Exploiting Symmetries to Test Programs

Arnaud Gotlieb
IRISA / INRIA
Campus Beaulieu
35042 Rennes Cedex, FRANCE
Arnaud.Gotlieb@irisa.fr

Abstract

Symmetries often appear as properties of many artifical settings. In Program Testing, they can be viewed as properties of programs and can be given by the tester to check the correctness of the computed outcome. In this paper, we consider symmetries to be permutation relations between program executions and use them to automate the testing process. We introduce a software testing paradigm called Symmetric Testing, where automatic test data generation is coupled with symmetries checking to uncover faults inside the programs. A practical procedure for checking that a program satisfies a given symmetry relation is described. The paradigm makes use of Group theoretic results as a formal basis to minimize the number of outcome comparisons required by the method. This approach appears to be of particular interest for programs for which neither an oracle, nor any formal specification is available. We implemented Symmetric Testing by using the primitive operations of the Java unit testing tool Roast [9]. The experimental results we got on faulty versions of classical programs of the Software Testing community tend to show the effectiveness of the approach.

1. Introduction

Testing imperative programs at the unit level requires to select test data from the input domain, to execute the program with the selected test data and finally to check the correctness of the computed outcome. For almost three decades, propositions have been made to automate this process. Structural test data generation relies on program analysis to find automatically a test set that guarantees the coverage of some criteria based on flow graphs [8, 5, 15, 14]. Functional testing is based on specifications to generate automatically test data [3, 20]. These techniques both require a formal description to be given as input : the source code of programs in the case of structural testing ; the formal specification of programs in the case of functional testing. However there are programs to be tested for which none of these formal descriptions is available. For example, commercial off-the-shelf components are usually delivered as "black-boxes", i.e. executable objects whose licenses forbid de-compilation back to the source code [17], and informal specification is used most of the time to describe their expected behaviour. In these situations, techniques such as random testing [12], boundary-value analysis [16] or local exhaustive testing [21] can be employed. All these methods have in common a focus on the generation of input values and are based on an underlying assumption which concerns the availability of a correct and complete oracle, i.e. a procedure able to predict the right outcome for any input data. Unfortunately, there are situations where this assumption seems to be unreasonable. As pointed out by Weyuker [19], some programs are considered to be non-testable. These are programs for which it is theoretically possible, but practically too difficult to determine the correct outcome. Consider programs intented to compute a function which is not accurately known or programs for which correct answers are too difficult to compute by hand. Third-party librairies and commercial components fall usually into the former case [11], whereas complex numerical programs fall into the latter [7].

In this paper, we introduce a software testing paradigm, called Symmetric Testing (ST), which aims at testing programs for which neither an oracle, nor any formal description is required. We consider symmetries to be permutation relations between program executions and use them to automate the testing process. Given the interface of a program and a symmetry relation that the program has to satisfy, ST combines automatic test data generation and symmetry checking to uncover faults within the program. Group theoretic results are used as a formal basis, conforming so the well-known adage *"Numbers measure size, Groups measure symmetry"* [2]. As a trivial example, consider a program p intended to compute the greatest common divisor (gcd) of two non-negative integers u and v

and suppose that p is tested with the following test datum ($u = 1309, v = 693$), automatically generated by a random test data generator. Although, we all know how to compute the gcd of two integers[1], it is not so easy to predict the expected value of $gcd(1309, 693)$ without the help of a calculator. Fortunately, gcd satisfies a simple symmetry relation :$\forall u \forall v, gcd(u, v) = gcd(v, u)$. So, if $gcd(1309, 693) \neq gcd(693, 1309)$ then the testing process will succeed to uncover a fault without the help of an oracle of gcd. We generalized this idea to obtain a formal definition of symmetry relation on imperative programs. Formally speaking, let p be a program which takes a vector of at least k values as input and returns a vector of at least l values, and let x and y be two vectors then a symmetry relation for p holds if :

$$\forall \theta \in S_k, \exists \eta \in S_l \text{ such as } y = \theta.x \implies p(y) = \eta.p(x)$$

where S_k (resp. S_l) is the symmetric group acting on k elements of x (resp. l elements of $p(x)$), $p(x)$ denotes the vector of values computed by the execution of p with x as input and $\theta.x$ denotes the image of x by the permutation θ. Symmetric Testing consists in finding test data that violate a given symmetry relation, i.e. finding x and θ such as :

$$y = \theta.x \wedge p(y) \neq \eta.p(x)$$

Symmetries relations are abstract generic properties and checking the correctness of programs with respect to these relations is a difficult task, likely undecidable in the general case[2]. However, there are circumstances when testing procedures can be used to check the correctness of properties against programs [5, 7, 13]. Hence, by using these procedures, it becomes possible to check that a given symmetry relation is satisfied by the program on a finite subset of its input space. Limitations of ST concern the weaknesses of symmetry relations to differentiate incorrect implementations from correct ones. In fact, there are lots of programs that satisfy a given symmetry relation and any incorrect implementation will not be necessary discovered by Symmetric Testing. Conversely, the approach does not report any spurious fault. In order to evaluate the fault revealing capacity of ST, we implemented it by using the four unit operations of the Java testing tool *Roast* [9] and we used it to reveal faults on several academic programs and on programs extracted from the third-party library `java.util.Collections` of the Java 2 plateform (std edition 1.4). These first experimental results show that ST is of particular interest when testing some of the "non-testable" programs.

The rest of the paper is organized as follows : section 2 presents the Group theoretic results as well as the necessary notations required to fully understand this paper. Section 3 details the principle of Symmetric Testing while section 4 reports the experimental results obtained with *Roast*. Related works are described in section 5 and finally section 6 indicates several perspectives to this work.

[1] with the Euclidian algorithm for example

[2] Although we are not aware of any proof of this, it can be hypothesized as a consequence of the undecidability of the halting problem

2. Group theory : notations and selected results

All the basic material on Group theory presented in this section is extracted from [2] and the JS Milne's lecture notes (available online `www.jmilne.org/math/CourseNotes`).

Definition 1 *(Group)*
A nonempty set G together with a composition law $\circ$ is a **group** *iff G satisfies the following axioms :*

- $\forall a, \forall b, \forall c \in G, a \circ (b \circ c) = (a \circ b) \circ c$ *(associativity)*
- $\exists e \in G$ *such as* $a \circ e = e \circ a = a \;\; \forall a \in G$ *(neutral)*
- $\forall a \in G, \exists a^{-1} \in G$ *such as* $a \circ a^{-1} = a^{-1} \circ a = e$ *(existence of an inverse)*

The notion of symmetric group is the corner-stone of Symmetric Testing. Let E be a nonempty finite set of n distinct elements, the set S_E of bijective mappings from E to itself is called the **symmetric group** of E (say S_E acts on E). This symmetric group has exactly $n!$ elements, called permutations. It is clear that S_E is a group because it is closed and associative under $\circ$, identity is its neutral element and each permutation possesses an inverse because the definition is restricted to bijective mappings only. e_{S_E} denotes its neutral element.

S_E can be identified with S_n : the symmetric group acting on $\{1, .., n\}$ as there is a trivial bijective relation (isomorphism) between S_E and S_n. A permutation in S_n is written : $\theta = \begin{pmatrix} 1 & .. & n \\ i(1) & .. & i(n) \end{pmatrix}$ where $i(1), .., i(n)$ denote the images of $1, .., n$ by the permutation θ. As previously said, when a permutation of S_n is applied to a vector x of size n, we will write $\theta.x$ to denote the image of x by the permutation θ (say θ acts on x). For the sake of clarity, we will extend our notations to program compositions. If p is a program, $p \circ \theta$ will denote the application of p to the permutation θ of the elements of its input vector. Conversely, $\eta \circ p$ will denote the permutation η applied to the output vector of p.

All the permutations can be expressed by using only a few ones. Consider for example the permutation
$\theta = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 3 & 4 & 1 & 5 & 2 \end{pmatrix}$ of S_5, the same permutation can be captured by the following notation $\theta = (13)(245)$ where each pair of brackets denotes a specific permutation called an r_cycle. A permutation $(a_1 a_2 .. a_r)$ of S_n is an **r_cycle**

iff it maps a_1 to a_2, .. a_{r-1} to a_r, a_r to a_1 and leave unchanged the other elements. A 2_cycle (written $(a_i a_j)$) is usually referred to as a transposition. A trivial property of transpositions is that they are their own inverse.
A subset X of elements of a finite group G is a set of **generators** iff every element of G can be written as a finite composition product of the elements of X. G is said to be generated by X. For example, it is well-known that S_3 is generated by the two transpositions $\tau_1 = (12)$ and $\tau_2 = (23)$ because each one of the six elements of S_3 can be written with a finite composition product of these two transpositions. More generally, S_n is generated by all the transpositions, but also by the following subset of transpositions : $\{(12), (23), .., (n-1, n)\}$. In this paper we will use the following proposition, given here without a proof (that can be found in [2]).

Proposition 1 *(generators of S_n)*
The transposition $\tau = (12)$ and the n_cycle $\sigma = (12..n)$ together generate S_n.

It can be shown that S_n cannot be generated by less than two permutations. Hence $\{\tau, \sigma\}$ is a set of generators of minimum size.

A fundamental notion in Group theory which will be helpfull when defining symmetries over programs is the notion of group homomorphism :

Definition 2 *(group homomorphism)*
A group homomorphism from a group G to a group G' over the same composition law $\circ$ is a map $\varphi : G \longrightarrow G'$ such that $\varphi(\theta \circ \theta') = \varphi(\theta) \circ \varphi(\theta')$.

Note that an isomorphism is simply a bijective homomorphism. As a consequence of this definition, the image of a group homomorphism from G to G' is a subgroup of G'. It is noted $Im(\varphi)$. Conversely, $Ker(\varphi)$ denotes the kernel of a group homomorphism, which is the set of permutations of G which are mapped to $e_{G'}$. $G/Ker(\varphi)$ denotes the group quotient of G by $Ker(\varphi)$, i.e. the set $\{g \circ h | g \in G, h \in Ker(\varphi)\}$. $Hom(G, G')$ denotes the set of group homomorphisms. To end this review of the Group theoretic results that are needed to introduce Symmetric Testing , we give the fundamental theorem of group homomorphisms :

Proposition 2 *(isomorphism theorem)*
Let G and G' be two groups and φ be an element of $Hom(G, G')$, then φ factors into the composite of a surjection, an isomorphism $\bar{\varphi}$ and an injection :

$$\begin{tikzcd} G \arrow[r, "\varphi"] \arrow[d, "surj"'] & G' \\ G/Ker(\varphi) \arrow[r, "\bar{\varphi}"] & Im(\varphi) \arrow[u, "inj"'] \end{tikzcd}$$

3. Principle of Symmetric Testing

3.1. Symmetry relations

The idea behind Symmetric Testing is to exploit user-defined symmetries to automate the testing process of sequential imperative programs[3]. We briefly discuss two possible kinds of symmetry that can be considered for program testing.

- **Symmetries over values**. A symmetry over values can be expressed as a relation between two program executions where there is a proportional link between the two input points. As a trivial example, consider a program p which takes two integers as arguments and verifies $p(x, y) = p(k * x, k * y)$. When $k = -1$, the two input points are symmetrical w.r.t. the origin of the input space ;

- **Symmetries over variables**. A symmetry over variables can be viewed as a relation between two program executions where there is a permutation relation beetwen the input points. $p(x, y) = p(y, x)$ is the most simple example of such a symmetry.

Symmetries over values can easily be recognized for programs that compute a mathematical function given by a formula (based on arithmetic or trigonometric operations). In some cases, local symmetries over these operations may be aggregated to determine a global symmetry over the formula, such as in the formula $p(x, y) = sin(xy) - cos(y)$ which satisfies a trivial symmetry over values w.r.t. the origin. However, in such a case, the formula itself can be used to check the correctness of the computed outcome. Hence, these symmetries over values appear to be irrelevant to our purpose. Conversely, symmetries over variables can be specified with a very little knowledge on the function being computed. Type informations are sometimes sufficient to see that a program has to satisfy a symmetry over variables. Further, they are abstract properties that can be easily extracted from an informal specification. These are the reasons why we will focus on such symmetries in this paper. Formally speaking, a symmetry is defined as follows :

Definition 3 *(symmetry)*
Let p be a program over a domain D that takes n references as input and returns m references[4], and let S_n (resp. S_m) be the symmetric group over n (resp. m) elements, then a **symmetry** *for p is a pair $< \theta, \eta >$ such that : $\theta \in S_n, \eta \in S_m$,*

$$y = \theta.x \implies p(y) = \eta.p(x) \quad \forall x, y \in D$$

[3] For the sake of simplicity, our presentation will be limited to the Imperative Programming paradigm only

[4] As usual in imperative programming, the value of an input reference may be modified within the program and considered so as an output variable

Note that every program p satisfies at least the trivial symmetry $< e_{S_n}, e_{S_m} >$ because only deterministic programs are considered here (two executions with the same input give the same result). Some of the references of the input vector may be left unchanged by the permutation θ of a symmetry $< \theta, \eta >$. So, the vector of k exchanged input references involved in the symmetry is called the *permutable input* set[5] whereas the vector of l exchanged output references for any θ is called the *permutable output* set. Symmetries can be grouped together by means of symmetry relations.

Definition 4 *(symmetry relation)*
Let p be a program over a domain D that has k permutable input data and l permutable output data, $\Psi_{k,l}$ is a **symmetry relation** *for p iff*

- $\forall \theta \in S_k,\ < \theta, \Psi_{k,l}(\theta) >$ *is a symmetry for p ,*
- $\Psi_{k,l} \in Hom(S_k, S_l)$ *(group homomorphism).*

Symmetry relations are required to be group homomorphisms because this allows to maintain interesting properties over the links between input and output of *p*. Informally speaking, this requirement guarantee to consider only the "right" symmetry relations. This will be made clearer in the following. Note that symmetry relations are difficult to state when the number of permutable input data increases (S_k contains $k!$ permutations), because they are to be given under the form of a set of symmetries $\{< \theta, \Psi_{k,l}(\theta) >\}_{\forall \theta \in S_k}$. Note also that $\Psi_{k,l}$ does not denote a unique symmetry relation because there is no requirement over the mapping properties of the homomorphism. In fact, $\Psi_{k,l}$ is identified with a class of symmetry relations that are group homomorphisms in $Hom(S_k, S_l)$.

Based on their formal definition, identifying such symmetry relations might appear to be difficult. Conversely, we argue that they can easily be specified by looking at the informal specification of programs, because they are often related to the type informations of program variables. Consider a program *p* taking an unordered set as argument, then we already know that *p* has to satisfy a symmetry relation because computing *p* with a permutation of the elements of the set does not modifiy the computed result. Numerous programs take unordered sets as arguments : consider sorting programs or graph-based programs just to name a few. Further, third-party libraries that contain lots of generic programs (for reusing purpose) often have to satisfy symmetry relations.

3.2. Examples

Consider the standard application programming interface specification of the `java.util.Collections.replaceAll` method given in Fig.1. If we consider the n_cycle σ (permutation $(12..n)$), then the method `replaceAll` has to satisfy a $< \sigma, \sigma >$ symmetry : let A (resp. B) be a vector of n symbolic references and A' (resp. B') be the resulting vector computed by invocation of `replaceAll` with the references *oldVal* and *newVal*, then $B = \sigma.A \implies B' = \sigma.A'$. A and B are two permutable input sets whereas A' and B' are the permutable output sets. Further, it is clear that `replaceAll` has to satisfy the same symmetry for all $\theta \in S_n$. Hence, this Java method has to satisfy a $\Psi_{|A|,|A|}$ symmetry relation, where $|A|$ denotes the size of the abstract collection A. Finally, this group homomorphism is the identity of $Hom(S_{|A|}, S_{|A|})$. By looking at the `java.util.Collections` class which contains 19 distinct methods[6] among 37, we found that 12 methods have to satisfy at least one non-trivial symmetry relation. This class was selected because it consists of methods that operate on collections, which can be specialized on multiset or sequences of objects. As a consequence, the results given here should not be extrapolated for any other classes. Table1 summarizes the symmetry relations that can easely be extracted by the tester. Permutable input and output are indicated in the two central columns. The symbol Ret denotes the returned reference or value of the Java method.

```
public static boolean replaceAll(List A,
                        Object oldVal,
                        Object newVal,
```

Replaces all occurrences of one specified value in a list with another. More formally, replaces with newVal each element e in A such that (oldVal==null ? e==null : oldVal.equals(e)). (This method has no effect on the size of the list.)
Parameters:
`A` - the list in which replacement is to occur.
`oldVal` - the old value to be replaced.
`newVal` - the new value with which oldVal is to be replaced.
Returns:
true if list contained one or more elements e such that `(oldVal==null ? e==null : oldVal.equals(e))`.
Throws:
UnsupportedOperationException - if the specified list or list-iterator does not support the set method.

Figure 1. API specification of `replaceAll`

`Max` and `min` have to satisfy a $\Psi_{|A|,1}$ symmetry relation because they return one of the elements of an unordered set, took as argument. Conversely, `fill, replaceAll, reverse, rotate, sort, shuffle,swap` have to satisfy a $\Psi_{|A|,|A|}$ symmetry relation because they modify ordered sequences or lists. In fact, these polymorphic functions have been extensively studied in the Functional Programming Community and the symmetry relations they have to satisfy should be spelled out from well-known properties of their type [18]. The informal specification of `enu-`

[5] in Group theory, this is called the support of a permutation

[6] methods which have distinct specifications, and not only distinct interfaces

Table 1. Examples of symmetry relations

Signature of Java methods	Perm. in	Perm. out	Sym. rel.
`void copy(List B,List A)`	A	B	$\Psi_{\lvert A\rvert,\lvert B\rvert}$
`Enumeration enumeration(Collection A)`	C	Ret	$\Psi_{\lvert A\rvert,\lvert Ret\rvert}$
`void fill(List A,Object obj)`	A	A	$\Psi_{\lvert A\rvert,\lvert A\rvert}$
`Object max(Collection A)`	C	Ret	$\Psi_{\lvert A\rvert,1}$
`Object min(Collection A)`	C	Ret	$\Psi_{\lvert A\rvert,1}$
`List nCopies(int n, Object O)`	O	Ret	$\Psi_{1,\lvert Ret\rvert}$
`boolean replaceAll(List A, Object oldVal, Object newVal)`	A	A	$\Psi_{\lvert A\rvert,\lvert A\rvert}$
`void reverse(List A)`	A	A	$\Psi_{\lvert A\rvert,\lvert A\rvert}$
`void rotate(List A, int distance)`	A	A	$\Psi_{\lvert A\rvert,\lvert A\rvert}$
`void shuffle(List A)`	A	A	$\Psi_{\lvert A\rvert,\lvert A\rvert}$
`void sort(List A)`	A	A	$\Psi_{\lvert A\rvert,1}$
`void swap(List A, int i, int j)`	A	A	$\Psi_{\lvert A\rvert,\lvert A\rvert}$

`meration` shows that $|Ret|$ is equal to $|A|$. Hence the method `enumeration` has indeed to satisfy a $\Psi_{|A|,|A|}$ symmetry relation. Note that `shuffle` uses a random permutation of its permutable input data. Although the computed list cannot be easily predicted, the symmetry relation that `shuffle` has to satisfy is specified without any difficulty. `nCopies` has to satisfy a $\Psi_{1,|A|}$ symmetry relation, which can be interpreted as follows : whatever is the argument of `nCopies`, the outcome should be a vector of equal references. To complete this panorama, consider the `copy` method which aims at copying the values contained into a list (the source list) into another one (the destination list). The method requires the destination list to be at least as long as the source list. If it is longer, the remaining elements in the destination list are unaffected and remain equal to their previous values. As a consequence, the method has to satisfy a $\Psi_{|A,|B|}$ symmetry relation where $|A|$ may not be equal to $|B|$. For example, copying $S = [1, 2, 3]$ into $D = [0, 0, 0, 0, 0]$ leads to $D' = [1, 2, 3, 0, 0]$. In section 4, we will give several other examples of non-trivial symmetry relations.

3.3. Symmetric Testing (ST)

These symmetry relations can be used to seek for a subclass of faults within an implementation. Formally speaking :

Definition 5 *(ST principle)*
Let p be a program over D and $\Psi_{k,l}$ be a symmetry relation for p, then ST aims at finding a triple $(x, p(x), p(\theta.x))$ such as $p(\theta.x) \neq \Psi_{k,l}(\theta).p(x)$ and $x, \theta.x \in D$

If found, such a triple $(x, p(x), p(\theta.x))$ represents a counter-example for the symmetry relation. We call this a *symmetry violation*. Note that there is no way to identify among the two test data x and $\theta.x$ the one that leads to an incorrect outcome for p. In the worst case, they can even be both faulty. So, given a set of test data and a symmetry relation, we get a naive procedure that can uncover a subclass of faults in p : it requires to compute p with all the permutations of the permutable input of each vector x in the test set and to check whether the outcome vectors are equals to a permutation of the vector returned by $p(x)$. These latter operations will be called *outcome comparisons* in the following[7]. The principle of ST is illustrated by the following diagram for which one can ask whether it is commutative or not :

$$\begin{tikzcd} x \arrow[r, "\theta"] \arrow[d, "p"'] & \theta.x \arrow[d, "p"] \\ p(x) \arrow[r, "\Psi_{k,l}(\theta)"'] & \Psi_{k,l}(\theta).p(x) \end{tikzcd}$$

Proving that this diagram is commutative is only a necessary condition for the correctness of p w.r.t. its specification because incorrect implementations of the function computed by p may also satisfy the same symmetry relation. Note that this procedure is independent of the test set being used. In fact it leaves the tester the possibility to use any automatic test data generators, bearing in mind that no oracle is required.

3.3.1 Finding all the symmetry violations

Exploiting the ST principle to test programs yields to consider the following question : would it be possible to detect all the symmetry violations within the input domain D ? To answer this question (at least partially), we propose to use a local exhaustive test data generator [21] tuned for our purpose. Local exhaustive testing requires to identify critical points around which input values are exhaustively selected and used as test data. In our framework, the exhaustive exploration is based on a Cartesian Product generator [9] which starts on the origin of a Cartesian domain. When the input domain D is infinite, as it is the case for the **replaceAll** method (Fig.1) which takes an unbounded list of objects as its first argument, the Cartesian Product generator will enumerate all the possible list sizes until a selected size will be reached. Hence, by combining local exhaustive testing with symmetry checking, ST should be able to detect all the symmetry violations that exist into the finite restricted subset of D being explored. In addition, the limitation of the input domain to be restricted and finite allows the technique to remain practical (when the number of input references k is small).

Under some circumstances that are discussed in Sec.3.4 and when no symmetry violation has been detected, we get a proof that the program satisfies a given symmetry relation in the subset of D being explored. Note that other ap-

[7] Checking equality of floating point values might be hazardous, hence the outcome comparison should be relaxed with an epsilon in this case

proaches, that make use of a (semi-)proving technique, can be followed [7, 13] to do such a proof.

3.3.2 Minimizing the number of outcome comparisons

The somehow naive procedure given above requires to do an outcome comparison for each possible permutation in the Symmetric Group S_k. Recalling that S_k contains $k!$ permutations, it is clear that the approach becomes impractical when k increases. We turn now to a more practical procedure that checks whether a program satisfies a given symmetry relation.

By using Prop.1, we known that no more than two permutations are required to generate S_k. This result can be used to minimize the number of outcome comparisons required by ST :

Proposition 3 *Let p be a program over D and $\Psi_{k,l}$ be a symmetry relation for p, let $\tau = (12)$ and $\sigma = (12..k)$, then we have*

$$\begin{cases} p \circ \tau = \Psi_{k,l}(\tau) \circ p \\ p \circ \sigma = \Psi_{k,l}(\sigma) \circ p \end{cases} \Longleftrightarrow p \circ \theta = \Psi_{k,l}(\theta) \circ p \quad \forall \theta \in S_k$$

Proof:
$\Leftarrow$. Taking $\theta = \tau$ and $\theta = \sigma$ yields the expected result.
$\Rightarrow$. Let $\theta \in S_k$ be a permutation (distinct from τ or σ). By using proposition 1, θ can be written as a finite composition of the two permutations τ and σ. Let $\theta = \tau \circ \sigma \circ ...$ be the beginning of such a composition (taking any other chain does not change the proof) then $p \circ \tau \circ \sigma \circ ... = \Psi_{k,l}(\tau) \circ p \circ \sigma \circ ...$ by applying one of the two hypothesis and recalling that $\circ$ is associative. Further, it is possible to iterate on the composition chain : $p \circ \tau \circ \sigma \circ ... = \Psi_{k,l}(\tau) \circ \Psi_{k,l}(\sigma) \circ p \circ ... = \Psi_{k,l}(\tau) \circ \Psi_{k,l}(\sigma) \circ ... \circ p$. This is repeated until the complete finite chain would have been processed. Finally, $\Psi_{k,l}(\tau) \circ \Psi_{k,l}(\sigma) \circ ...$ is equal to $\Psi_{k,l}(\theta)$ because $\Psi_{k,l}$ is a group homomorphism ■

This shows that only two outcome comparisons are required for each test datum to check a given symmetry relation. As a corollary, it is possible to characterize the subgroup of S_l, image of S_k by the homomorphism $\Psi_{k,l}$.

Proposition 4 *(Generators of $Im(\Psi_{k,l})$)*
$\Psi_{k,l}(\tau)$ and $\Psi_{k,l}(\sigma)$ together generate the subgroup $Im(\Psi_{k,l})$.

Proof: $\Psi_{k,l}(\theta) = \Psi_{k,l}(\tau) \circ \Psi_{k,l}(\sigma) \circ ...$ for all $\theta \in S_k$ hence every permutation of $Im(\Psi_{k,l})$ can be written as a finite composition product of $\Psi_{k,l}(\tau)$ and $\Psi_{k,l}(\sigma)$ ■

$Ker(\Psi_{k,l})$ denotes the set of permutations of S_k which leave the outcome of p unchanged. Because $G/Ker(\Psi_{k,l})$ is isomorphic to $Im(\Psi_{k,l})$ by the group isomorphism induced by $\Psi_{k,l}$ (isomorphism theorem), it is possible to determine precisely the mapping properties of $\Psi_{k,l}$ just by looking at the relation between the two generators $\Psi_{k,l}(\tau)$ and $\Psi_{k,l}(\sigma)$, but this is outside of the scope of the paper. In practice, giving just the two symmetries $< \tau, \Psi_{k,l}(\tau) >$ and $< \sigma, \Psi_{k,l}(\sigma) >$ is enough to fully determine the symmetry relation that p has to satisfy. The keypoint of our approach is that we have to consider only symmetry relations that are group homomorphisms. In section 3.1, we called these the "right" symmetry relations.

3.3.3 A practical procedure for ST

Based on the previous result, we can now provide a procedure able to find all the symmetry violations in a finite restricted subset of D, noted $\mathcal{D}$. This procedure is based on the fact that it is only required to show that $p(\tau.x) = \Psi_{k,l}(\tau).p(x)$ and $p(\sigma.x) = \Psi_{k,l}(\sigma).p(x)$ for all x and $\sigma.x$ in $\mathcal{D}$. When no violation is found, this proves that p satisfies not only the two selected symmetries but also all the symmetries $< \theta, \Psi_{k,l}(\theta) >$ where $\theta \in S_k$ on $\mathcal{D}$.

The procedure shown in Fig.2 takes a program p, the finite domain $\mathcal{D} \subseteq D$, and the two symmetry images $\Psi_{k,l}(\tau)$, $\Psi_{k,l}(\sigma)$ as arguments. When it terminates, it returns either the first found triple $(x, p(x), p(\theta.x))$ that violates the symmetry relation or a proof that p satisfies $\Psi_{k,l}$ over $\mathcal{D}$ ("Q.E.D"). The procedures requires only two outcome comparisons for each $x \in \mathcal{D}$. The computed outcome results of p can be kept into a table in order to avoid to compute p several times with the same input data (this is tackled implicitly within the **let** statement of the procedure). Note that some

while($\mathcal{D} \neq \emptyset$) do
  pick up $x \in \mathcal{D}$,
  $\mathcal{D} := \mathcal{D} \setminus \{x\}$;
  let $r = p(x)$, $r_\tau = p(\tau.x)$, $r_\sigma = p(\sigma.x)$ **in**
    if($r_\tau \neq \Psi_{k,l}(\tau).r$)
      then return (x, r, r_τ)
    if($r_\sigma \neq \Psi_{k,l}(\sigma).r$)
      then return (x, r, r_σ)
return("Q.E.D.");

Figure 2. A procedure for Symmetric Testing

test data are not required to be examined. For example, test data of the form $x = (v, v, .., v)$ can be eliminated from $\mathcal{D}$ because any permutation will leave x unchanged. Note also that some outcome comparisons are unecessary, such as the ones of $p(x)$ and $p(\tau.x)$ when $x = (v, v, w_1, .., w_{n-2})$ because $x = \tau.x$. These optimizations can easily be implemented by filtering the test data generated by the local exhaustive testing tool.

3.4. Discussion

As expected, the termination of the previous procedure cannot be guaranted because it requires the computation of p. Although $\mathcal{D}$ is required to be finite, nothing prevents p to iterate infinitely when computing $p(x), p(\tau.x)$ or $p(\sigma.x)$ and no general procedure can be used to decide the termination of p.

Under the strong hypothesis that p halts on all test data, $\mathcal{D}$ may have to be fully explored in the worst case (when no symetry violation is found). Let $\mathcal{D} = \mathcal{D}_1 \times \mathcal{D}_2 \times .. \times \mathcal{D}_n$ and let d be the number of elements of the greatest domain $\mathcal{D}_i$, then the procedure given in Fig.2 will have to enumerate $O(d^n)$ points in the worst case. From a practical point of view, it is crucial to maintain d and n as small as possible by limiting the size of the domains of permutable input data. Note also that the number of outcome comparisons is $O(d^n)$ by using the procedure of Fig.2 whereas it would have been $O(n!\ d^n)$ in the worst case by using the naive procedure that we first introduced.

Another limitation comes from the difficulty to establish that a user-defined symmetry relation is actually a group homomorphism. To exemplify this problem, consider the method `Vector min_nb(Vector A, int nb)` which computes a vector of nb minimum integer values extracted from A (given in Fig.3). We guess that this program has to satisfy a $\Psi_{|A|,|Ret|}$ symmetry relation but this is difficult to state even when the source code is available. The vector returned by the method is not sorted by default and its elements order depends strongly on the algorithm used to compute the minimum values. As a consequence, we really do not know whether the relation is an homomorphism or not. In addition, it is obviously difficult to verify at hand that $\Psi_{|A|,|Ret|}(\theta_1 \circ \theta_2) = \Psi_{|A|,|Ret|}(\theta_1) \circ \Psi_{|A|,|Ret|}(\theta_2)$ for all θ_1, θ_2. Our approach for this problem consists in using a symmetry relation over a composition product of programs. By composing the `min_nb` program with a sorting method s, we get that the resulting program has to satisfy a $\Psi_{|A|,1}$ symmetry relation. This allows us to apply ST to the composition product $s \circ min\_nb$ in place of min_nb.

```
public static Vector min_nb(Vector V, int nb) {
  if( (V == null) ) return null;

  int k = ((V.size() < nb) ? V.size() : nb);
  int j, i = 0 ;
  Vector R = new Vector();
  Integer max, cur;
  Collections c = null;

  while( i < k ) {
    R.addElement(V.elementAt(i));
    i++;
  }

  while(i < V.size()){
    j = 0;
    while(j < k){                                  // k is the size of R
      max = (Integer) c.max(R);                    // max value of R
      cur = (Integer) V.elementAt(i);              // current value of V

      if(max.intValue() > cur.intValue()){
        R.setElementAt(cur, R.indexOf(max));
        break;
      }
      j++;
    }
    i++;
  }
  return(R);
}
```

Figure 3. The `min_nb` program

4. Experimental results

4.1. Implementation

We implemented ST with the help of the primitive operations of the Java unit testing tool *Roast* [9]. The tool includes four unit operations (generate, filter, execute, and check) designed 1) to automate the generation of test tuples, 2) to filter some of the generated tuples, 3) to execute the program under test with the selected tuples and 4) to check the computed outcomes. *Roast* provides several test data generators such as a boundary-value generator and a Cartesian product generator. We used only this latter one to implement our local exhaustive test data generator. The filtering operation was used to eliminate the test data of the form $(v, v, .., v)$. Roast supports test templates (Perl macros) to compare the actual outcomes to the predicted ones. We used these templates in combination with our Java methods to check whether a computed outcome is a permutation of another one.

4.2. Experiments with ST

The goal was to study the capacity of ST to reveal faults in programs and to find cases where ST is able to prove that a given symmetry relation is satisfied. The experiment was performed on classical academic programs, where faults were injected by mutation.

Programs. Six programs were selected among which three were implemented and three came from the **java.lang.Collections** class : `replaceAll`, `sort` and `copy`. We implemented the `min_nb` method (given in Fig.3), the `GetMid` program (given in [7]) intended to compute the median of three integers, and the well–known triangle classification program `trityp` [10]. This program takes three positive integers as arguments that represent the relative lengths of the sides of a triangle and classifies the triangle as scalene (sca), isocele (iso), equilateral (equ) or illegal (illeg). To limit the size of the search space, we considered every input integer to belong to a range of 100 values (ranging from 0 to 99) for `GetMid` and `trityp`. For `min_nb`, `replaceAll`, **copy**, and **sort** we considered lists to contains at most 4 integers ranging from 0 to 24.

Mutants. Four mutants were created for `min_nb` and `GetMid`. `GetMid_1` and `min_nb_1` are versions of the original programs where two statements are removed. The first mutant has been studied in [7] because it contains a "missing path error" fault, which is considered as a difficult fault to reveal. The mutation of relational operators in `GetMid_2` and `min_nb_2` leads to the creation of infeasible paths (at least for the first program). 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[8] have been removed from the set of experiments, because they cannot be revealed by the mean of testing [10]. All the mutants are available at the url `www.irisa.fr/lande/gotlieb`

Symmetry relations. The results of `GetMid` and `trityp` must be invariant to every permutation of their three input values, leading to a $\Psi_{3,1}$ symmetry relation. As previously said, `min_nb` composed with the `sort` method has to satisfy a $\Psi_{|A|,1}$ symmetry relation. The symmetry relations for `replaceAll`, `copy` and `sort` are given in Table 1.

4.3. Experimental results and Analysis

All the computations were performed on a 1.8Ghz Pentium 4 personal computer by using the version 1.3 of *Roast*. The computations were repeated five times and only the worst resulting runtime was retained.

4.3.1 Revealling faults with ST

We first applied ST to reveal faults among the mutants of each program. The results are given in tables 4, 2, and 3. The values of $p(x)$, $p(\tau.x)$ and $p(\sigma.x)$ are given in each of three interior columns of the tables. In the case where a test datum that violates the symmetry relation is found, it is given in the first column and the mutant is said to be killed by the symmetry relation. When all the finite restricted input domain has been explored without finding such a test datum, the expression "not killed" is written. Incorrect results of the program p are noted with boldface to facilitate the results interpretation, but this was determined manually. For each relation, the CPU time spent to find the solution is given (including time spent garbage collecting or in system calls) in the last column.

[8] programs which compute the same outcome as the original program although a mutation operator has been applied

Table 2. Results for `GetMid`

Mutants	x	$p(x)$	$p(\sigma.x)$	$p(\tau.x)$	rtime (sec)
`GetMid_1`	(1,1,0)	**0**	1	**0**	0.1
`GetMid_2`	(1,0,0)	0	**1**	0	0.1
`GetMid`	$100^3 - 100 = 999900$ test data				9,4

Table 3. Results for `trityp`

Mutants	x	$p(x)$	$p(\sigma.x)$	$p(\tau.x)$	rtime (sec)
`trityp_1`	(3,1,1)	**iso**	illeg	illeg	0.2
`trityp_2`	not killed				9.6
`trityp_3`	(2,1,1)	illeg	**iso**	**iso**	0.2
`trityp_4`	(2,1,1)	**equ**	**sca**	**iso**	0.2
`trityp_5`	(2,1,1)	**iso**	illeg	illeg	0.2
`trityp_6`	(2,1,1)	illeg	illeg	**iso**	0.2
`trityp_7`	(2,1,1)	illeg	illeg	**iso**	0.2
`trityp_8`	(2,2,1)	**equ**	iso	**equ**	0.2
`trityp_9`	not killed				9.3
`trityp_10`	(2,1,1)	**equ**	illeg	illeg	0.2
`trityp_11`	not killed				9.2
`trityp_12`	(2,2,1)	**illeg**	iso	**illeg**	0.2
`trityp_13`	not killed				9.2
`trityp_14`	not killed				9.6
`trityp_15`	not killed				9.4
`trityp_16`	not killed				10
`trityp_17`	(3,2,1)	**sca**	illeg	**sca**	0.2
`trityp_18`	not killed				9.6
`trityp_19`	(2,2,1)	**sca**	iso	**sca**	0.2
`trityp_20`	(3,2,1)	**sca**	illeg	illeg	0.2
`trityp_21`	(3,2,1)	illeg	**sca**	illeg	0.2
`trityp_22`	(2,1,1)	illeg	illeg	**equ**	0.2
`trityp_23`	(2,1,1)	illeg	**equ**	**equ**	0.2
`trityp_24`	(2,1,1)	illeg	illeg	**equ**	0.2
`trityp_25`	(2,1,1)	**equ**	**iso**	illeg	0.2
`trityp_26`	(2,1,1)	illeg	**iso**	illeg	0.2
`trityp_27`	(2,1,1)	**equ**	illeg	illeg	0.2
`trityp_28`	(2,1,1)	illeg	**iso**	illeg	0.2
`trityp_29`	(2,1,1)	**equ**	**iso**	**iso**	0.2
`trityp_30`	not killed				9.9
`trityp_31`	(1,1,0)	illeg	**iso**	illeg	0.2
`trityp_32`	(1,0,1)	illeg	**iso**	**iso**	0.1
`trityp_33`	not killed				10
`trityp`	$100^3 - 100 = 999900$ test data				9,6

Test data that kill all the mutants of `min_nb` and `GetMid` are found. This illustrates the capacity of ST to reveal two difficult class of faults (missing path error and infeasible path) on these programs in a very short period of time. Among the 33 mutants of `trityp`, 10 were not detected as faulty versions (programs `trityp` _2,_9,_11,_13,_14,_15,_16,_18,_30, _33) by ST. Studying these programs leads to see that they are equivalent to the correct version of `trityp` in the following sense : both the mutant and the correct version satisfies the same symmetry relation. For example, the mutant `trityp_9` cannot be detected by ST because the fault has been introduced into a statement only reached by a sequence of three equal integers, which is invariant to permutation and in fact not even tried by the local exhaustive generator.

In some cases, the output vectors $p(x)$, $p(\sigma.x)$ and $p(\tau.x)$ are all incorrects (mutants _4 and _29). This illustrates a situation where a fault injected in the program leads to modify every computed outcome. Fortunately, this breaks also the symmetry relation that the program has to satisfy.

Table 4. Results for `min_nb`

Mutants	x	$p(x)$	$p(\sigma.x)$	$p(\tau.x)$	rtime (sec)
`sort ∘ min_nb_1`	vec:[1,1,0], nb:2	**[0,0]**	[0,1]	**[0,0]**	21.2
`sort ∘ min_nb_2`	vec:[1,0], nb:1	[0]	**[1]**	**[1]**	11.7
`sort ∘ min_nb`	$(25^4 + 25^3 + 25^2 - 75) * 5 =$ 2034000 test data				53,3

4.3.2 Checking that a program satisfies a given symmetry relation

Checking that the symmetry relations are satisfied by the correct versions of `min_nb`, `GetMid`, and `trityp` yields to the results shown in the last row of table 4, 2 and 3. As the size of the search space was arbitrarely limited, the proof is only valid on a small part of the input space. Nevertheless, we prefered to compromize the size of the input space rather than the time spent to search a counter-example. Although these proofs are done in restricted cases, they form a valuable step toward program correctness because the checking procedure is completely automated for these programs. Finally, we applied ST to check symmetry relations for the `copy`, `sort`, `replaceAll` methods of the class `java.lang.Collections`. The results are given in table 5 and show that the three methods satisfy their symmetry relation among the restricted input space.

Table 5. Results for `sort copy`, `replaceAll`

Mutants	x	$p(x)$	$p(\sigma.x)$	$p(\tau.x)$	rtime (sec)
`copy`	$25^4 + 25^3 + 25^2 - 75 = 406800$ test data				13.1
`sort`	$25^4 + 25^3 + 25^2 - 75 = 406800$ test data				13.7
`replaceAll`	$(25^4+25^3+25^2-75)*25 =$ 254250000 test data				4606.6

5. Related work

The idea of checking the computed results of a program by using several input data is not new. Ammann and Knight [1] described the data diversity approach which aims at executing the same program, on a related set of input points. To check the computed outcomes, a voting procedure is used as an acceptance test. As claimed by the authors, the reexpression algorithms, used to generate the input data in relation within an expression, must be tailored to the application at hand. Blum and Kannan [4] proposed the concept of *program checker*, which is a program able to play the role of a probabilistic oracle, under a set of restrictions. As an example, the authors considered the graph isomorphism problem and they provide a program checker which checks that a graph G' resulting from a random permutation of a graph G is isomorphic to a graph H only if G is isomorphic to H. Other program checkers that make use of mathematical properties are designed for programs that compute gcd,the matrix rank or programs that sort data. Conversely to these works, our approach is deterministic because each reported symmetry violation is given with certainty. Further, it focus on generic relations that can be easily extracted from an informal specification.

More recently, Chen et al. proposed in [6] to use existing relations over the input data and the computed outcomes to eliminate faulty programs, in a framework called Metamorphic Testing. They propose in [7] to use global symbolic evaluation techniques to prove that an implementation satisfies a given metamorphic relation for all input data. The technique yields to enumerate the paths of the program and to evaluate the statements along each path, by replacing variables by symbolic values. The procedure requires to compare several sets of constraints extracted from the program. The `GetMid` program is used as an example to illustrate the approach. For this program, a symmetry relation is provided and only a few permutations (transpositions only) are used to check the relation. However, their approach is only manual and requires several sets of constraints to be compared. In a previous work [13], we proposed to automate the generation of input data that violate a given metamorphic relation, by using Constraint Logic Programming techniques. During this work, symmetry relations appear to be of a great interest because of their simplicity and genericity. Although the preliminary ideas of ST are similar to those of the works of Chen and its collegues, our approach focus on symmetry relations and generalizes the approach which aims at exploiting these relations to test programs.

6. Perspectives

In this paper, we have introduced a new software testing paradigm called Symmetric Testing which aims at finding test data that violate a given symmetry relation. Group theoretic results are used to give a formal basis to this paradigm. In particular, a formal definition of symmetry relation is introduced and we have given a practical procedure for applying Symmetric Testing on imperative programs. However, the limits of our automated approach of Symmetric Testing have been identified. We forsee to replace the local exhaustive test data generator by a a constraint-based test data generator which makes use of constraints extracted from the source code. In this approach, constraints are used as relational expressions between input and output symbolic variables and this allows us to prove properties about the program and its test data. We believe that symmetry relations would be easily expressed as abstract program properties into such a framework. Further, program composition appears to be an interesting perspective to take into account programs for which it is not easy to specify symmetry relations. This would allow to specify symmetry relations over finite sequence of method invocations, providing a way to test programs at the integration level.

Acknowledgment

We would like to thank David Pichardie, Thomas Jensen and Bernard Botella for detailed comments on a preliminary version of this paper. Thanks also to the five anonymous referees for their helpful suggestions.

References

[1] P. E. Ammann and J. C. Knight. Data diversity: An approach to software fault tolerance. *IEEE Trans. on Computers*, 37(4):418–425, 1988.

[2] M. A. Armstrong. *Groups and Symmetry (Undergraduate Texts in Mathematics).* Springer Verlag, second edition, 1988.

[3] G. Bernot, M. C. Gaudel, and B. Marre. Software testing based on formal specifications: a theory and a tool. *Soft. Eng. Jour.*, 6(6):387–405, 1991.

[4] M. Blum and S. Kannan. Designing programs that check their work. *Jour. of the Assoc. for Computing Machinery*, 42(1):269–291, Jan. 1995.

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

[6] 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.

[7] 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.

[8] L. Clarke. A System to Generate Test Data and Symbolically Execute Programs. *IEEE Trans. on Soft. Eng.*, SE-2(3):215–222, September 1976.

[9] N. Daley, D. Hoffman, and P. Strooper. A framework for table driven testing of Java classes. *Soft. Prac. and Exper.*, 32(5):465–493, Apr. 2002.

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

[11] P. Devanbu and S. G. Stubblebine. Cryptographic verification of test coverage claims. In M. Jazayeri and H. Schauer, editors, *Proc. of the European Soft. Eng. Conf. (ESEC/FSE)*, pages 395–413. LNCS 1013, Sept. 1997.

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

[13] A. Gotlieb and B. Botella. Automated metamorphic testing. In *27th IEEE An. Int. Comp. Soft. and App. Conf. (COMPSAC)*, Dallas, TX, USA, Nov. 2003.

[14] 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.

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

[16] G. J. Myers. *The Art of Software Testing*. John Wiley, New York, 1979.

[17] J. M. Voas. Certifying off-the-shelf software components. *Computer*, 31(6):53–59, June 1998.

[18] P. Wadler. Theorems for free! In *FPCA'89, London, England*, pages 347–359. ACM Press, Sept. 1989.

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

[20] E. Weyuker, T. Goradia, and A. Singh. Automatically generating test data from a Boolean specification. *IEEE Trans. on Soft. Eng.*, 20(5):353–363, May 1994.

[21] T. Wood, K. Miller, and R. E. Noonan. Local exhaustive testing: a software reliability tool. In *Proc. of the Southeast regional conf.*, pages 77–84. ACM Press, 1992.