Automatic Unit Test Data Generation Using Mixed-Integer Linear Programming and Execution Trees

S. Lapierre[1], E. Merlo[2], G. Savard[2], G. Antoniol[3], R. Fiutem[3], and P. Tonella[3]

[1]*Bell Canada*, Quality Engineering and Research Group,
1050 Beaver Hall, 2nd floor, Montreal, Quebec, H2Z 1S4, Canada
sebastien.lapierre@bell.ca

[2]*GEGI, École Polytechnique*,
C.P. 6079, Succ. Centre Ville. Montréal, Québec, H3C 3A7, Canada
merlo@rgl.polymtl.ca, gilles@crt.umontreal.ca

[3]*IRST-Istituto per la Ricerca Scientifica e Technologica*, I-38050 Povo (Trento), Italy
antoniol@irst.itc.it, fiutem@irst.itc.it, tonella@irst.itc.it

ABSTRACT

This paper presents an approach to automatic unit test data generation for branch coverage using mixed-integer linear programming, execution trees, and symbolic execution. This approach can be useful to both general testing and regression testing after software maintenance and reengineering activities.

Several strategies, including original algorithms, to move towards practical test data generation have been investigated in this paper. Methods include:

- the analysis of minimum path-length partial execution trees for unconstrained arcs, thus increasing the generation performance and reducing the difficulties originated by infeasible paths
- the reduction of the difficulties originated by non-linear path conditions by considering alternative linear paths
- the reduction of the number of test cases, which are needed to achieve the desired coverage, based on the concept of unconstrained arcs in a control flow graph
- the extension of symbolic execution to deal with dynamic memory allocation and deallocation, pointers and pointers to functions

Execution trees are symbolically executed to produce Extended Path Constraints (EPC), which are then partially mapped by an original algorithm into linear problems whose solutions correspond to the test data to be used as input to cover program branches. Partially mapping this problem into a linear optimization problem avoids infeasible and non-linear path problems, if a feasible linear alternate path exists in the same execution tree.

The presented approach has been implemented in C++ and tested on C-language programs on a Pentium/Linux system. Preliminary results are encouraging and show that a high percentage of the program branches can be covered by the test data automatically produced. The approach is flexible to branch selection criteria coming from general testing as well as regression testing.

KEYWORDS

Automatic test data generation, execution tree, mixed-integer linear programming, symbolic execution, extended path constraint (EPC), path infeasibility.

1 INTRODUCTION

Software testing aims at identifying and removing software defects in the tested programs. However, difficulties arise when trying to find input data to be used as test cases. Two main approaches attack this problem and are termed black box and white box testing.

Black box testing is based on the "user point of view", on "functionality testing" in the perspective of the "verification of the correspondence between the results obtained and the expected results" [2]. On the other hand, white box testing relies on the internal structure of the software to be tested and uses this knowledge in order to generate input data to be run as test cases. Usually, white box testing tries to achieve coverage of a given structural criterion (statements, branches, def-use pairs, and so on), with different criterion varying in effectiveness to uncover bugs but also in difficulty to achieve coverage. When branch coverage is sought, coverage figures ranging in the 70%-85% are often considered acceptable because of infeasible paths [18, 25].

Our research concerns the automation of test data generation based on the white box approach and symbolic execution [14, 4, 5]. The presented approach could be used to generate regression test data which is an expensive maintenance process directed at validating modified software [22].

A detailed review of the literature concerning test data generation can be found in [16].

In this paper, we will present our approach to test data generation (section 2). The experimental phase of our research will be presented in section 3 and will afterwards be discussed in section 4. Conclusions and perspectives about future research works are presented in section 5.

2 TEST DATA GENERATION APPROACH

2.1 Execution tree

This notion was first mentioned in [13, 17]. Let $CFG = (V_{CFG},\ E_{CFG})$ be a Control Flow Graph for a given program, where each node in V_{CFG} corresponds to a statement in the analyzed program and each edge in E_{CFG} corresponds to a flow of control between statements. Let $ET = (V_{ET}, E_{ET})$ be an execution tree such that V_{ET} and E_{ET} be the sets of nodes and edges in the tree.

The set of nodes is defined as follows:

$$\begin{gathered} V_{ET} \overset{\text{def}}{=} \{w_p \mid \\ (\exists p =< v_{start}, ..., v_i, ..., v_m >, v_i \in V_{CFG}) \wedge \\ (\forall i, (0 \leq i \leq m \Leftrightarrow 1), < v_i, v_{i+1} > \in E_{CFG})\} \end{gathered} \tag{1}$$

Function $stm : V_{ET} \rightarrow V_{CFG}$ defines the correspondence between a node in the execution tree and its associated node in the program Control Flow Graph (CFG) as follows:

$$\begin{gathered} stm = \{< w_p, v_m > \mid \\ (\exists p =< v_{start}, ..., v_i, ..., v_m >, v_i \in V_{CFG}) \wedge \\ (\forall i, (0 \leq i \leq m \Leftrightarrow 1), < v_i, v_{i+1} > \in E_{CFG})\} \end{gathered} \tag{2}$$

The set of edges in the execution tree is defined as:

$$\begin{gathered} E_{ET} \overset{\text{def}}{=} \{< w_1, w_2 > \mid \\ (< stm(w_1), stm(w_2) > \in E_{CFG}) \wedge \\ (w_1 \in V_{ET})\ \wedge\ (w_2 \in V_{ET})\} \end{gathered} \tag{3}$$

Intuitively, an execution tree is a tree representing multiple partial executions, which contain some common subpaths as a prefix and which differ at a given CFG branch.

Equation (1) defines V_{ET}, the set of nodes of ET, as a set of nodes w_p whose associated node in the CFG can be reached through an execution path from the starting node of the CFG (v_{start}) and ending at CFG node $stm(w_p)$.

Equation (3) states that E_{ET} is composed of edges whose starting and ending nodes have associated nodes in the CFG which form an edge of E_{CFG}

2.2 Paths and partial-execution-tree leaves

Let us define a path in the execution tree as follows:

$$\begin{gathered} path(w_m) =< w_1, ..., w_m > \mid \\ ((w_i \in V_{ET}) \wedge \\ (\forall i, (0 \leq i \leq m - 1), < w_i, w_{i+1} > \in E_{ET})) \end{gathered} \tag{4}$$

In this context, a path can be represented as a series of adjacent nodes in ET. The function

$$max\_loops : V_{ET} \rightarrow N$$

can be defined as follows:

$$max\_loops(w) = \max_{u \in path(w)} \sum_{(z \in path(w))\ \wedge\ (u \neq z)} \delta(stm(u), stm(z)) \tag{5}$$

where

$$\delta(x, y) = \begin{cases} 1 & \text{if x=y} \\ 0 & \text{otherwise} \end{cases}$$

The function $max\_loops(w)$ returns the maximum number of times that any instruction in the CFG has been encountered while traversing a particular path in the execution tree from the root to node w.

Using max_loops, function

$$et\_leaves : V_{CFG} \times N \rightarrow 2^{V_{ET}}$$

can be defined as

$$\begin{aligned} et\_leaves(v, n) = \{w \in V_{ET} \mid \quad & (stm(w) = v) \wedge \\ & (max\_loops(w) = n)\} \end{aligned} \tag{6}$$

$ETL_{v,k} = et\_leaves(v, k)$ represents the set of leaves of the subtree of ET which contains all the partial executions ending in $v \in V_{CFG}$. Furthermore, any instruction belonging to such partial executions is encountered at most k times.

Let

$$et\_nodes : 2^{V_{ET}} \rightarrow 2^{V_{ET}}$$

where $2^{V_{ET}}$ is the power set of V_{ET}, be a function, which associates nodes belonging to a subset $V \subseteq V_{ET}$ to the nodes belonging to the subtree encountered while traversing ET from its root to nodes in V, and which is defined as follows:

$$et\_nodes(V) = \bigcup_{w \in V} \{z \mid z \in path(w)\}$$

$ETN_{v,k} = et\_nodes(et\_leaves(v, k))$ represents the nodes belonging to V_{ET} encountered while traversing ET from its root to all nodes in $ETL_{v,k}$.

2.3 Partial order

A partial order relation can be established on the function *et_leaves* (ETL) as follows:

$$ETL_{v_1,n_1} \preceq ETL_{v_2,n_2} \Leftrightarrow (v_1 = v_2) \wedge (n_1 \leq n_2) \quad (7)$$

where:

$$v_1 \in V_{CFG}, v_2 \in V_{CFG}, n_1 \in N, n_2 \in N.$$

It has to be noted that:

$$(ETL_{v_1,n_1} \preceq ETL_{v_2,n_2}) \not\Rightarrow (ETN_{v_1,n_1} \subseteq ETN_{v_2,n_2})$$

If any interpretation of the partial order is sought, it may be found more in the direction of Scott domains [11] than in that of set theory.

In our opinion, the average length of the paths from the root to a node in $et\_leaves(v,n)$, for some v and some n, grows with n, while the size of the subtree of ET from the root to all nodes in $ETL_{v,n}$ for some v and some n, grows exponentially with n.

Since the difficulties originated by infeasible paths increase with the path length [26], it makes sense to use the partial order induced by $et\_leaves(v,n)$ to prioritize the test data generation process.

An edge condition function

$$ec : E_{CFG} \rightarrow EC\_EXPR$$

is defined as a function taking an edge in the CFG and computing the condition under which control can follow such an edge.

The computation of significant edge condition is performed, in our approach, using symbolic execution techniques. Extensions have been made to deal with C-language constructs such as pointers, function pointers, and so on.

2.4 Test data generation problem

The test data generation problem can be formulated as the satisfiability problem of finding values of the problem variables associated with program inputs which satisfy the following boolean expression:

$$P_{CFG}(v,k) = \bigvee_{w \in et\_leaves(v,k)} pc(w)$$

where $P_{CFG}(v,k)$ represents a logical disjunction between all the path conditions $pc(w)$ associated with nodes in $ETL_{v,k}$. Path conditions can be computed by symbolic execution.

The boolean expression $P_{CFG}(v,k)$ can be expressed as a function $P_{ET}(w,v,k)$ on the execution tree, as follows:

$$P_{CFG}(v,k) \leftrightarrow P_{ET}(w_0,v,k)$$

where w_0 is the root of ET. The function can be recursively re-written as:

$$P_{ET}(w,v,k) =$$

$$\bigvee_{\substack{(u \in ETN_{v,k}) \wedge \\ (<w,u> \in E_{ET})}} \begin{array}{c} (ec(<stm(w), stm(u)>)) \wedge \\ (P_{ET}(u,v,k)) \end{array} \quad (8)$$

Using equations (8) coverage of a program's branches can be obtained by solving $P_{ET}(w_0,v_j,k)$ for every unconstrained arc $< v_i, v_j >$ in the program CFG, while considering increasing iteration limits k, $(0 \leq k \leq K_{max})$, as suggested by the partial order described in equation (7).

The boolean satisfiability problem represented by $P_{ET}(w_0,v,k)$ can be partially converted into a linear problem $LP_{ET}(w_0,v,k)$ defined on a set of linear constraints $LC_{ET}(w_0,v,k)$.

A linear problem LP can be represented as:

$$LP : z_{ip} = max\,\{zx \mid x \in S\} \quad (9)$$

where $S = \{x \mid LC_{ET}(w_o,v,k)\}$ is the solution space based on a set of linear constraints $LC_{ET}(w_o,v,k)$ which have the form $Ax \leq b$.

We assume that $z_{ip} < \infty$, i.e. that the objective function zx has a finite upper bound. For the purposes of this, the objective function to be maximized can be a trivial one which is maximized by the first solution of the linear problem encountered in the solution space S. In the future, this assumption could be refined to improve the quality of the test data generated in terms, for example, of fewer 0-1 variables, shorter execution paths to be tested, and so on. The purpose of this paper has been limited to the study of the existence and the practical feasibility of the mixed-integer linear programming problem associated with unit test data generation.

A linear constraint c_i can also be represented by the triple $c_i = (T_i, r_i, b_i)$, where T_i is a set of terms defined as

$$T_i = \{< a_{i,j}, x_{i,j} > \mid a_{i,j} \in \Re, x_{i,j} \in VarSpace\},$$

$VarSpace$ is the set of possible linear problem variables, $r_i \in \{\leq, \geq\}$ is a relational operator, and $b_i \in \Re$ is a constant.

Projection functions over the three components of a constraints c_i can be defined as

$$\Pi_1(c_i) = T_i,\ \Pi_2(c_i) = r_i,\ \Pi_3(c_i) = b_i.$$

The linear problem consists of finding some values of the linear problem variables associated with program inputs which satisfy all linear constraints in $LC_{ET}(w_0, v, k)$ for a given statement v and some number of iterations k and which maximize the objective function zx.

The set $LC_{ET}(w_0, v, k)$ can be further decomposed into two subsets of linear constraints:

$$LC_{ET}(w_0, v, k) = LC'_{ET}(w_0, v, k) \cup LC''_{ET}(w_0, v, k)$$

which are inductively defined as follows:

$$LC'_{ET}(w, v, k) =$$

$$\left\{ \bigcup_{\substack{(c \in LC'_{ET}(u,v,k)) \wedge \\ (u \in ETN_{v,k}) \wedge \\ (< w, u > \in E_{ET})}} \begin{array}{c} (\{\Pi_1(c) \cup \\ \{< m_c, var(w,u) >\}\}, \\ \Pi_2(c), \\ \Pi_3(c)) \end{array} \right\} \cup$$

$$\cup \left\{ \bigcup_{\substack{(u \in ETN_{v,k}) \wedge \\ (< w, u > \in E_{ET})}} lp(ec(< stm(w), stm(u) >)) \right\} \quad (10)$$

$$LC''_{ET}(w, v, k) =$$

$$\left\{ \left(\left\{ \bigcup_{\substack{(u \in ETN_{v,k}) \wedge \\ (< w, u > \in E_{ET})}} < 1, var(w,u) > \right\}, \leq, 1 \right) \right\} \cup$$

$$\cup \left\{ \bigcup_{\substack{(u \in ETN_{v,k}) \wedge \\ (< w, u > \in E_{ET})}} LC''_{ET}(u, v, k) \right\} \quad (11)$$

where $var(w, u)$ is a new 0-1 integer variable introduced by our approach, which is defined in $VarSpace$, i.e., the space of the linear problem variables, and which can take only the values 0 or 1. The following holds:

$$(\forall x \mid \exists w, \exists u, x = var(w, u)) \Leftrightarrow x \in \{0, 1\}.$$

m_c is a constant value large enough to make the constraint c relevant or irrelevant to the solution of the linear problem, depending on the value 0 or 1 of $var(u, v)$. The value of m_c is:

$$m_c = \left\{ \begin{array}{ll} \text{-MAX\_REAL} & \text{if } \pi_2(c) = '\leq' \\ \text{MAX\_REAL} & \text{if } \pi_2(c) = '\geq' \end{array} \right. \quad (12)$$

where MAX_REAL represents a proper large enough real number.

To partially map an edge condition into a set of linear constraints, function:

$$lp : EC\_EXPR \rightarrow LC$$

has been defined. Not all edge conditions can be converted into a set of linear constraints. If the conversion is not possible (and this depends only on the conditions written by programmers, since equations (10) and (11) are linear), function lp returns the empty set. Details about the partial conversion from arbitrary boolean expressions, which may include arbitrarily placed function calls, into a set of linear constraints can be found in [16].

The advantage of having reduced the boolean expression problem to a linear one is that we can take advantage of linear optimization techniques [19] to solve this problem in an efficient way. On the other hand, not all boolean satisfiability problems can in general be converted into a linear one. Approaches to generate test data in the presence of non-linear expressions can be found in [15].

A purpose of this paper is to study the program branch coverage practically attainable using such a problem reduction approximation.

Equation (10) takes into account the possible existence of alternative paths reaching node v in the CFG. To generate test data, it is sufficient to satisfy one of the alternative paths only.

Equation (11) specifies that alternative execution paths are mutually exclusive for test data generation, in agreement with the semantics of execution of a sequential program.

The solution process fails only if:

1. All paths between $w_0 \in V_{ET}$ and nodes in $et\_leaves(v, k)$ are non-linear, or

2. all linear paths between $w_0 \in V_{ET}$ and nodes in $et\_leaves(v,k)$ are infeasible

These failure conditions have rarely happened in the experiments described in section 3.2.

Figure 1 shows the code for program *wc* which has been taken from [10] and which will be used in the following to illustrate our approach.

```
   void main(void)
   {
     int inword, nl, nw, nc, c ;

1 :  inword = 0 ;
2 :  nl = 0 ;
3 :  nw = 0 ;
4 :  nc = 0 ;
5 :  c = getc(stdin) ;
6 :  while(c != EOF)
     {
7 :    nc = nc + 1 ;
8 :    if (c == '\n')
9 :      nl = nl + 1 ;
10:    if (c=='\n' || c=='\t' || c==' ')
11:      inword = 0 ;
12:    else if (inword == 0) {
13:      inword = 1 ;
14:      nw = nw + 1 ;
      }
15:    c = getc(stdin) ;
     }
16:  printf("value of nl : %d",nl) ;
17:  printf("value of nw : %d",nw) ;
18:  printf("value of nc : %d",nc) ;
   }
```

Figure 1: *Code for wc program*

In Fig. 2, the partial execution tree for program *wc* is shown. The tree has been computed to reach edge $< 12, 15 >$ while considering $k = 2$ iterations and it corresponds to the set of nodes $ETN'_{<12,15>,2}$.

Using program *wc*, $PBCC'_{ET}(1, < 12, 15 >, 1)$, after basic logical simplifications, is the boolean expression presented in figure 3. Note that, for this particular example, $PBCC'_{ET}(1, < 12, 15 >, 0)$ was automatically identified as being unsolvable by our analyzer, which then tried to analyze $PBCC'_{ET}(1, < 12, 15 >, 1)$.

The linear problem produced by our system is reported in figure 4.

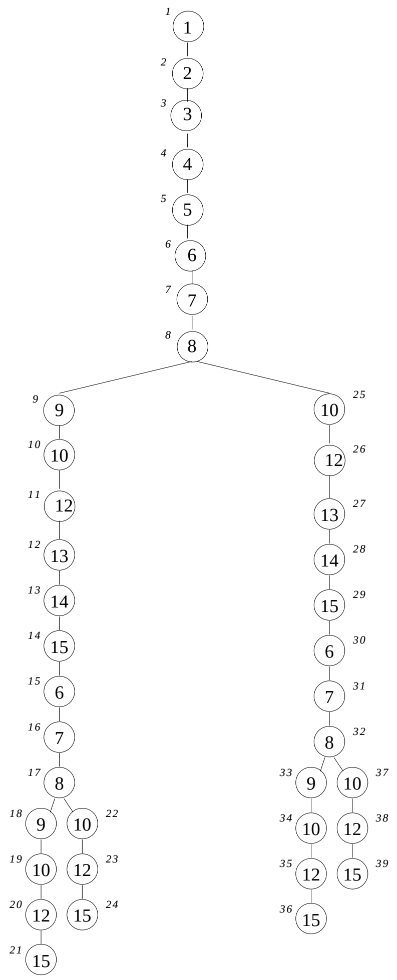

Figure 2: *Execution tree to reach CFG edge $< 12, 15 >$ in wc program for k=2 iterations, where the number appearing in each node represents stm for that node and the distinct number appearing right beside each node represents its distinct label in the tree.*

3 ANALYSIS TOOLS, EXPERIMENTAL SETUP, AND RESULTS

3.1 Tools

The presented approach has been implemented into a tool called *TAO* [1] developed at École Polytechnique de Montréal. *TAO* enables automatic test data generation by implementing the concepts presented in section 2. It has been coded in approximately 20 Kloc, in C++, and compiled under g++. Figure 5 represents the overall architecture of the system.

3.2 Experimental setup and results

Here are presented preliminary results that will be discussed in the following section.

[1]*TAO* is a French acronym which stands for *Tests Assistés par Ordinateur*, which means Computer-Assisted Testing.

(AND
(AND
($X_2 \neq EOF$) ($X_1 \neq' \backslash t'$) ($X_1 \neq' \ \ '$) ($X_1 \neq' \backslash n'$)
($X_1 \neq EOF$))
(OR
(AND
($X_2 \neq' \backslash n'$) ($X_2 \neq' \backslash t'$) ($X_2 \neq' \ \ '$) ($X_1 =' \backslash n'$))
(AND
($X_2 \neq' \backslash n'$) ($X_2 \neq' \backslash t'$) ($X_2 \neq' \ \ '$)))))

Figure 3: *Partially simplified logical condition given by* $PBCC'_{ET}(1, <12, 15>, 1)$ *for wc example, with input data being represented by* X_1 *and* X_2*. Note that special characters '\n', '\t', ' ' and EOF are converted to their numerical ASCII equivalent before proceeding with the test data generation process.*

A total of 10 C-language programs containing 39 units have been analyzed. Some of these programs were selected because they were mentioned in the literature (*wc* [10], *patternmatcher* [9] and *trityp* [8]) and others because they contained precise characteristics that we were interested in analyzing, such as :

- symbolic indexing
- pointers, pointer arithmetic
- function pointers

One program (*taotest*) belongs to the test set for our system. General characteristics of these programs are shown in table 1, namely their size in lines of code, their complexity according to McCabe's cyclomatic complexity metric [2], and the presence of precise characteristics, such as the presence of pointers, function pointers or symbolic indexing within some units of each program.

Program	LOC	McCabe metric	ptrs.	func. ptrs.	symb. indexing
1. wc	39	6	-	-	-
2. patternmatcher	77	8	X	-	-
3. prog6	68	6	-	-	-
4. taotest	239	74	X	X	-
5. minpath	194	64	X	-	X
6. quicksort	95	20	X	-	X
7. histogramme	45	6	-	-	X
8. what	130	3	X	-	-
9. trityp	62	16	-	-	-
10. exptree	389	190	X	-	X
average	134	39.3	-	-	-

Table 1: *Programs analyzed by TAO.*

The programs were first parsed and converted into an intermediate control flow graph language by CANTO [1]. Programs were then partitioned into functional units in order to proceed with unit testing. The unconstrained

```
Minimize
        z1

Subject To
c1:  X2 + M z1 <= EOF -1 + M
c2:  X2 + M z1 >= EOF + 1
c3:  X1 + M z2 <= '\t' -1 + M
c4:  X1 + M z2 >= '\t' + 1
c5:  X1 + M z3 <= ' ' -1 + M
c6:  X1 + M z3 >= ' ' + 1
c7:  X1 + M z4 <= '\n' -1 + M
c8:  X1 + M z4 >= '\n' + 1
c9:  X1 + M z5 <= EOF -1 + M
c10: X1 + M z5 >= EOF + 1
c11: X2 + M z8 - M z6 <= '\n' -1 + M
c12: X2 + M z8 + M z6 >= '\n' + 1
c13: X2 + M z9 - M z6 <= '\t' -1 + M
c14: X2 + M z9 + M z6 >= '\t' + 1
c15: X2 + M z10 - M z6 <= ' ' -1 + M
c16: X2 + M z10 + M z6 >= ' ' + 1
c17: X1 + - M z7 <= '\n'
c18: X1 + + M z7 >= '\n'
c19: X2 + M z11 - M z7 <= '\n' -1 + M
c20: X2 + M z11 + M z7 >= '\n' + 1
c21: X2 + M z12 - M z7 <= '\t' -1 + M
c22: X2 + M z12 + M z7 >= '\t' + 1
c23: X2 + M z13 - M z7 <= ' ' -1 + M
c24: X2 + M z13 + M z7 >= ' ' + 1

Bounds
-1 <= X1 <= 255
-1 <= X2 <= 255
0 <= z1..z13 <= 1
End
```

Figure 4: *Linear problem associated to problem in figure 3.*

edges were afterwards generated and used as goals for our path selector module. As a first result, the CFGs of the units analyzed contained a total of 684 edges. Computation of the unconstrained edges identified 124 edges (as reported in table 2) that had to be covered in order to ensure complete unit branch coverage. This shows a very significant reduction of 82% in the number of edges that had to be considered by test data generation while still potentially attaining full unit branch coverage.

The path selector module computed the execution trees that were to be analyzed through the symbolic executor. Given UE, the overall set of unconstrained edges in all unit CFGs, and k, a number of iterations, we can define

$$N_{ET}(k) = \frac{\sum_{e \in UE} card(ETN_{e,k})}{card(UE)}$$

$N_{ET}(k)$ is the average number of nodes in the execution trees of all the unconstrained edges, for all units, and is

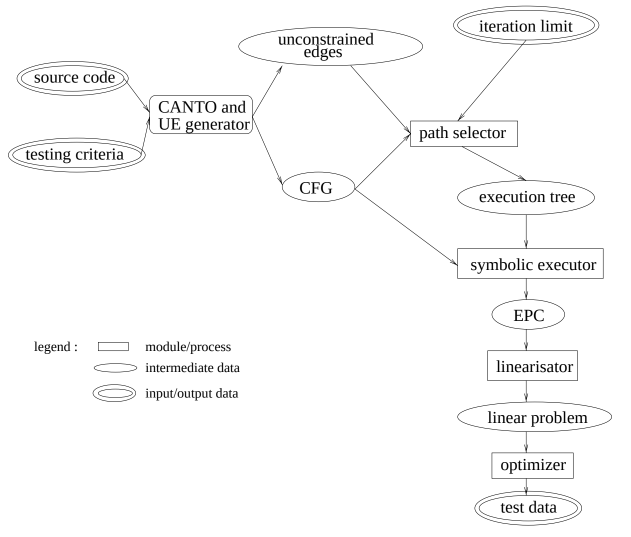

Figure 5: *TAO system architecture*

given for a fixed number of iterations k. Table 2 gives $N_{ET}(k)$ for the 10 programs, considering k=0, 1 and 2.

TAO's symbolic executor was given the task of generating EPCs for the produced execution trees. In the context of unit testing, stubs were created for the tested unit to accomplish its requested calls. The bodies of these stubs were composed of simple operators identifying those parameters being affected by modifications within that call, thus simulating the effects of the real function over the received parameters.

The symbolic executor performed very well in presence of pointers and function pointers, both at intra or inter procedural levels. In program *taotest*, dynamic circular double-linked lists were automatically generated as test data.

Symbolic indexing occurred within units in 4 different programs (see table 1). Indexes were taken from a previously randomly created input file and turned out to satisfy the EPC at the first attempt.

The produced expressions were simplified through a simplification module in order to reduce their size and to identify some easily detectable contradictions. As a result, our simplificator was able to conduct a reduction of 42% in size of the processed expressions.

The simplified EPCs were converted into linear problems and submitted to our linear optimizer. Characteristics about these linear problems are given in table 2, where is stated, for each program and number of iterations, N_v, the average number of 0-1 variables needed in the linear problems (generated by equation (11)) for all the unconstrained edges of the units of that program and for a given number of iterations, and N_c, which corresponds to:

$$N_c(v) = \frac{\sum_{e \in UE} card(LP_{CFG}(e,k))}{card(UE)}$$

and is the average number of linear constraints generated in the linear problems for all the unconstrained edges and for all units of that program and for a given number of iterations.

Problems with fixed number of iterations k were analyzed separately, starting from problems obtained with k=0 and extending to k=1 and 2. Table 2 shows the percentage ($\%_c$) of covered unconstrained edges for each program, considering all its units, and for each number of iterations k analyzed, with k=0, 1 and 2. Note that $\%_c$, for a given k, is given as an added coverage over the coverage already achieved with lower values of k.

As final results, table 3 shows the unconstrained edges coverage achieved for all analyzed programs and the total time needed to achieve such coverage. To clarify the result presentation, UE coverage and time results were added for all units of each program. Time performances given represent CPU time needed to complete the process described above, from abstract syntax tree representation to final test data generation. Experiments were conducted on a PentiumPro 166Mhz processor with 64Mb RAM and running the Linux operating system.

Program	UE coverage achieved for all units (%)	Time required (sec.)
1. wc	100	1.15
2. patternmatcher	100	1.48
3. prog6	100	0.68
4. taotest	100	11.23
5. minpath	94	14.97
6. quicksort	100	2.70
7. histogramme	100	0.78
8. what	100	1.29
9. trityp	94.7	12.67
10. exptree	100	36.5

Table 3: *Total UE coverage and associated time performances for unit test data generation for the 10 analyzed programs.*

An interesting result is also given in figure 6 : it shows the total test-case generation time (in seconds) as a function of the coverage attained (in percentage). The 10 programs analyzed are presented on the same figure and are clustered as three separate groups for which we have have given the range of corresponding McCabe's cyclomatic complexity metric.

4 DISCUSSION

In [26], it is reported that a path's infeasibility tends to increase exponentially as a function of it's length. In order to decrease path infeasibility problems, our search strategy makes use of the implications of uncon-

Program	N_u	N_{UE}	0 iteration				1 iterations				2 iterations				% left
			N_{ET}	N_c	N_v	$\%_c$	N_{ET}	N_c	N_v	$\%_c$	N_{ET}	N_c	N_v	$\%_c$	uncov.
1. wc	1	5	10.8	19.8	5.0	80	50.2	87.0	30.0	20	199.0	234.0	82.4	-	-
2. patternmatcher	2	7	8.9	9.9	2.9	100	23.9	29.0	9.4	-	49.6	61.9	21.1	-	-
3. prog6	1	4	9.5	13.5	4.0	100	39.8	98.8	34.5	-	190.0	505.8	182.0	-	-
4. taotest	11	20	7.3	9.1	1.8	100	17.9	24.4	4.8	-	38.5	49.2	10.2	-	-
5. minpath	5	17	11.4	10.1	1.0	70.5	103.4	45.6	9.4	23.5	1011.4	189.8	47.8	-	6
6. quicksort	4	8	5.6	3.6	0.1	87.5	36.8	3.6	0.1	12.5	183.5	2.6	0.1	-	-
7. histogramme	1	3	16.7	7.5	1.3	66.7	40.3	18.7	5.0	33.3	92.0	34.7	10.0	-	-
8. what	1	5	6.8	22.0	4.6	100	37.0	185.8	59.4	-	197.4	1050.4	360.4	-	-
9. trityp	1	19	155.3	27.3	6.2	94.7	155.3	27.3	6.2	-	155.3	27.3	6.2	-	5.3
10. exptree	12	36	9.8	9.5	2.1	94.4	93.7	64.3	16.0	5.6	743.9	410.4	128.8	-	-
average	3.9	12.4	31.7	13.0	2.7	91.1	77.5	50.0	13.0	7.3	423.6	226.7	71.7	-	1.6

Table 2: *Number of units, number of unconstrained edges, average values of the number of nodes in the execution tree, of the number of linear constraints needed to represent the problems, of the 0-1 variables generated and of the percentage of solved cases for coverage of unconstrained edges, and added percentage of unsolved cases, considering 0,1 and 2 number of iterations, for the 10 analyzed programs.*

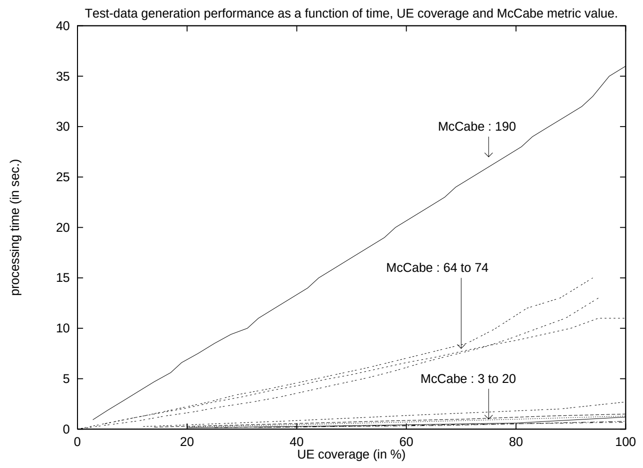

Figure 6: *Time performances as a function of coverage achieved for the 10 programs, with McCabe's cyclomatic complexity metric value given for program groups.*

strained edges in such a way that the search conducted in the unit CFG is stopped when the goal unconstrained edge is reached, thus limiting the lengths of the selected paths. To our knowledge, no other test data generation system made use of the implications of a path's length to conduct their path selection strategy. In fact, path selection has mostly been aimed at reducing the number of analyzed paths (as in [23]), an approach in which path infeasibility is not considered. The approach we implemented, which is based on [3], has the double advantage of significantly minimizing the number of paths to be considered (reduction of 82 % in our experiments) and of limiting path lengths, therefore reducing difficulties associated with path infeasibility.

The use of execution trees is another major factor in decreasing problems related to path infeasibility. In [24], it has been proven that determining the feasibility of a single program path is an undecidable problem. The use of execution trees helps us to get around this problem by potentially providing alternative feasible paths which will lead to a solution, thus overriding undecidability problems linked with any single path within the tree.

Execution trees play a similar role in helping us avoiding problems related to non-linear set of constraints. Using an equation solver module based on linear programming, our system would not be able to solve any single non-linear set of constraints. But as mentioned earlier, use of execution trees provides us with alternatives that potentially help us avoid non-linear problems. Furthermore, it is our opinion that non-linear problems can be mostly avoided by using execution trees; inability to solve an execution tree due solely to non-linear problems can only be caused if each path within that tree involves a non-linear problem. In our opinion, the probability for an execution tree to contain exclusively non-linear paths is very low, as supported by research data on frequency of non-linear problems in computer programs [6, 7].

The preliminary experiments we conducted show a very high degree of unconstrained edges coverage. In fact, only 1.6% of all the unconstrained edges were not covered by our analysis, which represents 2 unconstrained edges out of 124. In one case (*trityp*), complete coverage was not possible due to a clearly infeasible path produced by contradicting sequencing of assignments and tests on internal variables. In the other case (*minpath*), the not-covered unconstrained edge was part of a loop that had to be performed 5 times in order to become valid. Our analysis was limited to the treatment of 0,1 and 2 iterations and we therefore could not automatically solve the above mentioned problem.

In both cases, however, human interaction was quite easily provided and the identification of the cause of the inability to find a solution was quite straightforward. In our opinion, full automation of test data generation is unattainable, but human interaction can be limited to a very few cases.

Human involvement is also made easier in the context of unit testing where a mental model of the unit can be more easily created then for a bigger system. Moreover, as mentioned in [20], unit testing often is the most productive phase to find faults in software and is essential in lessening total testing costs. Unit testing therefore seems a very convenient approach to testing which is reinforced by the fact that the size of execution trees grows exponentially as a function of the size of their corresponding CFG. While the whole system may grow, often the average size of units tends to remain small. Unit testing enables us to limit the size of the CFG's to be analyzed and is therefore a straightforward way of limiting the explosion of the execution trees. However, a drawback of unit testing in the context of symbolic execution is that, with each unit being analyzed separately, more programs variables are left symbolic (e.g. function parameters and function call arguments passed to stubs). Cases of symbolic indexing are therefore more likely to appear then in integration testing, where function parameters and global variables have been assigned a specific value.

With regard to our specific symbolic indexing solution by user interaction, we are fully aware that the performance of our analysis is directly dependent upon human interaction. A large number of trials could be necessary in order to find conclusive values to be used as exact indexes. As a fact, in our experiments, all cases of symbolic indexing were solved at the first attempt to find exact indexes. Although we are aware that this situation is not necessarily representative of most symbolic indexing occurrences, we find this result to be promising and will continue experiments with the hope of gathering more data about symbolic indexing frequency and complexity.

Another exponential growth problem is related to the number of iterations analyzed by our path selector. Data (see table 2) shows averages in the number of nodes in the execution tree to pass from 31.7 to 77.5 to 423.6 for 0,1 and 2 iterations. We can also observe an exponential growth in the number of linear constraints and of 0-1 variables needed to represent the problem (see table 2), since they are related to the size of the execution tree. With present experimental data, however, we think that the impacts of problems related to iterative constructs can be maintained low and that human interaction can be limited to the few cases where automation tools prove inadequate. Furthermore, as average unit size tends to decrease in modern programming languages (Object-Oriented languages for example), we expect a corresponding decrease in execution tree sizes and therefore in related exponential growth. Results obtained (see table 2) show very high coverage even when considering only 0, 1 and 2 number of iterations. In fact, for the analyzed programs, the coverage of unconstrained edges was achieved within 0 and 1 iterations and no additional coverage was provided when the analysis was extended to deal with 2 iterations. Consequently, in our opinion, any number of iterations larger than 1 is likely to contribute with lower and lower additional coverage.

An interesting intuitive result of our experiment is that a program's complexity directly influences its difficulty to be tested. As seen in figure 6, higher McCabe cyclomatic metric values influence the time needed to achieve coverage.

As a final point of discussion, we can see from table 3 that time performances needed to achieve unit coverage in the 94%-100% range are very good. These results, even though they are preliminary and have not been applied to large programs, are very encouraging. Being based on unit testing, it is our opinion that our system would easily scale to process bigger systems composed of a larger number of units.

5 CONCLUSIONS

We have presented an approach for automatic test data generation based on the concepts of mixed-integer linear programming, execution trees, and symbolic execution. Our approach improves practicality of test data generation by reducing the number of test data needed to achieve branch coverage and by substantially reducing the effects of infeasible paths and of non-linear set of constraints.

Experiments have been conducted on C-language programs and show a very high degree of unit branch coverage with test data automatically generated by our system.

Future research work is necessary to assess the effectiveness of the presented test data generation approach on larger systems; also the frequency and the complexity of symbolic indexing should be investigated to better understand the problem and to be in better position for finding a practical solution; the integration of data-flow testing criterion to conduct test data generation [12, 21] could also be studied in the presented context; the exploration of alternative techniques for processing non-linear equations should be addressed, and the links with dynamic approaches when symbolic execution loses efficiency [15] should be evaluated.

ACKNOWLEDGEMENTS

We would like to thank Charles Leduc and Christian Meunier, from École Polytechnique of Montréal, who helped in building TAO's expression simplifier and unconstrained edges analyzer.

REFERENCES

[1] G. Antoniol, R. Fiutem, G. Lutteri, P.Tonella S. Zanfei and E. Merlo. "Program Understanding and Maintenance with the CANTO Environment", To appear In *Proceedings of the International Conference on Software Maintenance*, Sept 28-30 Oct 1-3, Bari, 1996.

[2] B. Beizer. "Software testing techniques, second edition", International Thomson computer press, 1990.

[3] A. Bertolino and M. Marré. "Automatic generation of path covers based on the control flow analysis of computer programs", *IEEE Trans. Soft. Eng.*, dec. 1994, pp.885–898.

[4] A. Cimitile, A. De Lucia and M. Munro. "Qualifying reusable functions using symbolic execution", *IEEE proc. Second Working Conf. on Reverse Eng.*, july 1995, pp.178–187.

[5] A. Cimitile, A. De Lucia and M. Munro. "Identifying reusable functions using specification driven program slicing: a case study", *IEEE proc. Int. Conf. on Soft. Maintenance"*, oct. 1995, pp.124–133.

[6] L. A. Clarke. "A system to generate test data and symbolically execute programs", *IEEE Trans. Soft. Eng.*, sep. 1976, pp.215–222.

[7] P. D. Coward. "Symbolic execution and testing", *Information and soft. technology*, 1991, pp.53–64.

[8] R. A. DeMillo and A. J. Offutt. "Constraint-based automatic test data generation", *IEEE Trans. Soft. Eng.*, sep. 1991, pp.900–910.

[9] P. G. Frankl and E. J. Weyuker. "An applicable family of data flow testing criteria", *IEEE Trans. Soft. Eng.*, oct. 1988, pp.1483–1498.

[10] K. B. Gallagher and J. R. Lyle, "Using program slicing in software maintenance", *IEEE Trans. Soft. Eng.*, 17(8), 1991, pp.751–761.

[11] C. A. Gunter and D. S. Scott. "Semantic Domains", pp. 635–677, from "Handbook of theoretical computer science", Elsevier Science Publishers, 1990.

[12] R. Gupta and M. L. Soffa. "Priority based data flow testing", *IEEE ICSM*, 1995, pp.348–357.

[13] R. Jasper, M. Brennan, K. Williamson and B. Currier. "Test data generation and feasible path analysis", *Int. symp. on soft. testing and analysis*, 1994, pp.95–107.

[14] J. C. King. "Symbolic execution and program testing", *ACM Programming Languages*, july 1976, pp.385–394.

[15] B. Korel. "Automated software test data generation", *IEEE Trans. Soft. Eng.*, aug. 1990, pp.870–879.

[16] S. Lapierre, E. Merlo, G. Savard, G. Antoniol, R. Fiutem and P. Tonella. "Description of *TAO* test data generation system", *École Polytechnique technical report*, aug. 1997.

[17] T. E. Lindquist and J. R. Jenkins. "Test-case generation with IOGen", *IEEE Soft.*, jan. 1988, pp.72–79.

[18] Y. K. Malaiya. "Antirandom testing : getting the most out of black-box testing", *ISSRE'95 (Sixth int. symp. on soft. reliability engineering)*, 1995, pp.86–95.

[19] G. L. Nemhauser and L. A. Wolsey. "Integer and combinatorial optimization", Wiley-Interscience series in discrete mathematics and optimization, 1988.

[20] A. J. Offut and S. D. Lee. "An empirical evaluation of weak mutation", *IEEE Trans. Soft. Eng.*, may 1994, pp.337–344.

[21] S. Rapps and E. Weyuker. "Data flow analysis techniques for test data selection", *IEEE proc. of ICSE-6*, 1982, pp.272–278.

[22] G. Rothermel and M. J. Harrold. "Analyzing regression test selection techniques", *IEEE Trans. Soft. Eng.*, aug. 1996, pp.529–551.

[23] H. S. Wang, S. R. Hsu and J. C. Lin. "A generalized optimal path-selection model for structural program testing", *Journal of Systems and Soft.*, 1989, pp.55–63.

[24] E. J. Weyuker. "The applicability of program schema results to programs", *Int. J. Comput. Inf. Sci.*, 1979.

[25] A. von Mayrhauser. "Software testing : opportunity and nightmare", *IEEE Int. Test Conf.*, 1992, pp.551–552.

[26] D. F. Yates and N. Malevris. "Reducing the effects of infeasible paths in branch testing", *ACM SIGSOFT Soft. Eng. Notes*, 1989, pp.48–54.