Testing by means of inductive program learning

Testing by Means of Inductive Program Learning

FRANCESCO BERGADANO and DANIELE GUNETTI
University of Torino

Given a program P and a set of alternative programs $\mathcal{P}$, we generate a sequence of test cases that are adequate, in the sense that they distinguish the given program from all alternatives. The method is related to fault-based approaches to test case generation, but programs in $\mathcal{P}$ need not be simple mutations of P. The technique for generating an adequate test set is based on the inductive learning of programs from finite sets of input-output examples: given a partial test set, we generate inductively a program $P' \in \mathcal{P}$ which is consistent with P on those input values; then we look for an input value that distinguishes P from P', and we repeat the process until no program except P can be induced from the generated examples. We show that the obtained test set is adequate with respect to the alternatives belonging to $\mathcal{P}$. The method is made possible by a program induction procedure which has evolved from recent research in machine learning and inductive logic programming. An implemented version of the test case generation procedure is demonstrated on simple and more complex list-processing programs, and the scalability of the approach is discussed.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing—*test data generators*; I.2.6 [**Artificial Intelligence**]: Learning—*induction*

General Terms: Algorithms, Reliability

Additional Key Words and Phrases: Program induction by examples

1. INTRODUCTION

Testing is concerned with the problem of detecting the presence of errors in programs by executing them on specific and well-chosen input data. An error is known to be present in a program P when the output is not consistent with the specifications or is perceived as mistaken by the tester. Both for checking the output and for generating meaningful test cases, we need some information about the "correct" program P_c.

This information may be more or less complete. In the best case, one might actually have available a correct version of the program, or an

This work was supported in part by an EU grant under contract 20237 (ILP2).
This article is a revision of a paper present at SIGSOFT '93.
Authors' address: Dipartimento di Informatica, University of Torino, corso Svizzera 185, 10149 Torino, Italy; email: {bergadan; gunetti}@di.unito.it.

executable specification, and program testing reduces to proving or giving probabilistic evidence of program equivalence by means of input/output examples. This corresponds to the first notion of program correctness discussed in Budd and Angluin [1982] in connection with testing and leads to various forms of algebraic [Tsai et al. 1990], syntax-directed [Bazzichi and Spadafora 1982; Gorlick et al. 1990], and specification-based testing [Choquet 1986; Hayes 1986]. The corresponding concept of an ideal test set is defined as follows [Budd and Angluin 1982; Denney 1991; Howden 1976]:

Definition 1.1. A test set T for a program P is *reliable* (with respect to a correct program P_c) iff $[\forall x \in T\ P(x) = P_c(x)] \rightarrow P \equiv P_c$. If P contains errors, this will be shown by the test cases in T.

Even if an executable specification P_c is not available, one may have a priori knowledge about the correct functionality of the desired program, such as domain and codomain of the needed subprograms, boundary, or otherwise critical values within these domains, and possible interactions among different input and output variables. This leads to techniques based on functional and domain testing [Clarke et al. 1982; Hoffman and Strooper 1991; Howden 1980; Strooper and Hoffman 1990] and may be combined with executable specifications [Tsai et al. 1990].

If no precise a priori information about the correct or desired behavior of the program is known, then errors can only be detected by the tester through a direct inspection of the outputs during the normal execution of the program. The "correct" program is basically defined as "the one the tester has in mind," not a very informative concept. In these cases it has been assumed that, although the correct program is unknown, there is a known *set of programs* that we may view as alternative implementations and should contain at least one correct solution. This is the second concept of program correctness analyzed in Budd and Angluin and leads to the following definition of meaningful test data (see also DeMillo and Offutt [1991] and Hamlet [1981]).

Definition 1.2. A test set T for a program P is *adequate* (with respect to a set $\mathcal{P}$ of alternative programs) iff it is reliable w.r.t. every program in $\mathcal{P}$.

It follows that, if $\mathcal{P}$ contains at least one correct program P_c, then the test set will also be reliable for P. Many diverse test case generation techniques correspond to this notion, including the method described in this article. When the set of alternatives $\mathcal{P}$ depends on the given program P, we are dealing with *white-box* test case generation methods: one needs to inspect the code of P in order to generate the test data, because P is needed for determining the alternative programs. In the approach presented here, we do not make any assumption about $\mathcal{P}$, and in general, this set need not depend on the particular structure of the program to be tested.

The idea of relating the problem of testing a program P to a set of possible alternatives $\mathcal{P}$ is fully undertaken and made explicit by fault-based methods [DeMillo and Offutt 1991; DeMillo et al. 1978; Howden 1982; King and Offutt 1991; Morell 1988]. Here, a general-purpose set of typical

programming errors is defined a priori, and alternative programs are obtained by inserting in P all of these errors. Actually, many approaches use single *mutation operators* that modify the basic components of P by simple syntactic changes, and therefore only one fault at a time is introduced: the set of alternative programs differs from P by only one mutation. If one assumes the "coupling effect" [DeMillo 1989; DeMillo et al. 1978; Offutt 1989], i.e., complex errors are detected by an analysis of simpler ones, and if at least one correct implementation is obtained from P by a sequence of allowed mutations, then all the errors in P will be detected. More precisely we may define *the coupling-effect assumption*:

> if a test set T is adequate w.r.t. single mutations of P, then T is also adequate w.r.t. any mutant of P.

The above definition is absolute, although the coupling effect is usually defined probabilistically, in the sense that single faults *tend* to reveal multiple faults as well. It is not always easy to generate test data that are adequate with respect to a large set of mutants, especially if the coupling effect is not assumed. Weak-mutation testing [Howden 1982] provides a different proposal:

The Weak-Mutation Assumption. Suppose $Q_1, \ldots, Q_n$ are the relevant components of program P; given a test T, let $T_1, \ldots, T_n$ be the data that are passed to $Q_1, \ldots, Q_n$ respectively, when P is called on the inputs of T. Weak-mutation testing assumes that if, for all i, T_i is adequate for Q_i w.r.t. its mutations, then T is adequate for P w.r.t. to any combination of these mutations.

In DeMillo and Offutt [1991], yet another assumption is made, suggesting that it may be enough to make the mutated program's state differ from P's state only after the statement where the mutation occurs (this is shown to be a necessary condition for test data adequacy). In general these kinds of assumptions make the testing system ignore the global effect of errors [Morell 1988] or the interaction between mutations at the global program level.

The method proposed in this article makes one step further in the direction of realizing the importance of an explicit set $\mathcal{P}$ of alternative programs. It is suggested that when we are given a program P for testing, the set of alternative implementations is often problem dependent, and much could be said about it by the programmer of P. We describe a test data generation procedure for distinguishing P from the given alternatives. This procedure cannot be similar to the ones used in mutation testing, as we have a user-given set of alternative programs and since locality assumptions are not made. Such sets may contain many programs that are not just simple mutations of P, but depend on the application domain and on the programmer's knowledge. The number of alternative implementations may be very large, and they cannot all be executed on the test data. Moreover, we would like to limit the size of the test set, so that the tester is not required to inspect an excessive number of output values for correctness.

The test case generation procedure is based on techniques for learning logic programs from examples.

2. INDUCTION AND TESTING

The relation between induction and testing has been analyzed in Cherniavsky and Smith [1987] and Weyuker [1983] and, only indirectly, in Budd and Angluin [1982], Hamlet [1981], and Kilpelainen and Mannila [1990]. An intuitive symmetry is almost immediately noticed: induction is the inference from examples to programs; test case generation is from programs to input values. Given a test set T of input values for a program P, the examples of P for T are defined as $E(P, T) = \{\langle i, o\rangle \mid i \in T \text{ and } P(i) = o\}$. Weyuker [1983] formalizes this notion:

Definition 2.1. A test set T is *inference adequate* for a program P intended to satisfy a specification S iff P_I is inductively inferred from E(P, T) and $P_I \equiv P \equiv S$. If only $P_I \equiv P$ is true, then T is said to be *program adequate*, and if only $P_I \equiv S$ is true, then T is said to be *specification adequate*.

In this article we are interested in the case where a specification S is not given, and therefore we will only use the notion of a program-adequate test set. The intuitive meaning of the above definition is as follows: if, given a set E(P, T) of input/output examples, we inductively infer a program $P_I \equiv P$, then T is likely to be useful for testing the program P.

This analysis leaves a number of unspecified notions. Most important, when we say "P_I is inductively inferred from T" we must have in mind some particular kind of learning procedure. The induction method of Summers [1977] is used by Weyuker [1983], but the above definition should not be tied to it in general; it may be more appropriate to characterize an inductive inference procedure by means of the basic requirements it must satisfy. But even in this case, we need to define this more precisely, because many and quite different proposals have been made and because there is no unique understanding of the word "induction." These include EX-identification [Cherniavsky and Smith 1987; Gold 1967], finite identification [Gold 1967], probably approximately correct (PAC) induction [Valiant 1984], and complete and consistent classification rules [Bergadano et al. 1992; Michalski 1980]. An inductive inference machine (IIM) M receives in input a sequence of examples $\langle i, o\rangle$ of a program P, and after every such example, it outputs a guess $P_I \in \mathcal{P}$. Given a set T of input values, an IIM M EX-identifies P from E(T, P) iff after some point it always makes the same correct guess $P_I \equiv P$.

The notions of EX-identification and testability are compared by Cherniavsky and Smith and are found to be quite different: many simple classes of programs are shown to be easily EX-identified from examples, even when a finite test set that is adequate in the sense of Definition 1.2 may not be found. Moreover, although an oracle for the halting problem will make all the partial recursive functions EX-identifiable, it will not make the previ-

$$\frac{E(T, P), \mathcal{P}}{P} \qquad \frac{P, \mathcal{P}}{T}$$

Induction | *Test Case Generation*

Fig. 1. Induction and test case generation compared.

ously analyzed testing problems any easier. We will then use, in this article, the alternative notion of finite identification [Gold 1967]:

Definition 2.2. An IIM finitely identifies a program $P \in \mathcal{P}$ from $E(T, P)$ iff after seeing a finite subset of $E(T, P)$, it can determine that the correct guess is P_I and $P_I \equiv P$. It will then ignore the remaining examples in $E(T, P)$.

This is closer to test case generation, because in both cases we are to deal with a finite set of input/output pairs. By contrast, with EX-identification, the induction procedure eventually obtains a correct guess, but is not aware of having reached that point.

In all of the above-cited approaches to induction, it is fundamental to determine beforehand the class $\mathcal{P}$ of programs that could be produced. In fact, this is generally observed in most approaches to induction. It is true for such diverse areas as Bayesian inference (where $\mathcal{P}$ takes the form of a prior probability) and connectionist induction (where $\mathcal{P}$ is defined by the structure of the neural net). The importance of restricting $\mathcal{P}$ through some form of *bias* was also a natural conclusion for the research in machine learning and inductive logic programming during the last few years [Bergadano 1993; Bergadano and Gunetti 1993; Kirschenbaum and Sterling 1991; Muggleton 1991; Utgoff 1986].

No matter which meaning of inductive inference we want to use in Definition 2.1, it is then appropriate to relativize it to the set of legal programs $\mathcal{P}$, that P_I must belong to. This is consistent with our discussion on testing, given in the introduction, where it was shown that all test case generation procedures that are not specification based refer (sometimes implicitly) to a set $\mathcal{P}$ of alternative programs. The intuitive symmetry between induction and test case generation is then pictured as in Figure 1. The basic proposal of Definition 2.1 can now be understood more precisely by defining "inductive inference" as "finite identification." Moreover, this is strongly related to the concept of test data adequacy (Definition 1.2):

Observation 2.3. T is adequate for $P \leftrightarrow$ P is finitely identified from $E(T, P)$.

The restriction to a finite set of alternative programs $\mathcal{P}$ seems to be acceptable for testing; for instance, all approaches to fault-based testing, such as mutation analysis, are based on this assumption.

Previous comparisons of induction and test case generation were either mainly theoretical [Budd and Angluin 1982; Cherniavsky and Smith 1987; Hamlet 1981] or devoted to the problem of *checking* test set adequacy [Weyuker 1983] rather than actually *generating* the adequate test cases. This was due, in part, to the fact that research in program induction was

not very well developed at that time [Weyuker 1983]. Although a general solution is not available at present, some progress has been made, particularly within the field of inductive logic programming [Muggleton 1991]. In this article we use a method that has evolved from this recent research [Bergadano 1993; Bergadano et al. 1988; Quinlan 1990] and adapt it to the needs of our test case generation procedure. This is, to our knowledge, the first implemented test case generation method that is based on machine-learning techniques, although some related work has been done for the problem of knowledge base validation [DeRaedt et al. 1991].

3. TEST CASE GENERATION

Test cases are generated through a sequence of inductive inferences of programs from examples. Initially, there are no examples, and in the end, the generated set of examples will be adequate in the sense of Definition 1.2.

Suppose P is the program to be tested. At any given moment, the examples generated so far are used to induce a program P′. New examples that distinguish P from P′ are added, and the process is repeated until no program P′ that is not equivalent to P can be generated. This procedure is described in more detail below.

Test case generation procedure:

Input: a program P to be tested
a finite set $\mathcal{P}$ of alternative programs
an induction procedure M

Output: an adequate test set T

$T \leftarrow \emptyset$;
loop forever:
$\langle P', T\rangle \leftarrow M(E(T, P), \mathcal{P})$
if P' = "fail" then return (T)
if $(\exists\ i)\ P'(i) \neq P(i)$
then $T \leftarrow T \cup \{i\}$
else $\mathcal{P} \leftarrow \mathcal{P} - P'$

The induction procedure M takes in input a set of examples E(T, P) and a set $\mathcal{P}$ of alternative programs and gives in output a program $P' \in \mathcal{P}$ that is consistent with E(T, P). In order to do so, M may need to add input values to T, yielding an extended set $T' \supseteq T$; this will be illustrated in Section 3.2. M must satisfy a basic admissibility requirement:

Definition 3.1. An induction procedure M is *admissible* iff

(1) $M(E(T, P), \mathcal{P}) = \langle P', T'\rangle \rightarrow P'$ is consistent with E(T′, P)
(2) $M(E(T, P), \mathcal{P}) = \langle \text{fail}, T'\rangle \rightarrow (\nexists P' \in \mathcal{P})\ P'$ is consistent with E(T′, P).

Requirement (1) is a form of correctness property, and (2) is a form of completeness. An admissible induction procedure is described in Section 3.2. In the test case generation procedure, the test set T is initially empty.

The main step in the loop consists of calling the induction procedure M and obtaining a program P′ which is consistent with the examples generated so far. Then, this program P′ will be ruled out either by (1) adding an input value to T or (2) removing it from $\mathcal{P}$. As a consequence, P′ will never be generated again. The condition in the second "if" statement addresses the question of program equivalence, which is of course undecidable, in general. If, for the particular set $\mathcal{P}$, a decision procedure is available for finding an input i such that P(i) ≠ P′(i), this is used directly in the above test generation method. As this is not true in general, examples are found by enumerating (in some random or ad hoc order) the possible inputs i and stopping when an i is found such that P(i) = o ≠ P′(i) = o′. This enumeration could also be guided by other test case generation techniques, such as path or functional testing. The requirement of decidable equivalence is not easily verified or accepted. Program equivalence was found to be a major theoretical [Budd and Angluin 1982] and practical [DeMillo 1989; DeMillo and Offutt 1991; King and Offutt 1991] issue in program testing. In the implementation of our test case generation method we approximate it by means of its time-bounded semidecision procedure. A better implementation would not use randomly generated examples for this purpose, but, for example, large sets of examples obtained through some black-box testing method. Except for this approximation, the system produces adequate test sets with respect to any finite class of programs $\mathcal{P}$. In fact, when M cannot find a program P′ ∈ $\mathcal{P}$ that is consistent with the examples, then the only programs with this property are P and those equivalent to it, i.e., the test set T is adequate, as proved in the following:

THEOREM 3.2. *If equivalence is decidable for programs in* $\mathcal{P}$, *then the above test case generation procedure produces an adequate test set* T *for* P.

PROOF. Since equivalence is decidable for programs in $\mathcal{P}$, the procedure terminates, as the condition (∃i) P′(i) ≠ P(i) always produces an answer. As a consequence each loop execution will make P′ inconsistent or remove it from the finite set $\mathcal{P}$. By virtue of Observation 2.3, it will then be sufficient to show that M finitely identifies P from E(T, P), where T is the test set obtained after termination. Since M is admissible, M(E(T, P), $\mathcal{P}$ ∪ {P}) = ⟨P_I, T′⟩, because P is consistent with E(T′, P) for any T′, and therefore M cannot fail. But, just before termination, M(E(T″, P), $\mathcal{P}$ − {P′ | P′ ≡ P}) = ⟨fail, T⟩, for some T″ ⊆ T, and, being M admissible, $\mathcal{P}$ − {P′ | P′ ≡ P} cannot contain any program consistent with E(T, P). Therefore, after seeing E(T, P), M can stop and determine that the correct and only consistent guess is P_I = P. □

3.1 Defining a Set of Alternative Programs

Suppose we wanted to test a program P. In order for the testing procedure to work, we must be able to (1) describe a set $\mathcal{P}$ of alternative programs and (2) use $\mathcal{P}$ and a partial test set T to learn inductively an intermediate program P′ = M(E(T, P), $\mathcal{P}$). Inductive learning is described in Section 3.2. Here we describe how a set $\mathcal{P}$ of alternative programs may be provided.

The basic philosophy in our approach is that no general-purpose alternative set of programs may be useful. Every problem, and every implementation, has special weaknesses, likely errors, and significant mutations. We have developed a language for describing mutations that are specific to a given program. This language refers to alternative programs that are written in Prolog, but the overall method does not require that the tested program be a Prolog program.

A full description of this "mutation description language" is found in Bergadano and Gunetti [1996], viewed there as a language for specifying inductive bias. Here we summarize a subset of it, by describing the notion of "clause sets" and "literal sets." In general, a set $\mathcal{P}$ of alternative programs is described by a pair ⟨*known clauses*, *possible clauses*⟩. A program P belongs to $\mathcal{P}$ if P = {known clauses ∪ P1}, where P1 is a subset of the possible clauses. The simplest syntactic tool for specifying inductive bias within the present framework is given by *Clause Sets*: known clauses are listed as in a Prolog program, while possible clauses are surrounded by brackets denoting a set. For instance, there follows a possible description of a priori information for learning a logic program for *member*:

```
member(X, [X|_]).
{member(X, [Y|Z]) :- cons(X, W, Z).}
cons(X, Y, [X|Y]).
{member(X, [Y|Z]) :- X ≠ Y, member(X, Z).
member(X, [Y|Z]) :- cons(Y, Z, W).
member(X, [Y|Z]) :- member(X, Z).}
```

This means that the learned program will have to include the first clause, which is known, possibly followed by the second clause "member(X, [Y|Z]) :- cons(X, W, Z)."; the third clause "cons(X, Y, [X|Y])." will have to follow. Finally, some or all of the remaining clauses may be appended to the program. There are 16 different logic programs satisfying these very strict requirements; among these some represent a correct implementation of *member*.

Unfortunately, a priori information about possible faults is not always so precise, and the set of possible clauses may be much larger. As a consequence, the user may find it awkward, or even impossible, to type them one after the other. To this purpose we define *literal sets*. If a clause occurs in a clause set, then some conjunctions of literals of its antecedent may be surrounded by brackets, denoting a set. In this case the clause represents an *expansion* given by all the clauses that may be obtained by deleting one or more literals from the set. Formally:

$$\{P :- A, \{B, C, \ldots\}, D\}$$
$$= \{P :- A, \{C, \ldots\}, D\} \cup \{P :- A, B, \{C, \ldots\}, D\}$$

In other words, the expansion of a clause is the set of clauses obtained by replacing the literal sets with a conjunction of any of its literals. With this

```
int([],¬[]).
{
int([X1|X2],Y,Z) :- {member(X1,Y), notmember(X1,Y)},
                    {int(X2,Y,Z), int(X2,Y,W), cons(X1,W,Z)}.
member(X,Y) :- {cons(X,Y,Z), cons(X,Z,Y), cons(W,Y,Z),
                    cons(W,Z,Y), member(X,Z), member(W,Z)}.
}
cons(X,Y,[X|Y]).
notmember(X,[]).
notmember(X,[Y|Z]) :- X≠Y, notmember(X,Z).
```

Fig. 2. Inductive bias for *intersection*.

syntactic mechanism, one can define in a concise way a large set of possible clauses. Figure 2 gives an example of inductive bias for learning *intersection*. The clause set of Figure 2 contains $2^5 + 2^6 = 96$ clauses, and there are 2^{96} permitted programs. Then, even if the given bias may seem quite informative, it still allows for a very large set of alternative programs.

This language, with the additional syntax for *term sets*, described in Bergadano and Gunetti [1996], has been used by the authors in connection with different program induction techniques [Bergadano and Gunetti 1993, 1994b; Bergadano et al. 1995] and was used or extended by other groups within the machine-learning community [Ade and de Raedt 1995; Nedellec and Rouveirol 1994]. In the present context, it may be viewed as a way to specify meaningful mutations in a concise way, depending on the special characteristics of the program to be tested.

3.2 Induction Procedure

We may view a logic program as defining an input/output transformation, associated to a "main" predicate p. The procedure defined in this section induces a logic program P belonging to a set $\mathcal{P}$ of legal programs, given a set of input values for its main predicate p. We require that every predicate occurring in $\mathcal{P}$ be also defined by some clause in P, or in a common external module.

Without loss of generality, we may also assume that all predicates q occuring in $\mathcal{P}$ are binary, where $\vdash$ q(x, y) will indicate that input x is transformed into output y. For a program P and its main predicate p, P(i) = o can now be written as P $\vdash$ p(i, o). During the induction process, more input values may be added to the test set T, both for the "main" predicate p and for other predicates occurring in $\mathcal{P}$. Such input values are added to T through a side effect of the procedure "covers" described below. The final generated program will then have to be consistent with all of these examples. Given a predicate q, we will write T(q) for the input values for q in the test set $T = \cup_q T(q)$.

The main loop in the proposed procedure follows the basic scheme of many machine-learning methods (e.g., Bergadano et al. [1988; 1992],

Michalski [1980], and Quinlan [1990]): clauses are generated one at a time, independently of each other, and the examples that they "cover" are deleted. We stop when all examples are covered by some generated clause. When generating the clauses, we will be careful that they are consistent with E(T, P). The main difference, in the method presented here, is that the program P to be tested is used to generate missing examples, and functionality constraints are used, so that negative examples are not necessary. The great advantage of these approaches is that we do not have to try all programs in $\mathcal{P}$ and pick the first which is consistent with the examples; we just need to try all possible clauses, generate the ones that are consistent with the examples, and add them to the logic program which is being developed. As programs are sets of clauses, the number of programs is exponential in the number of clauses, and being able to generate one clause at a time results in a learning procedure which is linear in the number of clauses. This is necessary for the feasibility of the approach. The learning algorithm follows:

Induction procedure M(E(T, P),$\mathcal{P}$):
$T' \leftarrow T$; $P' \leftarrow \emptyset$
while $(\exists\ q)\ T'(q) \neq \emptyset$ do
    generate_one_clause C = "q(X, Y) :- α"
    $P' \leftarrow P' \cup C$
    if C = "fail" then return ⟨fail, T⟩
    /* remove inconsistent clauses */
    $P' \leftarrow P' - \{C \mid (\exists\ i \in T)\ \text{covers}(C, r(i, o')) \wedge P \vdash r(i, o) \wedge o \neq o'\}$
    /* remove covered examples */
    $T' \leftarrow T - \{i \mid (\exists C \in P')\ \text{covers}(C, r(i, o)) \wedge P \vdash r(i, o)\}$
return ⟨P', T⟩

generate_one_clause C = "q(X, Y) :- α":
$\alpha \leftarrow$ "true"
while /* C is not consistent with E(T, P) */
  $(\exists i \in T(q))\ \text{covers}(C, q(i, o')) \wedge P \vdash q(i, o) \wedge o \neq o'$
do choose r(Z, W) s.t. "q(X, Y) :- α, r(Z, W)" belongs to
        some alternative program in $\mathcal{P}$ (a backtracking point)
  if no such r(Z, W) is left, then return ("fail")
  $\alpha \leftarrow \alpha \wedge r(Z, W)$
  if /* C never produces a correct output */
    $(\nexists\ i \in T'(q))\ \text{covers}(C, q(i, o)) \wedge P \vdash q(i, o)$
    then backtrack to a different literal choice

where **covers**("r(X, Y) :- β", r(i, o)) is true iff
$r(i, o)$:- $\beta[i/X, o/Y] \wedge$ transform (P) $\vdash r(i, o)$
and transform (P) is the same as P, except that
(1) no clause can unify with r(i, o) and
(2) whenever $s(Z, W) \in \beta$ and s(a, W) is resolved against some clause of P that needs to be tested (not a clause from an external module), a is added to T(s).

```
{
merge(X,Y,Z) :- {
                null(X), null(Y), X=Z, Y=Z, head(X,X1), tail(X,X2), head(Y,Y1), tail(Y,Y2),
                X1<Y1, X1>Y1, X1≤Y1, merge(X2,Y,W), merge(X,Y2,W),
                cons(X1,W,Z), cons(Y1,W,Z), X1≥Y1, X1=Y1, W=Z,
                }
}
```

Fig. 3. Set of alternative programs for testing *merge*.

Input values may then be added to T as a side effect of the function "covers(C, e)." Examples are removed when they satisfy one of the generated clauses. Clause generation is achieved by adding to the antecedent one predicate at a time. A predicate is chosen from the corresponding literal set in the description of the possible mutations, described by using the clause set language of Section 3.1. The antecedent α is completed when the clause C produces the correct output for at least one $i \in T'$ and gives no incorrect outputs. An example of how the induction procedure learns a clause C is given at the end of Section 4.

It may be proved [Bergadano and Gunetti 1993] that the above procedure M is admissible. As a consequence, the overall test case generation method will produce a test set which is adequate for the given program P with respect to the set $\mathcal{P}$ of alternative programs, as stated in Definition 1.2.

4. EXAMPLE

Suppose we wanted to test the following program P for merging two ordered lists:

```
merge(X, Y, Z) :- null(X), Y = Z.
merge(X, Y, Z) :- null(Y), X = Z.
merge(X, Y, Z) :- head(X, X1), head(Y, Y1), tail(X, X2), X1 ≤ Y1, merge(X2, Y,
                  W), cons(X1, W, Z).
merge(X, Y, Z) :- head(X, X1), head(Y, Y1), tail(Y, Y2), X1 > Y1, merge(X, Y2,
                  W), cons(Y1, W, Z).
```

There is an error related to the third clause: the comparison $X1 \leq Y1$ should be replaced by $X1 < Y1$, and another clause should be inserted for the case $X1 = Y1$. The effect of this error is that elements occurring in both input lists X and Y are repeated in the output Z. Suppose also, as a possible example, that the set of alternatives $\mathcal{P}$ is defined through Clause Sets in Figure 3 by using all of the literals occurring in P plus some additional literals (the last three) allowing further possible program changes (we will discuss this choice in Section 6).

There are 2^{18} possible clauses and $2^{2^{18}}$ alternative programs in $\mathcal{P}$, i.e., all possible subsets of the space of clauses. Among those there are versions of the correct implementation. The problem is feasible for our method because the induction procedure only needs to explore heuristically the space of possible clauses, not the space of possible programs. Moreover, once a

clause is found to be incorrect w.r.t. the generated examples, all of its specializations are ruled out; this results in a drastic pruning of the search space. Learning *merge* with this set of possible clauses takes between two and five minutes of elapsed time on a Sun SPARCstation 5, depending on which examples are given. This time can be brought to the desired performance by reducing the size of $\mathcal{P}$, i.e., by allowing for a smaller number of mutations.

The test case generation procedure will start with an empty test set T_0 of input values and call the induction procedure M. As $E(T_0, P)$ contains no examples, the empty program P_0 is an acceptable output of $M(E(T_0, P),\mathcal{P})$.

We now enumerate pairs of lists X and Y, so that $P_0 \vdash$ merge(X, Y, Z'), $P \vdash$ merge(X, Y, Z), and $Z \neq Z'$. The first such pair that was found is $\langle X, Y\rangle = \langle[], []\rangle$; for this input, P_0 produces no output, and P outputs Z = []. The new test set is then $T_1 = \{\langle[], []\rangle\}$.

$M(E(T_1, P), \mathcal{P})$ is called again, yielding P_1:

```
merge(X, Y, Z) :- X = Z.
```

This program is an acceptable output of M because merge ([], [], []) is derived from it, and the output is the same as the one of P.

We now enumerate pairs of lists X and Y, so that $P_1 \vdash$ merge(X, Y, Z'), $P \vdash$ merge(X, Y, Z), and $Z \neq Z'$. The first such pair that was found is $\langle X, Y\rangle = \langle[], [1]\rangle$; for this input, P_1 outputs Z = [] while P outputs Z = [1]. The new test set is then $T_2 = T_1 \cup \{\langle[], [1]\rangle\}$.

$M(E(T_2, P),\mathcal{P})$ is called again, yielding P_2:

```
merge(X, Y, Z) :- Y = Z.
```

This program is an acceptable output of M because merge([], [], []) and merge([], [1], [1]) are derived from it.

$T_3 = T_2 \cup \{\langle[1], []\rangle\}$

P_3:

```
merge(X, Y, Z) :- head(X, X1), X = Z.
merge(X, Y, Z) :- head(Y, Y1), Y = Z.
merge(X, Y, Z) :- null(X), Y = Z.
```

$T_4 = T_3 \cup \{\langle[1], [2]\rangle\}$

P_4:

```
merge(X, Y, Z) :- head(X, X1), tail(X, X2), merge(X2, Y, W), cons(X1, W, Z).
merge(X, Y, Z) :- null(Y), X = Z.
merge(X, Y, Z) :- null(X), Y = Z.
```

As a side effect of M, the procedure "covers" also added another input pair, yielding:

$T'_4 = T_4 \cup \{\langle[], [2]\rangle\}$.

$T_5 = T'_4 \cup \{\langle[2], [1]\rangle\}$

P_5:

```
merge(X, Y, Z) :- head(X, X1), head(Y, Y1), tail(X, X2), X1 < Y1, merge(X2, Y,
                  W), cons(X1, W, Z).
```

```
merge(X, Y, Z) :- head(X, X1), head(Y, Y1), tail(Y, Y2), X1 > Y1, merge(X, Y2,
                  W), cons(Y1, W, Z).
merge(X, Y, Z) :- null(Y), X = Z.
merge(X, Y, Z) :- null(X), Y = Z.
```

$T_6 = T_5 \cup \{([1], [1])\}$

P_6:

```
merge(X, Y, Z) :- head(X, X1), head(Y, Y1), tail(X, X2), X1 < Y1, merge(X2, Y,
                  W), cons(X1, W, Z).
merge(X, Y, Z) :- head(X, X1), head(Y, Y1), tail(Y, Y2), X1 > Y1, merge(X, Y2,
                  W), cons(Y1, W, Z).
merge(X, Y, Z) :- head(X, X1), head(Y, Y1), tail(X, X2), X1 = Y1, merge(X2, Y,
                  W), cons(X1, W, Z).
merge(X, Y, Z) :- null(Y), X = Z.
merge(X, Y, Z) :- null(X), Y = Z.
```

As $P_6 \equiv P$, it is removed from $\mathcal{P}$, and no test case is generated. A few more programs equivalent to P are then generated. Finally, no other program consistent with T_6 can be found, and M fails, ending the test generation process. T_6 is adequate, and it contains an input—namely, X = [1] and Y = [1]—that demonstrates the error of P, giving Z = [1, 1] as output. The correct output would be Z = [1].

Only seven examples were necessary to isolate the error, while many more would have been required by random testing, if there are many possible element values with respect to the average list length. Functional testing would succeed easily, if the rather usual criterion of having equal elements in input lists and vectors is adopted [Howden 1980]. Nevertheless, we view this not as a general criterion, but as a specific hypothesis about typical programming errors, that is made explicit in the set $\mathcal{P}$ of alternative programs. We share this philosophy with other fault-based testing methodologies, but methods presented in the literature and cited in our references would have problems with the above program. The reason is that the correct program is not a simple mutation of the program P to be tested: it requires one simple modification and the addition of one entire clause. In an imperative programming language, this would correspond to having a conditional or a similar piece of code to be added to P, in order to obtain the correct mutant. Most approaches to fault-based testing, instead, are only able to generate minor and syntactically simple modifications.

We conclude this section by showing how the first clause of program P_4 above is learned by the induction procedure. As a first step in **generate_one_clause**, C is set to "merge(X, Y, Z) :- true." For the input pair ([1], [2]) P computes o = [1, 2], whereas covers(C, merge([1], [2], o')) is true for any value of o'. Hence it is the case that $o \neq o'$, and **generate_one_clause** goes on. Suppose literal "head(X, X1)" is chosen[1] from the literal set of Figure 3. Then, C = "merge(X, Y, Z) :- head(X, X1)." Again, covers(C,

[1]To make the discussion short, we only show the "correct" (i.e., leading to a solution) choices made by the induction procedure. In practice, backtracking would occur when necessary.

merge([1], [2], o')) is true for any value of o', since o' is not instantiated by C, and we must continue. Let literals "tail(X, X2)" first, and "merge(X2, Y, W)" then, be chosen. At this point we have:

C = merge(X, Y, Z) :- head(X, X1), tail(X, X2), merge(X2, Y, W).

When calling covers(C, merge([1], [2], o')), "merge([], [2], W)" is evaluated using P, yielding W = [2]. The input pair ⟨[], [2]⟩ is added to T_4. Still o ≠ o', and let "cons(X1, W, Z)" be chosen. We have:

C = merge(X, Y, Z) :- head(X, X1), tail(X, X2), merge(X2, Y, W), cons(X1, W, Z).

Now covers(C, merge([1], [2], o')) is no longer true for any o' ≠ o (i.e., C produces only the output o = [1, 2] for the input pair ⟨[1], [2]⟩). C is returned to the main loop, and ⟨[1], [2]⟩ is removed from T'_4.

5. A MORE COMPLEX CASE STUDY

For a more complex example, consider the classical problem of inserting a new element into a balanced binary search tree with rebalancing. We refer to the well-known algorithm described by Knuth [1973] yielding an $O(n \log n)$ sorting procedure. We recall that a binary tree is *balanced* if the height of the left subtree of every node never differs by more than ±1 from the height of its right subtree. The *balance factor* within each node is +1, 0, or −1. This is computed as the height of the right subtree minus the height of the left subtree.

The program takes in input an element and a balanced tree, inserts the element at the right place, and rebalances the tree if necessary (we assume the inserted element does not already occur in the input tree). A second output variable, the *increase factor*, is set to 1 if the height of the tree increased after the insertion, to 0 otherwise. This is used to decide whether rebalancing is needed. The *insert* program assumes a tree being represented as a list of four elements: [key, left_subtree, right_subtree, balance_factor].

Suppose the following (incorrect) program P_e to be tested is given:

P_e:

1) insert(A, T, Y, Inc) :- null(T), Y = [A, nil, nil, 0], Inc = 1.

2) insert(A, T, Y, Inc)
   :- T = [H, L, R, B], A < H, B < 0, insert(A, L, NewL, IncL), IncL = 1, Inc = 0, rebalL([H, NewL, R], [NH, NL, NR]), Y = [NH, NL, NR, 0].

3) insert(A, T, Y, Inc)
   :- T = [H, L, R, B], A < H, B ≥ 0, insert(A, L, NewL, IncL), IncL = 1, Inc = 0, Y = [H, NewL, R, −1].

4) insert(A, T, Y, Inc)
   :- T = [H, L, R, B], A < H, insert(A, L, NewL, IncL), IncL = 0, Inc = 0, Y = [H, NewL, R, B].

5) insert(A, T, Y, Inc)
   :- T = [H, L, R, B], A > H, B > 0, insert(A, R, NewR, IncL), IncL = 1, Inc = 0, rebalR([H, L, NewR], [NH, NL, NR]), Y = [NH, NL, NR, 0].

```
6) insert(A, T, Y, Inc)
     :- T = [H, L, R, B], A > H, B ≤ 0, insert(A, R, NewR, IncL), IncL = 1,
     Inc = 1, Y = [H, L, NewR, 1].
7) insert(A, T, Y, Inc)
     :- T = [H, L, R, B], A > H, insert(A, R, NewR, IncL), IncL = 0, Inc = 0,
     Y = [H, L, NewR, B].
8) rebalL(H, NL, R, Y)
     :- NL = [LH, LL, LR, -1], YR = [H, LR, R, 0], Y = [LH, LL, YR, 0].
9) rebalL(H, NL, R, Y)
     :- NL = [LH, LL, LR, 1], LR = [LRH, LRL, LRR, 0], YL = [LH, LL, LRL,
     0], YR = [H, LRR, R, 0], Y = [LRH, YL, YR, 0].
10) rebalL(H, NL, R, Y)
     :- NL = [LH, LL, LR, 1], LR = [LRH, LRL, LRR, 1], YL = [LH, LL, LRL,
     1], YR = [H, LRR, R, 0], Y = LRH, YL, YR, 0].
11) rebalL(H, NL, R, Y)
     :- NL = [LH, LL, LR, 1], LR = [LRH, LRL, LRR, -1], YL = [LH, LL,
     LRL, 0], YR = [H, LRR, R, 1], Y = [LRH, YL, YR, 0].
12) rebalR(H, L, NR, Y)
     :- NR = [RH, RL, RR, 1], YL = [H, L, RL, 0], Y = [RH, YL, RR, 0].
13) rebalR(H, L, NR, Y)
     :- NR = [RH, RL, RR, -1], RL = [RLH, RLL, RLR, 0], YL = [H, L, RLL,
     0], YR = [RH, RLR, RR, 0], Y = [RLH, YL, YR, 0].
14) rebalR(H, L, NR, Y)
     :- NR = [RH, RL, RR, -1], RL = [RLH, RLL, RLR, -1], YL = [H, L, RLL,
     0], YR = [RH, RLR, RR, 1], Y = [RLH, YL, YR, -1].
15) rebalR(H, L, NR, Y)
     :- NR = [RH, RL, RR, -1], YR = [RL, RLH, RLR, 1], YL = [H, L, RLL,
     -1], YR = [RH, RLR, RR, 0], Y = [RLH, YL, YR, -1].
```

Clauses 3 and 6 are wrong. In fact, w.r.t. clause 3, two different clauses must be used to test separately for the balance factor being equal to or greater than 0. In the correct program, clause 3 must be replaced by the following two clauses:

```
insert(A, T, Y, Inc)
  :- T = [H, L, R, B], A < H, B > 0, insert(A, L, NewL, IncL), IncL = 1, Inc = 0,
  Y = [H, NewL, R, 0].
insert(A, T, Y, Inc)
  :- T = [H, L, R, B], A < H, B = 0, insert(A, L, NewL, IncL), IncL = 1, Inc = 1,
  Y = [H, NewL, R, -1].
```

And, similarly, clause 6 must be replaced by the two clauses:

```
insert(A, T, Y, Inc)
  :- T = [H, L, R, B], A > H, B < 0, insert(A, R, NewR, IncL), IncL = 1, Inc = 0,
  Y = [H, L, NewR, 0].
insert(A, T, Y, Inc)
  :- T = [H, L, R, B], A > H, B = 0, insert(A, R, NewR, IncL), IncL = 1, Inc = 1,
  Y = [H, L, NewR, 1].
```

```
{
insert(A,T,Y,Inc) :- {
                     null(T), Y=[A,nil,nil,0], T=[H,L,R,B],
                     A<H, A>H, B>0, B=0, B<0, Inc=0, Inc=1, IncL=0, IncL=1,
                     insert(A,L,NewL,IncL), insert(A,R,NewR,IncL),
                     Y=[H,NewL,R,0], Y=[H,NewL,R,-1], Y=[NH,NL,NR,0], Y=[H,NewL,R,B],
                     Y=[H,L,NewR,0], Y=[H,L,NewR,1], Y=[H,L,NewR,B],
                     rebalL([H,NewL,R],[NH,NL,NR]), rebalR([H,L,NewR],[NH,NL,NR]),
                     }
}
```

Fig. 4. Set of alternative programs for testing *insert*.

Moreover, it is worth noting that, if literal "Inc = 0" in clause 3 is replaced with literal "Inc = 1," then program P_e would be "almost" correct. In fact, it would always output balanced trees, but the balance factor and the increase factor could be wrong. Literal "Inc = 0" is responsible for producing unbalanced trees if an element is inserted as a left leaf and if the resulting tree needs rebalancing.

Let us suppose that the two rebalancing procedures *rebalL* and *rebalR* are known to be correct. As a consequence, they do not have to be tested and may be put in the set of *known clauses* (Section 3.1). This is also called, in the ILP literature, *background knowledge*. Through the Clause Set notation, we define the set of alternative programs $\mathcal{P}$ on the basis of P_e by just using all of the literals occurring in the part of the program that was to be tested, i.e., the first seven clauses.[2] The literal A = H was not used, as we assume that the inserted element is not already present in the tree. As noted in Section 3.1, even if not stated explicitly each mutation program will always contain the clauses in the background knowledge (the known clauses), plus a subset of the possible clauses, defined by the clause set of Figure 4. As a consequence, $\mathcal{P}$ contains $2^{2^{23}}$ alternative programs. Among the other alternatives, there are versions of the correct implementation. However, $\mathcal{P}$ does not contain P_e (in fact, this is not required by the test case generation system), but equivalent versions of P_e (as P_{15} below) are in $\mathcal{P}$.

To generate the test cases, a simple balanced-tree generator is used. Trees are generated in order of growing complexity w.r.t. the number of nodes. To test the *insert* program it is not important which elements form an input tree, but their relative values. As a consequence, the same keys are used by the generator to build a balanced tree. For each input tree, an appropriate element among those available is chosen, in order to produce an insertion into all possible positions for that tree. The second column of Table I clarifies this strategy. Only eight different numeric keys are used. Each input tree has always the same root key. Recursively, this is also true for each subtree, if present. The remaining keys are used to produce an

[2]Actually, literals $B \geq 0$ and $B \leq 0$ are split into $B = 0$, $B > 0$, and $B < 0$, in order to enlarge the set of allowed mutations.

Table I

examples	input pair	Pi-1 output	Pe output	time (sec)
e1:	50 -> nil	no	50 1	0
e2(*):	25 -> 50	25 1	-50 0 / 25	6
e3:	75 -> 50	no	+50 1 \ 75	26
e4:	90 -> +50 \ 75	+50 1 \ +75 \ 90	75 0 / \ 50 90	136
e5(*):	75 -> -50 / 25	no	+50 1 / \ 25 75	159
e6(*):	65 -> +50 \ 75	65 0 / \ 50 75	+50 0 \ -75 / 65	223
e7(*):	35 -> -50 / 25	-50 0 / +25 \ 35	35 0 / \ 25 50	292
e8(*):	15 -> 50 / \ 25 75	25 0 / \ 15 50 \ 75	50 0 / \ -25 75 / 15	354
e9(*):	35 -> 50 / \ 25 75	35 0 / \ 25 50 \ 75	-50 0 / \ +25 75 \ 35	554
e10(*):	65 -> 50 / \ 25 75	+50 1 / \ 25 -75 / 65	50 0 / \ 25 -75 / 65	623
e11:	90 -> 50 / \ 25 75	75 0 / \ 50 90 / 25	+50 1 / \ 25 +75 \ 90	919

Table I—*Continued*

examples	input pair	Pi-1 output	Pe output	time (sec)
e12(*):	5 -> -50 / \ -25 75 / 15	-50 0 / \ 15 75 / \ 5 25	-50 0 / \ -25 75 / -15 / 5	1171
e13(*):	65 -> -50 / \ -25 75 / 15	+50 1 / \ -25 -75 / / 15 65	-50 0 / \ -25 -75 / / 15 65	1327
e14(*):	5 -> -50 / \ +25 75 \ 35	no	-50 0 / \ -25 75 / \ 5 35	1478
e15(*):	5 -> +50 / \ 25 -75 / 65	-50 0 / \ -25 -75 / / 5 65	+50 0 / \ -25 -75 / / 5 65	1810

insertion into all possible positions. As a consequence of using always the same values, we can keep small the number of input pairs added by "covers." This is because input variables in the body of clauses handled by "covers" are instantiated always to the same values, and therefore such values need to be added to T only once. For example, in Table I there are three input pairs (examples e_{12}, e_{14}, and e_{15}) using the numeric key 5, but only the first time (i.e., for example e_{12}) the input pair ⟨5, nil⟩ is added to T(insert) (i.e., to T_{12}).

As before, the test case generation starts with an empty test set T_0 of input values, and the induction procedure outputs the empty program[3] P_0. Input pairs ⟨element, balanced tree⟩ are enumerated in order of growing complexity of the input tree. The first such pair for which P_e and P_0 differ is ⟨50, nil⟩. For this input, P_0 produces no output, whereas P_e outputs

Y = [50, nil, nil, 0], Inc = 1.

As a consequence, the new test set becomes $T_1 = T_0 \cup \{\langle 50, \text{nil}\rangle\}$. M(E(T1, P_e),$\mathcal{P}$) is called again, yielding P_1:

P_1: insert(A, T, Y, Inc) :- Y = [A, nil, nil, 0], Inc = 1.

[3]Actually the empty program plus the background knowledge.

plus clauses 8–15 for *rebalL* and *rebalR* as listed in P_e. The first enumerated input pair such that P_e and P_1 differ is ⟨25, [50, nil, nil, 0]⟩. For this input pair P_1 outputs

Y = [25, nil, nil, 0], Inc = 1

whereas P_e outputs

Y = [25, [50, nil, nil, 0], nil, −1], Inc = 0.

Also, this test case shows one of the errors of P_e, since the increase factor variable "Inc" should be set to 1 and not to 0. However, this may not be easily noticed, as the output tree is balanced. We set $T_2 = T_1 \cup \{\langle 25, [50, nil, nil, 0]\rangle\}$, and $M(E(T_2, P_e), \mathcal{P})$ is called, yielding P_2:

P_2:
insert(A, T, Y, Inc)
:- null(T), Y = [A, nil, nil, 0], Inc = 1.
insert(A, T, Y, Inc)
:- T = [H, L, R, B], A < H, insert(A, L, NewL, IncL), Inc = 0, Y = [H, NewL, R, −1].

plus clauses 8–15 for *rebalL* and *rebalR* as listed in P_e.

The test case generation procedure goes on in this way, as shown in Section 4. Table I summarizes the obtained results. In this table, trees are represented graphically. In each tree, node keys are positive integers. A positive or a negative sign is used to indicate that the balance factor of that node is +1 or −1. No sign means a 0 balance factor. The increase factor is reported next to the corresponding output tree. For each entry i ($1 \leq i \leq 15$), the second column contains the first input pair found (according to the adopted enumeration) for which P_{i-1} and P_e give different outputs. The third column is the output pair of P_{i-1} on the ith input pair. The fourth column is the output pair of P_e. As before, each test set T_i, used by the induction procedure M to learn P_i, is obtained by adding the ith input pair to T_{i-1}. The 15 examples $e_1, \ldots, e_{15}$ represent an adequate test case set for P_e. Examples marked with an asterisk show errors of P_e. Fifteen more examples have been added during the learning process by "covers." Experiments have been done on a Sun SPARCstation 5, with the induction procedure[4] written in C-Prolog (interpreted). The fifth column reports times (in seconds) required to learn program P_{i-1}. Program P_{15}, reported below, has been learned in 1831 seconds. This is the first program equivalent to P_e found by the induction procedure:

P_{15}:
1) insert(A, T, Y, Inc) :- null(T), Y = [A, nil, nil, 0], Inc = 1.

[4]The induction procedure is thoroughly described in Bergadano and Gunetti [1993] and is available through ftp at the Machine Learning archive at ftp.gmd.de. Send mail to ml-archive@gmd.de for instructions about the use of the archive.

```
2) insert(A, T, Y, Inc)
     :- T = [H, L, R, B], A < H, B < 0, insert(A, L, NewL, IncL), IncL = 1,
     Inc = 0, rebalL([H, NewL, R], [NH, NL, NR]), Y = [NH, NL, NR, 0].
3a) insert(A, T, Y, Inc)
     :- T = [H, L, R, B], A < H, B = 0, insert(A, L, NewL, IncL), IncL = 1,
     Inc = 0, Y = [H, NewL, R, -1].
3b) insert(A, T, Y, Inc)
     :- T = [H, L, R, B], A < H, B > 0, insert(A, L, NewL, IncL), IncL = 1,
     Inc = 0, Y = [H, NewL, R, -1].
4) insert(A, T, Y, Inc)
     :- T = [H, L, R, B], A < H, insert(A, L, NewL, IncL), IncL = 0, Inc = 0,
     Y = [H, NewL, R, B].
5) insert(A, T, Y, Inc)
     :- T = [H, L, R, B], A > H, B > 0, insert(A, R, NewR, IncL), IncL = 1,
     Inc = 0, rebalR([H, L, NewR], [NH, NL, NR]), Y = [NH, NL, NR, 0].
6a) insert(A, T, Y, Inc)
     :- T = [H, L, R, B], A > H, B = 0, insert(A, R, NewR, IncL), IncL = 1,
     Inc = 1, Y = [H, L, NewR, 1].
6b) insert(A, T, Y, Inc)
     :- T = [H, L, R, B], A > H, B < 0, insert(A, R, NewR, IncL), IncL = 1,
     Inc = 1, Y = [H, L, NewR, 1].
7) insert(A, T, Y, Inc)
     :- T = [H, L, R, B], A > H, insert(A, R, NewR, IncL), IncL = 0, Inc = 0,
     Y = [H, L, NewR, B].
```

plus clauses 8–15 for *rebalL* and *rebalR* as in P_e.

It is easy to check that programs P_e and P_{15} are equivalent (in P_{15} we have just reordered learned clauses and literals to make the comparison easier). In particular, clauses 3a and 3b (6a and 6b) of P_{15} are equivalent to clause 3 (6) of P_e. In fact, clause 3 (6) merges the two tests "B = 0" and "B > 0" ("B < 0") made separately in 3a and 3b (6a and 6b). However, the testing procedure has tested P_{15} and P_e for equivalence on all possible insertion cases into balanced trees from the empty tree up to trees of maximum height 3. As the outputs are the same, P_{15} is removed from $\mathcal{P}$, and no new test case is generated. Again, a few more programs equivalent to P_e are found, and then no other program consistent with T_{15} can be found; and M fails. Of the 15 examples generated, examples e_2, e_5, e_6, e_8, e_9, e_{10}, e_{12}, e_{13}, e_{14}, and e_{15} isolate the errors of P_e. Of particular interest are examples e_6 and e_{12}. For the input pairs of these two examples P_e outputs unbalanced trees. This is the consequence of the error in clause 3 of P_e discussed above.

Finally, it is worth noting that the learning times for programs P_0–P_{15} grow more or less linearly with the size and complexity of the target program. This is a consequence of the fact that clauses are learned independently of each other and shows that the approach is feasible w.r.t. the size and/or complexity of the program to be tested.

6. SCALABILITY

In this section we discuss whether the method can be scaled to test larger programs, and we describe additional experiments.

The first thing that must be noted is that, through the mechanism of Clause Sets, the set of alternative programs is actually tailored to the particular program P under testing. In fact, this is precisely what is done in the field of Theory Revision [Bergadano and Gunetti 1994a; Wrobel 1994]. The underlying assumption, both of Theory Revision and of our method, is that P may be wrong, but it is plausible to assume it is not completely wrong: the correct program will not be something completely different from P. Consequently, it can be used as a starting point: a template to define possible mutations by using its basic constituents. In the case of logic programs, they might be the clauses the literals and the terms P is made of. Actually, through the use of the Clause Sets language this can be done very easily, even automatically, by surrounding clauses, literals, and terms with brackets. In the simplest case, $\mathcal{P}$ may be defined by just using all of the literals in P, as we did for the *insert* clauses of program P_e of the previous section. Of course, in that case the size of $\mathcal{P}$ is exponential in the number of literals, and the induction procedure turns out to be exponential in the size of P. However, the tester may well be supposed to have some knowledge about the program to be tested. She or he may know some part of the program (i.e., some clause) to be correct and avoid testing it (i.e., put the clauses in the background knowledge). Other parts of the code (i.e., other clauses) may be suspected to be wrong, and a set of mutants for them may be defined with a clause set. A finer control is provided by Literal Sets. Possible faulty literals are put into a literal set, whereas correct literals may be left outside brackets, as part of any allowed mutation. The actual number of mutations defined in this way depends on the particular problem and on the knowledge the tester has on it, but this number can be controlled through a judicious use of the language. As an example, consider again program P_e of the previous section. By inspecting the part of code under testing we may observe the following:

(a) The nonrecursive clause of *insert* seems reasonably correct and may be put in the background knowledge;[5]

(b) the literal "T = [H, L, R, B]" is used to split the input tree into its basic elements and may well be part of every clause of every mutation program (i.e., it does not have to be tested);

(c) the subprogram {2, 3, 4} and the subprogram {5, 6, 7} are mutually exclusive, since the first tests for "A < H" and the second for "A > H." We may then define a mutation set for {2, 3, 4} using just the literals that occur there, and where each allowed clause has to test for "A < H." In the same way a mutation test may be defined for {5, 6, 7}, with each clause testing for "A > H."

[5]In fact this clause corresponds to the insertion of an element in the empty tree.

Note that this reasoning does not assume nor imply any knowledge of the faulty parts of the code, but it leads to a reformulation of the clause set of Figure 4 as follows:

```
{
insert(A, T, Y, Inc)
  :- T = [H, L, R, B], A < H,
  {
      B = 0, B < 0, B > 0, Inc = 0, Inc = 1, IncL = 0, IncL = 1,
      Y = [H, NewL, R, 0], Y = [H, NewL, R, −1], Y = [NH, NL, NR, 0], Y =
      [H, NewL, R, B],
      rebalL([H, NewL, R], [NH, NL, NR]), insert(A, L, NewL, IncL),
  }
insert(A, T, Y, Inc)
  :- T = [H, L, R, B], A > H,
  {
      B = 0, B < 0, B > 0, Inc = 0, Inc = 1, IncL = 0, IncL = 1,
      Y = [NH, NL, NR, 0], Y = [H, L, NewR, 0], Y = [H, L, NewR, 1], Y = [H,
      L, NewR, B],
      rebalR([H, L, NewR], [NH, NL, NR]), insert(A, R, NewR, IncL),
  }
}
```

This clause set defines now a mutation set $\mathcal{P}$ of "only" $2^{(2^{13}+2^{13})}$ alternative programs. $\mathcal{P}$ still contains a correct version of *insert* and can be used in place of the one of Section 5 with the same results, but with shorter learning times.

In general, we think the sketched situation is better than just having a standard set of mutations allowed. If the set of mutations is independent of the program to be tested, it may be the case that a particular kind of mutation is meaningless. Nonetheless, it will be applied, resulting in a waste of time. Consider again the *insert* example, where tests for *equal*, *greater_than*, and *smaller_than* are used. A "reasonable" set of mutations would replace any test with a different one. However, this would lead to a much larger number of clauses and hence to a much larger $\mathcal{P}$. For example, there is no need to build any mutation program checking whether "A = H," because of the assumption that an inserted element does not already occur in the input tree.

On the other hand, our approach does not rule out the possibility to employ a predefined set of mutations. A set of predefined "template literals" may be available and included together with—or in place of—the literals from P in a literal set, after a suitable instantiation. For example, a template such as "equal(var_1, var_2)" or "greater_or_equal(var_1, var_2)" may be instantiated to (some of) the variables found in P and used to allow for other mutation programs in $\mathcal{P}$. This is what we did with the last three literals of Figure 3 when testing *merge*.

As a second point, from the definition of the Test Case Generation Procedure in Section 3 it turns out that in the worst case the induction

procedure must be invoked $|\mathcal{P}|$ times. Of course this worst case is definitely unfeasible, as even for the simplest cases $\mathcal{P}$ may contain a very large number of programs. However, the actual situation is quite different. Consider the *merge* example of Section 4. There, $\mathcal{P}$ contains $2^{2^{18}}$ programs, but the test case generation procedure only loops seven times. In the case of *insert* we have $2^{2^{23}}$ mutation programs and 15 invocations of the induction procedure. In the experiment reported in Section 6.1 $2^{(2^{23}+2^{8}+2^{4})}$ alternative programs require 15 invocations of M. On the basis of all of the experiments we made, this seems to be the rule: even a very huge number of mutations leads to a very limited number of loops of the test case generation procedure.

Finally, a few words about the possibility of using a parallel implementation of the induction procedure M used in the reported experiments. The basic algorithm described in Section 3.2 is implemented in a system called FILP [Bergadano and Gunetti 1993]. FILP works in two main steps: a completion phase and an induction phase. In the completion phase a set of missing examples ME is added by the function covers(C, e), on the basis of E(T, P) and $\mathcal{P}$. In the induction phase a program $P \in \mathcal{P}$ consistent with E(T, P) $\cup$ ME is found. The missing examples are queried to the user or, for the case of test case generation, to the program under testing P. The completion step is made at the beginning, and it takes only a very small amount of the whole learning time.[6] For example, when learning P_{15} of the previous section, the completion phase takes about 60 seconds, whereas the induction phase takes the remaining up to 1831 seconds. What is important here is that, in the induction phase, clauses are learned independently of each other. As a consequence, this step can be performed in parallel on a pool of CPUs or hosts, each one learning a clause covering a different example. If a sufficient number of CPUs is available, the time needed to synthesize a program shrinks to the maximum time needed to synthesize one of its clauses. A parallel version of FILP, called PARFILP, is at the moment under development [Bergadano and Gunetti 1995]. Although extensive experiments are not available yet, and although expectations about parallel implementations of algorithms must always be taken with a grain of salt, PARFILP is expected to achieve a speedup almost proportional to the number of employed CPUs.

We conclude this section with an example of the scalability of the approach.

6.1 Scaling the *Insert* Testing Problem

Consider the *insert* program of the previous section. There, the two rebalancing procedures, *rebalL* and *rebalR*, were supposed to be correct and hence had not undergone the test case generation task. Now, we want to make them part of the code to be tested. Actually, these two procedures are correct, but of course that is not known to the testing procedure. Suppose, for example, that we want to test only part of the code of the rebalancing

[6]Of course, in the case of test case generation this also depends on the efficiency of P.

procedures; let us say the first two clauses of *rebalL* and *rebalR*. As was done for *insert*, we just use the literals of these particular clauses. Hence, clauses 8 and 9 are removed from the background knowledge, and a set of their mutations is defined through the following clause set:

```
{
rebalL(H, NL, R, Y)
  :- {
      NL = [LH, LL, LR, −1], YR = [H, LR, R, 0], Y = [LH, LL, YR, 0],
      NL = [LH, LL, LR, 1], LR = [LRH, LRL, LRR, 0],
      YL = [LH, LL, LRL, 0], YR = [H, LRR, R, 0], Y = [LRH, YL, YR, 0],
      }
}
```

Similarly, the set of mutations for clauses 12 and 13 is defined by:

```
{
rebalR(H, L, NR, Y)
  :- {
      NR = [RH, RL, RR, 1], YL = [H, L, RL, 0], Y = [RH, YL, RR, 0],
      NR = [RH, RL, RR, −1], RL = [RLH, RLL, RLR, 0],
      YL = [H, L, RLL, 0], YR = [RH, RLR, RR, 0], Y = [RLH, YL, YR, 0],
      }
}
```

Clauses 10, 11, 14, and 15 remain part of the background knowledge. The two new clause sets define 2^8 different clauses each, the first including clauses 8 and 9 and the second including clauses 12 and 13. Now, any program synthesized by the induction procedure can be made of clauses from the two new clause sets and from the clause set for *insert* given in Figure 4 (plus the clauses left in the background knowledge). Consequently, the new set of mutation programs $\mathcal{P}$ contains $2^{(2^{23}+2^8+2^8)}$ alternative programs.

The test case generation is repeated exactly as was done for P_e in Section 5. The same set of examples $\{e_1, \ldots, e_{15}\}$ is found, and 27 more examples are automatically added through "covers": fifteen for *insert* (the same that were added in Section 5), seven for *rebalL*, and five for *rebalR*. Since the tested subprogram {8, 9, 12, 13} is correct, no example showing errors in its clauses is found. After the generation of e_{15}, program P_{15} equivalent to P_e is discovered in 2075 seconds. The 244 extra seconds (w.r.t. the time required to learn P_{15} in the previous experiment) are employed to synthesize the four rebalancing clauses. In the previous steps of the test case generation task, when learning programs $P_0, \ldots, P_{14}$, a proportional amount of time varying from 0 to 244 seconds is required to learn the needed rebalancing clauses.

7. CONCLUSION

A test case generation method was presented that is based on the inductive inference of programs from input/output examples. A *set* $\mathcal{P}$ of *alternative programs* is required and may be defined on the basis of the program P to

be tested. The generated test set is adequate in the sense that it distinguishes the program to be tested from all alternatives. If $\mathcal{P}$ contains at least one correct implementation, then the obtained test set will also be reliable, i.e., it will isolate any errors that may be present. The method is related to fault-based test generation techniques.

The efficiency of the method depends mainly on the size of the space $\mathcal{P}$ of alternative programs. This is due to the fact that the most critical step in the described test generation process is the induction of a program $P' \in \mathcal{P}$ consistent with E(T, P), for the partial test set T. The complexity of this step is proportional to the number of possible clauses and is therefore logarithmic in the number $|\mathcal{P}|$ of possible programs. For every generated test case, one inductive inference step has to be performed.

By contrast, the size of the program P to be tested does not directly influence the complexity of the method. This may happen indirectly, in the sense that a more complex program may contain more errors and may lead the tester to define a larger set $\mathcal{P}$ of alternatives. Nevertheless, it is reasonable to assume that more-complex programs need more testing effort and more time devoted to defining a restricted, but still meaningful, set of alternative programs. The method scales up well when the size or the complexity of the program P to be tested grows. It stops being practical if the number of alternatives or possible errors becomes too large. This should be contrasted with usual approaches to fault-based testing, which may cause problems when the size of the program to be tested is too large. In fact, the larger the program, the higher the number of syntactical mutations that may be applied systematically to every part of the code.

The test case generation method presented in this article was made possible by recent advances in inductive logic programming, providing the basis for practical procedures for inducing programs from finite sets of input/output examples. The set $\mathcal{P}$ of alternative programs may then be viewed as a space of allowed inductive hypotheses, and the induced program P′ will have to belong to that space of logic programs. The program induction method described here has two important novelties: (1) it uses strong functionality and termination constraints, so that only positive examples are needed, and (2) it uses the program P to be tested for generating the missing examples. The second point deserves a quite interesting concluding remark.

The induction method is based on the extensional interpretation of the predicates occurring in a clause: given a clause "p(X, Y) :- . . . , q(Z, W), . . ." the procedure "covers" will treat the literal q(Z, W) either by calling external modules or by using the examples provided for q. This extensional interpretation is what allows the system to learn one clause at a time, without using previously generated clauses, with a complexity that is only logarithmic in the number $|\mathcal{P}|$ of possible programs. The price paid for it is that the examples of q(Z, W) must be available for all instantiations j of Z obtained by calling p(i, Y) with the given clause, for all $i \in T(p)$. Only in this context of program testing can we do that, because we can obtain any missing example by calling q(j, W) with the program P that we are testing. For this rather technical reason we contrast the concluding remark of

Weyuker [1983] which says that, if program induction techniques were sufficiently developed, there would be no need for testing, as one would generate correct programs automatically from input/output examples, and there would be no need to write a program by hand and test it afterward. But our program induction procedure is practical only because it can rely on the program to be tested.

ACKNOWLEDGMENT

We thank the anonymous referees who suggested many improvements to the article.

REFERENCES

ADE, H. AND DE RAEDT, L. 1995. Declarative bias for specific-to-general ILP systems. *Mach. Learning 20,* 1–2, 119–154.

BAZZICHI, F. AND SPADAFORA, I. 1982. An automatic generator for compiler testing. *IEEE Trans. Softw. Eng. 8*, 4, 343–353.

BERGADANO, F. 1993. Inductive database relations. *IEEE Trans. Data Knowl. Eng. 5*, 6, 969–972.

BERGADANO, F. AND GUNETTI, D. 1993. An interactive system to learn functional logic programs. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence* (Chambéry, France), R. Bajcsy, Ed. Morgan-Kaufmann, San Mateo, Calif., 1044–1049.

BERGADANO, F. AND GUNETTI, D. 1994a. Intensional theory revision. In *Proceedings of the ECML-94 Workshop on Theory Revision*, S. Wrobel, Ed. Univ. of Catania, Catania, Italy.

BERGADANO, F. AND GUNETTI, D. 1994b. Learning clauses by tracing derivations. In *Proceedings of the 4th International Workshop on Inductive Logic Programming* (Bonn, Germany).

BERGADANO, F. AND GUNETTI, D. 1995. PARFILP: A parallel implementation of an ILP system. Tech. Rep. 95.12.15, CS Dept., Univ. of Torino, Torino, Italy.

BERGADANO, F. AND GUNETTI, D. 1996. *Inductive Logic Programming: From Machine Learning to Software Engineering*. MIT Press, Cambridge, Mass.

BERGADANO, F., GIORDANA, A., AND SAITTA, L. 1988. Automated concept acquisition in noisy environments. *IEEE Trans. Patt. Anal. Mach. Intell. 10*, 4, 555–578.

BERGADANO, F., GUNETTI, D., NICOSIA, M., AND RUFFO, G. 1995. Learning logic programs with negation as failure. In *Proceedings of the 5th International Workshop on Inductive Logic Programming* (Leuven, Belgium). Katholieke Universiteit, Leuven, Belgium.

BERGADANO, F., MATWIN, S., MICHALSKI, R. S., AND ZHANG, J. 1992. Learning two-tiered descriptions of flexible concepts: The Poseidon System. *Mach. Learning 8*, 5–43.

BUDD, T. A. AND ANGLUIN, D. 1982. Two notions of correctness and their relation to testing. *Acta Informatica 18*, 31–45.

CHERNIAVSKY, J. C. AND SMITH, C. H. 1987. A recursion theoretic approach to program testing. *IEEE Trans. Softw. Eng. 13*, 7, 777–784.

CHOQUET, N. 1986. Test data generation using Prolog with constraints. In *Proceedings of the Workshop on Software Testing*. ACM, New York, 132–141.

CLARKE, L. A., HASSELL, J., AND RICHARDSON, D. J. 1982. A close look at domain testing. *IEEE Trans. Softw. Eng. 8*, 4, 380–390.

DEMILLO, R. A. 1989. Test adequacy and program mutation. In *Proceedings of the International Conference on Software Engineering*. IEEE Computer Society, Washington, D.C., 355–356.

DEMILLO, R. A. AND OFFUTT, A. J. 1991. Constraint-based automatic test data generation. *IEEE Trans. Softw. Eng. 17*, 9, 900–910.

DEMILLO, R. A., LIPTON, R. J., AND SAYWARD, F. G. 1978. Hints on test data selection: Help for the practicing programmer. *IEEE Comput. 11*, 4 (Apr.), 34–41.

DENNEY, R. 1991. Test case generation from Prolog-based specifications. *IEEE Softw. 8*, 2, 49–57.

DE RAEDT, L., SABLON, G., AND BRUYNOOGHE, M. 1991. Using interactive concept learning for knowledge base validation and verification. In *Validation, Verification and Testing of*

Knowledge Based Systems, M. Ayel and J. P. Laurent, Eds. John Wiley and Sons, London, 177–190.

GOLD, E. M. 1967. Language identification in the limit. *Inf. Contr. 10*, 5, 447–474.

GORLICK, M. M., KESSELMAN, C. F., MAROTTA, D. A., AND PARKER, D. S. 1990. Mockingbird: A logical methodology for program testing. *J. Logic Program. 8*, 95–119.

HAMLET, R. G. 1981. Reliability theory of program testing. *Acta Informatica 16*, 1, 31–43.

HAYES, I. I. 1986. Specification directed module testing. *IEEE Trans. Softw. Eng. 12*, 1, 124–133.

HOFFMAN, D. M. AND STROOPER, P. 1991. Automated module testing in Prolog. *IEEE Trans. Softw. Eng. 17*, 9, 934–943.

HOWDEN, W. E. 1976. Reliability of the path analysis testing strategy. *IEEE Trans. Softw. Eng. 2*, 3, 208–215.

HOWDEN, W. E. 1980. Functional program testing. *IEEE Trans. Softw. Eng. 6*, 2, 162–169.

HOWDEN, W. E. 1982. Weak mutation testing and completeness of test sets. *IEEE Trans. Softw. Eng. 8*, 4, 371–379.

KILPELAINEN, P. AND MANNILA, H. 1990. Generation of test cases for simple Prolog programs. *Acta Cybernetica 9*, 3, 235–246.

KING, K. N. AND OFFUTT, A. J. 1991. A Fortran language system for mutation-based software testing. *Softw. Pract. Exper. 21*, 7, 685–718.

KIRSCHENBAUM, M. AND STERLING, L. 1991. Refinement strategies for inductive learning of simple Prolog programs. In *Proceedings of the International Joint Conference on Artificial Intelligence*, J. Mylopoulos and R. Reiter, Eds. Morgan Kaufmann, San Mateo, Calif., 757–761.

KNUTH, D. E. 1973. *The Art of Computer Programming*. Vol. 3, *Sorting and Searching*. Addison-Wesley, Reading, Mass.

MICHALSKI, R. S. 1980. Pattern recognition as rule-guided inductive inference. *IEEE Trans. Patt. Anal. Mach. Intell. 2*, 349–361.

MICHALSKI, R. S. 1983. A theory and methodology of inductive learning. *Artif. Intell. 20*, 2, 111–161.

MORELL, L. J. 1988. Theoretical insights into fault-based testing. In *Proceedings of the Workshop on Software Testing, Verification and Analysis*. ACM, New York, 45–62.

MUGGLETON, S., Ed. 1991. *Inductive Logic Programming*. Academic Press, New York.

NEDELLEC, C. AND ROUVEIROL, C. 1994. Specifications of the Haiku system. Tech. Rep. D.LRI.2b, LRI, Universite de Paris-Sud, Paris, France.

OFFUTT, A. 1989. The coupling effect: Fact or fiction? In *Proceedings of the Workshop on Software Testing, Analysis and Verification*. ACM, New York, 131–140.

QUINLAN, R. 1990. Learning logical definitions from relations. *Mach. Learning 5*, 3, 239–266.

STROOPER, P. AND HOFFMAN, D. 1991. Prolog testing of C modules. In *Proceedings of the International Symposium on Logic Programming*, V. Saraswat and K. Ueda, Eds. MIT Press, Cambridge, Mass., 596–608.

SUMMERS, P. D. 1977. A methodology for LISP program construction from examples. *J. ACM 24*, 1, 161–175.

TSAI, W. T., VOLOVIK, D., AND KEEFE, T. F. 1990. Automated test case generation for programs specified by relational algebra queries. *IEEE Trans. Softw. Eng. 16*, 3, 316–324.

UTGOFF, P. 1986. *Machine Learning of Inductive Bias*. Kluwer Academic, Dordrecht, The Netherlands.

VALIANT, L. G. 1984. A theory of the learnable. *Commun. ACM 27*, 11 (Nov.), 1134–1142.

WEYUKER, E. J. 1983. Assessing test data adequacy through program inference. *ACM Trans. Program. Lang. Syst. 5*, 4, 641–655.

WROBEL, S. 1994. *Concept Formation and Knowledge Revision*. Kluwer, Dordrecht, The Netherlands.

Received May 1994; revised October 1995; accepted January 1996