Automatic Test Data Generation using Constraint Solving Techniques

Arnaud Gotlieb
Dassault Electronique
55 quai Marcel Dassault
92214 Saint Cloud, France
and also at
Université de Nice - Sophia Antipolis
Arnaud.Gotlieb@dassault-elec.fr

Bernard Botella
Dassault Electronique
55 quai Marcel Dassault
92214 Saint Cloud, France
Bernard.Botella@dassault-elec.fr

Michel Rueher
Université de Nice - Sophia Antipolis
I3S-CNRS Route des colles,
BP 145
06903 Sophia Antipolis, France
rueher@unice.fr

Abstract

Automatic test data generation leads to identify input values on which a selected point in a procedure is executed. This paper introduces a new method for this problem based on constraint solving techniques. First, we statically transform a procedure into a constraint system by using well-known "Static Single Assignment" form and control-dependencies. Second, we solve this system to check whether at least one feasible control flow path going through the selected point exists and to generate test data that correspond to one of these paths.

The key point of our approach is to take advantage of current advances in constraint techniques when solving the generated constraint system. Global constraints are used in a preliminary step to detect some of the non feasible paths. Partial consistency techniques are employed to reduce the domains of possible values of the test data. A prototype implementation has been developped on a restricted subset of the C language. Advantages of our approach are illustrated on a non-trivial example.

Keywords

Automatic test data generation, structural testing, constraint solving techniques, global constraints

1 INTRODUCTION

Structural testing techniques are widely used in unit or module testing process of software. Among the structural criteria, both statement and branch coverages are commonly accepted as minimum requirements. One of the difficulties of the testing process is to generate test data meeting these criteria.
From the procedure structure alone, it is only possible to generate input data. The correctness of the output of the execution has to be checked out by an "oracle".

Two different approaches have been proposed for automatic test data generation in this context. The initial one, called *path-oriented* approach [4, 7, 16, 20, 3], includes two steps which are :

- to identify a set of control flow paths that covers all statements (resp. branches) in the procedure ;
- to generate input test data which execute every selected path.

Among all the selected paths, a non-negligeable amount is generally non-feasible [24], i.e. there is no input data for which such paths can be executed. The static identification of non-feasible paths is an undecidable problem in the general case [1]. Thus, a second approach called *goal-oriented* [19] has been proposed. Its two main steps are :

- to identify a set of statements (resp. branches) the covering of which implies covering the criterion ;
- to generate input test data that execute every selected statement (resp. branch).

Assuming that every statement (resp. branch) is reachable, there is at least one feasible control flow path going through the selected statement (resp. branch). The goal of the data generation process is then to identify input data on which one such path is executed.

For these approaches, existing generation methods are based either on symbolic execution [18, 4, 16, 7, 10], or on the so called "dynamic method" [20, 19, 11, 21].

Symbolic execution consists in replacing input parameters by symbolic values and in statically evaluating the statements along a control flow path. The goal of symbolic execution is to identify the constraints (either equalities or inequalities) called "path conditions" on symbolic input values under which a selected path is executed. This method leads to several problems : the growth of intermediate algebraic expressions, the difficulty to deal with arrays (although some solutions exist [13, 8]), and the aliasing problem for pointer analysis. Using symbolic execution corresponds to an exhaustive exploration of all paths going through a selected point. Of course, this may be unacceptable for programs containing a large number of paths.

Korel proposes in [20] to base the test data generation process on actual executions of programs. Its method is called the "dynamic method". If an undesirable path is observed during the execution flow monitoring, then a function minimization technique is used to "correct" the input variables. [19] presents an extension of the dynamic method to the *goal-oriented approach*. This method is designed to handle arrays, dynamic structures, and procedures calls [21]. However, although the dynamic method takes into account some of the problems encountered with symbolic execution, it may require a great number of executions of the program.

This paper introduces a new method to identify automatically test data on which a selected point in the procedure is executed. The proposed method operates in two steps :

1. The procedure is statically transformed into a constraint system by the use of "Static Single Assignment" (SSA) form [23, 2, 9] and control-dependencies [12]. The result of this step is a set of constraints — called *Kset* — which is formed of :
   - the constraints generated for the whole procedure ;
   - the constraints that are specific to the selected point.
2. The constraint system *Kset* is solved to check whether at least one feasible path which goes through the selected point exists. Finally, test data corresponding to one of these paths are generated.

The key point of this method is to take advantages of current constraint techniques to solve the generated constraint system. In particular, global constraints are used in a preliminary step to detect some of the non-feasible parts of the control structures and partial consistency techniques are employed to reduce the domains of possible values of the test data. Search methods based on the combination of both enumeration and inference processes are used in the final step to identify test data. Furthermore, these techniques offer a flexible way to define and to solve new constraints on values of possible test data.

A prototype implementation of this method has been developped on a restricted subset of the C language.

Outline of the paper : the second section presents the generation of *Kset* while the third section is devoted to the resolution techniques. The fourth section describes the prototype implementation while the fifth section provides a detailed analysis of a non-trivial example that has been successfully treated with our method.

2 GENERATION OF THE CONSTRAINT SYSTEM

Application of our method is limited to a structured subset of a procedural language. Unstructured statements such as "*goto-statement*" are not handled in our framework because they introduce non-controled exits of loops and backward control flow.
Pointer aliasing, dynamic allocated structures, function's pointer involve difficult problems to solve in the frame of a static analysis. In this paper, we assume that programs avoid such constructions. The treatement of basic types such as char and floating point numbers is not presented. A few words in the fourth section are devoted to the extension of our method to these types.

The generation of the constraint system *Kset* is done in three steps :

1. Generation of the "Static Single Assignment" form ;
2. Generation of a set of constraints corresponding to the procedure p, called $pKset(p)$;
3. Generation of a set of constraints corresponding to the control-dependencies of a selected point n, called $cKset(n)$.

Kset is defined as :

$$Kset(p,n) \overset{def}{=} pKset(p) \cup cKset(n)$$

Now, let us introduce some basics used in the rest of the paper.

2.1 Basics

A procedure control flow graph (V, E, e, s) [1] is a connected oriented graph composed by a set of vertices V,

```
int f(int i)
        int j ;
1.      j := 1 ;
2.      while ( i ≠ 0 )
                do
3a                      j := j * i ;
3b                      i := i - 1 ;
                od ;
4.      if ( j = 2 )
5.              then i := 2 ;
        fi ;
6.      return j ;
```

Figure 1: Example 1

a set of edges E and two particular nodes, e the unique entry node, and s the unique exit node. Nodes represent the basic blocks which are sets of statements executed without branching and edges represent the possible branching between basic blocks. For instance, consider the procedure[1] given in figure 1, which is designed to compute the factorial function, and its control flow graph (CFG) shown in figure 2.

A *point* is either a node or an edge in the CFG. A *path* is a sequence $< v_i, \ldots, v_j >$ of consecutive nodes (edge connected) in (V, E, e, s). A *control flow path* is a path $< v_i, \ldots, v_j >$ in the CFG, where $v_i = e$ and $v_j = s$. A path is *feasible* if there exists at least one test datum on which the path is executed, otherwise it is *non-feasible*. For instance, the control flow path $< 1, 2, 4, 5, 6 >$ in the CFG of example 1 is non-feasible.

A node v_1 is *post-dominated* [12] by a node v_2 if every path from v_1 to s in (V, E, e, s) (not including v_1) contains v_2.
A node v_2 is *control-dependent* [12] on v_1 iff 1) there exists a path P from v_1 to v_2 in (V, E, e, s) with any v in $P \setminus \{v_1, v_2\}$ post-dominated by v_2 ; 2) v_1 is not post-dominated by v_2. For example, block 5 is control-dependent on block 4 in the CFG of example 1.

2.2 SSA Form

Most procedural languages allow destructive updating of variables ; this leads to the impossibility to treat a program variable as a logical variable. Initially proposed for the optimization of compilers [2, 23], the "Static Single Assignment" form [9] is a semantically equivalent 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 linear sequence of code is obtained by a simple renaming ($i \longrightarrow i_0$, $i \longrightarrow i_1, \ldots$) of the variables. For the control structures, SSA form introduces special assignments, called ϕ-functions, in the junction nodes of the CFG. A ϕ-function returns one of its arguments depending on the control flow. Consider the *if-statement* of the SSA form of example 1 in figure 3 ; the ϕ-function of statement 6 returns i_3 if the flow goes through the *then-part* of the statement, i_2 otherwise. For some more complex structures, the ϕ-functions are introduced in a special heading of the loop (as in the *while-statement* in figure 3). SSA Form is built by using the algorithm given in [5], which is designed to treat structured programs in one parsing step.
For convenience, a list of ϕ-assignments will be written with a single statement :
$x_2 := \phi(x_1, x_0), \ldots, z_2 := \phi(z_1, z_0) \Longleftrightarrow \vec{v_2} := \phi(\vec{v_1}, \vec{v_0})$

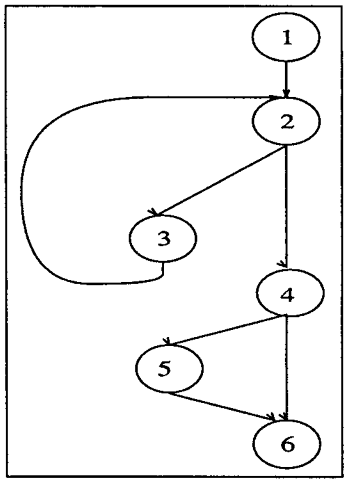

Figure 2: Control flow graph of example 1

```
int f(int i0)
        int j0 ;
1a.     j0 := 1 ;
        /* Heading */
1b.     j2 := φ(j0,j1) ;
1c.     i2 := φ(i0,i1) ;
2.      while (i2 ≠ 0 )
        do
3a.                 j1 := j2 * i2 ;
3b.                 i1 := i2 - 1 ;
        od
4.      if (j2 = 2)
5.                  then i3 := 2 ;
        fi
6.      i4 := φ(i3,i2) ;
        return (j2) ;
```

Figure 3: SSA Form of example 1

[1]For all the examples throughout the paper, a clear abstract syntax is used to indicate that our method is not designed to a particular language

2.3 Generation of *pKset*

pKset(p) is a set of both atomic and global constraints associated with a procedure *p*.

Informally speaking, an atomic constraint is a relation between logical variables. Global constraints are designed to handle more efficiently set of atomic constraints. For instance, global constraint ELEMENT/3 [2] : ELEMENT(k, L, v) constraints the k^{th} argument of the list L to be equal to v.

Let us now present how *pKset* is generated. The method is driven by the syntax. Each subsection, which is devoted to a particular construction, presents the generation technique.

2.3.1 *Declaration*

The variables of a procedure are either input variables or local variables. Parameters and global variables are considered as input variables while the other variables are considered as local. Each variable x which has a basic type declaration, is translated in atomic constraint of the form : $x \in [Min, Max]$ where Min (resp. Max) is the minimum (resp. maximum) value depending on the current implementation. An array declaration is translated into a list of variables of the same type while a record is translated into a list of variables of different types.
A specific variable, named "OUT", is devoted to the output value of the procedure.

2.3.2 *Assignment and Decision*

Elementary statements, such as assignments and expressions in the decisions are transformed into atomic constraints. For instance, the assignment of statement 3a in example 1 generates the constraint $j_1 = j_2 * i_2$. The decision of statement 2 generates $i_2 \neq 0$. A basic block is translated into a conjunction of such constraints. For example, statements 3a and 3b generate $j_1 = j_2 * i_2 \wedge i_1 = i_2 - 1$.

2.3.3 *Conditional Statement*

The conditional statement *if_then_else* is translated into global constraint ITE/3 in the following way :

$$pKset(\textbf{if } d \textbf{ then } s1 \textbf{ else } s2 \textbf{ fi } \vec{v_1} := \phi(\vec{v_2}, \vec{v_3})) = \text{ITE}(pKset(d), pKset(s1) \wedge \vec{v_1} = \vec{v_2}, pKset(s2) \wedge \vec{v_1} = \vec{v_3})$$

[2]where /3 denotes the arity of the constraint

This constraint denotes a relation between the decision and the constraints generated for the *then-* and the *else-*parts of the conditional. Note that ϕ-assignments are translated in simple equality constraints. The operational semantic of the constraint ITE/3 will be made explicit in section 3.2.

2.3.4 *Loop Statement*

The loop statement *while* is also translated in a global constraint W/5. Informally speaking, this constraint states that as long as its first argument is true, the constraints generated for the body (fifth argument) of the *while statement* are true for the required data.

$$\begin{aligned} pKset(\vec{v_2} := \phi(\vec{v_0}, \vec{v_1}) \;\; \textbf{while } d \textbf{ do } s \textbf{ od}) \\ = \text{W}(pKset(d),\ \vec{v_0},\ \vec{v_1},\ \vec{v_2},\ pKset(s)) \end{aligned}$$

The generated constraint requires three vectors of variables $\vec{v_0}, \vec{v_1}, \vec{v_2}$. $\vec{v_0}$ is a vector of variables defined before the *while-statement*. $\vec{v_1}$ is the vector of variables defined inside the body of the loop and $\vec{v_2}$ is the vector of variables referenced inside and outside the *while-statement*. Note here that the ϕ-assignments are only used to identify the vectors of variables.

The operational semantics of the constraint W/5 will also be given in section 3.2.

2.3.5 *Array and Record*

Both arrays and records are treated as list of variables, therefore we only present the generation of *pKset* on arrays.

Reference of an array is provided in the SSA Form by a special expression [9] : *access*. The evaluation of *access(a,k)* statement is the k^{th} element of a noted v.

For the definition of an array, the special expression *update* is used [9]. *update(a,j,w)* evaluates to an array a_1 which has the same size as a and which has the same elements as a, except for j where value is w.

Both expressions *access* and *update* are treated with the constraint ELEMENT/3 :

$$pKset(\text{ v:= } access(a, k)) = \{\text{ELEMENT}(k, a, v)\}$$

$$\begin{aligned} & pKset(a_1 := update(a, j, w)) \\ & = \bigcup_{i \neq j} \{\text{ELEMENT}(i, a, v) \wedge \text{ELEMENT}(i, a_1, v)\} \\ & \quad \cup \{\text{ELEMENT}(j, a_1, w)\} \end{aligned}$$

2.4 Generation of *cKset*

cKset(n) is a set of constraints associated with a point n in the CFG. It represents the necessary conditions under which a selected point is executed. These conditions are precisely the control-dependencies on the selected point. *cKset(n)* is then the set of constraints of the statements and the branches on which n is control-dependent. For example, node 5 is control-dependant on node 4 then : $cKset(5) = \{j_2 = 2\}$

2.5 Example

For the procedure given in figure 1 and the statement 5, the following sets are obtained :

$pKset(f) =$
$\{\ j_0 = 1,$
$\quad \text{W}(i_2 \neq 0, (i_0, j_0), (i_1, j_1), (i_2, j_2),$
$\qquad j_1 = j_2 * i_2 \wedge i_1 = i_2 - 1),$
$\quad \text{ITE}(j_2 = 2\ i_3 = 2 \wedge i_4 = i_3, i_4 = i_2),$
$\quad OUT = j_2\ \}$

$cKset(5) = \{j_2 = 2\}$

$Kset(f, 5) = pKset(f) \cup cKset(5)$

3 SOLVING THE CONSTRAINT SYSTEM AND GENERATION OF TEST DATA

Constraint programming has emerged in the last decade as a new tool to address various classes of combinatorial search problems. Constraint systems are inference systems based on such operations as constraint propagation, consistency and entailment. Inference is based on algorithms which propagate the information given by one constraint to others constraints. These algorithms are usually called partial consistency algorithms because they remove part of inconsistent values from the domain of the variables. Altough these approximation algorithms sometimes decide inconsistency, it is usually necessary to combine the resolution process with a search method. Informally speaking, search methods are intelligent enumeration process.
For a survey on Constraint Solving and Constraint Logic Programming, see [14] and [17].

Let us first introduce some basics notations on constraint programming required in the rest of the paper. These notations are extracted from [15].

A constraint system is consistent if it has at least one solution, i.e. if there exists at least one variable assignment which respects all the constraints. More formally, a set of constraints σ is called a *store* and the store is consistent if :

$$\models (\exists)\sigma$$

where $(\exists)\phi$ denotes the existential closure of the formula ϕ.
Entailment test checks out the implication of a constraint by a store. For example,

$$x \geq 0 \text{ is entailed by } \{x = y^2\}$$

The entailment test of the constraint c by the store σ is noted :

$$\models (\forall)(\sigma \implies c)$$

where $(\forall)\phi$ denotes the universal closure of ϕ.
Both consistency and entailment tests are NP-complete problems in the general case. For this reason, implementations of these tests are based on two approximations : domain-consistency and interval-consistency.

3.1 Local Consistency

Associated with each input variable x_i is both a domain $D_i \in \mathbb{Z}$ and an interval $D_i^* = [min(D_i), max(D_i)]$. A constraint $c(x_1, \ldots, x_n)$ is a n-ary relation between variables $(x_1, \ldots, x_n)$ which dénotes a subset of $\mathbb{Z}^n$.

Domain-consistency also called arc-consistency removes values from the domains and Interval-consistency only reduces the lower and upper bounds on the domains. Both are applied in a subtle combination by the constraint solver. Intuitivelly, when the domains contain a small number of values, domain-consistency is applied. Interval-consistency is applied on large domains. Precise definitions of these local consistencies are now given :

Definition 1. *(domain-consistency) [15]*
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.

Definition 2. *(interval-consistency) [15]*
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

```
int g(int x,int y )
      int z ;
      int t ;
1a.   z := x * y ;
1b.   t := 2 * x ;
2.    if (z ≤ 8)
             then
3a.                    t := t - y ;
3b.                    if (t = 1 ∧ x > 1 )
4.                            then ...
```

Figure 4: Example 2

$D_1^*, \ldots, D_{i-1}^*, D_{i+1}^*, \ldots, D_n^*$ such that $c(v_1, \ldots, v_n)$ holds.

A local treatment is associated to each constraint. The corresponding algorithm is able to check out both domain- and interval- consistencies for this constraint. The inference engine propagates the reductions provided by this algorithm on the other constraints. The propagation iterates until a fixpoint is reached. Informally speaking, a fixpoint is a state of the domains where no more prunnings can be performed.

Let us illustrate how interval-, domain- consistency and the inference engine may reduce the domains of possible values of test data on the example 2 given in figure 4. Consider the problem of automatic test data generation for statement 4.

Parameters are of non-negative integer type. The following set is provided :

$Kset(g, 4) = \{x_0, y_0 \in [0, Max], z_0 = x_0 * y_0,\ t_0 = 2 * x_0,\ z_0 \leq 8,\ t_1 = t_0 - y_0,\ t_1 = 1,\ x_0 > 1\}$

and the following resolution process is performed :

$z_0 = x_0 * y_0$ leads to $z_0 \in [0, Max]$
$t_0 = 2 * x_0$ leads to $t_0 \in [0, Max]$
$z_0 \leq 8$ leads to $z_0 \in [0, 8]$
$t_1 = 1$ leads to $t_1 \in \{1\}$
$x_0 > 1$ leads to $x_0 \in [2, Max]$
$z_0 = x_0 * y_0$ leads to $x_0 \in [2, 8]$ and $y_0 \in [0, 4]$
$t_0 = 2 * x_0$ leads to $t_0 \in [4, 16]$
$t_1 = t_0 - y_0$ leads to $y_0 \in \{3, 4\}$ and $t_0 \in \{4, 5\}$
$t_0 = 2 * x_0$ leads to $x_0 \in \{2\}$ and $t_0 \in \{4\}$
$t_1 = t_0 - y_0$ leads to $y_0 \in \{3\}$

Finally, $(x_0 = 2, y_0 = 3)$ corresponds to the unique test datum on which statement 4 in the program of figure 4 can be executed.

3.2 Global Constraints Definitions

For atomic constraints and some global constraints, the local treatment is directly implemented in the constraint solver. However, for user-defined global constraint, it is necessary to provide the algorithm. The key point of our approach resides in the use of such global constraints to treat the control structures of the program. The global constraints are used to propagate information on inconsistency in a preliminary step of the resolution process.

3.2.1 *Entailment Test Implementation*

The entailment test is used to construct these global constraints. The implementation of entailment test may be done as a proof by refutation. A constraint is proved to be entailed by a store if there is no variable assignment respecting both the store and the negation of the constraint.

The operational semantic of the user-defined global constraints is designed with properties which are "guarded" by entailment tests. Such properties are expressed by constraints added to the store. We have introduced in the section 2.3 two global constraints : ITE/3 and W/5. Let us give now their definitions.

3.2.2 *ITE/3*

Definition 3. (ITE/3)
ITE*(c, $\{c_1 \wedge \ldots \wedge c_p\}$, $\{c'_1 \wedge \ldots \wedge c'_q\}$)*

- *if* $\models (\forall)(\sigma \implies c)$ *then* $\sigma := \sigma \cup \{c_1 \wedge \ldots \wedge c_p\}$
- *if* $\models (\forall)(\sigma \implies \neg c)$ *then* $\sigma := \sigma \cup \{c'_1 \wedge \ldots \wedge c'_q\}$
- *if* $\models (\forall)(\sigma \implies \neg(c_1 \wedge \ldots \wedge c_p))$ *then* $\sigma := \sigma \cup \{\neg c \wedge c'_1 \wedge \ldots \wedge c'_q\}$
- *if* $\models (\forall)(\sigma \implies \neg(c'_1 \wedge \ldots \wedge c'_q))$ *then* $\sigma := \sigma \cup \{c \wedge c_1 \wedge \ldots \wedge c_p\}$

The first two features of this definition express the operational semantic of the control structure *if_then_else*. The last ones are added to identify non-feasible parts formed by one of the two branches of the control structure. Consider for example :

ITE$(i_0 \neq 0, i_1 = i_0 - 1 \wedge i_2 = i_1, i_2 = 1)$

Suppose that the store contains $i_2 = 0$; when applying the fourth feature of the ITE constraint we have to consider the consistency of the following set : $\{i_2 = 0\} \cup \{i_2 = 1\}$ It is inconsistant, meaning that the *else*-part of the statement is non feasible. Then, the constraints $i_0 \neq 0 \wedge i_1 = i_0 - 1 \wedge i_2 = i_1$ are added to the store.

3.2.3 *W/5*

The *while-statement* combines looping and destructive assignments. Hence W/5 behaves as a constraint generation program.

When evaluating W/5, it is necessary to allow the generation of new constraints and new variables. A substitution $subs(\vec{v_1} \leftarrow \vec{v_2}, c)$ is a mechanism which generates a new constraint having the same structure as c but where variables vector $\vec{v_1}$ has been replaced by vector $\vec{v_2}$. The following example illustrates this mechanism : if $\vec{v_1} = (x_1, y_1)$ and $\vec{v_2} = (x_2, y_2)$ then $subs(\vec{v_1} \leftarrow \vec{v_2}, x_1 + y_1 = 3)$ is $(x_2 + y_2 = 3)$

W/5 is now formally defined :

Definition 4. (W/5)
$W(c, \vec{v_0}, \vec{v_1}, \vec{v_2}, c_1 \wedge \ldots \wedge c_p)$

- *if* $\models (\forall)(\sigma \implies subs(\vec{v_2} \leftarrow \vec{v_0}, c))$ *then*
  $\sigma := \sigma \cup \{subs(\vec{v_2} \leftarrow \vec{v_0}, c_1 \wedge \ldots \wedge c_p) \wedge$
  $W(c, \vec{v_1}, \vec{v_3}, \vec{v_2},$
  $subs(\vec{v_1} \leftarrow \vec{v_3}, c_1 \wedge \ldots \wedge c_p))\}$
- *if* $\models (\forall)(\sigma \implies subs(\vec{v_2} \leftarrow \vec{v_0}, \neg c))$ *then*
  $\sigma := \sigma \cup \{\vec{v_2} = \vec{v_0}\}$
- *if* $\models (\forall)(\sigma \implies subs(\vec{v_2} \leftarrow \vec{v_0}, \neg(c_1 \wedge \ldots \wedge c_p)))$ *then*
  $\sigma := \sigma \cup \{subs(\vec{v_2} \leftarrow \vec{v_0}, \neg c) \wedge \vec{v_2} = \vec{v_0}\}$
- *if* $\models (\forall)(\sigma \implies \vec{v_2} \neq \vec{v_0})$ *then*
  $\sigma := \sigma \cup \{subs(\vec{v_2} \leftarrow \vec{v_0}, c) \wedge$
  $subs(\vec{v_2} \leftarrow \vec{v_0}, c_1 \wedge \ldots \wedge c_p) \wedge$
  $W(c, \vec{v_1}, \vec{v_3}, \vec{v_2}, subs(\vec{v_1} \leftarrow \vec{v_3}, c_1 \wedge \ldots \wedge c_p))\}$

The first two features represent the operational semantic of the *while* statement. As for the ITE/3 constraint, the other features identify non-feasible part of the structure. The third one is applied if it can be proved that the constraints of the body of the loop are inconsistent with the current store. This means the body cannot be executed even once, the output vector of variables $\vec{v_2}$ is then equated with the input vector $\vec{v_0}$. In the opposite, if $\vec{v_2} = \vec{v_0}$ is inconsistent in the current store, the fourth feature is applied meaning that the body of the loop is executed at least once.

Let us illustrate the treatment of W/5 on the *while*-statement of example 1 :

Suppose that the store contains $\{j_0 = 1, j_2 = 2\}$; when testing the consistency of

$$W(i_2 \neq 0, (i_0, j_0), (i_1, j_1), (i_2, j_2),$$
$$j_1 = j_2 * i_2 \wedge i_1 = i_2 - 1)$$

the fourth feature is applied twice and then gives the following store :

$$\{j_0 = 1, j_2 = 2, j_2 = j_1 * i_1, i_2 = i_1 - 1,$$
$$j_1 = j_0 * i_0, i_1 = i_0 - 1, i_2 = 0\}$$

Finally, $(i_0 = 2)$ is obtained.

3.3 Complete Resolution of the Example

Consider again the example of figure 1 and the problem of generating a test data on which a feasible path going through statement 5 is executed. The *Kset* provided by the first step of our method is :

$$Kset(f, 5) = pKset(f) \cup cKset(5) =$$
$$\{\ j_0 = 1,$$
$$W(i_2 \neq 0, (i_0, j_0), (i_1, j_1), (i_2, j_2),$$
$$j_1 = j_2 * i_2 \wedge i_1 = i_2 - 1),$$
$$\text{ITE}(j_2 = 2, i_3 = j_2 \wedge i_4 = i_3, i_4 = i_2),$$
$$OUT = j_2\} \cup \{j_2 = 2\}$$

The loop is executed twice, generating the following store :

$$\{j_0 = 1, i_0 \neq 0, i_1 \neq 0, i_2 = 0, i_2 = i_1 - 1, i_1 = i_0 - 1, j_2 = j_1 * i_1, j_1 = j_0 * i_0, j_2 = 2, i_3 = j_2, i_4 = i_3, OUT = j_2\}$$

Interval consistency is applied to solve the system, and yields to $i_0 = 2$. This is the unique test data on which statement 5 may be executed.

3.4 Search Process

Of course, local consistencies are incomplete constraint solving techniques [22]. The store of constraints can be domain-consistent though there is no solution in the domains (i.e. the store is inconsistent). Let us give an example of a classical pitfall of these techniques :
$x, y, z \in \{0, 1\}, \quad \sigma = \{x \neq y, y \neq z\}$
Testing $\models (\forall)(\sigma \implies (x = z))$ fails because the store $\{x \neq y, y \neq z, x \neq z\}$ is domain-consistent.

In order to obtain a solution, it is necessary to enumerate the possible values in the restricted domains [22, 15]. This process is incremental. When a value v is chosen in the domain D_x of the variable x, the constraint $(x = v)$ is added to the store and propagated. This may reduce the domains of the other variables. This process is repeated until either the domain of all variables is reduced to a single value or the domain of some variable becomes empty. In the former case, we obtain a solution of the test data generation problem, whereas in the latter we must backtrack and try another value $(x = w)$ until $D_x = \emptyset$.

In general, there are many test data on which a selected point is executed. As claimed in the Introduction, constraint solving techniques provide a flexible way to choose test data. The search process can be user-directed by adding new constraints on the input variables of the procedure. Our framework provides an elegant way to handle such constraints. These constraints are propagated by the inference engine as soon as they induce a reduction on the domains. Furthermore, these additional constraints may be used to insure that the generated input data are "realistic". They may have one of the two following forms :

- constraints on domains (for example $x_0 \in [-3, 17]$) ;
- constraints between variables (for example $y_0 > x_0$ meaning that a parameter y_0 of a procedure is strictly greater than another one x_0).

It is also possible to guide the search process with some well-known heuristics. For example :

- to select the variable with the smallest domain (first-fail principle) ;
- to select the most constrained variable ;
- to bissect the domains ($x \in [a, b]$ is transformed into $(D_x = [a, a + b/2]$ or $D_x = [a + b/2, b])$).

4 IMPLEMENTATION

INKA, a prototype implementation has been developed on a structured subset of language C. The extension to control structures such as *do-while* and *switch* statement is straightforward. Characters are handled in the same way as integer variables. Floating point numbers do not introduce new difficulties in the constraints generation process, but they require another solver. Although the domains remain finite, it is of course not possible to enumerate all the values of a floating point variable. Resolution of the constraint system is therefore more problematic. References on these solvers can be found in [17]. The extension of our method to pointer variables falls into two classical problems of static analysis : the aliasing problem and the analysis of dynamic allocated structures.

```
int sample(int a[3], int b[3], int target)
    int i, fa, fb, out ;
1a. i := 1 ;
1b. fa := 0 ;
1c. fb := 0 ;
2.  while (i ≤ 3)
        do
3.          if (a[i] = target)
4.              then fa := 1 ;
            fi ;
5.          i := i + 1 ;
        od
6.  if (fa = 1)
        then
7a.         i := 1 ;
7b          fb := 1 ;
8.          while (i ≤ 3)
                do
9.                  if (b[i] ≠ target)
10.                     then fb := 0 ;
                    fi ;
11.                 i := i + 1 ;
                do ;
    fi ;
12. if (fb = 1)
13.     then out := 1 ;
14.     else out := 0 ;
    fi ;
15. return out ;
```

Figure 5: Program SAMPLE

INKA includes 5 modules :

- A C Parser
- A generator of SSA form and control-dependencies
- A generator of *Kset*
- A constraints solver
- A search process module

The constraint solver is provided by the CLP(FD) library of Sicstus Prolog 3.5 [6].

5 EXAMPLE

We present now the results of our method on a non-trivial example adapted from [11] : the SAMPLE program given in figure 5. For the sake of simplicity, it is written in the abstract syntax used in this paper. Size of array have been reduced to 3 for improving the presentation.

Consider the problem of automatic test data generation to reach node 13.

INKA has generated the *Kset(SAMPLE,13)* constraint system. The following set of constraints on domains are added :

$$a[1], a[2], a[3], b[1], b[2], b[3], target \in [1, 9]$$

Table 1 reports only the results of the constraint solver and search process module. Experiments are made on a Sun Sparc 5 workstation under Solaris 2.5

First experiments concern the search of solutions without adding any kind of constraints on input data. The line 1 of the table 1 indicates the time required to obtain the first solution and all solutions of the problem. The exact test data is provided in the former case while the number of solutions is only provided in the later one.

Then, we have considered that the user wants the input data to satisfy the additional constraint :

$$a[3]^2 = a[1]^2 + a[2]^2$$

Second line reports the results of generation when the additional constraint is checked out after the search process and the third line reports the results when the constraint is added to the current store and propagated.

A first fail enumeration heuristic has been used for these experiments. *Test data* are given in vector form $(a[1], a[2], a[3], b[1], b[2], b[3], target)$ and *CPU time* is the time elapsed in the constraint solving phase. Note that a complete enumeration stage would involve to try $9^7 = 4782969$ values.

These experiments are intended to show what we have called the flexible use of constraints. First, the CPU time elapsed in the first and second experiments are approximatively the same to obtain all the solutions. In both cases, the search process has enumerated all the possible values in the reduced domains. The only difference is that, in the second case, the added constraint has been checked out after the enumeration step. This illustrate a generate and test approach. On the contrary, note that the results presented in the third line of table 1 show an important improvement factor due to the use of the additional constraint in the resolution process. In the third case, the additional constraint is used to prune the domains and thus the time elapsed in the search process module is dramatically reduced.

Of course, further experiments are needed to show the effectiveness of our approach and to compare the method with other approaches.

Table 1: Results

First solution	CPU time	All solutions	CPU time
(1,1,1,1,1,1,1)	1.0s	1953 solutions	287s
(3,4,5,3,3,3,3)	53s	(3,4,5,3,3,3,3), (3,4,5,4,4,4,4), (3,4,5,5,5,5,5), (4,3,5,3,3,3,3), (4,3,5,4,4,4,4), (4,3,5,5,5,5,5)	292s
(3,4,5,3,3,3,3)	1.3s	(3,4,5,3,3,3,3), (3,4,5,4,4,4,4), (3,4,5,5,5,5,5), (4,3,5,3,3,3,3), (4,3,5,4,4,4,4), (4,3,5,5,5,5,5)	2.4s

6 CONCLUSION

In this paper, we have presented a new method for the automatic test data generation problem. The key point of this approach is the early detection of some of the non-feasible paths by the global constraints and thus the reduction of the number of trials required for the generation of test data. First experiments on a non-trivial example made with a prototype implementation tend to show the flexibility of our method. Future work will be devoted to the extension of this method to pointer variables and experimentations with floating point numbers ; an experimental validation on real applications is also forseen.

Acknowledgements

Patrick Taillibert and Serge Varennes gave us invaluable help on preliminary ideas to design the global constraints introduced. Thanks to Xavier Moulin for its helpful comments on earlier drafts of this paper.
This work is partially supported by A.N.R.T.
This research is part of the software testing project DEVISOR of Dassault Electronique.

References

[1] Aho, A., Sethi, R., and Ullman, J. *Compilers Principles, techniques and tools.* Addison-Wesley Publishing Company, Inc, 1986.

[2] Alpern, B., Wegman, M. N., and Zadeck, F. K. Detecting Equality of Variables in Programs. In *Proc. of Symposium on Principles of Programming Languages* (New York, January 1988), ACM, pp. 1–11.

[3] Bertolino, A., and Marré, M. Automatic Generation of Path Covers Based on the Control Flow Analysis of Computer Programs. *IEEE Transactions on Software Engineering 20*, 12 (December 1994), 885–899.

[4] Boyer, R., Elspas, B., and Levitt, K. SELECT - A formal system for testing and debugging programs by symbolic execution. *SIGPLAN Notices 10*, 6 (June 1975), 234–245.

[5] Brandis, M. M., and Mössenböck, H. Single-Pass Generation of Static Single-Assignment Form for Structured Languages. *Transactions on Programming Languages and Systems 16*, 6 (November 1994), 1684–1698.

[6] Carlsson, M. *SICStus Prolog User's Manual, Programming over Finite Domains.* Swedish Institute in Computer Science, 1997.

[7] Clarke, L. A System to Generate Test Data and Symbolically Execute Programs. *IEEE Transactions on Software Engineering SE-2*, 3 (September 1976), 215–222.

[8] Coen-Porisini, A., and de Paoli, F. Array Representation in Symbolic Execution. *Computer Languages 18*, 3 (1993), 197–216.

[9] Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. Efficently Computing Static Single Assignment Form and the Control Dependence Graph. *Transactions on Programming Languages and Systems 13*, 4 (October 1991), 451–490.

[10] DeMillo, R. A., and Offut, A. J. Constraint-Based Automatic Test Data Generation. *IEEE Transactions on Software Engineering SE-17*, 9 (September 1991), 900–910.

[11] Ferguson, R., and Korel, B. The Chaining Approach for Software Test Data Generation". *ACM Transactions on Software Engineering and Methodology 5*, 1 (January 1996), 63–86.

[12] Ferrante, J., Ottenstein, K. J., and Warren, J. D. The Program Dependence Graph and its use in optimization. *Transactions on Programming Languages and Systems 9-3* (July 1987), 319–349.

[13] Hamlet, D., Gifford, B., and Nikolik, B. Exploring Dataflow Testing of Arrays. In *Proc. of the International Conference on Software Engineering* (Baltimore, May 1993), IEEE, pp. 118–129.

[14] Hentenryck, P. V., and Saraswat, V. Constraints Programming : Strategic Directions. *Constraints 2*, 1 (1997), 7–34.

[15] Hentenryck, P. V., Saraswat, V., and Deville, Y. Design, implementation, and evaluation of the constraint language cc(fd). In *LNCS 910* (1995), Springer Verlag, pp. 293–316.

[16] Howden, W. Symbolic Testing and the DISSECT Symbolic Evaluation System. *IEEE Transactions on Software Engineering SE-3*, 4 (July 1977), 266–278.

[17] Jaffar, J., and Maher, M. J. Constraint Logic Programming : A Survey. *Journal of Logic Programming 20*, 19 (1994), 503–581.

[18] King, J. C. Symbolic Execution and Program Testing. *Commun. ACM 19*, 7 (July 1976), 385–394.

[19] Korel, B. A Dynamic Approach of Test Data Generation. In *Conference on Software Maintenance* (San Diego, CA, November 1990), IEEE, pp. 311–317.

[20] Korel, B. Automated Software Test Data Generation. *IEEE Transactions on Software Engineering 16*, 8 (august 1990), 870–879.

[21] Korel, B. Automated Test Data Generation for Programs with Procedures. In *Proc. of ISSTA'96* (San Diego, CA, May 1996), vol. 21(3), ACM, SIGPLAN Notices on Software Engineering, pp. 209–215.

[22] Mackworth, A. K. Consistency in Networks of Relations. *Artificial Intelligence 8*, 1 (1977), 99–118.

[23] Rosen, B. K., Wegman, M. N., and Zadeck, F. K. Global Value Numbers and Redundant Computations. In *Proc. of Symposium on Principles of Programming Languages* (New York, January 1988), ACM, pp. 12–27.

[24] Yates, D. F., and Malevris, N. Reducing The Effects Of Infeasible Paths In Branch Testing. In *Proc. of Symposium on Software Testing, Analysis, and Verification (TAV3)* (Key West, Florida, December 1989), vol. 14(8) of *Software Engineering Notes*, pp. 48–54.