A CLP Framework for Computing Structural Test Data

Arnaud Gotlieb[1], Bernard Botella[1], and Michel Rueher[2]

[1] Thomson-CSF Detexis, Centre Charles Nungesser 2, av. Gay-Lussac
78851 Elancourt Cedex, France
{Arnaud.Gotlieb,Bernard.Botella}@detexis.thomson-csf.com
[2] Université de Nice–Sophia-Antipolis, I3S
ESSI, 930, route des Colles - B.P. 145
06903 Sophia-Antipolis, France
rueher@essi.fr, http://www.essi.fr/~rueher

Abstract. Structural testing techniques are widely used in the unit testing process of softwares. A major challenge of this process consists in generating automatically test data, i.e., in finding input values for which a selected point in a procedure is executed. We introduce here an original framework where the later problem is transformed into a CLP(FD) problem. Specific operators have been introduced to tackle this kind of application. The resolution of the constraint system is based upon entailment techniques. A prototype system — named InKA— which allows to handle a non-trivial subset of programs written in C has been developed. First experimental results show that InKa is competitive with traditional ad–hoc methods. Moreover, InKa has been used successfully to generate test data for programs extracted from a real application.

1 Introduction

Structural testing techniques are widely used in the unit or module testing process. Structural testing requires:

1. Identifying a set of statements in the procedure under test, the covering of which implies the coverage of some criteria (e.g., statement or branch coverage);
2. Computing test data so that each statement of the set is reached.

The second point —called ATDG[1] problem in the following— is the corner stone of structural testing since it arises for a wide range of structural criteria. The ATDG problem is undecidable in the general case since it can be reduced to the halting problem. Classical ad-hoc methods fall into three categories:

– Random test data generation techniques which blindly try values [Nta98] until the selected point is reached;

[1] Automatic Test Data Generation.

- Symbolic-execution techniques [Kin76,DO93] which replace input parameters by symbolic values and which statically evaluate the statements along the paths reaching the selected point;
- Dynamic methods [Kor90,FK96] which are based on actual execution of procedure and which use heuristics to select values, e.g. numerical direct search methods.

The limit of these techniques mainly comes from the fact that they "follow one path" in the program and thus fail to reach numerous points in a procedure. A statement in a program may be associated with the set of paths reaching it, whereas a test datum on which the statement is executed follows a single path. However, there are numerous non-feasible paths, i.e., there is no input data for which such paths can be executed. Furthermore, if the procedure under test contains loops, it may contain an infinite number of paths.

We introduce here an original framework where the ATDG problem is transformed into a CLP problem over finite domains. Roughly speaking, this framework can be defined by the following three steps:

1. Transformation of the initial program into a CLP(FD) program with some specific operators which have been introduced to tackle this kind of application ;
2. Transformation of the selected point into a goal to solve in the CLP(FD) system ;
3. Solving the resulting constraint system to check whether at least one feasible control flow path going through the selected point exists, and to generate automatically test data that correspond to one of these paths.

The two first steps are based on the use of the "Static Single Assignment" form [CFR+91] and control-dependencies [FOW87]. They have been carefully detailed in [GBR98].

In this paper, we mainly analyze the third step: the constraint solving process. The key-point of our approach is the use of constraint entailment techniques to drive this process efficiently. In the proposed CLP framework test data can be generated without following one path in the program.

To validate this framework, a prototype system — named InKA— has been developed over the CLP(FD) library of Sicstus Prolog. It allows to handle a non-trivial subset of programs written in `C`. The first experimental results show that InKa overcomes random generation techniques and is competitive with other methods. Moreover, InKa has been used successfully to generate test data for programs extracted from a real application.

Before going into the details, let us illustrate the advantage of our approach on a very simple example.

1.1 Motivating Example

Let us consider the small toy–program given in Fig. 1. The goal is to generate a test datum, i.e. a pair of values for (x, y), for which statement 10 is executed.

"Static Single Assignment" techniques and control-dependencies analysis yield the following constraint system[2]:
$\sigma_1 = (x, y, z, t_1, t_2, u \in (0..2^{32} - 1) \wedge$
$\quad (z = x * y) \wedge (t_1 = 2 * x) \wedge (z \leq 8) \wedge (u \leq x) \wedge (t_2 = t_1 - y) \wedge (t_2 \leq 20)$
Variables t_1 and t_2 denote the different renaming of variable t.

```
int foo(int x, int y)
     int z, t, u;
1.   { z = x * y;
2.     t = 2 * x;
3.     if (x < 4)
4.       u = 10;
       else
5.       u = 2;
6.     if (z <= 8)
7.       { if (u <= x)
8.           { t = t - y;
9.             if (t <= 20)
10.              { ...
```

Fig. 1. Program foo

Local consistency techniques like *interval–consistency* [HSD98] cannot achieve any significant pruning: the domain of x will be reduced to $0..2^{16} - 1$ while no reduction can be achieved on the domain of y. So, the search space for (x, y) contains $(2^{16} - 1) \times (2^{32} - 1)$ possible test data. However, more information could be deduced from the program. For instance, the following relations could be derived from the first `if_then_else` statement (lines 3,4,5):
$\quad (x \geq 4 \wedge u = 2)$ holds if $\neg(x < 4 \wedge u = 10)$ holds
$\quad (x < 4 \wedge u = 10)$ holds if $\neg(x \geq 4 \wedge u = 2)$ holds
Entailment mechanisms allow to capture such information. Indeed, since $\neg(x < 4 \wedge u = 10)$ is entailed by $u \leq x$, we can add to the store the constraint $(x \geq 4 \wedge u = 2)$. Filtering $x \geq 4 \wedge u = 2 \wedge \sigma_1$ by *interval-consistency* reduces the domain of x to 4..11 and the domain of y to 0..2.
This example shows that entailment tests may help to drastically reduce the search space. Of course, the process becomes more tricky when several conditional statements and loop statements are inter-wound.

Outline of the Paper. The next section introduces the notation and some basic definitions. Section 3 details how the constraint system over CLP(FD) is generated. Section 4 details the constraint solving process. Section 5 reports

[2] In this context, an **int** variable has an unsigned long integer value, i.e. a value between 0 and $2^{32} - 1$.

the first experimental results obtained with InKa, while section 6 discusses the extensions of our framework.

2 Notations and Basic Definitions

A *domain* in FD is a non-empty finite set of integers. A variable which is associated to a domain in FD is called a FD_variable and will be denoted by an upper-case letter. *Primitive constraints* in CLP(FD) are built with variables, domains, the $\in$ operator, arithmetical operators in $\{+, -, \times$ `div`, `mod` $\}$ [3] and the relations $\{>, \geq, =, \neq, \leq, <\}$. Note that the negation of a primitive constraint is also a primitive constraint. In the following, c possibly subscripted denotes exclusively a primitive constraint. A *constraint–store* σ is a conjunction of primitive and non-primitive constraints.

Non-primitive constraints are composed of combinators and guarded–constraints. *Combinators* are boolean combination of constraints. For example, the constraint `element`(I, L, V) which express that V is the I^{th} element in the list L is a combinator.

Guarded–constraints are built by using the blocking ask operator [HSD98] and are denoted $C_1 \longrightarrow C_2$, where C_1 and C_2 stand for constraints. C_1 is called the guard. The operational semantic of $C_1 \longrightarrow C_2$ is given by the following rules:

- The constraint $C_1 \longrightarrow C_2$ is removed and C_2 is added to σ when C_1 is entailed by σ;
- The constraint $C_1 \longrightarrow C_2$ is just removed when $\neg C_1$ is entailed by σ;
- The constraint $C_1 \longrightarrow C_2$ is suspended when neither C_1 nor $\neg C_1$ are entailed by σ;

Note that C_1 and C_2 are not restricted to be primitive and that checking whether $\neg C_1$ is entailed may require to compute the negation of a non-primitive constraint.

Entailment operations are based on partial consistencies. Two partial entailment tests have been introduced in [HSD98]: *domain-entailment* and *interval-entailment*. They are based upon *domain-consistency* and *interval-consistency*. Let $X_1, \ldots, X_n$ be FD_variables, let $D_1, \ldots, D_n$ be domains and let C be a constraint[4].

Definition 1 *(Domain-Consistency)*
A constraint C is domain-consistent if for each variable X_i and value $v_i \in D_i$ there exists values $v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n$ in $D_1, \ldots, D_{i-1}, D_{i+1}, \ldots, D_n$ such that $C(v_1, \ldots, v_n)$ holds. A store σ is domain-consistent if for every constraint C in σ, C is domain-consistent.

Interval consistency is based on an approximation of finite domains by finite sets of successive integers. More precisely, if D is a domain, D^* is defined by the set $\{\min(D), \ldots, \max(D)\}$ where $\min(D)$ and $\max(D)$ denote respectively the minimum and maximum values in D.

[3] `div` and `mod` represent the Euclidean division and remainder.
[4] We assume that all the constraints are implicitly defined on $X_1, \ldots, X_n$.

Definition 2 *(Interval-Consistency)*
A constraint C is interval-consistent if for each variable X_i and value $v_i \in \{min(D_i), max(D_i)\}$ there exist values $v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n$ in $D_1^*, \ldots, D_{i-1}^*, D_{i+1}^*, \ldots, D_n^*$ such that $C(v_1, \ldots, v_n)$ holds. A store σ is interval-consistent if for every constraint C in σ, C is interval-consistent.

The following relaxations of entailment are introduced in [HSD98]:

Definition 3 *(Domain-Entailment)*
A constraint $C(X_1, \ldots, X_n)$ is domain-entailed by $D_1, \ldots, D_n$ iff, for all values $v_1, \ldots, v_n$ in $D_1, \ldots, D_n$, $C(v_1, \ldots, v_n)$ holds.

Definition 4 *(Interval-Entailment)*
A constraint $C(X_1, \ldots, X_n)$ is interval-entailed by $D_1, \ldots, D_n$ iff, for all values $v_1, \ldots, v_n$ in $D_1^*, \ldots, D_n^*$, $C(v_1, \ldots, v_n)$ holds.

We introduce here another partial entailment test which is based on refutation:

Definition 5 *(abs-Entailment)*
A constraint C is abs-entailed by a store σ iff, filtering $\sigma \wedge \neg C$ by domain-consistency or interval-consistency yields an empty domain.

3 Generation of the Constraint System

Let P be a single procedure written in an imperative language, let n be a point (either a statement or a branch) in P. Solving the ATDG problem requires to compute a vector of input[5] values of P such that n is executed.

For the sake of simplicity, we first introduce the constraint system generation technique for an `array_if_while` language over integers. Procedure calls are handled in our framework but we assume that there is only one mechanism for passing arguments: the call–by–value mechanism. Programs must be well-structured and must avoid floating-point variables. A procedure is assumed to have a single return statement.

Next subsection recalls the general principles of the "Static Single Assignment" form [CFR+91]. The following subsections detail the transformation process of a program under SSA form into a CLP program.

3.1 Static Single Assignment Form

The SSA form is a version of a procedure on which every variable has a unique definition and every use of a variable is reached by this definition. The SSA form of a basic block is obtained by a simple renaming ($i = i + 1$ yields $i_2 = i_1 + 1$). For the control structures, SSA form introduces special assignments, called ϕ-functions, to merge several definitions of the same variable. For example, the

[5] An input variable is either a formal parameter or a referenced global variable.

```
if ($x < 4$)                      if ($x < 4$)
        $u = 10$;                         $u_1 = 10$;
else                              else
        $u = 2$;                          $u_2 = 2$;
                                  $u_3 = \phi(u_1, u_2)$;


$j = 1$;                              $j_1 = 1$;
                                      /* Heading - while */
                                      $j_3 = \phi(j_1, j_2)$;
while ($j * u \leq 16$)               while ($j_3 * u_3 \leq 16$)
        $j = j + 1$                           $j_2 = j_3 + 1$;
```

Fig. 2. SSA form of control statements

SSA form of the `if_then_else` statement is illustrated in the top of Fig. 2. The ϕ-function of the statement $u_3 = \phi(u_1, u_2)$ returns one of its argument: if the flow comes from the *then-* part then the ϕ-function returns u_1, otherwise it returns u_2.
For other structures such as loops, the ϕ-functions are introduced in a special heading which is executed at every iteration. The ϕ-functions work as usual: this explains the counter-intuitive renaming of variables (see Fig. 2).

For convenience, a list of ϕ-functions will be written with a single statement: $x_2 := \phi(x_1, x_0), \ldots, z_2 := \phi(z_1, z_0) \iff \boldsymbol{v_2} := \phi(\boldsymbol{v_1}, \boldsymbol{v_0})$ where $\boldsymbol{v_i}$ stands for a vector of variables.

3.2 Generation of the CLP Program

The basic idea is to translate each statement of the SSA form into a primitive constraint or a combinator, in order to build a CLP program. A clause is generated for each procedure P of the program. The head of the clause has several arguments:

- A list of FD_variables associated with the parameters of P ;
- A list of FD_variables associated with the referenced globals of P ;
- A list of FD_variables associated with the local variables used inside the decisions of P ;
- A list of FD_variables associated with the globals defined inside P ;
- A single FD_variable associated with the expression returned by P.

Now, let us detail the transformation process.

Declaration. A type declaration of a variable x_i is translated into a primitive constraint of the form: $X_i \in Min_T..Max_T$ where Min_T (resp. Max_T) is the minimum (resp. maximum) value of the type T. Such a constraint prevents overflows of values, a condition which is required to generate a test datum on which a selected point is reached.

Array. SSA form provides special expressions to handle arrays: $access(a_0, k)$ which evaluates to the k^{th} element of a_0, and $update(a_0, j, w)$ which evaluates to an array a_1 which has the same size and the same elements as a_0, except for j where value is w. *access* and *update* expressions are transformed into `element/3` constraints:

- $v = access(a_0, k)$ is translated into $\texttt{element}(K, A_0, V)$;
- a definition statement $a_1 = update(a_0, j, w)$ is translated into $\texttt{element}(J, A_1, W) \quad \bigwedge_{I \neq J}(\texttt{element}(I, A_0, V) \wedge \texttt{element}(I, A_1, V))$.

Conditional. The *if_then_else* statement is treated by using a combinator, called `ite/3`. For example, the *if_then_else* statement of Fig. 2 is translated into: $\texttt{ite}(X < 4,\ U_1 = 10 \wedge U_3 = U_1,\ U_2 = 2 \wedge U_3 = U_2)$. The conditional statement express an exclusive disjunction between two paths. So, $\texttt{ite}(c, C_1 \wedge \ldots \wedge C_n, C'_1 \wedge \ldots \wedge C'_m)$ holds iff $(c \wedge C_1 \wedge \ldots \wedge C_n)$ or $(\neg c \wedge C'_1 \wedge \ldots \wedge C'_m)$ holds, where c is a primitive constraint, and $C_1, \ldots, C_n, C'_1, \ldots, C'_m$ are primitive or non-primitive constraints. The operational semantic of combinator `ite/3` is based on the following rules:

Definition 6 `ite/3` *(Operational Semantic)*
$\texttt{ite}(c, C_1 \wedge \ldots \wedge C_n,\ C'_1 \wedge \ldots \wedge C'_m)$ *is reduced to the four following guarded–constraints:*

- $c \longrightarrow C_1 \wedge \ldots \wedge C_n$
- $\neg c \longrightarrow C'_1 \wedge \ldots \wedge C'_m$
- $\neg(c \wedge C_1 \wedge \ldots \wedge C_n) \longrightarrow (\neg c \wedge C'_1 \wedge \ldots \wedge C'_m)$
- $\neg(\neg c \wedge C'_1 \wedge \ldots \wedge C'_m) \longrightarrow (c \wedge C_1 \wedge \ldots \wedge C_n)$

The first two guarded–constraints result from the operational semantic of the *if_then_else* statement in an imperative language. The last two are introduced to allow a more effective pruning. c and $\neg c$ are included in the guards to facilitate the detection of inconsistencies by *abs-entailment* (see section 4).

Loop. Unlike the conditional, the *while* statement under SSA form cannot be translated directly. A *while* statement in SSA form is of the general form: $\boldsymbol{v_2} = \phi(\boldsymbol{v_0}, \boldsymbol{v_1})$ *while* (c) $\{C_1; \ldots; C_p\}$ where $\boldsymbol{v_0}$ is the vector of input variables of the *while*, $\boldsymbol{v_1}$ is the vector of variables defined inside the body of the *while*, and $\boldsymbol{v_2}$ is the vector of variables used inside and outside the *while*. This statement is transformed into a $\texttt{w}(c, \boldsymbol{V_0}, \boldsymbol{V_1}, \boldsymbol{V_2}, C_1 \wedge \cdots \wedge C_p)$ combinator, which is a constraint generation program.
$\texttt{w}(c, \boldsymbol{V_0}, \boldsymbol{V_1}, \boldsymbol{V_2}, C_1 \wedge \ldots \wedge C_p)$ holds iff $(\neg\dot{c} \wedge \boldsymbol{V_0} = \boldsymbol{V_2})$ or $(\dot{c} \wedge \dot{C_1} \wedge \ldots \wedge \dot{C_p} \wedge \texttt{w}(c, \boldsymbol{V_1}, \boldsymbol{V_3}, \boldsymbol{V_2}, \ddot{C_1} \wedge \ldots \wedge \ddot{C_p}))$ holds, where c is a primitive constraint, $\dot{c} = subs(\boldsymbol{V_2} \leftarrow \boldsymbol{V_0}, c)$; $\boldsymbol{V_0}, \boldsymbol{V_1}$ and $\boldsymbol{V_2}$ are three vector of FD_variables, $\boldsymbol{V_3}$ is a newly created vector of FD_variables, $\dot{C_1} = subs(\boldsymbol{V_2} \leftarrow \boldsymbol{V_0}, C_1), \ldots, \dot{C_p} = subs(\boldsymbol{V_2} \leftarrow \boldsymbol{V_0}, C_p)$, and $\ddot{C_1} = subs(\boldsymbol{V_1} \leftarrow \boldsymbol{V_3}, C_1), \ldots, \ddot{C_p} = subs(\boldsymbol{V_1} \leftarrow \boldsymbol{V_3}, C_p)$; *subs* being the substitution of variables over a term.

The operational semantic of `w/5` is defined by the following rules:

Definition 7 `w/5` *(Operational Semantic)*
`w`*$(c, \boldsymbol{V_0}, \boldsymbol{V_1}, \boldsymbol{V_2}, C_1 \wedge \ldots \wedge C_p)$ is reduced to the four following guarded–constraints:*

- $\dot{c} \longrightarrow (\dot{C_1} \wedge \ldots \dot{C_p} \wedge w(c, \boldsymbol{V_1}, \boldsymbol{V_3}, \boldsymbol{V_2}, \ddot{C_1} \wedge \ldots \wedge \ddot{C_p}))$
- $\neg\dot{c} \longrightarrow \boldsymbol{V_0} = \boldsymbol{V_2}$
- $\neg(\dot{c} \wedge \dot{C_1} \wedge \ldots \dot{C_p}) \longrightarrow (\neg\dot{c} \wedge \boldsymbol{V_0} = \boldsymbol{V_2})$
- $\neg(\neg\dot{c} \wedge \boldsymbol{V_0} = \boldsymbol{V_2}) \longrightarrow (\dot{c} \wedge \dot{C_1} \wedge \ldots \dot{C_p} \wedge w(c, \boldsymbol{V_1}, \boldsymbol{V_3}, \boldsymbol{V_2}, \ddot{C_1} \wedge \ldots \wedge \ddot{C_p}))$

The first two guarded–constraints result from the behavior of the *while* statement. Whenever the decision of the statement $\dot{c}$ is verified, then the body is executed and another `w/5` is stated. When the decision is refuted, the body is skipped and the input variables of the statement are equated to the vector of used variables.
The third guarded–constraint is based on the following observation: if the constraints of the body are inconsistent w.r.t the current information in the store, then the loop cannot be performed. The last guarded–constraint comes from the following observation: if the value of a variable is different before and after the *while* statement, then the body of the loop must be executed at least once. Note that the guards of both combinators are either primitive constraints or negations of conjunction of constraints, so the implementation of *abs–entailment* becomes straightforward (see section 4).
Let us illustrate how `w/5` works on the example of Fig. 2. The *while_do* statement is translated into: `w`$(J_3 * U_3 \leq 16, [J_1], [J_2], [J_3], J_2 = J_3 + 1)$. If the store contains $J_1 = 1, J_3 = U_3$, then the fourth guarded–constraint is activated because $\neg(\neg(J_1 * U_3 \leq 16) \wedge J_1 = J_3)$ is entailed by the store. So, the following constraints are added to the store: $J_1 * U_3 \leq 16 \wedge J_2 = J_1 + 1 \wedge$ `w`$(J_3 * U_3 \leq 16, [J_2], [J_\#], [J_3], J_\# = J_3 + 1)$ where $J_\#$ is a newly created variable.

Procedure Call. A procedure call is translated into a goal to solve. For example, a statement such as $v = foo(x, 29)$ is translated into
`foo`$([X, 29], [], \textit{Liste\_of\_locals}, [], V)$, where `foo` is the name of the clause generated for the procedure foo and *Liste_of_locals* is a the list of FD_variables associated to local variables and referenced in the decisions of the procedure. Such a mechanism allows the treatment of recursive procedure.

3.3 Generation of the CLP Goal

The decisions which must be verified to reach a given point in a procedure are called the *control-dependencies* [FOW87]. They are syntactically determined in well–structured procedures. For loop statements, these decisions are computed dynamically. Let $C(foo, 10)$ be the *control-dependencies* associated with point 10 in the procedure foo of Fig. 1. So, we have: $C(foo, 10) = (Z \leq 8) \wedge (U_3 \leq X) \wedge (T_2 \leq 20)$. The selected point determines a goal to solve with the clauses of the generated CLP(FD) program :

$$\longleftarrow C(foo, 10), foo([X, Y], [], [X, Z_1, U_3, T_2], [], RET)$$

The generated CLP program for program `foo` and the goal associated with point 10 are given in the Fig. 3.

```
foo([X, Y], [], [X, Z_1, U_3, T_2], [], RET) ⟵
    X, Y, Z, T_1, T_2, U_1, U_2, U_3, RET ∈ 0..2^32 − 1,
    Z = X ∗ Y,
    T_1 = 2 ∗ X,
    ite(X < 4, U_1 = 10 ∧ U_3 = U_1, U_2 = 2 ∧ U_3 = U_2),
    ite(Z ≤ 8,
        ite(U_3 ≤ X, T_2 = T_1 − Y ∧
            ite(T_2 ≤ 20,
                ...
    RET = ...


⟵ (Z ≤ 8) ∧ (U_3 ≤ X) ∧ (T_2 ≤ 20), foo([X, Y], [], [X, Z_1, U_3, T_2], [], RET)
```

Fig. 3. CLP Program generated for the program foo

4 Solving the Goal

In our framework, the constraint solving process is based on:

1. A filtering process based on partial consistency techniques and entailment techniques ;
2. A search procedure which combines an enumeration process and a constraint propagation step.

In view of the operational semantics of combinators introduced in the previous section, there are several operations to be implemented. They include an entailment test, an algorithm for processing the guarded–constraints, and the implementation of the combinators themselves.

4.1 Entailment Test

Three levels of entailment relaxations may be used to achieve entailment tests: *domain–entailment*, *interval–entailment* and *abs–entailment*, defined in section 2.

Consider the following example: $\sigma = (X \in 1..100) \wedge (Y \in 9..11) \wedge (X \neq Y)$ and the question " is $(X * Y \neq 100)$ entailed by σ ?".
The constraint is neither interval-entailed, nor domain-entailed because $(X = 10, Y = 10)$ does not verify the constraint. Thus, in our framework, we have implemented *abs–entailment* which is more effective —at least on our problems—

than *domain-entailment* and *interval-entailment*. Practically, we add the negation of the considered constraint C to the store before starting a filtering step by interval–consistency. When the domain of one variable is reduced to an empty set, constraint C is entailed ; when all the constraints are interval–consistent, no deduction can be done and the previous store must be restored. For instance, filtering the store $\sigma \wedge \neg C = (x \in 1..100) \wedge (y \in 9..11) \wedge (x \neq y) \wedge (x * y = 100)$ by interval-consistency leads to an empty domain for both variables, and then proves that the constraint $x * y \neq 100$ is *abs-entailed.*

This relaxation of entailment can be seen as a proof by refutation. Technically, *abs-entailment* requires to compute the negation of the considered constraint C. Since we only test the entailment of primitive constraints or the negation of conjunctions of constraints in our framework, this computation becomes straightforward.

Note also that no suspension will remain in the constraint store at the end of the resolution, since the last step of the solving process is an enumeration step.

4.2 Processing Guarded–Constraints

The guarded–constraints are evaluated iteratively in the store. The algorithm for processing guarded–constraints is given in Fig. 4.

```
/* Let $C_1, C_2$ be two constraints and $\sigma$ be the current store */
/* Process $C_1 \longrightarrow C_2$ in $\sigma$ */

    if filtering $\sigma \wedge \neg C_1$ by interval–consistency yields an inconsistency
    then    /* $C_1$ is abs–entailed by $\sigma$ */

            $\sigma \leftarrow (\sigma \cup \{C_2\}) \setminus \{C_1 \longrightarrow C_2\}$;

    if filtering $\sigma \wedge C_1$ by interval–consistency yields an inconsistency
    then    /* $\neg C_1$ is abs–entailed by $\sigma$ */

            $\sigma \leftarrow \sigma \setminus \{C_1 \longrightarrow C_2\}$;

    if neither $C_1$ nor $\neg C_1$ are abs–entailed by $\sigma$
    then continue
            /* The guarded–constraint $C_1 \longrightarrow C_2$ is suspended in $\sigma$ */
```

Fig. 4. Algorithm for processing guarded–constraints

Note that the second rule can be ignored until the end of the computation because it does not add any constraint to the store. Two kind of problems may occur with this algorithm:

- The store may contain other guarded–constraints which are activated as soon as a filtering is started ;
- The store may contain a non–terminating combinator. In fact, some `w/5` combinator may introduce guarded–constraints which will recursively put other `w/5` combinators in the store. This pitfall can be seen as a consequence of the halting problem.

A practical solution for both difficulties consists in ignoring any other guarded–constraint or combinator of the store during the filtering of $\sigma \wedge \neg C_1$. Other awakening policies exist[Got00] but are not discussed in this paper.

4.3 Search Process

Filtering by partial consistencies does not always yield a solution, thus a search step is necessary. Note that, up to this point, no choice point has been set up. In fact, the disjunctions introduced by the combinators are "captured" by the entailment tests. As usual, the search is interleaved with constraint propagation. Since the class of programs is unbound, experiments are the best way to determine a good heuristic for the ATDG problem. We have tested the *first-fail*, *first-fail constrained*, *domain-splitting* heuristics among others. Iterative domain–splitting yields the best results in average [Got00].

The search process stops in one of the following states:

- **Success: A solution of the constraint system was found.** In our framework, such a solution is a test datum on which the selected point n is reached in the procedure P, hence it is a solution to the ATDG problem.
- **Success: The inconsistency of the constraint system has been detected.** If an inconsistency of the store is detected during the initial filtering step or during the search process, we can state that n is unreachable in P, i.e. there is no test datum on which n is executed[6]; hence, the ATDG problem has no solution. This is an important information for the tester.
- **Failure: The search process did not reach a success state during the allowed amount of CPU time.** This can result from the non–termination problem of `w/5`. Consider a reachable point n in a procedure containing a loop which does not terminate for certain input values. If such an input is tried during the search process, the `w/5` combinator will not terminate.
  Note that no information can be deduced when the process is stopped before the end. It is not possible to determine whether it is a consequence of an infinite loop or just a very long search. In both cases, we say that our technique fails to find a solution of the ATDG problem.

[6] Sometimes called dead code.

5 First Experimental Results

We compare our CLP framework with a random test data generation method and the dynamic approach of [Kor90,FK96]. We implemented the random method by using the `drand48` C function, which generates pseudo-random numbers with the well-known linear congruential algorithm and 48-bit integer arithmetic. TESTGEN is an implementation of the dynamic method for Pascal programs. The tool is not available, hence we base the comparison on the results published in [FK96]. The symbolic execution method has been implemented in a tool called GODZILLA [DO93] but the tool is dedicated to mutation analysis of Fortran programs making the experimental comparison very difficult.

5.1 Our Prototype System

INKA operates on a restricted subset of the C language. Unstructured statements such as *goto* statement are not handled in our framework. Pointer arithmetic, dynamic allocated structures, pointer functions, type casting, involve difficult problems to solve. Pointers are only partially supported by INKA (see section 6). Although, floating point numbers are finite in essence, they introduce problems[7] which cannot be solved within the framework introduced here. All the types of integer variables (char,short, long,...) and almost all the C operators (34 out of 42) are handled (by capturing their behavior into user–defined constraints).

INKA includes a C parser, a SSA form generator and a Constraint system producer over the `clp(fd)` library of Sicstus Prolog.

5.2 Experiments

We only present our experiments on three classical academic programs of the Software Testing Community and one real–world program but INKA has been used successfully on several other programs [Got00]. The academic programs[8] are 1) “bsearch” [DO93] which is a binary search in a sorted array; 2) a program published in [FK96] named “sample” which contains arrays, loops and a lot of dependencies; 3) the famous program “trityp” [DO93] which contains numerous non-feasible paths.

Finally, we introduce the results for a real–world program extracted from an avionic project, named “ardeta03”. This program mainly contains complex C structures and bitwise operations but does not contain loops.

5.3 Test Procedure

For each program, a test datum for each basic block (sequence of statements without branching) is generated. Of course, this approach is not optimal to

[7] The evaluation process of an arithmetical expression in a CLP system and the evaluation of the same expression in the operational software may yield different results.

[8] The source code of these programs are available at `http://www.essi.fr/~rueher/trityp.htm`.

reach a complete block coverage since no coverage information is reused between two generations.

For each selected block, we have compared InKa to the random method, and to the published results of Testgen. We have performed our experiments on a 300Mhz Sun UltraSparc 5. A time-out of 10 seconds per block was set. In 10 seconds, the random method generated approximatively 10^5 test data, while InKa generated only one test datum. To limit the factor of "bad luck" which may occur with the random method, we repeated 10 times the generation with different initial values for the linear congruential algorithm, and we only considered the best results.

[FK96] introduces the results of Testgen on the three academic programs among others. The Testgen technique starts with a random generation of value which determines the success of the method. They performed their experiments on a PC with 60Mhz–Pentium processor. A time–out was set to 5 minutes and the same test procedure as ours was applied, except that they repeated 10 times their search for each block. Their "coverage represents the percentage of nodes for which at least one try was successful in finding input data" [FK96]. According to this definition, they found 100% for each program.

5.4 Results

The results are shown in Fig. 5. The number of lines of code and the number of statement blocks are reported in the first two columns; whereas an estimate of the search space is reported in the third column (number of possible test data). The last three columns contain the results of block coverage obtained with the three different approaches.

Programs	**loc**	**blocks**	**test data**	Testgen*	**Random****	InKa**
bsearch	21	10	$> 10^{50}$	100%	100%	100%
sample	33	14	$> 10^{100}$	100%	93%	100%
trityp	40	22	$> 10^{10}$	100%	86%	100%
ardeta03	157	38	$> 10^{60}$	–	74%	100%

(*) 50 minutes on PC Pentium (60Mhz) for each block
(**) 10 seconds on Sun Sparc 5 (300Mhz) under Solaris 2.5 for each block

Fig. 5. Comparison on block coverage

5.5 Analysis

Testgen did allow 50 minutes per block whereas InKa did not spent more than 10 seconds on each block. The tests with Testgen have been done on a PC with 60Mhz-Pentium processor while InKa was run on 300Mhz Sun UltraSparc 5. If we assume that there is less than a factor 30 between these two computers, InKa is still 10 time faster than Testgen[9].

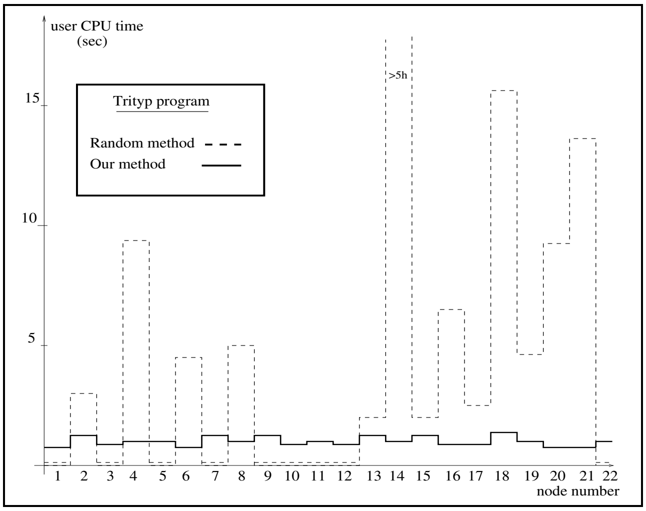

Fig. 6. Time required to generate a solution for each block

Let us see in more details what appends on one of the programs. We report in Fig. 6 the curve of times required to generate a solution for the program "trityp" by the last two methods. First, note that the time required by the random method is smaller on some blocks. In fact, InKa requires a nominal time to generate the constraint system and to solve it, even if it is very easy to solve. Second, note that the random method fails on some blocks. For instance, the block 14 which requires for the random method to generate a sequence of three equal integers. On the contrary, this block does not introduce a particular difficulty for InKa, because such a constraint is easily propagated.

6 Perspective

First experiments are promising but, of course, more experiments have to be performed on non-academic programs to validate the proposed approach. The main extension of our CLP framework concerns the handling of pointer variables. Unlike scalars, pointer variables cannot directly be transformed into logical variables because of the aliasing problem. In fact, an undirect reference and a variable may refer to the same memory location at some program point. In [Got00], we proposed to handle this problem for a restricted class of pointers: pointers to

[9] Note that InKa is written in Prolog while Testgen is written in C.

stack–allocated variables. Our approach, based on a pointer analysis, does not handle dynamically allocated structures. For some classes of applications, this restriction is not important. However, the treatment of all pointer variables is essential to extend our CLP framework to a wide spread of real–world applications.

Acknowledgements

Patrick Taillibert and Serge Varennes gave us invaluable help on the work presented in this paper. Thanks to François Delobel for its comments on an earlier draft of this paper. This research is part of the Systems and Software Tools Support department of THOMSON-CSF DETEXIS.

References

CFR+91. Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficently Computing Static Single Assignment Form and the Control Dependence Graph. *Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

DO93. R. A. Demillo and A. J. Offut. Experimental Results from an Automatic Test Case Generator. *Transactions on Software Engineering Methodology*, 2(2):109–175, 1993.

FK96. Roger Ferguson and Bogdan Korel. The Chaining Approach for Software Test Data Generation". *ACM Transactions on Software Engineering and Methodology*, 5(1):63–86, January 1996.

FOW87. Jeanne Ferrante, Karl J. Ottenstein, and J. David Warren. The Program Dependence Graph and its use in optimization. *Transactions on Programming Languages and Systems*, 9-3:319–349, July 1987.

GBR98. Arnaud Gotlieb, Bernard Botella, and Michel Rueher. Automatic Test Data Generation Using Constraint Solving Techniques. In *Proc. of the Sigsoft International Symposium on Software Testing and Analysis*, Clearwater Beach, Florida, USA, March 2-5 1998. *Software Engineering Notes*,23(2):53-62. disponible at `http://www.essi.fr/~rueher/`.

Got00. A. Gotlieb. *Automatic Test Data Generation using Constraint Logic Programming.* PhD thesis, PHD Dissertation (in French), Université de Nice–Sophia Antipolis, January 2000.

HSD98. Pascal Van Hentenryck, Vijav Saraswat, and Yves Deville. Design, implementation, and evaluation of the constraint language cc(fd). *Journal of Logic Programming*, 37:139–164, 1998. Also in CS-93-02 Brown–University 1993.

Kin76. James C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, July 1976.

Kor90. Bogdan Korel. Automated Software Test Data Generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, august 1990.

Nta98. S. Ntafos. On Random and Partition Testing. In *Proceedings of Sigsoft International Symposium on Software Testing and Analysis*, volume 23(2), pages 42–48, Clearwater Beach, FL, March,2-5 1998. ACM, SIGPLAN Notices on Software Engineering.