Automated Metamorphic Testing

Arnaud Gotlieb
IRISA / INRIA
35042 Rennes Cedex, France
Arnaud.Gotlieb@irisa.fr

Bernard Botella
THALES Airborne Systems
78851 Elancourt Cedex, France
Bernard.Botella@fr.thalesgroup.com

Abstract

Usual techniques for automatic test data generation are based on the assumption that a complete oracle will be available during the testing process. However, there are programs for which this assumption is unreasonable. Recently, Chen et al. [3, 4] proposed to overcome this obstacle by using known relations over the input data and their unknown expected outputs to seek a subclass of faults inside the program. In this paper, we introduce an automatic testing framework able to check these so-called metamorphic relations. The framework makes use of Constraint Logic Programming techniques to find test data that violate a given metamorphic–relation. Circumstances where it can also prove that the program satisfies this relation are presented. The first experimental results we got with a prototype tool build on the top of the test data generator INKA*, show that this methodogy can be completely automated.*

1. Introduction

Most of the current program testing techniques require again a huge amount of manual work. On the one side, structural testing techniques are based on complex program analysis to select test data that execute some program elements. On the other side, fault-based testing techniques that aim at demonstrating that a given class of faults is absent within the program is mainly an intellectual process. Structural testing and fault-based testing techniques have in common to focus on the generation of input values and propositions have been made to automate this generation. They fall into two main categories :

- Static methods rely on program analysis to derive a constraint system targeted to sensitize a given element or to uncover a given fault. Techniques for extracting the constraint system are based either on symbolic execution [12, 7] or on static single assignment form [9] ;
- Dynamic methods are based on actual executions of the program to select the test data. Random test data generation [8] blindly tries values in the input domain until a given element is sensitized, while dynamic test data generation [13] uses heuristics to select values.

Both static and dynamic methods work under the assumption that a complete oracle, e.g. a procedure able to predict every expected output, will be available when checking the outcome. Unfortunately, there are numerous situations where this assumption is unreasonable. As pointed out by Weyuker [17], some programs are considered as being non-testable. To name a few, consider programs written to provide an answer to an unsolved problem, or programs for which correct answers are too difficult to compute by hand.

Metamorphic Testing. Recently, Chen et al. introduced in [3, 4] a new software testing paradigm called Metamorphic Testing (MT). The idea behind MT is to use existing relations over the input data and the computed output values, to test the program. As a consequence, there is no need of an oracle. Formally speaking, let $\{I_1, .., I_n\}_{n>1}$ be n distinct test data for a program intended to compute a function f and suppose that given a relation r over $\{I_1, .., I_n\}$, the results $f(I_1), .., f(I_n)$ must satisfy a property r_f, then we have : $r(I_1, .., I_n) \implies r_f(f(I_1), .., f(I_n))$. It is clear that every correct implementation p of f must satisfy :

$$r(I_1, .., I_n) \implies r_f(p(I_1), .., p(I_n))$$

This property is called a metamorphic–relation and it is only a necessary condition for the correctness of p w.r.t. f. For example, consider a program p intended to compute the greatest common divisor of two integers and the following test datum $(1309, 693)$. Although we all know how to compute the gcd[1], it is not so easy to predict the expected value of $p(1309, 693)$ without the help of a calculator. Fortunately, gcd satisfies lots of metamorphic–relations and among them :

$$\begin{cases} I_1 = (u, v) \\ I_2 = (v, u) \end{cases} \implies p(I_2) = p(I_1) \quad \forall u, \forall v \in N^*$$

[1] with the Euclidian algortihm

Proceedings of the 27th Annual International Computer Software and Applications Conference (COMPSAC'03)
0730-3157/03 $ 17.00 © 2003 IEEE

So, MT requires to select $(693, 1309)$ as next test datum to check the given metamorphic–relation. If $p(1309, 693) \neq p(693, 1309)$ then the testing process will succeed in discovering a fault in p without the help of an oracle. Chen et al. proposed in [4] to combine MT with the theory of fault-based testing [16]. In [5], the use of global symbolic evaluation has been proposed to prove that an implementation satisfies a given metamorphic–relation for all inputs. The technique leads to enumerate the paths within the program and to evaluate the statements along each path by replacing variables by symbolic values. Iterations are treated by a complex and manual loop analysis. The procedure requires to compare several sets of constraints extracted from the program. In [5], the technique is illustrated on two example programs for which it is proved (by hand) that given metamorphic–relations are satisfied.

Automated Metamorphic Testing. In this paper, we introduce a software testing framework, named Automated Metamorphic Testing (AMT), able to check automatically metamorphic–relations. Given the source code of a program written in a non-trivial subset of C and a metamorphic–relation, AMT tries to find a test datum which violates the relation. The framework is an extension of the one we proposed for generating automatically test data for structural testing [9, 10]. The underlying method is mainly based on the translation of the code into an "equivalent" constraint logic program over finite domains (a clp(fd) program). A fault-based model of the metamorphic–relation is converted into a goal to solve with the rules of the clp(fd) program. Solving the resulting constraint system relies on constraint propagation, constraint entailment and an instanciation procedure, which are usual techniques used in the Constraint Logic Programming scheme [11, 14]. When found, a solution of the constraint system can be converted into a set of test data that reveals a fault in the program. Conversely, a contradiction of the constraint system shows that the metamorphic–relation is satisfied in every cases. However, the solving process may fail to get a solution in a given amount of time leading to the failure of our approach. Assuming that programs are run on machines with arbitrarily large amounts of storage, it is well known that no general automatic testing procedure can be used to prove program correctness. We implemented AMT on the top of the test data generator INKA [1] and we used it to reveal faults for a set of academic programs. First experimental results show that the MT paradigm can be completely automated and it is of particular interest for "non-testable" programs.

Outline of the paper. Section 2 illustrates our approach on a motivating example. In Section 3, the background required to fully understand the paper is given. Section 4 details the principle of AMT while Section 5 reports the experimental results obtained with our prototype implementation. Section 6 indicates several perspectives.

2. A motivating example

```
typedef unsigned short ush;
ush a[100];

ush p(ush a[], ush e) {
    ush l, h, mid, ret ;
    l = 0 ;  h = 99 ;  ret = 0 ;
    while (h > l)  {
        mid = (l + h)/2 ;
        if (e == a[mid])
            ret = 1 ;
        if (e > a[mid])
            l = mid + 1 ;
        else
            h = mid − 1;  }
    mid = (l + h)/2 ;
    if (ret != 1 && e == a[mid]/2 )
        ret = 1 ;
    return ret}
```

Figure 1. Example program

Consider the program p of Fig.1 and suppose that the precise function of p is unknown. It is just known that a is a sorted array of small integers and p is expected to implement a function which is invariant to any increasing function of its arguments. This can be converted into the following metamorphic–relation :

$$\begin{cases} I_1 = (a, e) \\ I_2 = (a', e') \end{cases} \implies p(I_2) = p(I_1)$$

where a' and e' are defined as $(a'[i] = f(a[i])_{\forall i \in \{0,..,99\}}$ and $e' = f(e)$ for any increasing function f over N^*. We select arbitrarily the function which computes the square although other functions may also be selected.

AMT works as follows. First, p is automatically translated into a clp(fd) program. Let $clause_p([A_0, A_1, .., A_{99}, E], [O])$:- .. be the head of the clause[2] generated, where the free logical variables $A_0, .., A_{99}, E, O$ represent the input and output values of p. The way this clause is generated has extensively been described [9, 10]. Second, a fault-based model of the metamorphic–relation is converted into a goal to solve :

$? - A_0 \leq A_1 \leq .. \leq A_{99}$, % array a is sorted
$A'_0 = (A_0)^2, A'_1 = (A_1)^2, ..., A'_{99} = (A_{99})^2, E' = E^2$,
$clause_p([A_0, A_1, .., A_{99}, E], [O])$,
$clause_p([A'_0, A'_1, .., A'_{99}, E'], [O'])$,
$O \neq O'$, `labelling`$([A_0, .., A_{99}, E])$.

[2] for the sake of clarity, the concrete syntax of Prolog is used to illustrate our approach

Third, the goal is solved by using a clp(fd) interpreter with some labelling strategy. This leads here to the following valuation that satisfy all the constraints :
$(A_0 = A_1 = .. = A_{99} = 9,\ E = 4,\ O = 1)$,
$(A'_0 = A'_1 = .. = A'_{99} = 81, E' = 16, O' = 0)$
Hence, we get a couple of test data $((a[0] = .. = a[99] = 9, e = 4)$ and $(a[0] = .. = a[99] = 81, e = 16))$ which reveals a fault in p. These test data can then be used to locate the fault by using a dynamic debugger, for example. The keypoint is that the fault is automatically detected without the help of an oracle.

In fact, the program p is expected to implement a binary search into a sorted array but a fault has been inserted in the last conditional test which would have been :
if $((ret$!= $1)$ && $(e == A[mid]))$.

When fixing the fault and restricting the range of variables, AMT proves that the goal doesn't have any solution. Hence, it determines that this program satisfies the metamorphic–relation for every input values in the given ranges, which is a first step toward program correctness.

3. Background

3.1. A subset of the C language

AMT works on a restricted subset of the C language (see Fig.2). The method studies out the unit level testing only, so we assume that programs are made of a single structured procedure. Pointer variables, function's pointers and dynamic allocation statements are outside the scope of this paper, as they introduce specific problems which have to be discussed [10]. Floating point variables are not currently handled although extensions of this framework for such variables have been proposed [15]. The generation of

pgm ::=	*decl proc*	
qual ::=	**signed** | **unsigned**	
type ::=	*qual* **int** | *qual* **short** | *qual* **long**	
decl ::=	ϵ | *type lexp* | *decl ; decl*	
proc ::=	ϵ | *type* id(*decl*){ *decl stmt* **return** (*exp*) }	
stmt ::=	ϵ | *lexp* = *exp*	
	| **if** (*exp*) { *stmt* } **else** { *stmt* }	
	| **while** (*exp*) { *stmt* }	
	| *stmt ; stmt*	
exp ::=	constant | id | id[*exp*]	
	| *exp* **op_ari** *exp*	**op_ari** in $\{+, -, *, \backslash\}$
	| *exp* **op_rel** *exp*	**op_rel** in $\{==, !=, >, ...\}$
lexp ::=	id | id[*exp*]	

Figure 2. A subset of the C language

the clp(fd) program is based on the Static Single Assignment form [6], which allows to translate each statement into a constraint. The generation of the SSA form and the clp(fd) program are now described in more details. Principles of constraint solving over finite domains are also recalled.

3.2. Static Single Assignment form

Initially introduced to ease the computation of dependencies in optimizing compilers, the SSA form is an equivalent version of a procedure on which every variable posses a unique definition. The variables in a SSA basic block are re-

```
ush p(ush a[], ush e) {
    ush l, h, mid, ret ;
    l_1 = 0 ;  h_1 = 99 ;  ret_1 = 0 ;

    /* Heading - while */
    l_4 = φ(l_1, l_3);  h_4 = φ(h_1, h_3);
    mid_3 = φ(mid_1, mid_2);  ret_4 = φ(ret_1, ret_3);
    while (h_4 > l_4) {
        mid_2 = (l_4 + h_4)/2 ;
        if (e == access(a, mid_2))
            ret_2 = 1 ;
        ret_3 = φ(ret_2, ret_4);

        if (e > access(a, mid_2))
            l_2 = mid_2 + 1 ;
        else
            h_2 = mid_2 − 1 ;
        l_3 = φ(l_2, l_4);  h_3 = φ(h_2, h_4); }

    mid_4 = (l_4 + h_4)/2 ;
    if ((ret_4 != 1) && (e == access(a, mid_3)/2) )
        ret_5 = 1 ;
    ret_6 = φ(ret_5, ret_4);
    return ret_6}
```

Figure 3. SSA form of the example program

named ($i = i+1; j = j*i$ leads to $i_2 = i_1+1; j_2 = j_1*i_2$). For the control structures, SSA introduces ϕ-functions, to merge several definitions of the same variable. For example, the SSA form of the example program is given in Fig.3. For convenience, we will write a list of ϕ-functions with a single statement :
$x_2 = \phi(x_1, x_0), .., z_2 = \phi(z_1, z_0) \iff \vec{v_2} = \phi(\vec{v_1}, \vec{v_0})$.
SSA provides special expressions to handle arrays : $access(a, k)$ which evaluates to the k^{th} element of a, and $update(a_0, j, v)$ 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 v.

3.3. Constraint Logic Programming over finite domains

Following the definitions of [14], a clp(fd) program is a set of clauses of the form $A : -B$ where A is a user-defined constraint and B is a goal. A goal is a sequence of either primitive constraints, or user-defined constraint. Primitive constraints are built with variables, domains, arithmetical operators in $\{ +, -, \times, \backslash \}$ and relations $\{ >, \geq, =, \neq, \leq, < \}$. Variables of the clp(fd) program (called fd_variables) take their values into a non-empty finite set of integers.

Combinators are language constructs (primitive– or user-defined– constraint) expressing a high-level relation between other constraints. For example, the combinator $\texttt{element}(I, L, V)$ holds if V is the I^{th} element in the list L of fd_variables.

When considered for solving, a goal leads to build dynamically a *constraint system*. Informally speaking, the solving process of a constraint system is based on a constraint propagation mechanism which makes use of the constraints to prune iteratively the domains of the fd_variables until a fixpoint is reached, on a constraint entailment mechanism which tries to infer new constraints from existing ones with the help of guarded–constraints (noted $C_1 \ -\!\!\rightarrow C_2$), and on a labelling procedure which tries to give a value to a fd_variable one by one and propagates throughout the constraint system.

3.4. Automatic generation of a clp(fd) program

The idea behind AMT consists in translating the imperative program into a clp(fd) program. As this process has already been described [10], we just recall here its main principle. A single clause is generated for the program p. The two arguments[3] of this clause are :
- a list of fd_variables for the formal parameters and the referenced globals (the inputs of p) ;
- a list of fd_variables for the globals defined and the return expression (the outputs of p).

Each statement under SSA form is inductively translated into a primitive–constraint or a combinator. The type declarations are converted into domain constraints. For example, a signed short declaration of x is converted into : $x \in -2^{31}..2^{31}-1$. Assignment statements and decisions are translated with the help of aritmetical and relational operators. *access* and *update* expressions are transformed in `element/3` combinators:
• access(a, k) is translated into $\texttt{element}(K, A, Tmp)$ where Tmp is a newly created temporary variable
• $a_1 = \textit{update}(A_0, j, w)$ is translated into
$\bigwedge_{i \neq j}(\texttt{element}(I, A_0, V) \wedge \texttt{element}(I, A_1, V))$
$\wedge\ (\texttt{element}(J, A_1, W))$

Conditional statement The conditional statement is treated with a user–defined combinator `ite/6`, for which arguments are composed of the variables that appear in the ϕ-functions and the constraints generated from the then– and the else– parts of the statement. Note that other combinators may be nested into the arguments of `ite/6`. An SSA **if_else** statement : **if**(*exp*){*stmt*}**else**{*stmt*} $\vec{v_2} = \phi(\vec{v_1}, \vec{v_0})$ is converted into $\texttt{ite}(c, \vec{v_0}, \vec{v_1}, \vec{v_2}, C_{Then}, C_{Else})$ where c is a primitive–constraint generated by the analysis of *exp* and C_{Then} and C_{Else} are sets of constraints generated for the two branches. The combinator `ite/6` is defined as :

[3] In fact, the clause had two more arguments which are irrelevant here

Definition 1 `ite/6`
$\texttt{ite}(c, \vec{v_0}, \vec{v_1}, \vec{v_2}, C_{Then}, C_{Else}) : -$
$c \ -\!\!\rightarrow C_{Then} \wedge \vec{v_2} = \vec{v_1}$
$\neg c \ -\!\!\rightarrow C_{Else} \wedge \vec{v_2} = \vec{v_0}$
$\neg(c \wedge C_{Then} \wedge \vec{v_2} = \vec{v_1}) \longrightarrow \neg c \wedge C_{Else} \wedge \vec{v_2} = \vec{v_0}$
$\neg(\neg c \wedge C_{Else} \wedge \vec{v_2} = \vec{v_0}) \ -\!\!\rightarrow c \wedge C_{Then} \wedge \vec{v_2} = \vec{v_1}$

Iterative statement. An SSA **while** statement $\vec{v_2} = \phi(\vec{v_0}, \vec{v_1})$ **while** (*exp*) { *stmt* } is treated with the combinator $\texttt{w}(c, \vec{v_0}, \vec{v_1}, \vec{v_2}, C_{Body})$. When evaluating `w/5`, it is necessary to allow the generation of new constraints and new variables. This is done with the help of a substitution mechanism. `w/5` is defined as[4] :

Definition 2 `w/5`
$\texttt{w}(c, \vec{v_0}, \vec{v_1}, \vec{v_2}, C_{Body}) : -$
$c \ -\!\!\rightarrow (C_{Body} \wedge \texttt{w}(c, \vec{v_1}, \vec{v_3}, \vec{v_2}, C_{Body}))$
$\neg c \ -\!\!\rightarrow \vec{v_2} = \vec{v_0}$
$\neg(c \wedge C_{Body}) \ -\!\!\rightarrow (\neg c \wedge \vec{v_2} = \vec{v_0})$
$\neg(\neg c \wedge \vec{v_0} = \vec{v_2}) \ -\!\!\rightarrow (c \wedge C_{Body} \wedge \texttt{w}(c, \vec{v_1}, \vec{v_3}, \vec{v_2}, C_{Body}))$

Note that the vector $\vec{v_3}$ is a vector of fresh variables. By construction, boths combinators `ite/6` and `w/5` model faithfully the operational semantic of the conditional and iteration statements within an imperative program [10].

For example, the clp(fd) program of the example program is given in Fig.4.

4. The AMT framework

4.1. Metamorphic–relations (μ–rel)

Let us recall that a metamorphic–relation is of the form [4] : $r(I_1, .., I_n) \implies r_f(p(I_1), ..., p(I_n))$ where $n > 1$ and I_i is a tuple of input values. These tuples can be either real input values or symbolic ones. Although the use of properties to check the correctness of the program outputs is not new [17], the originality behind the MT paradigm resides in the generality of the required properties. As pointed out by Chen et al., they are not limited to identity relations. As example, consider the following metamorphic–relation that has to be satisfied by any implementation of gcd :

$$\begin{cases} I_1 = (u, v) \\ I_2 = (k_1 u, k_2 v) \end{cases} \implies p(I_2) \geq p(I_1) \ \ \forall k_1, \forall k_2 \in N^*$$

Of course, lots of programs satisfy a given metamorphic–relation, including incorrect programs intended to compute f. So, it is clear that any incorrect implementation of f will not be necessarily discovered by the Metamorphic Testing paradigm.

In our framework AMT, r and r_f are restricted to be n_ary relations build with the arithmetic operators $\{+, -, *, \backslash\}$, the relational operators $\{=, \neq, >, >=, <, =<, \wedge, \vee, \neg\}$, integer constants and symbolic input values.

[4] For the sake of clarity, the constraint c generated throw the substitution mechanism isn't distinguished from c itself

```
p([A0, A1, ..., A99, E], [R]) :-
   A0, A1, .., A99, E ∈ 0..2^32 − 1,
   L1 = 0, H1 = 99, Ret1 = 0
   w(H4 > L4,
      [L1, H1, Mid1, Ret1],
      [L3, H3, Mid2, Ret3],
      [L4, H4, Mid3, Ret4],
      Mid2 = (L4 + H4)/2,
      element(Mid2, [A0, A1, ..., A99], Tmp1),
      ite(E = Tmp1, [Ret2], [Ret4], [Ret3],
         Ret2 = 1, true)

      element(Mid2, [A0, A1, ..., A99], Tmp2),
      ite(E > Tmp2, [L2, H2], [L4, H4], [L3, H3],
         L2 = Mid2 + 1,
         H2 = Mid2 − 1))

   Mid4 = (L4 + H4)/2,
   element(Mid3, [A0, A1, ..., A99], Tmp3),
   Tmp4 = (Ret4 ≠ 1 ∧ E = Tmp3 \ 2),
   ite(Tmp4 = 1, [Ret5], [Ret4], [Ret6],
      Ret5 = 1, true)
   R = Ret6.
```

Figure 4. Clp(fd) prog. automatic. generated

Hence, the possible shapes of metamorphic–relations are limited to numeric expressions over integers[5].

4.2. A fault-model

Let $clause_p(I, O)$ denotes the clp(fd) program generated for p where I is a tuple of logical variables for the inputs and O is a tuple for the outputs. A fault-model is given by the following constraint : $r(I_1, ..., I_n) \wedge \neg r_f(O_1, ..., O_n)$ which express the contradiction of the metamorphic–relation. Note that $\neg r_f(O_1, ..., O_n)$ can be expressed in the language of metamorphic–relations, because it is closed under negation. In AMT, this constraint is converted into a goal to solve as follows :
$? - \ r(I_1, .., I_n) \wedge \neg r_f(O_1, .., O_n)$,
$clause_p(I_1, O_1), .., clause_p(I_n, O_n)$, `labelling`$(I_1)$.
Only I_1 is required to be valuated because $I_2, .., I_n$ are composed of the same input values as I_1.

4.3. Computing a solution

A clp(fd) interpreter is used to compute the first solution to this goal. If found, a solution is a valuation of $(I_1, .., I_n)$ that satisfies r, $\neg r_f$ and the relations that exist between the input and the output of p. Hence, such a solution can be converted into a set of test data that reveal a fault in p. Computing a single solution is enough to discover the fault although all the solutions may be found upon backtracking. In AMT, only the inputs are labelled because the other variables are either initialized or computed with the input values.

Note that there is no way to identify among the test data $(I_1, .., I_n)$, the ones that lead to incorrect outputs for p.

[5] this current restriction will be removed with the next version of INKA which will handle floating-point variables

4.4. Limits of our approach

There are cases where the solving process cannot succeed in a reasonable amount of time. First, solving a non-linear constraint system over finite domains is a NP_complete problem [14]. Second, the `w/5` combinator, which models faithfully the semantic of the **while** statement relies on the termination problem. Whenever a non-terminating loop exists in p for several input values, the `w/5` combinator might not terminate too. Hence, AMT might fail to find a solution because the combinator is entering an infinite loop. We view this limitation as a consequence of the halting problem.

A practical solution for both problems is to set a time–out during the solving process. However it is important to note that no information can be deduced when it arises because one cannot differentiate an infinite loop from a too long search.

4.5. Proving necessary properties for program correctness

When the constraint system is shown to be contradictory then there doesn't exist any solution satisfying the goal. Because the constraint system is made of symbolic input, this demonstrates that the metamorphic–relation is satisfied by every possible test data.

Under the strong hypothesis that p terminates on every test datum of its finite input domain, AMT is guaranteed either to find a solution if there exists one or to demonstrate the absence of solutions, because this can be determined in finite time. However, note that the search space may have to be fully explored in the worst case.

5. Experimental results

5.1. Our prototype implementation

We implemented AMT on the top of the automatic test data generator INKA [1]. This tool computes test data that insure a high-level coverage of the criterium *all_decisions*. INKA operates on a subset of the C and C++ languages. It includes a parser, a SSA form generator for structured programs and a clp(fd) program generator. The core of INKA is written in Java and Prolog. AMT makes use of the primitives of INKA to generate the clp(fd) program and it uses the clp(fd) library of Sicstus Prolog [2] to write and solve the goals associated with the given metamorphic–relation.

5.2. First experiments with AMT

The goal was to study the capacity of AMT 1) to reveal faults in programs and 2) to prove that given metamorphic–relations are satisfied by programs. The experiment was performed on classical academic programs, where faults were injected by mutation.

Programs. Three programs were selected : `bsearch`

(given in Fig.1), `GetMid` [5] intended to compute the median of three integers, and the well–known triangle classification program `trityp` [7], which takes three integers as arguments and classifies the corresponding triangle as scalene (output = 1), isocele (2), equilateral (3) or illegal (4). To limit the size of the search space[6], we consider that every integer variable belongs to a range of 100 values.

Mutants. Four mutants were created for `bsearch` and `GetMid`. The mutant `bsearch_1` contains an infinite loop for some input data because of the mutation of a relational operator, whereas `bsearch_2` is given in Fig.1. `GetMid_1` contains a "missing path error" (removal of two statements), which is considered as a difficult fault to reveal. It has been studied in [5]. `GetMid_2` contains an infeasible path due to the mutation of an operator. Finally, thirty three mutants were manually created for `trityp`. The strategy used to create the mutants was to exchange operators, values or variables in a systematic manner. Equivalents mutants have been removed from the set of experiments, because they cannot be revealed by the mean of testing [7]. All the mutants are available at the url `www.irisa.fr/lande/gotlieb`

Metamorphic–relations. For the `bsearch` program, we used the metamorphic–relation given in section 2 :

$$\begin{cases} I_1 = (a, e) \\ I_2 = (a', e') \end{cases} \implies p(I_2) = p(I_1) \qquad (\mu\text{–rel1})$$

where a' satisfies $(a'[i] = f(a[i])_{\forall i \in \{0,..,99\}}$ and $e' = f(e)$ for any increasing function f over N^*. We selected the function $f : x \longrightarrow x^2$ as increasing function to illustrate the capacity of AMT to handle non-linear functions. However, other functions lead to similar results.

The results of `GetMid` and `trityp` must be invariant to every permutation of their three input values :

$$I_2 = \pi(I_1) \implies p(I_1) = p(I_2) \quad \forall \pi \in S_3$$

where S_3 is the Group that contains the 6 possible permutations over 3 elements. As explained in [5], it is only necessary to check the permutations $\tau_1 = \begin{pmatrix} 1 & 2 & 3 \\ 1 & 3 & 2 \end{pmatrix}$ and $\tau_2 = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 1 & 3 \end{pmatrix}$ to check all the elements of S_3. Hence, the selected metamorphic–relation μ–rel2 is :

$$I_2 = \tau_1(I_1) \wedge I_3 = \tau_2(I_1) \implies p(I_1) = p(I_2) = p(I_3)$$

Note that the fault–model of this metamorphic–relation requires to set three clause invocations. Note also that the negation of $p(I_1) = p(I_2) = p(I_3)$ can be expressed with the help of a user-defined combinator, called $notallequal/1$, which prunes efficiently the search space [14]. Another metamorphic–relation is selected for `trityp` based on the fact that homothetic triangles are of the same classification :

$$\begin{cases} I_1 = (u, v, w) \\ I_2 = (2u, 2v, 2w) \end{cases} \implies p(I_2) = p(I_1) \quad (\mu\text{–rel3})$$

[6] Its size is d^n in the worst case where d is the greatest number of values in the domains of the n fd_variables

5.3. Experimental results and Analysis

All the computations were performed on a 1.8Ghz Pentium 4 personal computer and a time-out (called to) has been set to 600 sec.

Revealling faults with AMT. We first applied AMT to reveal faults among the mutants of each program, leading to the results given in Tab.1, Tab.2 and Tab.3. If found, a couple of test data that violate the metamorphic–relation is shown (the mutant is said to be killed in this case). For each relation, the CPU time spent to find the solution is given.

Table 1. Results for `bsearch` (to = 600sec)

Mutants	(I,..,L)–O denotes a real test datum with (I,..,L) as inputs and O as the computed output	rtime (sec)
	μ–rel1 with $x \longrightarrow x^2$	
`bsearch_1`	not found	to
`bsearch_2`	with up : (0,..,0,2,1)–1, (0,..,0,4,1)–0	0,4
`bsearch_2`	with down : (9,..,9,4)–1, (81,..,81,16)–0	5,7

Table 2. Results for `GetMid`

Mutants	Test data	rtime (sec)
	μ–rel2	
`GetMid_1`	(1,1,0)-0, (1,0,1)-1	0.4
`GetMid_2`	(0,0,1)-1, (0,1,0)-0	0.1
`GetMid` correct version		516,8

As expected, `bsearch_1` is not killed in the given amount of time (600sec). It is likely due to the non-terminating loop introduced into the program. Depending on the labelling strategy ("up" stands for an increasing trail within the domains, while "down" stands for a decreasing one), a solution is given for `bsearch_2` in a very short period of CPU time. Note that an exhaustive test of `bsearch`[7] is impossible in a reasonable amount of time.

Test data are found for killing the two mutants of `GetMid`. This illustrates the capacity of AMT to reveal two difficult class of faults (missing path error and infeasible path) on this program and confirm some of the results given in [5]. Among the thirty three mutants of `trityp`, seven were not detected as faulty versions (programs `trityp_2`,`_9`,`_13`,`_14`,`_15`,`_18`,`_30`) by any of the two metamorphic–relations. Studying these programs leads to see that they are equivalent to the correct version of `trityp` in the following sense : any couple of test data satisfying the metamorphic–relation yield the same incorrect results for both the mutant and the correct version. For example, the mutant `trityp_9` cannot be detected with μ–rel2 because the fault has been introduced into a statement only reached by a sequence of three equal integers, which is invariant to permutation. However a permutation-based relation such as μ–rel2, reveal lots of faults for `trityp` (23 mutants killed over 33). Furthermore, μ–rel2 and μ–rel3 are not killing

[7] trying each of the 100^{100} possible test datum

Table 3. Results for `trityp` (to = 600sec)

Mut.	Test data	rtime	Test data	rtime
	μ–rel2		μ–rel3	
_1	(1,2,2)–4,(2,1,2)–4	0,2	not found	to
_2	not found	to	not found	to
_3	(1,2,1)–2,(2,1,1)–4	0,2	not found	to
_4	(1,1,2)–1,(1,2,1)–2	0,1	not found	to
_5	(1,1,2)–4,(1,2,1)–2	0,1	not found	to
_6	(1,1,2)–4,(1,2,1)–2	0,1	not found	to
_7	(1,1,2)–4,(1,2,1)–2	0,1	not found	to
_8	(2,1,2)–2,(2,2,1)–3	18,4	not found	to
_9	not found	to	not found	to
_10	(1,2,1)–4,(2,1,1)–3	0,2	not found	to
_11	not found	to	not found	to
_12	(2,1,2)–2,(2,2,1)–4	16	not found	to
_13	not found	to	not found	to
_14	not found	to	not found	to
_15	not found	to	not found	to
_16	not found	to	(2,3,4)–4,(4,6,8)–1	24,1
_17	(1,2,3)–4,(1,3,2)–1	0,2	not found	to
_18	not found	to	not found	to
_19	(2,1,2)–2,(2,2,1)–2	36,9	not found	to
_20	(1,3,2)–4,(3,1,2)–1	0,2	not found	to
_21	(1,2,3)–4,(2,1,3)–1	0,2	not found	to
_22	(1,1,2)–4,(1,2,1)–3	0,1	(1,1,2)–3,(2,2,4)–4	0,2
_23	(1,2,1)–3,(2,1,1)–4	0,1	not found	to
_24	(1,1,2)–4,(1,2,1)–3	0,1	not found	to
_25	(1,1,2)–2,(1,2,1)–4	0,1	not found	to
_26	(1,1,2)–2,(1,2,1)–4	0,1	(1,1,2)–2,(2,2,4)–4	0,1
_27	(1,2,1)–4,(2,1,1)–3	0,1	not found	to
_28	(1,1,2)–2,(1,2,1)–4	0,1	(2,2,1)–2,(4,4,2)–3	23,8
_29	(1,2,1)–2,(2,1,1)–3	0,2	not found	to
_30	not found	to	not found	to
_31	(1,0,1)–2,(1,1,0)–4	15,8	not found	to
_32	(0,1,1)–2,(1,0,1)–4	0,1	not found	23,2
_33	not found	to	(0,1,1)–4,(0,2,2)–2	0,1
`trityp` correct version		2388		3280,2

the same mutants. For example, `trityp_16` is only killed by μ–rel3 whereas `trityp_1` is only killed by μ–rel2. This shows how critical is the choice of the metamorphic–relations for AMT.

Proving that a program satisfies a given metamorphic–relation. The time required to do such a proof for `GetMid` and `trityp` is shown in the last row of Tab.2 and Tab.3. Because the size of the search space has been arbitrarely limited to hundred values for each variable, the proof is only valid in the limited ranges. For `bsearch`, the proof is done on a reasonnable amount of time only when values belong to a range of no more than twenty values. Although these proofs are partials, they form a valuable step toward automatic program correctness. However, this shows also that even if the use of constraints allows to prune the search space, the space left to explore is still too wide and when the program is correct, this left space has to be fully examined. Note that the labelling strategies used to explore this space have been initially designed to find solutions rather than to prove quickly inconsistencies.

6. Perspectives

According to our knowledge, this work is the first attempt to generate automatically test data to reveal faults by the mean of Metamorphic Testing. Experimental results were provided to show evidence of the potential interest of this paradigm for testing non-testable programs.

However, the limits of the automated approach have been identified and lead to forsee the usage of special metamorphic–relations, such as permutation-based relations, to speed up the search among the possible test data. Perspectives of this work concern also the extension of our framework to other constructs and data. In particular, it will be evaluated soon on numeric programs over floating–point variables, because the design of an automatic procedure able to check these programs without the help of any oracle is of great interest.

References

[1] Axlog Ingenierie and Thales Airborne Systems. *INKA–V1 User's Manual*, dec. 2002.

[2] M. Carlsson, G. Ottosson, and B. Carlson. An Open–Ended Finite Domain Constraint Solver. In *Prog. Lang., Impl., Logics, and Programs (PLILP)*, 1997.

[3] F. Chan, T. Chen, S. C. Cheung, M. Lau, and S. Yiu. Application of metamorphic testing in numerical analysis. In *IASTED Conf. in Soft. Eng.*, pages 191–197, 1998.

[4] T. Chen, T. Tse, and Z. Zhou. Fault-based testing in the absence of an oracle. In *IEEE Int. Comp. Soft. and App. Conf. (COMPSAC)*, pages 172–178, 2001.

[5] T. Chen, T. Tse, and Z. Zhou. Semi-proving: an integrated method based on global symbolic evaluation and metamorphic testing. In *ACM Int. Symp. on Soft. Testing and Analysis (ISSTA)*, pages 191–195, 2002.

[6] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. on Prog. Lang. and Sys.*, 13(4):451–490, 1991.

[7] R. A. Demillo and A. J. Offut. Constraint-Based Automatic Test Data Generation. *IEEE Trans. on Soft. Eng.*, 17(9):900–910, Sep. 1991.

[8] J. Duran and S. Ntafos. An evaluation of random testing. *IEEE Trans. on Soft. Eng.*, 10(4):438–444, Jul. 1984.

[9] A. Gotlieb, B. Botella, and M. Rueher. Automatic Test Data Generation Using Constraint Solving Techniques. In *ACM Int. Symp. on Soft. Testing and Analysis (ISSTA). Soft. Eng. Notes*,23(2):53-62, 1998.

[10] A. Gotlieb, B. Botella, and M. Rueher. A clp framework for computing structural test data. In *Computational Logic (CL)*, pages 399–413, LNAI 1891, 2000.

[11] J. Jaffar and J. Lassez. Constraint Logic Programming. In *ACM Princ. of Prog. Lang. (POPL)*, pages 111–119, 1987.

[12] J. C. King. Symbolic Execution and Program Testing. *CACM*, 19(7):385–394, 1976.

[13] B. Korel. Automated Software Test Data Generation. *IEEE Trans. on Soft. Eng.*, 16(8):870–879, Jul. 1990.

[14] K. Marriott and P. J. Stuckey. *Programming with Constraints : an Introduction.* The MIT Press, 1998.

[15] C. Michel, M. Rueher, and Y. Lebbah. Solving constraints over floating-point numbers. In *Constraint Prog. (CP)*, pages 524–538, LNCS 2239, 2001.

[16] L. J. Morell. A theory of fault-based testing. *IEEE Trans. on Soft. Eng.*, 16(8):844–857, Aug. 1990.

[17] E. Weyuker. On testing non-testable programs. *Computer Journal*, 25(4):465–470, 1982.