

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Exploiting Symmetries to Test Programs

Arnaud Gotlieb

N˚4810

Avril 2003

THÈME 2



ISSN 0249-6399 ISRN INRIA/RR--4810--FR+ENG



Exploiting Symmetries to Test Programs

Arnaud Gotlieb*

Thème 2 — Génie logiciel
et calcul symbolique
Projet Lande

Rapport de recherche n ° 4810 — Avril 2003 — 20 pages

Abstract: Symmetries often appear as properties of many artifical settings. In Program Testing, they can be viewed as properties of programs and can be used to check the correctness of the computed outcomes. 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 program executions 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 *Roast* : a Java unit testing tool developped by N. Daley, D. Hoffman and P. Strooper. The experimental results we got on faulty versions of classical programs of the Software Testing community show the effectiveness of the approach.

Key-words: Automated Software Testing, Symmetry, Group Theory, Symmetric Testing

(Résumé : tsvp)

* Projet Lande – IRISA / INRIA – Campus de Beaulieu, 35042 Rennes Cedex, FRANCE

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : 02 99 84 71 00 - International : +33 2 99 84 71 00
Télécopie : 02 99 84 71 71 - International : +33 2 99 84 71 71

Exploiter les symmétries pour automatiser le test de programmes impératifs

Résumé : La notion de symmétrie apparait souvent comme une propriété de nombreux objets artificiels. Dans le domaine du Test Logiciel, celle-ci peut être vue comme une propriété de certains programmes impératifs et peut être utilisée pour contrôler la correction des sorties calculées par ces programmes. Dans ce rapport, nous considérons qu'une symmétrie est une relation de permutation entre deux exécutions du programme et nous l'utilisons pour automatiser le processus de test. Nous introduisons un paradigme nommé Test Symmétrique où une procédure de génération automatique de données de test est couplée avec un contrôle de la symmétrie dans le but de détecter des fautes à l'intérieur des programmes. Une méthode pratique pour contrôler qu'un programme satisfait une relation de symmétrie donnée est décrite. Le paradigme fait usage de résultats de la Théorie des Groupes comme d'une base formelle pour minimiser le nombre d'exécutions requis par cette méthode. Cette approche semble être d'un certain intérêt pour les programmes qui ne disposent ni d'un oracle, ni d'une spécification formelle. Le Test Symmétrique a été développé et expérimenté à l'aide d'un outil de test unitaire de programmes Java, nommé *Roast* et mis au point par N. Daley, D. Hoffman et P. Strooper. Les résultats expérimentaux que nous avons obtenus sur des versions incorrectes de programmes classiques de la Communauté du Test Logiciel montrent l'efficacité de cette approche.

Mots-clé : Test logiciel automatisé, Symmétrie, Théorie des Groupes, Test symmétrique

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 outcomes. 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 [1, 2, 3, 4]. Functional testing is based on the specifications analysis to generate automatically test data [5, 6]. 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 no one 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 [7], and informal specification is used most of the time to describe their expected behaviour. In these situations, techniques such as random testing [8], boundary-value analysis [9] or local exhaustive testing [10] can be employed. Random testing aims at selecting randomly the values inside the input domain by using pseudo-random values generators, whereas boundary-value analysis relies on selecting the boundaries of each individual or dependent domains [11] of the input space. Local exhaustive testing requires to identify critical points around which input values will be exhaustively selected. All these methods have in common to 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 [12], 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 [13], whereas complex numerical programs fall into the latter [14].

In this paper, we introduce a software testing paradigm, called Symmetric Testing (ST), which aims at testing imperative 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, ST combines automatic test data generation and symmetries 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"* [15]. As a trivial example, consider the 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

[1] with the Euclidian algorithm for example

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[2] :

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

where S_k (resp. S_l) is the symmetric group acting on k elements of x (resp. l elements of y). Symmetric Testing consists in finding test data that violate a given symmetry relation, i.e. finding θ such as for all η :

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

Symmetries relations are generic properties and checking the correctness of programs in regards with these relations is a difficult task, likely undecidable in the general case[3]. However, there are circumstances when testing procedures can be used to check the correctness of properties against programs [2, 14, 16]. 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* [11] 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 the 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.

2 Group theory : notations and selected results

All the basic material on Group theory presented in this section is extracted from [15] and the JS Milne's lecture notes (available online `www.jmilne.org/math/CourseNotes`).

[2] $p(x)$ denotes the vector of values computed by the execution of p with x as input

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

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* $a \circ e = e \circ a = a \quad \forall a \in G$ *(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 symmetric group notion 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, which are named 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.

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. 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_2 to a_3, .. a_{r-1} to a_r 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 [15]).

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 is 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 $id_{G'}$. $G/Ker(\varphi)$ 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 (in fact, it is also a Group). To end this review of the Group theoretic results that we need here, 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 :

$$\begin{array}{ccc} G & \xrightarrow{\varphi} & G' \\ {\scriptstyle surj}\downarrow & & \uparrow{\scriptstyle inj} \\ G/Ker(\varphi) & \xrightarrow{\bar{\varphi}} & Im(\varphi) \end{array}$$

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 imperative programs. Lots of definitions of symmetry have been proposed in various contexts [15]. Some of them can be adapted for our purpose. We briefly discuss two possible choices that can be considered for program testing.

- **Symmetries over values**. A symmetry over values can be expressed as a relation between two program executions when there is a geometric relation (for example, an isometry) between the two input points. As a trivial example, consider a program p which takes two integers as arguments and verifies $p(x, y) = p(-x, -y)$. In this case, 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. Nevertheless, 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 useless for our purpose. Conversely, symmetries over variables can be specified with a very few 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 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** *is a pair $< \theta, \eta >$ such as : $\theta \in S_n, \eta \in S_m$,*

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

Note that every program p satisfies at least the trivial symmetry $< id_{S_n}, id_{S_m} >$ because imperative programs are considered to be deterministics here (two executions with the same input give the same result). Some of the references of the input vector may be leaved 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 is called the *permutable output* set. Such symmetries can be grouped together by the mean 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 outcomes, $\Psi_{k,l}$ is a **symmetry relation** *for p iff*

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

The reason why symmetry relations are required to be group homomorphisms is based on our will to characterize the links between permutable outcomes. This will be made clearer in the following. Note that symmetry relations are very difficult to check when the number of permutable input data increases (because S_k contains $k!$ elements). It is important to see 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

[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

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

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) have often 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

```
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`

(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 $Hom(S_{|A|}, S_{|A|})$, which is only one of the possible symmetry relations represented by $\Psi_{|A|,|A|}$. 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

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

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. Table 1 summarizes the symmetry relations found for these methods. Permutable input and output are indicated in the two central columns. The symbol *Ret* denotes the returned reference or value of the Java method.

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

Table 1: Examples of symmetry relations

The `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 can be derived from the well-known properties of their type [17]. `enumeration` has to satisfy a $\Psi_{|A|,|Ret|}$ symmetry relation but $|Ret|$ is equal to $|A|$ in this case, hence the program has to satisfy the same symmetry relation than the other programs. 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 difficulties. `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 the section 4, we give several examples of non-trivial symmetry relations.

3.3 Symmetric Testing

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

Definition 5 *Symmetric Testing*
Let p be a program and $\Psi_{k,l}$ be a symmetry relation for p, then Symmetric Testing aims at finding a triple $(x, p(x), p(\theta.x))$ such as $p(\theta.x) \neq \Psi_{k,l}(\theta).p(x)$
If found, such a triple $< x, p(x), p(\theta.x) >$ represents a counter-example for the symmetry relation. This shows that at least one of the two test data x and $\theta.x$ reveals a fault in p. So, given a set of test data and a symmetry relation, we get a naive procedure that can check whether program outcomes are incorrects. It is required to compute all the permutations of the permutable input of a vector x, to execute p with all these input data and to check whether the outcome vectors are equals to a permutation of the vector returned by $p(x)$. This principle can be illustrated by the following commutative diagram.

$$\begin{array}{ccc} x & \xrightarrow{\theta} & \theta.x \\ {\scriptstyle p}\downarrow & & \downarrow{\scriptstyle p} \\ p(x) & \xrightarrow[\Psi_{k,l}(\theta)]{} & \Psi_{k,l}(\theta).p(x) \end{array}$$

It is only a necessary condition for the correctness of p w.r.t. its specification because incorrect implementations of p may also satisfy the same symmetry relation. Note that this procedure is independant of the test set being used. In fact, it leaves the tester the possibility to use any automatic test data generators, because it is not required to produce an oracle for the expected outcomes. However, it should be recalled that the number of possible permutations in S_k is $k!$, leading to an impractical number of program calls whenever k increases. No hypothesis are made on the type of the permutable input data (Object references, integers, ...) but checking the equality between floating point values might be hazardous because programs that manipulate such variables depend strongly on the evaluation order of expressions. So, the equality relation between the computed results should be relaxed for these types.

3.4 Checking a given symmetry relation

We turn now on a more practical procedure that checks whether a program satisfies a given symmetry relation.

3.4.1 Reducing the number of permutations

In order to limit the number of program calls, we propose to check only two permutations when checking a symmetry relation. In fact, by using the proposition 1, we known that only two permutations are required to generate S_k. As a consequence, we get the following proposition :

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

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

Proof: $\Longleftarrow$. $\tau \in S_k$ and $\sigma \in S_k$, hence taking $\theta = \tau$ and $\theta = \sigma$ yields the expected result.
$\Longrightarrow$. 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.

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 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 $\Psi_{k,l}$, it is possible to determine precisely the mapping properties of $\Psi_{k,l}$ just by looking at the link between the two generators $\Psi_{k,l}(\tau)$ and $\Psi_{k,l}(\sigma)$, but this is outside the scope of this paper.

3.4.2 Checking only τ and σ

We will provide here a procedure to check whether a program p satisfies the two symmetries $< \tau, \Psi_{k,l}(\tau) >$ and $< \sigma, \Psi_{k,l}(\sigma) >$ over an input space D. In fact, it is 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 $x \in D$. To achieve such a goal by the mean of testing, we propose to use an local exhaustive test data generator [10]. However, other approaches, that make use of a (semi-)proving technique can be followed [14, 16] and are discussed in the section 5 of this paper. In general, input domains are infinites, as illustrated by the **replaceAll** method (Fig.1), which takes an unbounded list as first argument. In this case, a local exhaustive test data generator will enumerate all the possible lists until a selected size will be reached. So, any proof of symmetry relation satisfaction would be limited to the input domain being exhaustively explored. Fortunately, the limitation of the input space allows the approach to remain practical.

The keypoint of our approach is that we just have to know whether $p(\tau.x)$ and $p(\sigma.x)$ are permutations of $p(x)$. As previously said, a precise knowledge of $\Psi_{k,l}(\tau)$ and $\Psi_{k,l}(\sigma)$ is not

required here because $\Psi_{k,l}$ represents an entire class of symmetries. The procedure shown in Fig.2 takes a program p and the input space D as arguments and returns the first found triple $< x, p(x), p(\theta.x) >$ that violates the symmetry relation, among the portion of the input space being explored. If such a counter-example cannot be found, this proves that p satisfies not only the two selected symmetries but also all the permutations of S_k of the input vector in the input space D. Note that some test data are not required to be examined :

```
while(D ≠ ∅) {
      pick up x ∈ D,
      D := D \ {x};
      if( p(τ.x) is not a permutation of p(x) )
         return < x, r, p(τ.x) >
      if( p(σ.x) is not a permutation of p(x))
         return < x, r, p(σ.x) >
      }
      return("Check complete");
```

Figure 2: A procedure for Symmetric Testing

test data of the form $x = (v, v, .., v)$ can be eliminated from D because any permutation will leave x unchanged. Note also that non-permutable input data may be leaved constants because these input data do not play any role in the symmetry relation. However, by doing these, we restrict the proof when the procedure explores the complete domain.

3.5 Discussion

As expected, terminaison of the previous procedure cannot be guaranted. Although the input space of the program p 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 of its finite input space, ST is guaranted either to find a counter-example of the symmetry relation if there exists one, or to show that p satisfies the symmetry relation. However, the input space of p may have to be fully explored in the worst case. Let $D_1 \times D_2 \times .. \times D_n$ be the finite input space of p and let d be the number of elements of the greatest domain 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 smallest as possible by limiting the size of the domains of permutable input data. Note also that the number of program calls 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 an extracted symmetry relation is actually a group homomorphism, for some programs. 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 can guess that this program has to satisfy a $\Psi_{|A|,|Ret|}$ symmetry relation. However, the vector returned by the program is not sorted and its elements order depends strongly on the algorithm used. Further, it is 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. An approach for

```
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_R, cur_V;
  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_R = (Integer)c.max(R)    ; // max value of R
      cur_V = (Integer)V.elementAt(i); // current value of V

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

Figure 3: The `min_nb` program

this problem would be to use symmetry relations over compositions of programs. For example here, composing `min_nb` with a sorting program of the resulting vector yields a $\Psi_{|A|,1}$ symmetry relation for the composition.

4 Experimental results

4.1 Implementation

We implemented ST with the help of the primitive operations of the Java unit testing tool *Roast* [11]. 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 the latter to implement our local exhaustive test data generator. Roast supports test templates (Perl macros) to compare actual outcomes to predicted ones. We used these templates in combination with our Java methods to check whether a computed outcome was 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 circumstances where ST checks 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 [14]) intended to compute the median of three integers, and the well–known triangle classification program `trityp` [18]. This program takes three 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 19.

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 [14] 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[7] have been removed from the set of experiments, because they cannot be revealed by the mean of testing [18]. All the mutants are available at the url `www.irisa.fr/lande/gotlieb`

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

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` has to satisfy a $\Psi_{|A|,|Ret|}$ symmetry relation. Finally, Table 1 contains the expected symmetry relations for `replaceAll`, `copy` and `sort`.

4.3 Experimental results and Analysis

Mutants	x	$p(x)$	$p(\sigma.x)$	$p(\tau.x)$	rtime (sec)
`min_nb_1`	vec:[1,1,0], nb:2	**[0,0]**	[1,0]	**[0,0]**	6.8
`min_nb_2`	vec:[1,0], nb:1	[0]	**[1]**	[0]	4
`min_nb`	3367220 test data				57

Table 2: Results for `min_nb`

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`	999900 test data				9,4

Table 3: Results for `GetMid`

All the computations were performed on a 1.8Ghz Pentium 4 personal computer by using the version 1.3 of *Roast*.

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 2, 3, and 4. If found, a test datum x that violates the symmetry relation is given. The values of $p(x)$, $p(\tau.x)$ and $p(\sigma.x)$ are given in each of three interiors columns of the tables. In the case where a test datum is found , the mutant is said to be killed by the symmetry relation. 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.

Test data are found for killing the two mutants of `min_nb` and `GetMid`. This illustrates the capacity of ST to reveal two difficult class of faults (missing path error and infeasible path) on these programs. 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

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 found				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 found				9.3
`trityp_10`	(2,1,1)	**equ**	illeg	illeg	0.2
`trityp_11`	not found				9.2
`trityp_12`	(2,2,1)	**illeg**	iso	**illeg**	0.2
`trityp_13`	not found				9.2
`trityp_14`	not found				9.6
`trityp_15`	not found				9.4
`trityp_16`	not found				10
`trityp_17`	(3,2,1)	**sca**	illeg	**sca**	0.2
`trityp_18`	not found				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 found				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 found				10
`trityp`	999900 test data tried				9,6

Table 4: Results for `trityp`

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

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 2, 3 and 4. 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.

Mutants	x	$p(x)$	$p(\sigma.x)$	$p(\tau.x)$	rtime (sec)
`copy`	168361 test data				14.4
`sort`	168361 test data				5.4
`replaceAll`	67344400 test data				38240.5

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

5 Related work

The idea of checking the computed results of a program by using several input data is not new. Ammann and Knight [19] 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 [20] 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 [21] 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 [14] 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 [16], 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 allow us to prove properties about the program and its test data. We believe that symmetry relations would be easily expressed as 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 so a way to test programs at the integration level.

References

[1] L. Clarke, "A System to Generate Test Data and Symbolically Execute Programs," *IEEE Trans. on Soft. Eng.*, vol. SE-2, pp. 215–222, September 1976.

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

[3] B. Korel, "Automated Software Test Data Generation," *IEEE Trans. on Soft. Eng.*, vol. 16, pp. 870–879, Jul. 1990.

[4] 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.

[5] G. Bernot, M. C. Gaudel, and B. Marre, "Software testing based on formal specifications: a theory and a tool," *Soft. Eng. Jour.*, vol. 6, no. 6, pp. 387–405, 1991.

[6] E. Weyuker, T. Goradia, and A. Singh, "Automatically generating test data from a Boolean specification," *IEEE Trans. on Soft. Eng.*, vol. 20, pp. 353–363, May 1994.

[7] J. M. Voas, "Certifying off-the-shelf software components," *Computer*, vol. 31, pp. 53–59, June 1998.

[8] J. Duran and S. Ntafos, "An evaluation of random testing," *IEEE Trans. on Soft. Eng.*, vol. 10, pp. 438–444, Jul. 1984.

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

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

[11] N. Daley, D. Hoffman, and P. Strooper, "A framework for table driven testing of Java classes," *Soft. Prac. and Exper.*, vol. 32, pp. 465–493, Apr. 2002.

[12] E. Weyuker, "On testing non-testable programs," *Computer Journal*, vol. 25, no. 4, pp. 465–470, 1982.

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

[14] 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)*, pp. 191–195, 2002.

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

[16] A. Gotlieb and B. Botella, "Automated metamorphic testing," PI 1516, IRISA Tech. Report - Rennes, France, 2003.

[17] P. Wadler, "Theorems for free!," in *FPCA'89, London, England*, pp. 347–359, ACM Press, Sept. 1989.

[18] R. A. Demillo and A. J. Offut, "Constraint-Based Automatic Test Data Generation," *IEEE Trans. on Soft. Eng.*, vol. 17, pp. 900–910, Sep. 1991.

[19] P. E. Ammann and J. C. Knight, "Data diversity: An approach to software fault tolerance," *IEEE Trans. on Computers*, vol. 37, no. 4, pp. 418–425, 1988.

[20] M. Blum and S. Kannan, "Designing programs that check their work," *Jour. of the Assoc. for Computing Machinery*, vol. 42, pp. 269–291, Jan. 1995.

[21] 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)*, pp. 172–178, 2001.

Contents

1 Introduction **3**

2 Group theory : notations and selected results **4**

3 Principle of Symmetric Testing **6**
3.1 Symmetry relations 6
3.2 Examples 8
3.3 Symmetric Testing 9
3.4 Checking a given symmetry relation 10
3.4.1 Reducing the number of permutations 10
3.4.2 Checking only τ and σ 11
3.5 Discussion 12

4 Experimental results **13**
4.1 Implementation 13
4.2 Experiments with ST 14
4.3 Experimental results and Analysis 14
4.3.1 Revealling faults with ST 15
4.3.2 Checking that a program satisfies a given symmetry relation 15

5 Related work **16**

6 Perspectives **17**



Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399