Verification by Testing for Recursive Program Schemes

Daniel Le Métayer, Valérie-Anne Nicolas, Olivier Ridoux

Verification by Testing for Recursive Program Schemes

Daniel Le Métayer, Valérie-Anne Nicolas, and Olivier Ridoux

Irisa/Inria, Campus de Beaulieu, F-35042 Rennes Cedex, France
{lemetayer|vnicolas|ridoux}@irisa.fr

Abstract. In this paper, we explore the testing-verification relationship with the objective of mechanizing the generation of test data. We consider program classes defined as recursive program schemes and we show that complete and finite test data sets can be associated with such classes, that is to say that these test data sets allow us to distinguish every two different functions in these schemes. This technique is applied to the verification of simple properties of programs.

1 Introduction

The only way to improve confidence that a program really achieves its intended purpose is to confront it with other means of expressing this purpose. Typically, such means can be properties that the program is supposed to satisfy or test data sets with oracles characterizing the expected behavior of the program. However, despite the fact that they both contribute to the same final objective, verification and testing remain two independent research areas and we haven't seen much cross-fertilization between them so far (except in specific domains like protocol design). We believe that testing can be formalized in a fruitful way in order to cooperate harmoniously with verification. Our goal in this paper is to support this claim by putting forward a technique for the automatic verification of (simple) properties of programs that relies both on program analysis and program testing.

Since the systematic construction of complete test data sets is out of reach in general, we propose to tackle this problem by restricting it to classes of programs and properties. The key idea underlying this work is a transposition to recursive programs of the well-known property that $n+1$ values are sufficient to identify a polynomial of degree n. We introduce a hierarchy of common recursive program (or property) schemes which define infinite classes of functions. The main result of the paper shows that each scheme can be associated with a finite complete test data set. The test data sets are complete in the sense that they are sufficient to distinguish any two distinct functions in the class.

This result essentially provides a way to reduce program equivalence to program testing (with respect to a given hierarchy of program schemes). In this paper, we show how this technique can also be used in conjunction with abstract interpretation to prove simple properties of programs. One must have in

mind that we do not want to prove the complete correctness of a program via testing. Instead, partial correctness properties will be proved via testing. More than on a list of scheme hierarchies, we want to dwell on a new method for proving program properties, relying on the association of test data sets to schemes, independently of the syntactic shape of these schemes.

In the following section we introduce a simple hierarchy of unary recursive schemes to illustrate our ideas, and we proceed with the technical contribution of the paper, which makes it possible to associate schemes with complete test data sets. Section 3 extends this result to some more complex unary and binary scheme hierarchies our method can deal with. Section 4 describes the use of these results to prove properties of programs, and Section 5 shows the relevance of our method on some application examples. Section 6 sketches the more general context of this work.

2 A Simple Hierarchy of Recursive Schemes

The process that we describe in Section 4 relies on our ability to associate complete test data sets with schemes. In this section, we provide a sketch of the proof of this result for a simple hierarchy of recursive schemes introduced in Definition 2. The interested reader can find in [NIC98] the definition of the whole framework.

A test data set is complete with respect to a scheme if it allows to decide the equality of any two functions of the scheme. This is stated in Definition 1.

Definition 1
D is a complete test data set for a class C of functions if and only if

$$\forall f \in C,\ \forall g \in C,\ (f \neq g \Rightarrow \exists x \in D \ s.t.\ (f(x) \neq g(x)))$$

In other words, D is a complete test data set for a class C of functions if and only if $\forall f \in C,\ \forall g \in C,\ ((\forall x \in D\ ,\ f(x) = g(x)) \Rightarrow (f = g))$.

Definition 2

$$\begin{aligned} S_1^1 &= \{\lambda x.\ Succ^k(x) \mid k \in I\!N\} \bigcup \{\lambda x.\ k \mid k \in I\!N\} \\ S_{i+1}^1 &= \{f \mid f(0) = k \\ &\qquad f(n+1) = g(f(n))\ ,\ k \in I\!N\ ,\ g \in S_i^1\} \end{aligned}$$

The scheme of lower level in the hierarchy S^1 (S_1^1) is made of all the successor and all the constant unary functions on $\mathbb{N}$. Functions in scheme S_i^1 follow a recursive pattern where the result is a constant in the basic case, and a composition of a recursive call with a function belonging to a scheme of lower level in the hierarchy in the recursive case. The schemes S_i^1 are inspired by previous work on inductive data types and the associated inductive program schemes [PDM89]. These schemes are called unary schemes because they allow the definition of some unary functions (in Section 3, we consider some binary schemes to express functions on pairs of natural numbers). The first observation to be made about

functions of the S_i^1 schemes is that they can be split into two different classes of functions: the first class contains increasing functions,[1] in fact even only *separable* functions in a sense made precise below. Functions of the second class, that we call *periodic* have a finite codomain. For example, in S_1^1, the class of separable functions is the set of successor functions, and the class of periodic functions is the set of constant functions.

Definition 3
Two functions f and g from $\mathbb{N}$ to $\mathbb{N}$ are said α-separable if and only if there exists α open intervals $I_1, \ldots, I_\alpha$ with $I_i =]A_i, A_{i+1}[$, $A_1 = -1$, $A_{\alpha+1} = \infty$, $i > j \Rightarrow A_i > A_j$ and

$$\forall i \in [1, \alpha], (even(i) \Rightarrow (\forall x \in I_i, (f(x) < g(x)) \quad and \quad f(A_{i+1}) \geq g(A_{i+1})))$$

$$\forall i \in [1, \alpha], (odd(i) \Rightarrow (\forall x \in I_i, (f(x) > g(x)) \quad and \quad f(A_{i+1}) \leq g(A_{i+1})))$$

Two functions f and g are α-separable if it is possible to decompose $\mathbb{N}$ into α intervals $I_1, \ldots, I_\alpha$ such that one of the two functions is strictly greater than the other on each interval. In the definition, α is the least value satisfying this property. The relevance of α-separability for testing stems from Property 1, which follows directly from Definition 3 ($card(D)$ denotes the cardinality of the set D):

Property 1
If f and g are α-separable and D is a subset of $\mathbb{N}$ such that $\forall x \in D, f(x) = g(x)$ then $card(D) < \alpha$.

Property 1 means that two α-separable functions can at most be equal on $\alpha - 1$ values. It is thus necessary and sufficient to test them on α values to distinguish them. For example, $\{0, \ldots, \alpha - 1\}$ is a complete test data set for any pair of α-separable functions, and so for any set of α-separable functions.

We now turn our attention to periodic functions. A periodic function begins returning some distinct values, afterwards it has a cyclic behavior.

Definition 4
A function f from $\mathbb{N}$ to $\mathbb{N}$ is said δ-periodic if and only if $\exists \lambda \geq 0$, $\exists \pi > 0$, s.t. $(\lambda + \pi \leq \delta)$ and

$$\begin{aligned} &\forall x \geq \lambda \;.\; \forall y \geq \lambda \;.\; (x \bmod \pi = y \bmod \pi \Rightarrow f(x) = f(y)) \\ \wedge\; &\forall x < \lambda + \pi \;.\; \forall y < \lambda + \pi \;.\; (f(x) = f(y) \Rightarrow x = y) \end{aligned}$$

where mod is the modulo function.

Notice that the whole behavior of a δ-periodic function can be determined by just knowing its behavior on the first $(\lambda + \pi + 1)$ natural numbers (the λ first values give the initial non-cyclic behavior, the following π values give the cyclic behavior and the last one the value of the period).

The relevance of periodicity for testing is expressed by the following property, which is a direct consequence of Definition 4:

[1] A unary function f is increasing if $\forall x. f(x) \geq x$.

Property 2
If C is a set of δ-periodic functions then $\{0, \ldots, \delta\}$ is a complete test data set for C.

The notions of separability and periodicity can be generalized to sets of functions and the above results can be gathered as follows:

Property 3
If a class C of functions is the union of a class C_1 of δ-periodic functions and of a class C_2 of increasing α-separable functions then $\{0, \ldots, \mu\}$ is a complete test data set for C with $\mu = max(\delta, \alpha - 1)$.

This result can be proven using Property 1 and Property 2 and showing that the test values necessary to distinguish two functions from C_1 or C_2 respectively are sufficient to distinguish a function from C_1 and a function from C_2. It relies mainly on the fact that functions of C_2 are injective whereas functions of C_1 are not (the test of a function of C_1 on the value δ will yield a value already obtained by its test on the previous test data).

The observation made at the beginning of this section can now be stated formally:

Property 4
$\forall i,\ S_i^1 = C_i^1 \bigcup C_i^2$ *with* C_i^1 *i-periodic and* C_i^2 *increasing i-separable.*

This property is proven by induction on i, relying on lemmas which establish the propagation of separability and periodicity through the hierarchy of schemes [NIC98].

Property 4 and Property 3 joined together allow us to derive Property 5, which is the fundamental property on which all the other results presented in this paper rely.

Property 5
$\forall i,\ \{0, \ldots, i\}$ *is a complete test data set for* S_i^1.

We have considered only one simple hierarchy of schemes so far. It has mainly allowed us to introduce the key ideas of our method. In the next section, we show how the result presented above can be used to derive complete test data sets for other unary and binary schemes.

3 Extension to Some Other Scheme Hierarchies

In this section, we first present a larger hierarchy of program schemes where the recursive call is not direct (the recursive call to the function does not need to be applied to the argument n directly) and then consider some binary functions in Section 3.2. We can benefit from the result presented in the previous section to associate test data sets to these more complex scheme hierarchies.

3.1 A More Complex Unary Scheme Hierarchy

In this section, we consider a slight generalization of the unary scheme hierarchy S^1 presented in Section 2. The difference is that our new $\{S_n^2,\ n \in \mathbb{N}\}$ hierarchy allows recursive calls which do not need to apply directly to the argument n:

Definition 5

$$S_i^2 = \{f \mid f(0) = k \\ f(n+1) = g(f(h(n)))\ ,\ k \in I\!N,\ g \in S_i^1,\ h \in S_i^1\}$$

One first difference with respect to S_n^1 schemes is that S_n^2 do not contain only total functions. Depending on the value of the function h, the recursive call to f can apply to an argument greater than the initial one. Consequently, we begin with characterizing a definition domain for the functions of S_n^2.

Property 6
Consider the following definition of f

$$f(0) = k \\ f(n+1) = g(f(h(n)))$$

with $g \in S_i^1$ and $h \in S_i^1$.
Function f is total if and only if f is defined on each value of the set $\{1, \ldots, i\}$.

For a function h which is i-periodic, the resulting function f, when it terminates, is neither periodic (as defined in Definition 4) nor increasing α-separable. However, it has a cyclic behavior (just like periodic functions) but it may repeat certain values where periodic functions ensure distinct values. Nevertheless, the following property allows us to distinguish this new kind of function from the periodic and increasing α-separable ones.

Property 7
Consider the following definition of f

$$f(0) = k \\ f(n+1) = g(f(h(n)))$$

with $h \in S_i^1$ i-periodic and $g \in S_i^1$.
If f is total then the set $\{0, \ldots, 2i\}$ is sufficient to determine f (i.e. to determine the value of its period and its behavior on a period).

Gathering the above results, we can prove that: $\forall n\ .\ S_{n+1}^1 \subset S_n^2$. We are now able to derive a complete test data set for the total functions of the S_n^2 scheme.

Property 8
$\forall n\ .\ \{0, \ldots, 2n\}$ is a complete test data set for the total functions of the S_n^2 scheme.

3.2 Some Binary Scheme Hierarchies

All the functions considered so far are unary. We now turn our attention to binary functions (n-ary functions can be treated in a similar way). The following schemes capture some common recursive definition patterns:

Definition 6

$$\begin{aligned} B^1(X_1, X_2) = \{f \mid\ & f(0,m) = h(m) \\ & f(n+1,m) = g(f(n,m)) \ ,\ g \in X_1,\ h \in X_2\} \end{aligned}$$

$$\begin{aligned} B^2(X_1, X_2) = \{f \mid\ & f(0,m) = h(m) \\ & f(n+1,m) = f(n,g(m)) \ ,\ h \in X_1,\ g \in X_2\} \end{aligned}$$

$$\begin{aligned} B^3(X) \quad = \{f \mid\ & f(0,m) = k \\ & f(n+1,m) = g(m,f(n,m)) \ ,\ k \in I\!N,\ g \in X\} \end{aligned}$$

Binary schemes B^1 and B^2 are parameterized by the unary schemes associated with the functions occurring in their definitions, and binary scheme B^3 is parameterized by one of the binary schemes of the definition. Property 4 shows that each unary scheme S_i^1 is made of two classes of functions corresponding to δ-periodic functions and increasing α-separable functions. In order to establish the required results for binary schemes, we need to consider these two subclasses separately. We call them P_i and I_i (for Periodic and Increasing respectively).

Definition 7

$$\begin{aligned} P_1 \quad &= \{\lambda x.\ k \mid k \in I\!N\} \\ P_{i+1} &= \{f \mid f(0) = k \\ &\qquad f(n+1) = g(f(n)) \ ,\ k \in I\!N,\ g \in P_i\} \end{aligned}$$

$$\begin{aligned} I_1 \quad &= \{\lambda x.\ (x+k) \mid k \in I\!N\} \\ I_{i+1} &= \{f \mid f(0) = k \\ &\qquad f(n+1) = g(f(n)) \ ,\ k \geq i-1,\ g \in I'_i\} \end{aligned}$$

$$\begin{aligned} I'_1 \quad &= \{\lambda x.\ (x+k) \mid k > 0\} \\ I'_{i+1} &= \{f \mid f(0) = k \\ &\qquad f(n+1) = g(f(n)) \ ,\ k \neq 0,\ g \in I'_i\} \\ &\bigcup \{f \mid f(0) = 0 \\ &\qquad f(n+1) = g(f(n)) \ ,\ g \in I''_i\} \end{aligned}$$

$$\begin{aligned} I''_1 \quad &= \{\lambda x.\ (x+k) \mid k > 1\} \\ I''_{i+1} &= \{f \mid f(0) = k \\ &\qquad f(n+1) = g(f(n)) \ ,\ k > 1,\ g \in I'_i\} \\ &\bigcup \{f \mid f(0) = 0 \\ &\qquad f(n+1) = g(f(n)) \ ,\ g \in I''_i\} \\ &\bigcup \{f \mid f(0) = 1 \\ &\qquad f(n+1) = g(f(n)) \ ,\ g \in I''_i\} \end{aligned}$$

The sub-class of functions I'_i is used to exclude the possibility that $g = id$ in the definition of I_{i+1}. The intermediate scheme I''_i allows us to remove syntactically the different possible definitions of the identity function and the successor function.

We can now state the main results concerning binary schemes.

The following properties allow us to associate a complete test data set to most of the parameterized schemes of Definition 6. The first property concerns the scheme $B^1(X_1, X_2)$.

Property 9
$(\{0\} \times \{0, \ldots, j-1\}) \cup (\{1\} \times \{0, \ldots, i-1\})$ *is a complete test data set for* $B^1(I_i, I_j)$.
$\{0, \ldots, i\} \times \{0, \ldots, j\}$ *is a complete test data set for* $B^1(I_i, P_j)$.
$\{0, \ldots, i+1\} \times \{0, \ldots, j\}$ *is a complete test data set for* $B^1(P_i, P_j)$.

The next properties establish the same kind of results as Property 9 for the schemes $B^2(X_1, X_2)$ and $B^3(X)$ respectively.

Property 10
$(\{0\} \times \{0, \ldots, i-1\}) \cup (\{1\} \times \{0, \ldots, j-1\})$ *is a complete test data set for* $B^2(I_i, I_j)$.
$(\{0\} \times \{0, \ldots, i-1\}) \cup (\{1\} \times \{0, \ldots, j\})$ *is a complete test data set for* $B^2(I_i, P_j)$.

Property 11
$\{(0,0)\} \cup (\{1\} \times \{1, \ldots, i\}) \cup (\{1, \ldots, j\} \times \{1\})$ *is a complete test data set for* $B^3(B^1(I_i, I_j))$.

The whole proof of these properties is detailed in [NIC98].

Note that we have proposed some complete test data sets for the more common instantiations of the binary schemes. The ones which have not been treated correspond to unusual combinations in real programs. For example, the scheme $B^1(P_i, I_j)$ is defined by:

$$B^1(P_i, I_j) = \{f \mid f(0, m) = h(m) \\ f(n+1, m) = g(f(n, m)) \ , \ g \in P_i, \ h \in I_j\}$$

Actually, it is the recursive application of a periodic function, ending with a call to an increasing function.

Furthermore, it is interesting to notice that these unusual combinations lead to not easily testable program schemes. For the scheme $B^1(P_i, I_j)$ for example, the difficulty comes from the fact that there is no way to ensure that a finite number of arguments of the increasing function h produces a set of results covering the domain of g (modulo its periodicity).

In Section 4 we illustrate the use of the results presented in Sections 2 and 3 to derive complete test data sets to prove properties of programs.

4 Verification of Properties Using Complete Test Data Sets

Let us consider a simple program for reversing lists, written in a first-order functional language. Function *Rev* reverses a list, and function *App* adds an element (the second argument) at the end of a list (the first argument):

$$\begin{aligned} Rev(nil) &= nil \\ Rev(n:l) &= App(Rev(l), n) \end{aligned}$$

$$\begin{aligned} App(nil, m) &= m : nil \\ App(n:l, m) &= n : App(l, m) \end{aligned}$$

One property that a *reverse* program must satisfy is the fact that the length of its result must be equal to the length of its argument. In order to check this property, we have to express it as a function computing the expected length of the result of *Rev* from the length of its argument. Obviously, this function is the identity Id here.

The next stage consists in deriving an abstract version of *Rev* computing the length of the result of *Rev* from the length of its argument. Though the choice of an actual abstraction is dependent on the property to be proved and is not automatic, its application to the program can be achieved automatically, applying the abstract interpretation technique [CC77]. We choose the natural numbers $\mathbb{N}$ as the abstract domain and the abstraction function associates each list with its length. Non-list values are abstracted in a one point domain because they are not relevant to the analysis considered here. Rather than keeping these dummy arguments, we abstract a function with some non-list arguments into a function with fewer arguments. The primitives of interest here are basically the list constructor (denoted by ":" in our programming language) and the empty list nil. Not surprisingly, their abstractions are, respectively, the successor function $Succ = \lambda x.(x + 1)$ and the constant $\lambda x.0$. Thus, we get the following abstract interpretation *Lrev* for the *Rev* function.[2]

$$\begin{aligned} Lrev(0) &= 0 \\ Lrev(n+1) &= Lapp(Lrev(n)) \end{aligned}$$

$$\begin{aligned} Lapp(0) &= 1 \\ Lapp(n+1) &= Succ(Lapp(n)) \end{aligned}$$

Now we are left with comparing *Lrev* with the identity function. Of course, in this simple case we could rely on symbolic manipulations and inductive proof techniques to show that *Lapp* is equivalent to the function *Succ* and then replace it in the body of *Lrev*. But it is well known that mechanizing these techniques is difficult in general. What we do instead is to analyze the definitions of *Lrev* and

[2] Note that the second argument of *App* is not of type list, which explains why *Lapp* has a single argument

Id to derive a complete test data set to decide their equivalence (or provide a counter-example if they turn out to be different). The goal of this simple syntactic analysis (called *scheme inference*) is to identify the scheme (or skeleton) of each function and find its position in the hierarchy of schemes.

We do not dwell on the scheme inference algorithm here. It is achieved by pattern matching on the structure of the definition of the functions and relies on a set of inference rules akin to a type inference system. Details about its implementation can be found in [NIC98]. For example, the definition of $Lrev$ matches the generic pattern of the schemes S_i^1 defined in Section 2 :

$$\begin{aligned} f(0) &= k \\ f(n+1) &= g(f(n)) \end{aligned}$$

with $k = 0$ and $g = Lapp$. The definition of $Lapp$ matches the generic pattern with $k = 1$ and $g = Succ$, which belongs to scheme S_1^1. So $Lapp$ is associated with scheme S_2^1 and $Lrev$ is associated[3] with scheme S_3^1.

It is not difficult to show that the schemes S_i^1 define a hierarchy which is strictly increasing with respect to set inclusion (in other words $i < i' \Rightarrow S_i^1 \subset S_{i'}^1$). Id belongs to the scheme S_1^1, so we have to take the least upper bound of S_1^1 and S_3^1, which is S_3^1. This shows that it is enough to test Id and $Lrev$ on the values 0, 1, 2, and 3 to decide their equality. In order to express these values in terms of the original program, we just have to use the correspondence relation between the abstract and the concrete domains. Here, this means that it is enough to test the program Rev on four randomly chosen lists of lengths 0, 1, 2, and 3 to decide if Rev always returns lists of the same length as its argument. In practice, one can prefer to choose lists made of distinct elements, which have a greater power of discrimination and can allow the detection of bugs apart from the property of interest.

To summarize, the four main stages of the test data derivation process are the following:

Abstraction of the program: Program → Abstract program

↓

Scheme inference: Abstract program → $Scheme_1$
Property → $Scheme_2$

↓

Abstract test data inference: $Lub(Scheme_1, Scheme_2)$ → Abstract test data

↓

Concrete test data generation: Abstract test data → Concrete test data

[3] Since both Id and $Lrev$ are semantically equal to the identity function, we could have expected that they are just associated with S_1^1, but we have to keep in mind that this knowledge is not available at this stage (in fact, it is exactly what we are trying to prove).

5 The Method at Work

In the previous section, we have used the *Reverse* program and the Id property to explain the different stages of the method introduced in Section 2. In this section, we present further examples illustrating it. We are still considering list functions, and our aim is to prove properties about the lengths of their arguments and results. So, we are using the same abstraction as the one used in the previous section. We start with a replacement program, which is supposed to return a list whose length is the product of the lengths of its arguments ; we continue with two sort programs returning a list of the same length as their argument.

5.1 A Replacement Program

Let us consider a program *Rep* replacing each element of its first list argument by its second argument. This program can be written:

$$\begin{array}{ll} Rep(nil, l_2) & = nil \\ Rep(n : l_1, l_2) & = Apnd(l_2, Rep(l_1, l_2)) \end{array}$$

$$\begin{array}{ll} Apnd(nil, l_2) & = l_2 \\ Apnd(n : l_1, l_2) & = n : Apnd(l_1, l_2) \end{array}$$

We would like to check that the length of the result of *Rep* is the product of the lengths of its argument. The product function can be written as follows:

$$\begin{array}{ll} Mult(0, m) & = 0 \\ Mult(n + 1, m) & = Add(m, Mult(n, m)) \end{array}$$

$$\begin{array}{ll} Add(0, m) & = m \\ Add(n + 1, m) & = Succ(Add(n, m)) \end{array}$$

The abstract interpretation outlined in Section 4 returns the following abstract function for *Rep*:

$$\begin{array}{ll} Lrep(0, n_2) & = 0 \\ Lrep(n_1 + 1, n_2) & = Lapnd(n_2, Lrep(n_1, n_2)) \end{array}$$

$$\begin{array}{ll} Lapnd(0, n_2) & = n_2 \\ Lapnd(n_1 + 1, n_2) & = Succ(Lapnd(n_1, n_2)) \end{array}$$

The scheme inference algorithm associates the scheme $B^3(B^1(I_1, I_1))$ with both *Lrep* and $Mult$ (the scheme returned for *Lapnd* and *Add* is $B^1(I_1, I_1)$ since $\lambda x.x$ and *Succ* both belong to I_1). So $D = \{(0, 1),\ (1, 0),\ (1, 1)\}$ is a complete test data set for *Lrep* and it is sufficient to test *Rep* on lists of the lengths indicated by D to decide if the length of its result is indeed the product of the lengths of its arguments.

5.2 A Selection Sort Program

A selection sort program $Selsort$ can be defined as follows in our functional programming language:

$$\begin{array}{ll} Selsort(nil) & = nil \\ Selsort(n:l) & = \textbf{let } (n_1, l_1) = Maxl(l, n) \\ & \quad \textbf{in } n_1 : Selsort(l_1) \end{array}$$

$$\begin{array}{ll} Maxl(nil, m) & = (m, nil) \\ Maxl(n:l, m) & = \textbf{let } (n_1, l_1) = Maxl(l, n) \\ & \quad \textbf{in } (Max(m, n_1), Min(m, n_1) : l_1) \end{array}$$

The abstract interpretation returns the following abstract function for this program:

$$\begin{array}{ll} Lselsort(0) & = 0 \\ Lselsort(n+1) & = Succ(Lselsort(Lmaxl(n))) \end{array}$$

$$\begin{array}{ll} Lmaxl(0) & = 0 \\ Lmaxl(n+1) & = Succ(Lmaxl(n)) \end{array}$$

Note that $Lmaxl$ is of arity 1 since $Maxl$ has only one list argument, which allowed us to simplify the function by removing the *let* expression. The scheme inference returns the scheme S_2^2 for $Lselsort$ ($Lmaxl$ being associated with the scheme S_2^1). Since the identity function belongs to S_1^2, $\{0, 1, 2, 3\}$ is a complete test data set to decide if $Lselsort = Id$; it is thus sufficient to test the program $Selsort$ on four randomly chosen lists of length 0, 1, 2, and 3 to decide if it possesses the required property.

Note that a standard proof of this property (that program $Selsort$ respects the length) would have used an induction on the length of the argument. So, it would have required a non-trivial proof technique, while our method boils down to the comparison of test outputs.

However, it should be clear that only errors which have an impact on the length of the result are guaranteed to be detected using this test data set (since it is the very purpose of this test). Let us imagine for example that we have forgotten the introduction of the value $Min(m, n_1)$ in the result of $Maxl$:

$$\begin{array}{ll} Maxl(nil, m) & = (m, nil) \\ Maxl(n:l, m) & = \textbf{let } (n_1, l_1) = Maxl(l, n) \\ & \quad \textbf{in } (Max(m, n_1), l_1) \end{array}$$

The mistake would be revealed through the application of $Selsort$ to a list of length 2. But if we had inadvertently replaced Max by Min, the bug would not necessarily be captured by a test data set including random list of lengths 0, 1, 2, and 3. In this case however, if we consider the extra condition that the lists of the test data set contain different elements, then the bug is detected.

5.3 An Insertion Sort Program

Insertion sort can be defined as follows:

$$\begin{aligned} Insort(nil) &= nil \\ Insort(n:l) &= Insert(Insort(l), n) \end{aligned}$$

$$\begin{aligned} Insert(nil, m) &= m : nil \\ Insert(n:l, m) &= Max(n, m) : (Insert(l, Min(n, m))) \end{aligned}$$

The abstract interpretation of this program returns the following abstract function:

$$\begin{aligned} Linsort(0) &= 0 \\ Linsort(n+1) &= Linsert(Linsort(n)) \end{aligned}$$

$$\begin{aligned} Linsert(0) &= 1 \\ Linsert(n+1) &= Succ(Linsert(n)) \end{aligned}$$

The scheme inferred for $Linsort$ is S_3^1, so $Insort$ has the same complete test data set as $Selsort$. As an illustration of the accurateness of this test data set, let us consider a wrong definition of $Insert$:

$$\begin{aligned} Insort'(nil) &= nil \\ Insort'(n:l) &= Insert'(Insort'(l), n) \end{aligned}$$

$$\begin{aligned} Insert'(nil, m) &= m : nil \\ Insert'(n:l, m) &= Max(n, m) : (Min(n, m) : nil) \end{aligned}$$

The abstract interpretation of $Insort'$ is the function

$$\begin{aligned} Linsort'(0) &= 0 \\ Linsort'(n+1) &= Linsert'(Linsort'(n)) \end{aligned}$$

$$\begin{aligned} Linsert'(0) &= 1 \\ Linsert'(n+1) &= 2 \end{aligned}$$

$Linsert'$ can be cast into the S_i^1 schemes as:

$$\begin{aligned} Linsert'(0) &= 1 \\ Linsert'(n+1) &= \bar{2}(Linsert'(n)) \end{aligned}$$

where $\bar{2}$ denotes $\lambda x.\ 2$, the constant function which returns 2.

So $Linsert'$ also belongs to S_2^1; as a consequence $Linsort'$ and $Linsort$ belong to the same scheme S_3^1.

It turns out that the erroneous definition $Insort'$ returns correct results for lists of length less than or equal to 2. Thus, it is indeed necessary to include a list of length 3 into the test data set to capture this bug.

6 Related Work

6.1 Program Testing and Program Verification

The work presented here stands at the crossroad of three main trends of activities: program testing, program analysis and program verification. It presents similarities but also differences with each of them.

The main departure with respect to "traditional" verification techniques and formal development methods like Z [SPI92], VDM [JON90], LARCH [GH93], B [ABR96] is that we trade generality for mechanization. Our goal is not to provide complete correctness proofs of a program but rather to "formally test" it against specific properties. This strategy is shared by the program analysis community, but the verifications that are made possible by our method are out of reach of static analysis techniques. These techniques rely on iterative algorithms to compute fixed points of recursive equations [CC77]. Restrictions have to be introduced in order to ensure the termination of these iterations. One typical restriction is to impose that the abstract domains are finite (or, more generally, that no infinitely increasing chain of values can be constructed by the algorithm). The kind of restriction introduced in this paper is of a different nature: it is a restriction on the structure of the definition of the program. One advantage of this kind of restriction (which is due to its syntactic nature) is that it can also be used in a top-down process, to favor the construction of more easily testable programs. Another advantage is that it can be checked mechanically (in contrast with classical test hypotheses). Further work is needed to decide if traditional program analysis techniques can be extended to take advantage of such restrictions.

Most of the properties we have proven using testing in this paper also could have been proven using inductive proof techniques. Our method can be seen as a factorization of the proving effort in the association between a test data set and a scheme. Moreover, in the context of a syntactically restricted formalism, determining the scheme of a function is easier than directing a proof by induction.

The research activities in software testing can be classified into two very distinct categories:

1. General theories of testing have been proposed including notions like test data adequacy [BA82, WEY83, DO91], testability [FRE91, GAU95], robustness [GG75], reliability [DO91], ideal test data sets [GG75], valid and unbiased test data sets [GAU95], test hypotheses [GAU95], etc. But test criteria with the desired qualities usually lead to infinite test data sets or test hypotheses which do not necessarily hold.
2. On the practical side, a number of test coverage criteria have been put forward [BEI90, NTA88, RW85]. Some of them are supported by test coverage measure tools [OW91]. These tools are automatic but they only provide *a posteriori* information about the test coverage of a given test data set. In any case, these criteria are not exactly formal in the sense that there is no link

between the satisfaction of a test coverage criterion (at least for the effective ones) and the correctness of the program.

A distinctive feature of our work with respect to testing (which makes it closer to verification) is that it is not limited to the detection of bugs in a program: we know that a program which passes a complete test data set satisfies the tested property. Also, since our test data generation process takes both the program and a property into account, it can be seen as an hybrid of structural and functional testing. Such an integration has already been advocated in a more general framework in the past [RC85], but without any mechanical procedure. A similar approach has been successfully investigated for protocol testing [FJJV96, BP94], but these contributions focus on the control aspects of programs (rather than on data). They are thus complementary to our work.

6.2 Program Testing and Program Learning

By another way, one can compare our testing technique, and more generally the test generation process, with techniques from the program learning community. These techniques are about synthesizing programs from examples [BIE78] (as opposed to generating test data sets from programs). This trend of research is concerned with both the learning procedures and the classes of functions that can be learned. The framework that is common to all these methods is called *inductive inference*.

The programs that are learned belong in fact to restricted fragments of a programming language. This is because inductive learning uses a notion of *learning bias* to narrow the search space. The role of the learning bias is to make the learning process feasible and as efficient as possible. A learning bias restricts the language in which the concept to be learned is expressed. Such a bias makes the learning process incomplete because the intended concept may be better expressed outside the learning bias, or even may not be expressible in the learning bias. However, a learning bias establishes a formal relation between the examples and the concept that is learned.

Tools and methods exist for program testing and program learning. However, a deeper examination of these two activities shows that they are based on very different hypotheses.

In the learning activity, a finite suite of examples or traces is used to generate a program that is guaranteed to satisfy the examples. In other words, if one considers the suite of examples as a test data set, it is certain that the generated program will pass it. Note also that the input document is assumed to be correct. A learning bias is used to narrow the search space and to infer only regular programs. This restriction is in the same spirit as testing hypotheses, which also assume regularity in the tested programs.

In the context of structural testing, test data sets are generated from a program according to a test criterium (e.g., all-instructions, all-def-use). But finite test data sets are generally not robust (which means that they can accept incorrect programs). Finally, the input document (the program) cannot be assumed

to be correct. This shows that the situation is less favorable than in the context of program learning. The basic reason is that classical approaches to the generation of test data sets deal with general programs (i.e., written in Turing-complete languages), whereas program learning deals with biased programs.

A solution to make the situation of test data sets generation more favorable is to borrow from program learning some of its hypotheses. We will call *testing bias* a syntactic restriction that corresponds in program testing to a learning bias in program learning. Our method goes in this direction, taking inspiration from automated program learning to do test generation via a testing bias. Here, testing biases are defined as recursive function schemes and our method deals with hierarchies of testing biases.

The relation between program learning and program testing has been recognized in the past by several authors [WEY83, BG96]. In fact, Bergadano *et al.* actually use program learning as a means for generating test data sets. In their case, a program learning process generates incrementally a family of programs that are "close" to the program to be tested. Each time a new program is produced, a new test case is added to the set of examples. The new test case must be such that it distinguishes the new program from the program to be tested. Our technique does not actually perform program learning. We mainly use it as a fruitful metaphor to establish a tight relation between a finite set of input/output data, a program and a property. The key idea is that the program must belong to a "learnable" family. On the contrary, usual testing theories either involve infinite sets of data, or lack a well defined relation between test data sets and programs.

7 Conclusion

The approach put forward in this paper is based on a tight integration of static analysis and testing techniques for program verification. These techniques are traditionally studied by different communities without much cross-fertilization. Furthermore, considering the three types of documents used in programming, i.e., properties (e.g., specifications), programs, and data (e.g., test data and examples), one can observe that all point-to-point relations between these documents have been explored in both directions (e.g., the program-data relation corresponds to testing and program learning). However, it is seldom the case that the relation between the three types of documents is considered globally. We think that great benefits can be gained from a better understanding of the connections between them [LMNR98].

Our method is both formal and automatic. Once the abstraction is chosen (that is to say, the abstract domain and the abstraction function), all the different stages of the method are fully automated. There is no way to find automatically the abstraction, it is the only point in our method for which the user has to be a bit intuitive. From a practical point of view, it is possible to construct libraries of abstractions by associating some different generic abstractions to

several inductive types. This could be an help for the user. Our method has been implemented in a prototype system which, as a consequence, does not require any specific knowledge from the user. This system is powerful enough to deal with all the examples used in this paper. It is our belief that there is plenty of room for software engineering tools between the following two extremes: unrestricted, but only semi-automated, techniques requiring significant efforts from highly qualified users, and fully automated processes with restricted power [LM97].

Of course, the price to pay for complete mechanization is to limit one's ambitions: we have introduced restrictions on both programs and properties to be verified. Note however that the restriction on programs is weaker than the restriction on properties since it is only their abstraction that must belong to one scheme of the hierarchy. Because of the restriction on properties, our method should be seen as an extended type checker rather than a program verification technique. For instance, a typical type checker can verify that a program returns a result of type list, when our technique can also provide information about the length of this list. Further works are needed to assess the impact of the current limitations of the method and to suggest ways to enhance it to increase its practical significance. We just sketch now some extensions which are currently under investigation.

So far, we have studied only the *length* abstraction presented in this paper. We are now considering other properties on integers (like size, or depth) and other structured types (like trees, or general inductive types). One important constraint on the schemes (which plays a crucial role in the proofs of our results) is their uniformity with respect to the structured data type (natural numbers here). This choice is inspired by previous work on inductive data types and the associated inductive program schemes [PDM89]. Uniformity means that conditions in programs are based only on the structure of the arguments. It can be seen as a programming discipline, favoring the construction of programs which can be tested or verified more easily. It is also possible to alleviate this limitation on the source programs since it is only their abstraction that must belong to a scheme. One possible solution is to derive two approximate abstract versions of the program representing a lower bound and an upper bound of the property of the result. Consider for example a modification of the replacement program *Rep* of Section 5.1 to include a conditional statement on the elements of its first argument, replacing only the values different from zero. We can then derive two abstract functions corresponding to the two extreme cases: the first one returns its first argument (when no element is replaced) and the second one is the product (when all the elements are replaced). Further work is needed to assess the significance of this extension.

More generally, different works on protocol testing and verification of properties by model-checking have already shown that restricted formalisms could be of great use in the search for automation. We believe that domain specific languages could also gain benefits from our method and be the ideal target to exercise it.

References

[ABR96] J.-R. ABRIAL. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[BA82] T.A. BUDD and D. ANGLUIN. Two Notions of Correctness and Their Relation to Testing. *Acta Informatica*, 18, 1982.

[BEI90] B. BEIZER. *Software Testing Techniques, 2^{nd} Edition*. Van Nostrand Reinhold, 1990.

[BG96] F. BERGADANO and D. GUNETTI. Testing by means of inductive program learning. *ACM transactions on Software Engineering and Methodology*, 5(2), 1996.

[BIE78] A. BIERMANN. The inference of regular LISP programs from examples. *IEEE transactions on Systems, Man, and Cybernetics*, 8(8), 1978.

[BP94] G.V. BOCHMANN and A. PETRENKO. Protocol Testing: Review of Methods and Relevance for Software Testing. *Proceedings of ISSTA*, August 1994.

[CC77] P. COUSOT and R. COUSOT. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. *Proceedings of the 4^{th} POPL*, 1977.

[DO91] R.A. DEMILLO and A.J. OFFUTT. Constraint-Based Automatic Test Data Generation. *IEEE Transactions on Software Engineering*, 17(9), September 1991.

[FJJV96] J.-C. FERNANDEZ, C. JARD, T. JÉRON, and C.G. VIHO. Using on-the-fly verification techniques for the generation of test suites. *Proceedings of the Conference on Computer-Aided Verification*, July 1996.

[FRE91] R.S. FREEDMAN. Testability of Software Components. *IEEE Transactions on Software Engineering*, 17(6), June 1991.

[GAU95] M-C. GAUDEL. Testing can be formal, too. *Proceedings of TAPSOFT*, 1995.

[GG75] J.B. GOODENOUGH and S.L. GERHART. Toward a Theory of Test Data Selection. *IEEE Transactions on Software Engineering*, 1(2), June 1975.

[GH93] J.V. GUTTAG and J.J. HORNING. Larch: languages and tools for formal specification. *Texts and Monographs in Computer Science*, 1993.

[JON90] C.B. JONES. *Systematic software development using VDM*. Prentice Hall International, second edition, 1990.

[LM97] D. LE MÉTAYER. Program analysis for software engineering: new applications, new requirements, new tools. *ACM Sigplan Notices*, (1), Janvier 1997.

[LMNR98] D. LE MÉTAYER, V.-A. NICOLAS, and O. RIDOUX. Exploring the Software Development Trilogy. *IEEE Software*, November 1998.

[NIC98] V.-A. NICOLAS. *Preuves de Propriétés de Classes de Programmes par Dérivation Systématique de Jeux de Test*. PhD thesis, Université de Rennes 1, December 1998.

[NTA88] S.C. NTAFOS. A Comparison of Some Structural Testing Strategies. *IEEE Transactions on Software Engineering*, 14(6), June 1988.

[OW91] T.J. OSTRAND and E.J. WEYUKER. Data Flow-Based Test Adequacy Analysis for Languages with Pointers. *Proceedings of POPL*, January 1991.

[PDM89] B. PIERCE, S. DIETZEN, and S. MICHAYLOV. Programming in Higher-Order Typed Lambda-Calculi. *Research report CMU-CS-89-111*, March 1989.

[RC85] D.J. RICHARDSON and L.A. CLARKE. Partition Analysis: A Method Combining Testing and Verification. *IEEE Transactions on Software Engineering*, 11(12), December 1985.

[RW85] S. RAPPS and E.J. WEYUKER. Selecting Software Test Data Using Dataflow Information. *IEEE Transactions on Software Engineering*, 11(4), April 1985.

[SPI92] M. SPIVEY. *The Z notation - A reference manual*. International Series in Computer Science. Prentice Hall International, second edition, 1992.

[WEY83] E.J. WEYUKER. Assessing test data adequacy through program inference. *ACM Transactions on Programming Languages and Systems*, 5(4), October 1983.