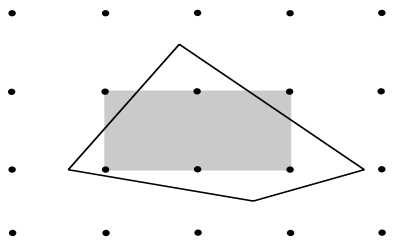

**Figure 10. The *enclose* process: the grey box contains all the integer points inside the polyhedron**

updating its (rational) bounds. Finally, integer rounding shaves the domain to fit with integer solutions only. For example, if a call to the simplex returns $\frac{7}{2}$ as an upper bound for $X$, then variable $X$ has to be lower or equal to 3. The algorithm of Fig.9 computes a fixpoint by successively applying constraint propagation and domain pruning via DLRs. We describe this process on the following example.

$$P : int\ x, y, z;\ z := x * y;\ z = x + y;\ assert\ z \leq 4;$$

VC generation for $P$ gives:

$$\begin{aligned} E(P) := \quad & X, Y, Z \in -2^{31}..2^{31} - 1 \wedge Z = X * Y \wedge \\ & Z = X + Y \wedge A \Leftrightarrow Z \leq 4 \\ V(P) := \quad & A = 0 \end{aligned}$$

For the sake of simplicity, suppose that the domains of variables $X$,$Y$ and $Z$ have been pruned to $-10..10$. The same reasoning applies if one keeps the original domains. When solving $E(P) \wedge V(P)$, constraint propagation leads to $D_X = D_Y = -5..10$ and $D_Z = 5..10$ by using $Z = X + Y$ and $Z < 4$. Using DLRs on this resulting constraint system

**Input:**
$R$, a set of linear inequalities
**Output:**
A set of integer intervals $D$ defining an hypercube that contains all the integer solutions of $R$

enclose($R$)
**forall** $X \in vars(R)$ **do**
$\underline{X} := \lceil minimize(X, R) \rceil$ ; $\overline{X} := \lfloor maximize(X, R) \rfloor$
$D_X := [\underline{X}, \overline{X}]$

**Figure 11. Algorithm for the $enclose$ process**

leads to:

$$\{-5 \leq X \leq 10, -5 \leq Y \leq 10, 5 \leq Z \leq 10,$$
$$Z = X + Y,$$
$$Z + 5.X + 5.Y + 25 \geq 0,$$
$$10.X - Z + 50 - 5.Y \geq 0,$$
$$10.Y + 50 - Z - 5.X \geq 0,$$
$$100 - 10.Y - 10.X + Z \geq 0\}$$

The *enclose* process prunes the domains of $X$ and $Y$ to $D_X = D_Y = 0..8$ while keeping $D_Z = 5..10$. Then, constraint propagation is restarted and leads to $D_X = D_Y = 1..8$. A more precise linear relaxation is built, taking into account these new domains and the $enclose$ process deduces $D_X = D_Y = 2..8$. Then, constraint propagation is restarted and leads to $D_X = D_Y = 2..5$. Finally, the linear relaxation is built on these domains and detects inconsistency of the linear relaxation problem. On this example, this is only the combination of constraint propagation and DLRs that shows the inconsistency of the constraint system. As a result, the assertion is shown to be satisfied.

# 5 Experimental results

## 5.1 Implementation

We implemented DLRs within the test data generator TAUPO. This tool implements a technique originating from the INKA test data generator [10]. TAUPO handles a non-trivial subset of the C programming language, including floating point numbers [2], pointer variables [11], function calls, dynamic structures. But, the tool does not currently handle unconstrained pointer arithmetic, function pointers, and non-trivial type casting.

Firstly, TAUPO translates a C program into static single assignment form. In addition, this process normalizes the expressions in 3-addresses code and associates boolean variables to the truth value of conditions (reification). Secondly, the tool generates the Verification Conditions from the code annotated with the assertions that come from properties, as described in Sec. 3.1. TAUPO exploits the following constraint libraries:

1. the *clpfd* library of Sicstus Prolog [4] which implements a finite domains constraint solver ;
2. the *clpq* library [14] that implements a linear programming solver based on simplex.

We made the two solvers cooperate by implementing our own constraint propagation queue and by building a dedicated constraint propagation solver.

For our experiments, we selected several programs from the literature and compared our implementation with and without DLRs. All the results have been computed using an Intel Pentium M 2Ghz with 1Gb of RAM.

## 5.2 The `Tritype` program

The first program is the `Tritype` program, which comes from the Software Testing folklore. An implementation of this program can be found in [6]. The program takes three unsigned short integers as input and checks whether these three numbers can be interpreted as the lengths of the sides of an isoscele, an equilateral or a scalene triangle. Although quite simple, this program is considered as difficult to analyze as it contains many non-feasible paths (more than forty over fifty-seven). Moreover, it contains many nested conditionals and disjunctions that both introduce non-linear constraints. We evaluated our approach based on DLRs by checking several assertions extracted from [6]. We describe our results for the following:

```
((i+j <= k)||(j+k<=i)||(i+k<=j))
                          ==> result == 4

!((i+j <= k)||(j+k<=i)||(i+k<=j)) &&
      (i==j) && (j==k) ==> result == 3
```

The first property states that when the three inputs $i, j$, and $k$ verify the condition, the corresponding figure is not a triangle while the second states that the triangle is equilateral. Checking these properties with TAUPO (with DLRs) requires 5sec of CPU time for the first and 0.5s of CPU time for the second while it requires more than 10 min of CPU time if one disconnects the DLRs in TAUPO. Thanks to the DLRs, the properties are checked without launching any enumeration step, which explains this remarkable result.

## 5.3 The `product` program

The second program implements the function:
$(a, b, c, d) \rightarrow |a - b| * |c - d|$ and contains disjunctions

and non-linear expressions (multiplication over variables). We exploited TAUPO to show that the program returns only non-negative values. Using DLRs, this check requires less than 0.3sec of CPU time. Indeed, DLRs infer that each absolute value is positive and, thus, that the product is also positive. Without DLR no such information are deduced during constraint propagation and a costly enumeration step is required. For domains restricted to $0..1000$, the enumeration step takes 192 sec. Note that the enumeration process requires a time proportional to the size of the domain whereas the deductions performed with DLRs are independent from this size. Thus, on this example, the benefits of DLRs can be made as important as desired and we limited our experience to a small domain for $a$. Although quite simple, this experience validates the idea presented in this paper: relaxing non-linear constraints can drastically speed up the constraint solving process in the context of Constraint-Based Testing.

### 5.4 The `divide` program

Our third example is the `divide` program shown in Fig. 1. Fig. 12 presents the total CPU time required for checking the three properties given in the first section of the paper (see Fig. 3) in function of the maximum number of loop unfolding. Recall that this parameter corresponds to the conversion of while loops into conditionals. In both

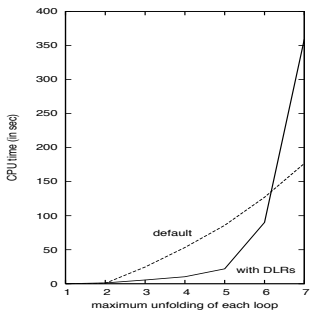


**Figure 12. CPU time required for the `divide` program**

cases of this example, the CPU time required to detect inconsistency is exponential w.r.t. the number of loop unfolding. From the value 7, the overhead introduced by the computations of DLRs makes the approach less interesting. But, for smallest values of this parameter, the tight overhead is undoubtly rewarded by the gain it makes on deductions during constraint propagation.

| $a..b$ | with DLRs | with enumeration |
|---|---|---|
| $-100..100$ | 0.15s | 0.05s |
| $-500..500$ | 0.81s | 1.39s |
| $-1000..1000$ | 1.91s | 5.53s |
| $-5000..5000$ | 24.8s | 138s |

**Table 1. Execution time of the solving process in presence of a slow convergence**

### 5.5 Slow Convergence Phenomenon

The algorithm presented on Figure 9 implements a fix point computation that interleaves calls to the simplex algorithm with classic propagation. On some examples, a slow convergence phenomenon appears. For example, consider the code
$w := x * y; z := x * y;\ assert(w = z + 1)$ and the following Verification Constraints are $W = X * Y, Z = X * Y, W = Z + 1$. Suppose that variables $X$ and $Y$ are in the domain $a..b$ and that $W$ and $Z$ have domains corresponding to the `int` type. With these domains, constraint propagation does not prune the domain of $X$ and $Y$. On the contrary, when we compute the linear relaxations of these constraints and call function $enclose$, $D_X$ and $D_Y$ are reduced to $a + 2..b - 2$. If we recomputed the approximation and recall function $enclose$, domains are reduced to $a+4..b-4$. And this phenomenon continues until domains become empty. This examples exhibits a slow convergence phenomenon that can compromise the interest of DLRs on some specific cases. However, without the DLRs a full enumeration of the domain of $X$ is required to detect the inconsistency of the system. Table 1 shows the CPU time required to detect the inconsistency in function of the interval $a..b$ in both cases.

## 6 Conclusion

In this paper, we introduced Dynamic Linear Relaxations (DLRs) as an improvement of constraint propagation in Constraint-Based Testing techniques. We showed that DLRs boost our current implementation called TAUPO on several programs extracted from the literature. We also pointed out that our approach could be responsible of a slow convergence phenomenon. DLRs are general and could be integrated within other Constraint-Based Testing tools. The price to pay is a tight overhead that is rewarded by more precise deductions during constraint propagation.

## References

[1] F. A. Al-Khayyal and J. E. Falk. Jointly constrained biconvex programming. *Mathematics if Operations Research*,

8(2):273–286, 1983.

[2] B. Botella, A. Gotlieb, and C. Michel. Symbolic execution of floating-point computations. *The Software Testing, Verification and Reliability journal*, 16(2):pp 97–121, June 2006.

[3] M. D. C. Pasareanu and W. Visser. Finding feasible abstract counter-examples. *International Journal on Software Tools for Technology Transfer*, 5(1), 2003.

[4] M. Carlsson, G. Ottosson, and B. Carlson. An open–ended finite domain constraint solver. In *Proc. of Programming Languages: Implementations, Logics, and Programs*, 1997.

[5] F. Charreteur, B. Botella, and A. Gotlieb. Modelling dynamic memory management in constraint-based testing. In *AIC-PART (Testing: Academic and Industrial Conference)*, Windsor, UK, Sep. 2007.

[6] H. Collavizza and M. Rueher. Exploration of the capabilities of constraint programming for software verification. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'06)*, pages 182–196, 2006.

[7] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of Symp. on Principles of Programming Languages*, pages 84–96. ACM, 1978.

[8] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Zadeck. Efficently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Language and Systems*, 13(4):451–490, Oct. 1991.

[9] R. DeMillo and J. Offut. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, September 1991.

[10] A. Gotlieb, B. Botella, and M. Rueher. Automatic test data generation using constraint solving techniques. In *Proc. of ISSTA'98*, pages 53–62, Clearwater Beach, FL, USA, March 1998.

[11] A. Gotlieb, T. Denmat, and B. Botella. Constraint-based test data generation in the presence of stack-directed pointers. In *20th IEEE/ACM International Conference on Automated Software Engineering (ASE'05)*, Long Beach, CA, USA, Nov. 2005. 4 pages.

[12] A. Gotlieb, T. Denmat, and B. Botella. Goal-oriented test data generation for pointer programs. *Information and Software Technology*, 49(9-10):1030–1044, Sep. 2007.

[13] N. Gupta, A. Mathur, and M. Soffa. Automated Test Data Generation Using An Iterative Relaxation Method. In *Foundations on Software Engineering*, Orlando, FL, Nov. 1998. ACM.

[14] C. Holzbaur. *OFAI clp(q,r) Manual*. Austrian Research Institutefor Artificial Intelligence, Vienna, 1.3.3 edition.

[15] D. Jackson and M. Vaziri. Finding bugs with a constraint solver. In *Proc. of ISSTA'00*, pages 14–25, 2000.

[16] D. Lewin, L. Fournier, M. Levinger, E. Roytman, and G. Shurek. Constraint satisfaction for test program generation. In *IEEE International Phoenix Conference on Communication and Computers, 1995*, 1995.

[17] G. P. McCormick. Computability of global solutions to factorable nonconvex programs: Part 1 - convex underestimating problems. *Mathematical Programming*, 10:147–175, 1976.

[18] C. Meudec. ATGen: automatic test data generation using constraint logic programming and symbolic execution. *Software Testing, Verification and Reliability*, 11(2):81–96, June 2001.

[19] P. M. N. Williams, B. Marre and M. Roger. Pathcrawler: Automatic generation of path tests by combining static and dynamic analysis. In *In Proc. Dependable Computing - EDCC'05*, pages 281–292, 2005.

[20] S. Sankaranarayanan, M. A. Colòn, H. Sipma, and Z. Manna. Efficient strongly relational polyhedral analysis. In *Proc. of VMCAI'06*, pages 115–125, 2006.

[21] B. Seljimi and I. Parissis. Using clp to automatically generate test sequences for synchronous programs with numeric inputs and outputs. In *Proc. of ISSRE'06*, pages 105–116, 2006.

[22] K. Sen, D. Marinov, and G. Agha. Cute: a concolic unit testing engine for c. In *Proc. of ESEC/FSE-13*, pages 263–272. ACM Press, 2005.

[23] G. Sierksma. *Linear and Integer Programming*. Dekker, 1996.

[24] S. Visvanathan and N. Gupta. Generating test data for functions with pointer inputs. In *Proceedings of the 17th IEEE Int. Conf. on Automated Software Engineering (ASE'02)*, Edinburgh, UK, September 2002.