Interface Mutation: An Approach for Integration Testing

Márcio E. Delamaro, *Member*, *IEEE Computer Society*,
José C. Maldonado, *Member*, *IEEE Computer Society*, and Aditya P. Mathur, *Member*, *IEEE*

Abstract—The need for test adequacy criteria is widely recognized. Several criteria have been proposed for the assessment of adequacy of tests at the unit level. However, there remains a lack of criteria for the assessment of the adequacy of tests generated during integration testing. We present a mutation-based interprocedural criterion, named Interface Mutation (IM), suitable for use during integration testing. A case study to evaluate the proposed criterion is reported. In this study, the UNIX sort utility was seeded with errors and Interface Mutation evaluated by measuring the cost of its application and its error revealing effectiveness. Alternative IM criteria using different sets of Interface Mutation operators were also evaluated. While comparing the error revealing effectiveness of these Interface Mutation-based test sets with same size randomly generated test sets we observed that in most cases Interface Mutation-based test sets are superior. The results suggest that Interface Mutation offers a viable test adequacy criteria for use at the integration level.

Index Terms—Test adequacy criteria, mutation testing, mutation analysis, integration testing, testing tool.

1 Introduction

Software testing may be considered to be an incremental activity that pervades most, if not all, of the software development cycle. During the development of a software system, one begins with the development of individual modules or units. For our purpose, we assume that a unit is a testable program that provides functionality to a software system. There may be cases when already developed and tested units from a previous project are selected for reuse in a new project. We assume that a unit is tested before it is integrated with other units to form a subsystem or the desired system itself. In the discussion below, we do not distinguish between a subsystem and a system. After having integrated units into a subsystem, one tests the subsystem for conformance to its specifications. During the testing of a unit one constructs one or more test inputs and executes the unit against these inputs in some suitably chosen order. The collection of test inputs is referred to as a *test set*. A similar procedure is used for testing a subsystem. In either case, one develops a test set, a collection of test cases, to test the unit or the subsystem in hand.

Considerable research has gone into the development of criteria for evaluating how "good" a test set is for a given unit. As a result, several criteria are available to evaluate a test set for a given program against a given specification. Such criteria are also known as *test adequacy criteria* or simply *adequacy criteria*. For example, data flow [22] and mutation testing [8], [15] provide a variety of adequacy criteria. A tester may use one or more of these criteria to assess the adequacy of a test set for a given unit and then decide whether to stop testing this unit or to enhance the test set by constructing additional test cases needed to satisfy a criterion. In principle, such criteria and the procedure may also be applied for the assessment of tests for subsystems [12], [14], [16], [17]. Concerns of inefficiency due to program size, redundancy because multiple test teams might test the same part of the code, and incompleteness because the large program size might prohibit testers from applying the criteria exhaustively, need to be dealt with.

Interface Mutation, abbreviated as IM, is proposed to evaluate how well the interactions between various units have been tested. Interface Mutation is an extension of mutation testing and is applicable, by design, to software systems composed of interacting units. The development of Interface Mutation was motivated by the need to assess test sets for subsystems which come about due to the integration of two or more units. For a given subsystem S that consists of two or more units, a related question is: *How good is a test set for S*?

At the unit level, mutation-based criteria have been found to be relatively more powerful in their error detection effectiveness than other code coverage based criteria [19], [23], [27], [29]. However, in practice, the large number of mutant programs that need to be executed can be a strong deterrent in the use of mutation testing. Alternative approaches [23], [25], [26], [28], [20], [21] have shown how to reduce the number of mutant programs and keep the cost of testing and the reduction in error detection effectiveness within acceptable limits. However, these approaches are also too expensive when applied to completely or partially

- *M.E. Delamaro is with DIN-UEM, Department of Informatics, Maringá State University, Maringá, PR, 87020-900, Brazil. E-mail: delamaro@din.uem.br.*
- *J.C. Maldonado is with the ICMC-USP, Department of Computer Sciences and Statistics, University of São Paulo, São Carlos, SP, 13560-970, Brazil. E-mail: jcmaldon@icmc.sc.usp.br.*
- *A.P. Mathur is with the Software Engineering Research Center, 1398 Department of Computer Sciences, Purdue University, W. Lafayette, IN 47907. E-mail: apm@cs.purdue.edu.*

Manuscript received 29 Aug. 1997; revised 22 Sept. 1998; accepted 27 May 1999.
Recommended for acceptance by M.L. Soffa.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 105567.

integrated software of the order of several hundred thousand lines of code. Our approach to overcome this problem is based on three key ideas: 1) restrict mutant operators to model integration errors, 2) one at a time, test connections between two units separately, and 3) apply the integration mutant operators only on those parts of the units that are related to unit interfaces, such as function calls, parameters, or global variables.

Using Proteum [5], a mutation testing tool for C programs, a pilot study was conducted to evaluate Interface Mutation [6]. In this study, the Unix SORT utility was seeded with several integration errors and then tested with Interface Mutation. The results indicate that: 1) this approach contributes to reducing the cost of mutation testing, thereby motivating further investigation to make it suitable for testing large systems and 2) the error revealing effectiveness was as good as that of mutation testing in that 96 percent of the test sets created using Interface Mutation revealed the errors and each seeded error was revealed by at least one of the test sets. The pilot study also revealed that further examination of costs associated with mutant generation and execution is necessary to make Interface Mutation an efficient and cost-effective criterion for test set evaluation during the integration phase.

Based on earlier experience, we have defined a mutant operator set and developed *PROTEUM/IM*. This tool supports the application of Interface Mutation criterion and exploration of alternative mutation criteria [21], [25] at the integration testing phase.

In summary, the objective of this paper is to introduce the Interface Mutation criterion to support, complement, and improve integration testing. The ideas and concepts are illustrated by a case study. The paper is organized as follows: Initially, we provide a brief overview of mutation testing. In the following two sections, Sections 3 and 4, we discuss integration testing, types of integration errors, and the most relevant approaches for integration testing. Section 5 presents our approach to integration testing based on Interface Mutation. Section 6 presents the set of mutant operators designed specifically for Interface Mutation. Section 7 reports a case study, conducted using *PROTEUM/IM*, to evaluate Interface Mutation. Our conclusions appear in Section 8.

2 Mutation Testing

Mutation testing is an error-based testing adequacy criterion proposed by DeMillo et al. [8], initially with the name "Mutant Analysis." Given a program P, a set of alternative programs called $\Phi(P)$ is considered in order to measure the adequacy of a test set T, according to Budd's definition [2]:

> A test set T is adequate to P in relation to $\Phi(P)$ if for each program $Q \in \Phi(P)$, either Q is equivalent to P or Q differs from P on at least one test case $t \in T$.

In practice, the criterion is applied by creating the set of alternative programs called mutants of P. The mutants differ from P only on simple syntactic changes determined by a set of **mutant operators**. In accordance with the **Competent Programmer Hypothesis** and the **Coupling Effect**, a test set T adequate with respect to such a mutant set should be able to reveal a significant fraction of the errors in the program P.

To assess the adequacy of a test set T, each mutant, as well as the program P, has to be executed against the test cases in T. If the observed output of mutant Q is the same as that of P for all test cases in T, then Q is considered **live**, otherwise it is considered **dead** or **distinguished**. A live mutant Q can be equivalent to program P. An equivalent mutant can not be distinguished and is discarded from the mutant set as it does not contribute to the adequacy of T. Test adequacy is measured by the **mutation score** computed as follows:

$$MS(P,T) = \frac{\#\text{ of dead mutants}}{\#\text{ total mutants} - \#\text{ of equivalent mutants.}}$$

The need to execute each mutant against the test cases in the test set and the undecidability of mutant equivalence might have an adverse effect on the cost of mutation testing. Some approaches [20], [21], [23], [25], [26], [28] have been proposed to reduce this cost by selecting a small subset of mutants thereby constraining the analysis.

3 Integration Testing

Unit testing is often the first step in testing software. Its intent is to build confidence in the correctness of a unit. A unit may not be a complete program by itself in which case it is tested using a driver and one or more stubs. The driver is a unit that coordinates the test—for example, the `main` function in C programs. It is responsible for supplying test cases, passing them as parameters to the unit under test, collecting results from the unit, and presenting them to the tester. A stub serves as a replacement for a subordinate unit. In most cases, a stub is a unit that mimics the behavior of a module used by the unit being tested.

In general, drivers and stubs are simplified versions of the units that, upon completion, will be used in the desired system. The simplification allows one to establish a controlled environment and to test the units at an early stage. Simplification also limits unit testing as the use of drivers and stubs, in general, fails to reproduce the complete environment where the unit being tested would be if it was interacting with the actual units in the system.

The next step in the testing process is the integration of units. Why should a program, built from units that work properly when tested individually, not function when tested after two or more units have been integrated? This is primarily because of the differences in the types of errors that one discovers at various levels of testing. At the unit level a tester is mostly interested in algorithmic aspects, verifying whether each unit performs its required function. The goal of integration testing is to put the units in their intended environment and exercise their interactions as completely as possible. Regardless of what approach is used for integration, incremental or otherwise, at some point during development, it is necessary to exercise the connections between units and it is useful to have one or more quantitative criteria to evaluate how well an interface has been exercised. Interface Mutation provides one such criterion.

There have been a few efforts to extend structural criteria for evaluating interprocedural test set adequacy. The first approach was proposed by Haley and Zweben [14]. The authors identify integration errors as those that occur when incorrect values are exchanged between units leading to incorrect output. In addition, they classify integration errors to be computational and domain integration errors. Given a unit *F* that calls unit *G*, the first occurs when a computational error in *G* produces an incorrect value that, returned to *F*, leads *F* to produce incorrect output. The second occurs when a domain error in *G* causes an incorrect output from *F*. Based on this classification, they suggest the retesting of some specific paths, assuming that each unit has been tested independently. These paths are calculated based on the *called* unit's interface. The number of paths aiming at computational integration errors is proportional to the number of formal parameters of such a unit. The number of paths aiming at domain integration errors is proportional to the number of input formal parameters of such a unit.

Linnenkugel and Müllerburg [18], defined a family of interprocedural control flow- and data flow-based criteria. The first group of criteria makes use of a Call Graph to represent a program and to identify the structures to be covered. A Call Graph is a directed multigraph where the nodes represent the units of the programs and the edges represent the calls between units. Using such a model, some control flow criteria were defined.

- **All-modules** requires that every unit is executed at least once, i.e., that every node on the graph is covered.
- **All-multiple-relations** requires that every call is executed, i.e., that every edge is covered.
- **All-call-sequences** requires that every descending sequence of calls is executed at least once, i.e., that every path in the graph is covered.

The other criteria proposed by Linnenkugel and Müllerburg define interprocedural data flow associations that must be covered by the test cases. Given a "communication variable"—as the authors named formal parameters and global variables—v of unit *F*, some data flow criteria are:

- **INT-all-defs** requires for every call to *F*, the execution of a definition free path with respect to v from each definition of v, prior to the call, to some use of v in *F* and from each definition of v in *F* to some use of v after the return from *F*.
- **INT-all-uses** requires for every call to *F*, the execution of a definition free path with respect to v from each definition of v, prior to the call, to every use of v in *F* and from each definition of v in *F* to every use of v after the return from *F*.

Harrold and Soffa [16], also proposed a technique for determining interprocedural definition-use structures allowing the application of dataflow adequacy criteria at the integration level. Harrold and Soffa's technique includes the solution to the problem of variable aliasing. At the interprocedural level, the calculation of definition-use associations is harder than at intraprocedural level due to the use of a variable with different names when a variable is passed as a reference parameter. The authors divide the application of their technique in four steps:

1. abstract control-flow and dataflow information, representing each unit by a graph that summarizes the data required for interprocedural analysis,
2. represent interprocedural control-flow and data-flow, combining the graphs obtained in Step 1,
3. get interprocedural information by propagating local information at nodes in the subgraphs throughout the interprocedural flow graph, and
4. compute interprocedural definition-use information, using local information from Step 1 and propagated information from Step 3.

They also present an algorithm to check whether or not a given test case covers the selected interprocedural associations.

Jin and Offut [17] have proposed another integration testing technique called coupling-based testing. They identify several ways two units can interact, also known as the coupling between the units, and, based on that classification, they establish requirements to be fulfilled by the test cases. The couplings are classified in 12 levels, for example:

- **level 0—Independent coupling**: Unit *A* does not call unit *B* and *B* does not call *A* and there are no common variable references or common references to external media between *A* and *B*.
- **level 5—Stamp control coupling**: A record in unit *A* is passed as actual parameter to *B* and there is a P-use in *B*.
- **level 11—Tramp coupling**: A formal parameter in unit *A* is passed as actual parameter to *B*; *B* passes the corresponding formal parameter to another unit without having accessed or changed the variable.

For each of these coupling levels Jin and Offut define a criterion, for instance:

- **Criterion 0—No coupling**: There is no coupling between *A* and *B* so no test requirement is defined.
- **Criterion 5—Stamp control coupling path**: Requires that for each record parameter x and each definition that reaches the call, a test case executes at least one path from the definition to the call and to each of the first P-uses of the formal parameter corresponding to x.
- **Criterion 11—Tramp coupling path**: Requires that for each parameter x defined in *A*, passed through *B* and used in *C* and each definition of x that reaches the call from *A* to *B*, a test case executes at least one path from the definition of x, through *B*, and to each of the first uses in *C*.

Interface Mutation differs from the above approaches. It does not require the computation of interprocedural associations or interprocedural paths that need to be covered, thus avoiding the complicated techniques required to deal with variable aliasing. Inherent to mutation testing is its flexibility, in the sense of being able to adapt the mutation operator set—either in selecting a subset or adding new operators—to the requirements and constraints of a specific domain or application such as: criticality, cost,

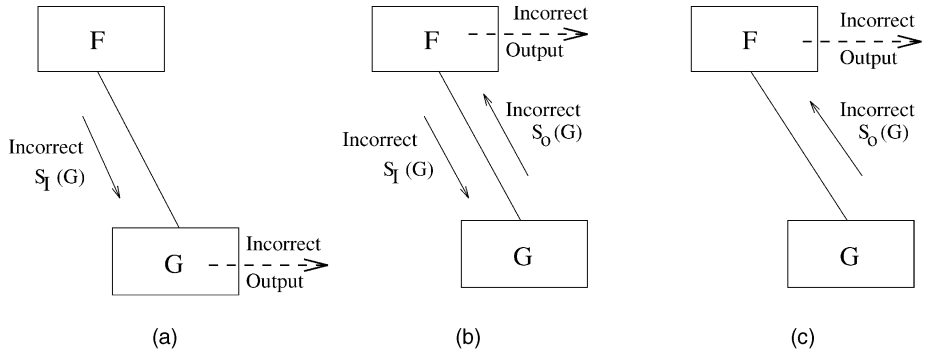


Fig. 1. Types of integration errors that might arise when units F and G interact.

and time. The use of Interface Mutation provides an opportunity to explore the complementary aspects of adequacy criteria at the integration level, along lines that have been pursued at the unit level [13], [29].

4 Integration Errors

Following Haley and Zweben [14], we suppose that an integration error occurs when an incorrect value is passed through a unit connection. Based on this observation, we classified integration errors into three categories described below. Consider a program P and a test case t for P. Suppose that in P there are units *F* and *G* such that *F* makes calls to *G*. Consider $S_I(G)$ to be the *n*-tuple of values passed to *G* and $S_O(G)$ the *n*-tuple of values returned by *G*. When executing P on test case t, an integration error is identified in a call to *G* from *F* when:

1. Type 1: Upon entering *G*, $S_I(G)$ does not have the expected values and these values cause an erroneous output (a failure) before returning from *G*.
2. Type 2: Upon entering *G*, $S_I(G)$ does not have the expected values and these values lead to an incorrect $S_O(G)$, which in turn causes an erroneous output (a failure) after returning from *G*.
3. Type 3: Upon entering *G*, $S_I(G)$ has the expected values, but incorrect values in $S_O(G)$ are produced inside *G* and these incorrect values influence an erroneous output (a failure) after returning from *G*.

The above classification does not specify the location of the fault responsible for causing incorrect outputs. It simply considers the existence of incorrect values entering or exiting a unit call. This excludes, for example, the case when $S_I(G)$ has the expected values, but a fault in *G* produces an erroneous output before returning from *G*. In this case, there is no error propagation through the connection *F*-*G*. This type of error is expected to have already been detected during unit testing.

In addition, more than one integration error can be associated with (or caused by) a single fault. For example, consider a program with three units R, S, and T such that R calls S and, upon returning from S, calls T. Suppose that in unit S an incorrect value $\mathtt{x} \in S_O(S)$ is returned and this value is also part of $S_I(T)$. Suppose that due to `x`, T produces an incorrect output. Thus, a fault in S produced a type 3 error in the connection R-S and a type 1 error in the connection R-T.

Finally, the *n*-tuples $S_I(G)$ and $S_O(G)$ depend, in part, on the language a program is written in. For example, for the C language a unit is a function and *n*-tuples $S_I(G)$ and $S_O(G)$ can be defined as:

- $S_I(G)$: The *n*-tuple of input values in a call to a function *G* is determined by
  - the input parameters used in the function call and
  - the global variables used in *G*.
- $S_O(G)$: The *n*-tuple of output values in a call to a function *G* is determined by
  - the output parameters used in the function call,
  - the global variables used in *G*, and
  - the values returned by *G*.

For example, a type 1 error occurs when an actual parameter or a global variable is passed to a unit and that called unit produces an incorrect output. The flow in this case is shown in Fig. 1a. In a type 2 error, there is an incorrect value entering the called unit and an incorrect value leaving the unit. This leads to an incorrect output in the calling unit (Fig. 1b). A type 3 error has one or more incorrect values leaving the called unit. In this case, a unit is called with correct input parameters but performs an incorrect computation which results in an incorrect return value which in turn leads to an incorrect output. This situation is illustrated in Fig. 1c.

5 Overview of Interface Mutation

To explain Interface Mutation, we begin by examining the nature of data exchange between units. In a call from a unit *F* to a unit *G*, we identify four ways in which data can be exchanged between the calling and the called units:

- Data can be passed to *G* via input parameters (parameters passed by value).
- Data can be passed to *G* and/or returned to *F* through input/output parameters (parameters passed by reference).
- Data can be passed to *G* and/or returned to *F* through global variables.
- Data can be returned to *F* through return values (as in `return` commands in C).

The above types of data transfer are not mutually exclusive. To evaluate a test set using Interface Mutation,

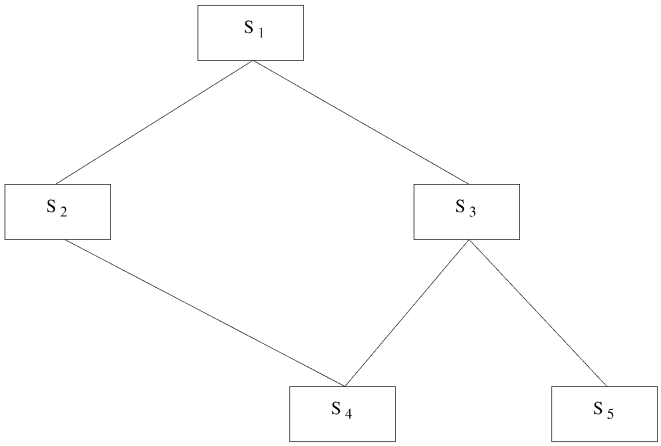


Fig. 2. Unit interconnections in a program with five units.

the first step is to perturb the system under test by making a simple syntactic change. The perturbation process is also known as *mutation* and the perturbed program as a *mutant*. When applying Interface Mutation, unlike traditional mutation, the syntactic changes are made only at the interface related points—or connections—between units. Variables and expressions concerned with the four types of data transfers mentioned above are possible candidates for mutation. Each type of mutation is carried out by applying a transformation to the program under test. The transformation is determined by an Interface Mutation operator. The application of one Interface Mutation operator on a program might generate zero or more mutants. The set of Interface Mutation operators is discussed later in Section 6. For now, it is sufficient to state that an Interface Mutation operator is intended to mutate the program in ways analogous to the errors that may be committed by a programmer during program development.

Once the mutants are generated, the next steps in Interface Mutation are the same as in traditional mutation testing: to execute the mutants, to evaluated test set adequacy, and to decide mutant equivalence.

5.1 An Illustrative Example

Fig. 2 shows a hypothetical system under integration. Consider an Interface Mutation operator *OP*(v) that perturbs a variable v in some manner. The tester can choose, for example, connection $S_1 - S_3$ to test. In this case, *OP* is applied to variables related to this connection.

Introducing type 1 or type 2 errors: To introduce these errors, *OP* should mutate an input to unit S_3. This input should be used inside S_3 and should return an incorrect value to unit S_1. There are at least two ways *OP* can be applied. First, in the case of values passed through parameters, *OP* can be applied either to the actual parameter in the call to S_3 which occurs inside S_1. Second, *OP* can be applied to the references to the formal parameters inside S_3. Consider the simple program in Fig. 3. If only the first approach is used, *OP* is applied to the argument a at line 5. This will generate at least one mutant. Thus, at least one test case is necessary to distinguish this mutant. However, this might leave two out of three uses of parameter x at lines 14, 16, and 18 untested. If the second approach is used, one mutant is generated for each use of the formal parameter x. Now, one test case is required to distinguish each of these three mutants. This will cause each reference to x to be tested at least once.

When values are passed to S_3 through global variables there is an additional problem of mutating variables prior to calling the unit. To perform such mutations would require the identification of all definitions of global variables that reach the call and are used inside the called unit. Note that, having an incorrect value being passed to S_3 in a call from S_1, does not automatically imply that this incorrect value was set inside S_1. It only implies that the value was set somewhere prior to calling S_3.

Introducing type 3 errors: This error can be introduced by mutating a value returned to S_1. If this value is a parameter passed by reference then the mutation can be done inside S_3 at each definition of the formal parameter; otherwise, mutation can be carried out inside S_1 by applying *OP* to the actual parameter immediately following the call to S_3. As in the previous example, if the latter approach is taken,

```
 1      S1()
 2      {
 3      int a, b;
 4
 5         b = S3(a);
 6         printf("%d", b);
 7      }
 8
 9      int S3(int x);
10      {
11         switch (x)
12         {
13            case 1:
14               return x*2;
15            case 2:
16               return x;
17            case 0:
18               return x+1;
19         }
20         return -1;
21      }
```

Fig. 3. Sample code for units S_1 and S_3 of Fig. 2.

only one mutant is created by *OP* and not all points of return are required to be exercised; the first approach raises the chances of exercising each return point. For values exchanged via global variables, a similar situation occurs and *OP* should be applied inside S_3.

For either reference parameters and global variables, the mutation could also be done after the return from the unit call at places where the value is used. This would allow each use of the data to be perturbed at least once. But this approach tends to impose higher cost, as every location in the program where the variable used is a potential location for mutation, even if it is not reachable by the definition inside S_3 or the definition is not live at that point.

Thus, for the three types of errors, Interface Mutation applies mutations at places where S_1 calls S_3, where global variables and formal parameters are used inside S_3 and where values are returned from S_3 to S_1 (as at *return* statements in C). These aspects were considered in the design of the Interface Mutation operators described in Section 6.

5.2 Testing "Interfaces" versus Testing "Units"

It is important to note that the Interface Mutation operator focuses on the testing interaction between units and not on testing a unit. For example, suppose that connection $S_3 - S_4$ in Fig. 2 is being tested and operator *OP* produces mutations inside unit S_4. These mutants may be distinguished by test cases that exercise calls to S_4 made from inside S_2, in which case the desired connection is not tested. This means that possible errors that lead to misuses of values passed to (returned from) S_4 from (to) S_3 might go untested. Thus, the application of *OP* must take into consideration the place from where a unit is called and only apply the mutation if this place is the desired one.

The above example points out a subtle difference between testing units, as in traditional mutation, versus testing interfaces as in Interface Mutation. This difference can be formalized by examining the necessary and sufficient conditions for a test case to distinguish a mutant in ordinary mutation testing. As stated in [10], three conditions are required for a test case t to distinguish a mutant *M* from the original program P:

1. *Reachability*: The execution of *M* with test case t must cause the program control to reach the point where the mutation was introduced.
2. *Necessity*: The state of *M* must differ from the state of P immediately following the execution of the mutated statement.
3. *Sufficiency*: The difference in the states of P and *M* must propagate such that different outputs are produced for P and *M*.

When the Interface Mutation operator is applied inside the called function, we need to alter the reachability condition. It can be restated as follows:

- *Interface Mutation reachability condition*: The execution of the mutant *M*, related to a connection *A*-*B*, with test case t must, through a call from *A* to *B*, cause the program control to reach the point where the mutation was introduced.

For some languages, for example C, this type of mutation requires a run time decision on whether the mutation should be applied or not. For example, to test connection $S_3 - S_4$ in Fig. 2, a mutation inside S_4 must have a "guard" in the point where S_3 calls S_4. This guard would enable the mutation. If S_4 is called from S_2, the absence of such a guard would not enable the mutation and S_3 should behave exactly as in the original program.

5.3 Handling Unit Coupling

Coupling of units via global variables raises interesting issues in Interface Mutation. Although Interface Mutation is applied pairwise, as it focuses on one connection each time, various indirect couplings may also be considered. For example, suppose that unit S_5 in Fig. 2 defines a variable v. When testing connection $S_3 - S_5$, operator *OP*(v) is applied inside S_5. To distinguish the mutant so created, it is necessary to have a test case that causes a use of v to be exercised in order to produce an incorrect output. This use of v may be located in S_4, for example. This implies that to distinguish a mutant created by applying an operator inside S_5, one requires a test case that exercises an indirect coupling between S_5 and S_4. Now, suppose that S_1 also has a use of v. In this case, it is possible to distinguish the mutant generated by applying *OP*(v) in S_5 by exercising only the use of v in either S_4 or S_1. This may cause a global coupling to go untested. Therefore, it is possible to exercise some indirect global interactions but it is not guaranteed that all of them will be exercised.

6 Interface Mutation Operators for the C Programming Language

The definition of mutant operators is dependent on the programming language of choice, although the same concepts can be applied to a variety of languages. Here, we present the Interface Mutation operator set for the C language. Our experience with the design of mutant operators for C and Fortran languages indicates that the set of operators for two languages can differ significantly due primarily to the difference in their respective syntax and, therefore, the nature of common errors programmers might make.

For purposes of the discussion below, a unit in C is a function that implements some related services specified by an interface. For a function *F*, its interface is precisely defined by its formal parameters, the global variables it accesses, and the code it implements. The connection between units are defined by function calls.

Interface Mutation operators and traditional mutant operators have similarities and differences. The idea behind both is the same: to produce subtle discrepancies between the state of the original program and the state of the mutant. In addition, the syntactic changes applied to create the mutants are similar and based mainly on reproducing common programming errors. As mentioned before, the application of an Interface Mutation operator differs from that of traditional mutant operators. For example, when a function *F* makes two calls to a function *G*, as in Fig. 4, to each such call corresponds a set of mutants.

Let G' be the function created by applying an Interface Mutation operator inside the function *G*. Then, as shown in Fig. 5, there are two different mutants to be considered when testing the connection *F*-*G*. The first one is created by replacing the first call to *G* by a call to G'. The second mutant is created by replacing the second call to *G* by a call to G'.

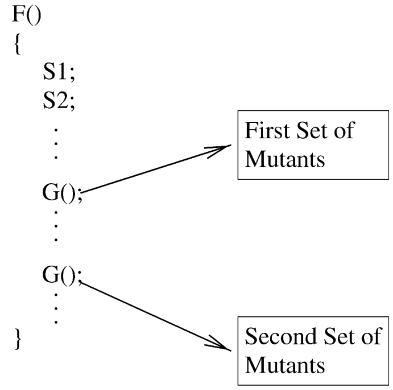


Fig. 4. Sets of mutants generated when applying Interface Mutation operators to the connection *F-G*.

Two groups of Interface Mutation operators are defined. Considering a connection *F-G*, operators in the first group are applied inside the body of function *G*. Operators in the second are applied to the call to *G* inside *F*. The set of Interface Mutation operators is presented below. Each operator has a name that identifies it uniquely among all Interface Mutation operators. Tables 1 and 2 list the name and meaning of the Interface Mutation operators.

6.1 Group I—Operators Applied Inside the Called Function

These operators are designed to cause changes in the values entering and/or leaving the function. To apply these operators, it is necessary to identify the place from where the function was called. The implementation of such operators requires a mechanism to enable or disable the mutation, depending on whether *G* is called from *F* or not. This mechanism, as implemented in $\mathcal{PROTEUM/IM}$, consists of a control variable that must be set when *F* calls *G* and enables the mutation. In the point where the mutation is really made this variable is tested. If it is unset, the original code is executed, otherwise, the modified code is executed. More details can be found in the description of $\mathcal{PROTEUM/IM}$ [4]. To define the operators in this group, the following sets are needed. Suppose that a connection *A-B* is being tested:

P(B): The set of formal parameters of *B*. This set also includes dereferences to pointer and array parameters. For example, when a parameter `v` is defined `int *v` or `int **v`, `v`, and `*v` belong to this set.

G(B): The set of global variables accessed by *B*.

L(B): The set of variables declared in *B* (local variables).

E(B): The set of global variables not accessed in *B*.

C(B): The set of constants used in *B*.

To avoid confusion, from now on, variables in set *G* are called "Globally Accessed Variables" and variables in set *E* are called "External Variables."

In addition, one more set *R* is defined; it does not depend on *G*. This set is the set of "required constants." It contains some special values relevant for each data type and associated operators. Table 3 summarizes these constants.

```
F()                                  F()
{                                    {
   S1;                                  S1;
   S2;                                  S2;
   .                                    .
   .                                    .
   .                                    .
   G'();                                G();
   .                                    .
   .                                    .
   .                                    .
   G();                                 G'();
   .                                    .
   .                                    .
   .                                    .
}                                    }
```

Fig. 5. Two mutants of *F* obtained by applying an Interface Mutation operator two different calls to *G*.

6.1.1 Direct Variable Replacement Operators

One way to perturb the value of an interface variable, i.e., a variable in sets *P* or *G*, is to directly replace each use or definition of the variable by another variable or constant. Operators *DirVarRep* in Table 1 do so. These operators replace each occurrence of a member of sets *P* and *G* by different groups of variables and constants and any level of dereferences and indexing of variables belonging to one of these sets.[1] They intend to perturb a value entering or exiting a function. It is important to note that only replacements between objects of compatible types are carried out.

6.1.2 Indirect Replacement Operators

Another way to perturb the value of an interface variable is to change another variable or a constant that can influence its final value. The operators *IndVarRep* in Table 1 do so. They are applied at places where variables or constants in sets *L* and *C* (noninterface variables) are used. These operators are applied only in the following two situations:

- When the variable/constant to be changed is in a `return` statement.
- When a variable from sets *P* or *G* is used in the same statement of the variable/constant to be changed.

For example, if `x` is an interface variable and `i` is a local variable, `i` is a candidate for mutation in the following cases:

- `if (x < i) ...`
- `j = x[i];`
- `x <<= i;`
- `return i;`

but is not a candidate for mutation in:

- `i += 10;`
- `printf("%d", i);`

1. From this point on, each time a variable is a candidate for mutation, a dereference, or an indexing to this variable is also a candidate for mutation, unless stated otherwise.

TABLE 1
List of Interface Mutation Operators—Group I

GROUP I	
Name	Meaning
DirVarRepPar	Replaces interface variable by each element of P
DirVarRepGlo	Replaces interface variable by each element of G
DirVarRepLoc	Replaces interface variable by each element of L
DirVarRepExt	Replaces interface variable by each element of E
DirVarRepCon	Replaces interface variable by each element of C
DirVarRepReq	Replaces interface variable by each element of R
IndVarRepPar	Replaces non interface variable by each element of P
IndVarRepGlo	Replaces non interface variable by each element of G
IndVarRepLoc	Replaces non interface variable by each element of L
IndVarRepExt	Replaces non interface variable by each element of E
IndVarRepCon	Replaces non interface variable by each element of C
IndVarRepReq	Replaces non interface variable by each element of R
DirVarIncDec	Inserts/removes increment and decrement operations at interface variable uses
IndVarIncDec	Inserts/removes increment and decrement operations at non interface variable uses
DirVarAriNeg	Inserts arithmetic negation at interface variable uses
IndVarAriNeg	Inserts arithmetic negation at non interface variable uses
DirVarLogNeg	Inserts logical negation at interface variable uses
IndVarLogNeg	Inserts logical negation at non interface variable uses
DirVarBitNeg	Inserts bit negation at interface variable uses
IndVarBitNeg	Inserts bit negation at non interface variable uses
RetStaDel	Deletes `return` statement
RetStaRep	Replaces `return` statement
CovAllNod	Coverage of all nodes
CovAllEdg	Coverage of all edges

These restrictions also hold for "Increment and Decrement" and "Unary Operator Insertion" operators when applied to noninterface variable and constants.

6.1.3 Variable Increment and Decrement Operators

There are two operators in this class. The first, *DirVarIncDec*, inserts a predecrement operator ($--$) and a preincrement operator (++) at each reference to an interface variable, i.e., to variables that belong to the sets P and G. Operator *IndVarIncDec* operates similarly, considering variables in set L. Operator *IndVarIncDec* also adds and subtracts one from each constant reference, i.e., each reference to constant `C` is replaced by expression `C + 1`.

TABLE 2
List of Interface Mutation Operators—Group II

GROUP II	
Name	Meaning
ArgRepReq	Replaces arguments by each element of R
ArgStcAli	Switches arguments of compatible type
ArgStcDif	Switches arguments of non compatible type
ArgDel	Remove argument
ArgAriNeg	Inserts arithmetic negation at arguments
ArgLogNeg	Inserts logical negation at arguments
ArgBitNeg	Inserts bit negation at arguments
ArgIncDec	Argument Increment and Decrement
FunCalDel	Removes function call

TABLE 3
Required Constant Sets

Variable Type	Required Constants
signed integer signed char signed long	-1, 1, 0, `MAXINT`, `MININT`
unsigned integer unsigned char unsigned long enum	-1, 1, 0, `MAXUNSIGNED`
float double	-1.0, 1.0, 0.0, -0.0

The constants MAXINT, MININT and MAXUNSIGNED correspond respectively to the largest positive integer, smallest negative integer and largest unsigned integer. These values are machine and data type dependent.

6.1.4 Unary Operator Insertion

The operators in this class insert an arithmetic, logical, or bit negation operator before the use of each variable and constant. Operator *DirVarAriNeg* inserts an arithmetic negation operator (-) before each interface variable reference. Similarly, operator *DirVarLogNeg* inserts a logical negation (!) and *DirVarBitNeg* inserts a bit negation (~).

Similar operators exist for noninterface variables and constants. They are applied on elements of sets *L* and *C*. These operators are *IndVarAriNeg*, *IndVarLogNeg*, and *IndVarBitNeg*.

6.1.5 Return Statement Operators

The operators in this class perturb the value returned by `return` statements. The first operator, *RetStaDel*, eliminates each `return` statement in the called function, one at a time. This forces the function to continue execution when it should stop and return a value.

Another operator, *RetStaRep*, replaces the expression used in a `return` statement by all the other expressions used in other `return` statements in the called function, one at a time, creating one mutant for each such replacement. Thus, for a given test case, the mutated function performs the same computations as the original function but only the return value is replaced by some other possible return value.

6.1.6 Coverage Operators

These operators are designed to assure that the test set used to test a connection reaches a minimal degree of coverage for the called function. The operators are: *CovAllNod* and *CovAllEdg* that correspond to the coverage of all-nodes and all-edges criteria [22], respectively.

Coverage operators have been included in this set primarily to maintain congruence with other mutant operator sets. Previous work on mutant operator design [1], [3] has always included coverage operators such as the "Trap on Statement Execution" operator. Thus, the authors, although aware of practical restrictions of such operators, decided to include node and edge coverage. Inclusion of these operators will facilitate empirical studies comparing and relating mutation testing and code coverage.

The all-nodes coverage criterion is accomplished by inserting, at the beginning of each basic block, a `TRAP_ON_EXECUTION` function that forces the mutation to be distinguished when the function is executed.

The all-edges coverage criterion is more complex. Coverage of all edges is obtained by inserting, for each expression in a conditional branch, the functions `TRAP_ON_FALSE` and `TRAP_ON_TRUE` that force the mutants to be distinguished if the expression evaluates to zero or a value different than zero, respectively. In addition, each branch in a `switch` statement is expected to be covered. This is achieved by using the same `TRAP_ON_EXECUTION`, as shown below, to force the execution of branch "`case 2:`"

```
switch(a)
{
    case 1:
         s1;
         goto __label__;
    case 2:
         TRAP_ON_EXECUTION();
         __label__:
         s2;
    case 3:
         s3;
}
```

In the above example, it is necessary to insert the `goto` statement to avoid the execution of the trap function from the first branch of the `switch` statement, instead of from the desired branch.

6.2 Group II—Operators Applied Inside the Calling Function

Given a connection *A-B*, these operators are applied to the statements where *A* calls *B* and to the call arguments. When an argument is an expression, the operators in this group are applied to the entire expression, not to objects inside the expression, like variables or constants.

6.2.1 Argument Replacement

The operator *ArgRepReq* is applied to each argument in the call statement. The arguments are replaced, one at a time, by the compatible constants in set *R*.

6.2.2 Argument Switch

The first operator in this group is *ArgStcAli*. It changes the order of the arguments in a call. Each argument is switched with each other compatible argument. A similar operator is *ArgStcDif*, which performs the same kind of switch between arguments with noncompatible types. In C, this error is easily caught at compile time, if the arguments have noncompatible types. On the other hand, they may pass unnoticed if the called function does not have a previously declared prototype. In this case, it may be worth to use *ArgStcDif*.

6.2.3 Argument Elimination

This operator, *ArgDel*, deletes each argument, one at each time. The same remark above about statically detecting such errors is valid here.

6.2.4 Unary Operator Insertion

These operators are similar to those defined for Group I but are applied to the arguments. The first, *ArgAriNeg*, inserts an arithmetic negation before each argument. The second, *ArgLogNeg*, inserts a logical negation and the last, *ArgBitNeg*, inserts a bit negation before each argument.

6.2.5 Function Call Deletion

The operator of this class is applied to the whole function call, instead of to the arguments. Operator *FunCalDel* eliminates the call to the function *G* in the connection *F-G*. If the function is a `void` function, then no care is necessary to implement the operator. On the other hand, if the function is used inside an expression, then instead of simply deleting the call, this operator replaces it by the required constants in the set *R*.

6.3 Comments on Interface Mutation Operators

Defining a set of operators is possibly the most relevant point for making the use of mutation testing feasible and effective. The set of operators presented above is not complete by any objective criterion. It is based on our experience with the use of mutant operators in traditional mutation. It intends to enable the construction of a test set that exercises all possible ways in which a pair of functions can interact. The effectiveness and utility of this Interface Mutation operators can only be completely assessed by further exploring its characteristics in other studies.

These operators are applied to any location in a C program where their application would not produce a syntactic error. The only restrictions to their application are:

- Testing of indirect connections implemented by function calls through pointers is not supported because it is not possible to statically realize how the interface of the called function is;
- Operators of Group I are not applied to directly recursive functions calls. This is primarily an implementation rather than a conceptual problem. Those operators require changes in two locations: in the call point to enable the mutation and inside the called function where the actual change takes place. For directly recursive functions, these two points are in the same function, which complicates the implementation; and
- Testing of connections to a function with a variable number of parameters is also not supported. This is also an implementation issue.

6.3.1 The Need for Parameterizable Operators

Mutant operators model simple faults frequently found when programming [1], [3], [9]. The Interface Mutation operators apply these faults to certain specific places related to a unit interface aiming at leading to an integration error. However, it is important to recognize that each program has different characteristics and may require different testing strategies. The application of the entire set of operators may create a large number of mutants resulting in prohibitive time required to execute and examine them. Therefore, it is necessary to allow the tester to adjust the set of operators to the specific application in terms of testing requirements and cost. A way to customize the application of the operators is to apply "constrained mutation," [20], [21], [25], [26]. So far, the constraints proposed are of two types: 1) randomly select a "small" percentage of the complete set of mutants and 2) apply only a restricted set of operators. It has been found that constraining mutation testing using any of these two approaches leads to a significant reduction in the number of mutants without significant loss in the error detection effectiveness of the adequate test sets. Any implementation of mutant operators must take such mutation constraints into consideration and allow this kind of tuning.

The implementation of Interface Mutation operators, besides allowing constrained mutation, incorporates an additional form of parameterization. The tester can determine the maximum number of mutants to be generated by an operator for each mutation point. For example, consider the use of a variable `x` that should be replaced by other variables, `y`, `z`, `w` by an operator like *DirVarRepPar*. We can limit the number of mutants generated by this operator to, say, two. Thus, `x` would be replaced by two out of the three possibilities selected randomly. This approach attempts to reduce the number of mutants by eliminating mutants that tend to behave alike as observed in our earlier study [6].

6.3.2 Discussion of Interface Mutation Operators

The Interface Mutation operators are designed to allow tests to be applied incrementally. The tester can choose to apply a subset of the operators and generate a reduced set of mutants or to apply all the operators and generate the complete set of mutants. The decision always depends on the testing requirements and criticality of each particular application. As each Interface Mutation operator is applied to a connection, any subset of the Interface Mutation operators applied to the entire system guarantees that each connection, or each function call is exercised at least once.

To reduce the number of mutants, the set of operators focuses on values related to the interaction between two functions. Thus, in the case of operators applied to points not directly related to this interaction, as to local variables in the called function, the mutations are restricted to places that can

interfere with or be influenced by an interface value. This approach may greatly reduce the number of mutants.

Operators in Group II are designed to model common errors in terms of function calls and parameter passing. In most cases, operators in Group II force the selection of test cases that show that each parameter has a distinct role in the function interface. If a parameter is not referenced or does not influence any computation inside the called function, a mutation applied to the arguments can not be distinguished and the parameter probably could be eliminated from the interface. The application of Group II operators to every connection in a system guarantees the coverage of every call in the program, a minimal requirement for any interprocedural criterion.

The operators in Group II tend to generate fewer mutants as compared to the operators in Group I. We have shown that the theoretical complexity of Interface Mutation (number of mutants generated in the worst case) using solely Group II operators is proportional to the number of formal parameters of the called unit. Using Group I operators, complexity is proportional to the number of accessible variables (global or local) in the called unit times the number of accesses to such variables in the called unit [4].

It is probably true that the problem of equivalence is easier to decide for Group II mutants because it might not require a complete understanding of the internal structure of the called function. Hence, these operators are good candidates to generate an initial set of mutants at the beginning of test set evaluation.

Operators in Group I have more diverse characteristics than operators in Group II trying to address interfaces in a more complete way, but at higher expense. In an incremental test approach, our experience and (mainly) intuition would recommend that after operators of Group II, the classes of operators that apply mutation directly to interface variables and `return` statement operators should be used. They directly interfere in the interaction between the two connected functions and are expected to produce more effective test cases, in terms of integration errors revealing. Next, we recommend the application of the set of indirect Interface Mutation operators, i.e., mutations applied to noninterface variables and to constants. In addition, it seems a good approach to parameterize the application and to start by creating a minimum set of mutants for these sets of operators and incrementally raise this number, as determined by cost and criticality constraints. Low cost requirements will tend to reduce the number of operators applied and the places where they are applied; high criticality requirements will tend to lead just to the opposite.

The coverage operators are designed to obtain control-flow coverage for each connection. The number of mutants created by these operators tends to be low. However, it is arguable whether all of them are meaningful to exercise the connection being tested. Thus, these operators, if used at all, may either be used during the early phases of test set evaluation or towards the end.

7 Experimental Evaluation of Interface Mutation

We report a case study conducted to investigate the error detection effectiveness of Interface Mutation and its cost. In this case study a Unix program was selected and several incorrect versions were created by seeding errors. Test sets were then generated using Interface Mutation and evaluated. The cost was measured in terms of the number of mutants generated. The error detection effectiveness was measured in terms of the percentage of the seeded errors revealed by Interface Mutation based test sets.

TABLE 4
Metrics Related to Functions in *Sort* Program

Function	LOC's	McCabe	Connections
main	109	35	19
sort	50	16	13
merge	77	29	10
disorder	7	2	2
newfile	5	2	4
setfil	8	3	0
oldfile	5	3	3
safeoutfil	11	7	1
cant	2	1	2
diag	4	1	1
term	10	3	4
cmp	88	38	3
cmpa	5	3	0
skip	30	14	1
eol	3	2	0
copyproto	6	2	0
field	48	16	1
number	6	2	0
qsort	67	14	1
TOTAL	541	193	65

7.1 Experimental Procedure and Data Collected

The selection of the Unix *Sort* utility was based on the following characteristics: 1) as illustrated in Table 4 and Fig. 6, it is a simple but nontrivial program[2] and 2) it makes use of many of the constructions and structures of the C language, allowing the application of the Interface Mutation operators.

Wong describes a set of 25 errors seeded in *Sort* [23]. From these errors, we selected 11 that were the most difficult to reveal. To determine the difficulty of revealing each error, we computed the **Hardness Index** (HI) for each of the 25 errors and selected those with HI less than or equal to 25 percent. The procedure for computing HI for an error is given below.

1. A set of 500 test cases is generated randomly.
2. The erroneous program is executed against each of the 500 test cases.
3. The *Sort* program (without errors) is executed against the same test set and used as an oracle.
4. The results of the erroneous version of the program are compared with those generated by the oracle.
5. HI is the percentage of test cases for which the

2. The number of connections includes calls to functions inside the *Sort* program, as well as calls to the C library functions. It does not include indirect calls.

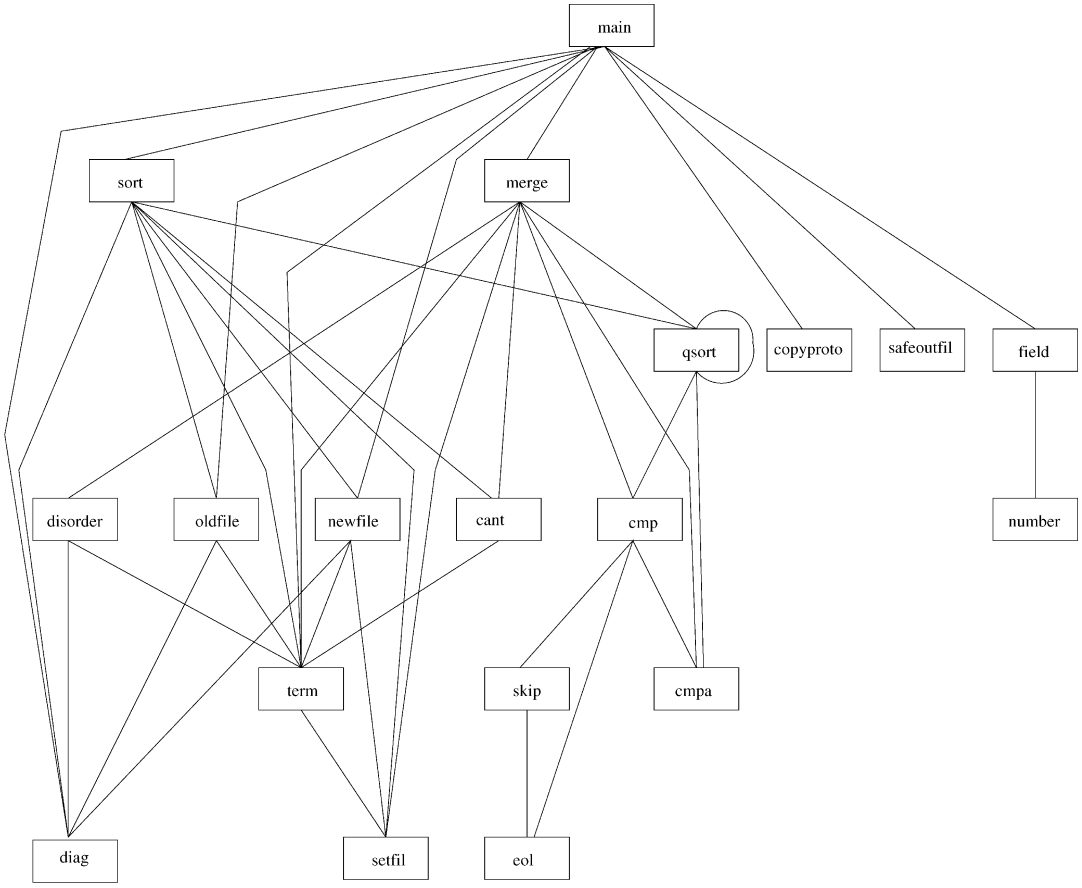


Fig. 6. Call Graph of the *Sort* Program.

TABLE 5
Errors Seeded in the *Sort* Program

Error	Function	Original code	Erroneous code	HI
I	merge	`while(...ip[-1]->l) < 0`	`while(...ip[-1]->l) <= 0`	3.99
II	cmp	`if (pb < lb && *pb==tabchar)`	`if (pb < lb || *pb==tabchar)`	4.63
III	skip_cmp	`c == ' ' || c == '\t'`	`c == ' ' && c == '\t'`	25.20
IV	field	`p->ignore = dict+128;`	`p->ignore = dict+290;`	7.17
V	main	`break;`	`continue;`	5.57
VI	qsort	`for (k=lp+1; k<=hp;)`	`for (k=lp+1; k<hp;)`	13.20
VII	cmp	`if (b = *--ipb - *--ipa)`	`if (b = *--ipb - --*ipa)`	10.60
VIII	merge	`while ((*dp++ = *cp++) != '\n');`	`while ((*++dp = *cp++) != '\n');`	13.77
IX	cmp	`if (*--ipa != '0')`	`if (--*ipa != '0')`	11.53
X	cmpa	`*pb == '\n' ? -fields[0].rflg`	`*pb == '\n' ? -fields[0].nflg`	23.23
XI	merge	`while (++k < j)`	`while (++j < k)`	2.87

output from the erroneous program does not agree with that from the oracle.

Errors with a higher HI are "easier" to reveal. Errors selected in this process are listed in Table 5. The HI presented in the table is the average of the HI computed with 30 different sets of 500 randomly generated test sets each. Erroneous versions of *Sort* are hereafter referred to as version I, version II, . . . , version XI. Thus, the hardest error is error number XI, followed by error number I, etc.

For each erroneous version, Interface Mutation was applied to seven connections as described in Table 6. The first column in this table shows the name of the function that makes calls to the function in the second column; the third column lists the number of times the connection occurs in the program; this number is the same as the number of calls made.[3]

For the connections mentioned in Table 6, all Interface Mutation operators, except *CovAllNod* and *CovAllEdg*, were applied. Also, based on previous studies [6], we restricted the application of replacement operators because they tend to produce a high number of mutants with similar behavior. The use of operators *DirVarRep* was restricted so that each operator created at most two mutants at each mutation point. The use of operators *IndVarRep* was restricted such that each of them created at most one mutant per mutation

3. The number of connections is the same for all the original and erroneous programs, because the errors introduced did not change the number of connections.

TABLE 6
Connections to which Interface Mutation was Applied

Calling function	Called Function	# of calls
main	sort	1
merge	qsort	1
sort	qsort	1
qsort	qsort	2
merge	disorder	2
cmp	skip	4
cmp	cmpa	1

point. Coincidentally, for all the erroneous versions, the number of mutants created is the same and is shown in Table 7.

Using the set of mutants generated as above, 30 test sets were constructed for each erroneous version. To create such an IM-based test set, an independent randomly generated pool with 500 test cases was constructed resulting in a total of 330 such pools for this study. The test cases were extracted from these pools to form the IM-based test sets. This process is illustrated in Fig. 7. In some cases, the test set obtained was not IM-adequate due to the process used for detecting equivalent mutants as described below. For all test sets the Interface Mutation score could never reach 100 percent but was always higher than 95 percent (that is why we call the sets IM-based instead of IM-adequate).

The process of marking the equivalent mutants was also automated. Prior to the creation of the IM-based test sets, all the mutants were executed against 500 randomly generated test cases; mutants not distinguished were considered equivalent. Certainly, this is an approximate and optimistic method for determining equivalent mutants. It does not allow for the estimation of the effort necessary to identify equivalent mutants and may result in an inflated mutation score. Some nonequivalent mutants may be marked as equivalent resulting in the exclusion of one or more test cases from the IM-based test sets. This exclusion can only lower the measures of effectiveness reported in this study. Thus, the effectiveness results reported here are conservative in the sense that if the process of marking equivalent mutants is improved, the effectiveness of Interface Mutation may also improve.

Once an IM-based test set is obtained, its effectiveness is measured by checking whether it reveals the seeded error or not. This is accomplished by comparing the execution of the erroneous version and the execution of the version with no errors (used as an oracle) with the test cases in the IM-based set. If, for any of the test cases, the results differ then the test set is able to reveal the seeded error. The 30 repetitions, creating 30 IM-based test sets, are intended to reduce the bias that could arise if only one or a couple of test sets were evaluated. Table 8 shows, for each erroneous version I to XI, the average and standard deviation of the mutation score obtained by the IM-based test sets, the cardinality of such sets, and the percentage of the test sets that revealed the error.

PROTEUM/IM allows the selection of subsets of mutants. Therefore, it was possible to evaluate the testing session with different characteristics. We defined some subsets of mutants, based on: 1) subsets of Interface Mutation operators and 2) random selection of a percentage of the generated mutants. Then, we measured the effectiveness of the test sets created using these subsets instead of the complete set of mutants. Tables 9 and 10 show the error detection effectiveness of several Constrained Interface Mutation criteria. The subsets we used are:

- Only 10 percent of the entire set of mutants, randomly selected.
- Only the operators of Group I are applied.
- Only the operators of Group II are applied.
- Only operators that directly replace interface variables (*DirVar* operators) are applied.

TABLE 7
Number of Mutants Generated

Operator	Mutants	Operator	Mutants
DirVarRepPar	114	DirVarRepGlo	46
DirVarRepLoc	118	DirVarRepExt	232
DirVarRepCon	118	DirVarRepReq	116
IndVarRepPar	98	IndVarRepGlo	83
IndVarRepLoc	107	IndVarRepExt	147
IndVarRepCon	73	IndVarRepReq	67
DirVarIncDec	240	IndVarIncDec	230
DirVarAriNeg	58	IndVarAriNeg	67
DirVarLogNeg	58	IndVarLogNeg	67
DirVarBitNeg	58	IndVarBitNeg	67
RetStaDel	12	RetStaRep	10
ArgRepReq	0	ArgStcAli	6
ArgStcDif	1	ArgDel	8
ArgAriNeg	4	ArgLogNeg	4
ArgBitNeg	4	ArgIncDec	52
FunCalDel	44	TOTAL	2309

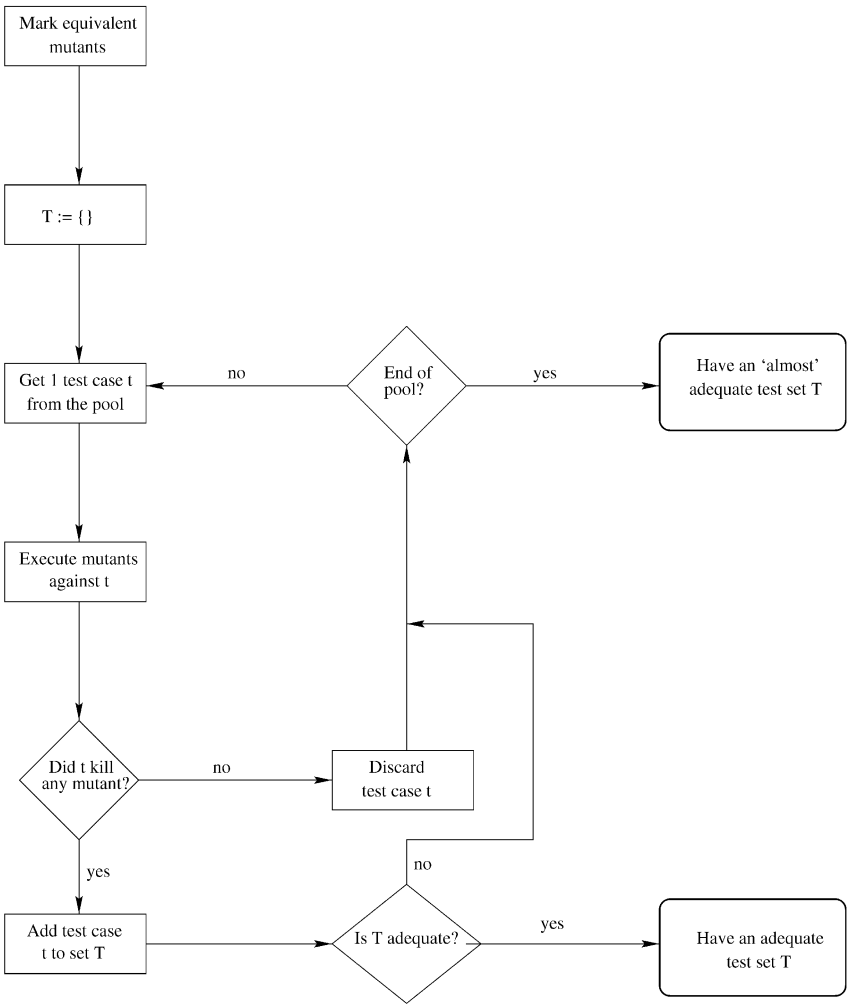


Fig. 7. Generation of an Interface Mutation-based test set.

- Only operators that replace noninterface variables (*IndVar* operators) are applied.
- Only operators that mutate the `return` statements (*RetSta* operators) are applied.
- Selective mutation [21] is used. A N-selective criterion is the one that applies all the operators, except those N which generate more mutants. For example, a 2-selective criterion would exclude the two operators with the largest number of mutants. In our case, as shown in Table 7, *DirVarIncDec* and *DirVarRepExt* are excluded in a 2-selective criterion (column S6 presents the 6-selective criterion, etc.).

The last line in Tables 9 and 10 shows, for each criterion, the number of mutants analyzed.

As shown in Table 11, the correspondence between connections and errors was also established. The number of mutants analyzed for each connection is listed in the last row of Table 11.

To obtain an evaluation of the effectiveness results, test sets with the same cardinality of IM-based sets were randomly generated and their ability to reveal errors was compared against the effectiveness of the test sets generated by the complete Interface Mutation criterion, the 19-selective mutation criterion, and the 10 percent randomly selected mutants. To obtain this comparison, the following procedure was used for each IM-based set:

1. A test set with size equal to that of the IM-based test set was generated randomly.
2. The test set so generated was executed against each erroneous version to determine whether or not it revealed the error.
3. Only n test cases were considered from the randomly generated test set, n being the cardinality of the 19-selective adequate test set and their effectiveness measured. 19-selective criterion was used because it is the last selective criterion with an average of over 95 percent error revealing test sets.
4. Only m test cases are considered in the randomly generated test set, m being the cardinality of the 10 percent adequate test set and their effectiveness measured.

Using the above procedure, it was possible to obtain a measure about IM-based test sets. The goal was to investigate whether or not the effectiveness obtained was due to the way test cases were selected or due to the sizes of test sets. If the size were the main factor, randomly generated test sets with the same sizes should obtain the same effectiveness score as IM-based test sets and

TABLE 8
Effectiveness Results

ERROR	MUTATION SCORE	NUMBER OF TEST CASES	PERCENTAGE OF SETS REVEALING THE ERROR
I	0.993883	44.10	100
	0.005509	5.42	
II	0.980643	44.77	100
	0.005505	7.47	
III	0.982352	42.30	100
	0.004102	4.54	
IV	0.944547	45.77	100
	0.007089	7.22	
V	0.969903	45.00	100
	0.003867	4.21	
VI	0.984760	45.13	100
	0.038822	6.31	
VII	0.986067	41.33	100
	0.005450	10.01	
VIII	0.983761	46.70	100
	0.040316	7.59	
IX	0.987760	42.33	100
	0.003815	7.40	
X	0.962617	40.17	100
	0.005973	5.63	
XI	0.999986	17.13	96
	0.000079	10.24	
Avg	0.979662	41.34	99.64
Dev	0.015465	8.28	1.21

Interface Mutation would not have contributed to the effectiveness results.

The average sizes of test sets, for each of the three criteria above, are reported in Table 12. For each criterion there are two sizes. The first is the actual size of the set. The second, called "restricted test set size," is based on the number of test cases necessary to reveal an error. In other words, this number gives the size of the IM-based test set that includes all test cases up to the first error-revealing test case. For example, if a test set has 10 test cases but the fourth one

TABLE 9
Constrained Interface Mutation Effectiveness Results (1)

ERROR	10%	G-I	G-II	DirVar	IndVar	RetSta
I	96	100	70	100	100	90
II	100	100	86	100	100	60
III	100	100	100	100	100	86
IV	100	100	80	100	100	60
V	100	100	96	100	100	66
VI	100	100	100	100	100	76
VII	100	100	100	100	100	93
VIII	100	100	96	100	100	70
IX	100	100	100	100	100	93
X	100	100	100	100	100	100
XI	96	96	96	96	96	96
Average	99.27	99.64	93.09	99.64	99.64	80.91
Deviation	1.62	1.21	10.09	1.21	1.21	14.94
Mutants	223	2188	123	1158	1006	22

TABLE 10
Constrained Interface Mutation Effectiveness Results (2)

ERROR	S6	S17	S19	S20	S21	S23	S24	S25
I	100	100	100	96	93	83	20	13
II	96	96	96	86	66	60	33	23
III	100	100	100	100	96	90	83	76
IV	100	96	93	93	66	56	36	30
V	100	100	96	83	66	63	33	30
VI	100	100	100	100	76	63	36	33
VII	100	100	100	100	96	96	56	50
VIII	100	100	96	86	73	66	66	63
IX	100	100	100	100	93	90	56	46
X	100	100	100	100	100	96	70	70
XI	96	96	96	96	96	96	96	96
Average	99.27	98.91	97.91	94.55	83.73	78.09	53.18	48.18
Deviation	1.62	1.87	2.55	6.59	14.16	16.39	23.86	25.55
Mutants	1224	307	191	139	93	37	27	19

reveals an error, then the restricted size for this test set is four. This second number may more accurately reflect the testing activity in the sense that a test set is built until an error occurs or the adequacy to a criterion is achieved.

Restricted test set size is also used in the comparison with random test sets generated as described above. The results are presented in Table 13. This table shows the percentage of random test sets that reveal an error in the program, considering the size and the restricted size of each Interface Mutation criterion. For example, line I shows that, considering the sizes of the randomly generated test sets to be the same as that of complete IM-based test sets, 93 percent of the random sets reveal an error and considering the restricted size, 50 percent reveal an error. In the same way, 70 percent and 43 percent for full and restricted size of 10 percent criterion and 60 percent and 33 percent for 19-selective criterion.

7.2 Data Analysis

The analysis in this section is concerned with the cost of applying Interface Mutation and its error revealing effectiveness. It is important to highlight the limited scope of this case study that has used a single program and a small set of errors. In addition, the cost measure may be biased by the fact that only the number of mutants and the number of test cases required were considered. To permit the automation of the experiment, factors such as the effort to identify equivalent mutants—that may be the hardest part in mutation testing—were not considered. On the other hand, this number may be roughly estimated from the total number of mutants. Despite its limited scope, this case study serves as the starting point to evaluate a new technique proposed herein for integration testing. Another case study [7] with a more complex program has been conducted. The results obtained in that study confirm those

TABLE 11
Effectiveness of Individual Connection

ERROR	cmp skip	merge qsort	cmp cmpa	sort qsort	main sort	qsort qsort	merge disorder
I	96	96	93	0	0	33	0
II	96	100	73	100	53	80	0
III	100	100	100	100	93	93	36
IV	96	100	60	96	96	30	0
V	100	70	53	43	36	33	3
VI	100	100	0	100	100	100	0
VII	100	100	100	100	100	100	10
VIII	96	96	73	0	3	16	43
IX	96	100	96	100	100	93	3
X	100	90	100	100	100	100	0
XI	96	96	96	96	96	90	20
Average	97.82	95.27	76.73	75.91	70.64	69.82	10.45
Deviation	2.09	8.96	30.64	41.11	40.39	33.93	15.69
Mutants	1080	264	250	264	367	20	64

TABLE 12
Cardinality of IM-Based Test Sets

	FULL IM		10%		19-SELECT	
ERROR	U. Size†	R. Size‡	U. Size†	R. Size‡	U. Size†	R. Size‡
I	44.10	15.13	20.17	11.70	16.73	9.37
II	44.77	13.57	23.30	10.40	16.43	8.97
III	42.30	4.67	18.13	4.50	15.73	4.50
IV	45.77	9.73	22.23	8.17	17.53	7.77
V	45.00	11.87	20.37	8.43	15.73	7.67
VI	45.13	6.33	22.37	5.73	17.37	5.47
VII	41.33	6.90	21.23	5.47	14.70	5.30
VIII	46.70	4.83	22.07	4.03	17.20	4.20
IX	42.33	7.77	21.33	6.07	15.57	5.73
X	40.17	4.00	20.47	3.70	15.23	3.50
XI	17.13	17.13	11.63	11.63	10.27	10.27
Average	41.34	9.27	20.30	7.62	15.68	6.61
Deviation	8.28	4.55	3.19	2.98	2.02	2.30

†: *Unrestricted size.* ‡: *Restricted size.*

herein presented and encourage the authors to recommend the use of Interface Mutation.

We started with a set of 25 errors from which we selected 11 to be used in the case study. Using Hardness Index (HI), we avoided the use of errors that would likely be revealed by almost any test set. The final set has errors with HI ranging from 2.87 to 25.20.

The effectiveness obtained by applying the full Interface Mutation is near 100 percent for all the erroneous versions. Of the 330 IM-based test sets, only one was unable to reveal an error. This happened for the error with the lowest HI. The alternate criteria, presented in Tables 9 and 10, also provided high error revealing effectiveness. In terms of cost, full Interface Mutation turns out to be an expensive criterion when compared, for example, with the 10 percent randomly selected criterion or with the 19-selective criterion. The first, with a reduction of approximately 90 percent in the number of mutants analyzed, provided 99.27 percent of error revealing test sets and the second, with a reduction of almost 92 percent, provided 97.91 percent of effectiveness. The number of test cases required to satisfy the criteria also exhibited a large reduction for these alternative criteria; 51 percent for the 10 percent criterion and 62 percent for the 19-selective criterion.

These results suggest a strategy for Interface Mutation application. One may start using Interface Mutation with a restricted set of mutants and expect an error to be revealed. In the study, for example, an error was revealed by more than 50 percent of the test sets using only 27 mutants produced by the 24-selective criterion. We assume that when an error is revealed it is removed from the program and the test set evaluation is restarted. When an error is not

TABLE 13
Random Generated Set Effectiveness

	FULL IM		10%		19-SELECT	
ERROR	U. Size†	R. Size‡	U. Size†	R. Size‡	U. Size†	R. Size‡
I	93	50	70	43	60	33
II	86	43	63	30	40	33
III	100	56	100	56	100	56
IV	93	40	73	36	66	33
V	86	50	63	43	53	40
VI	100	60	100	60	96	60
VII	93	50	76	43	76	40
VIII	100	30	93	30	86	30
IX	100	63	100	56	100	53
X	100	53	93	46	80	46
XI	40	40	26	26	13	13
Average	90.09	48.64	77.91	42.64	70.00	39.73
Deviation	17.49	9.71	22.69	11.45	27.24	13.55

†: *Unrestricted size.* ‡: *Restricted size.*

revealed, additional mutants are generated and additional test cases might need to be constructed to distinguish the newly generated mutants. Several different sets of Interface Mutation operators can be used in this incremental approach. Two obvious sequences are the *N* percent criteria and the *N*-selective criteria.

Analyses of Tables 12 and 13 confirms that for our set of 11 seeded errors, the use of full Interface Mutation produces test sets large enough so that randomly generated test sets, with the same sizes, have error revealing effectiveness close to that of the IM-based test sets. For five errors, the effectiveness remains unchanged and for the remaining six errors, the effectiveness of randomly generated test sets is at least 7 percent lower. For 10 percent and 19-selective criteria the size of the adequate test sets is reduced significantly and the effectiveness of randomly generated test sets reduces to 77.91 percent and 70 percent, respectively. In addition, upon analyses of the "restricted size" of the test sets, we find that the IM-based test sets tend to select error revealing test cases much faster than randomly generated test sets. Using restricted size, the average effectiveness of the randomly generated test sets remains below 50 percent. This leads to believe that the size of IM-based test sets is not the primary factor that controls the effectiveness.

As shown in Table 11, the test of a connection may be able to reveal errors created by unrelated faults. This leads to the question: "For *Sort*, which connections should we test first?" The obvious answer to this question is that one should first test connections with a relatively higher probability of revealing errors. However, such connections are not easy to determine.

For example, what makes cmp-skip a more suitable connection to start a test than merge-disorder? When a mutation is applied to connection *A-B*, a value related to this connection is perturbed. To distinguish such a mutant, this changed value must be used, leading to an output that differs from the original output. This use may occur anywhere in the program. Now, suppose that the mutated value in the connection is only used, besides possibly in *A* and *B*, in function *C*. In this case, all the test cases selected to test *A-B* must traverse a path that includes function *C*. This might limit the set of paths covered by such test cases. As a result, errors in other paths may go undetected. For example, this is what happens with connection merge-disorder. The 64 mutants may be distinguished, on the average, by 2.24 test cases that cover only a small fraction of the code and unable to reveal several errors. On the other hand, connections like cmp-skip and merge-qsort require a larger number of test cases that execute a larger set of paths and consequently force most errors to be revealed. Perhaps the way to determine the best connections is to start the test by analyzing the code and identifying connections where the primary data structures would be mutated which might lead to exercising larger and most relevant parts of the code.

8 Conclusions

The concept of mutation testing has been extended to and investigated at the interprocedural level, thereby providing mechanisms for the evaluation of integration testing. A test adequacy criterion named Interface Mutation has been defined. Interface Mutation is designed to be applicable for the assessment of tests of complete systems or subsystems and is aimed primarily at testing the interaction between a pair of units within a software system.

Mutation testing has been used for program testing and also for other testing scenarios [11], [12], [30]. More specifically, Fabbri et al. [12] have investigated the use of mutation for protocol testing. Woodward [30] has applied mutation testing to algebraic specification. These studies also make Interface Mutation applicable to interprocedural testing. In addition, the characteristics of unit pairwise testing eases the incremental use of Interface Mutation criterion in most of the integration testing strategies even when the system has been integrated using the "big-bang" strategy.

An Interface Mutation operator set for C language has been designed based on the authors' experience and an abstract integration error model. Perhaps it is possible to exclude some of operators from this set without losing the effectiveness of Interface Mutation. This possibility is to be addressed in future studies, similar to those by Offut et al. [20], to determine an "essential" set of Interface Mutation operators.

The study reported herein used the Unix *Sort* program. The results enhance our belief that Interface Mutation is an effective error-revealing criterion. It also reinforces our belief that the number of mutants can be kept below an acceptable limit allowing for the application of Interface Mutation during the testing of nontrivial programs. Interface Mutation has been shown effective in this case study; the error revealing effectiveness achieved is almost 100 perent for all the IM-based test case sets. In addition, by comparing Interface Mutation with randomly generated test sets of the same size—as done by Wong et al. [24]—it could be shown that its effectiveness is not due to secondary characteristics such as the size of test sets. It should be pointed out though that the cost of Interface Mutation might still impose some constraints to its use in practice. As in unit testing, the alternative mutation criteria, such as random and constrained mutation or selective mutation, may be considered to reduce the associated cost. Parameterization facilities are available in PROTEUM/IM and have been explored in the *Sort* study to further reduce the cost of Interface Mutation.

Results similar to those described in this paper have been obtained by applying Interface Mutation for testing Space [4], [7]—a program developed by the European Space Agency (ESA) for describing the configuration of antennas. These results provide additional support for the use of Interface Mutation.

The use of better interface definition with, for instance, IDL or Object Oriented Design and Programming may reduce the integration problems and the need for integration testing. On the other hand, we will need to focus on issues such as the validation of an IDL description or the integration at class level instead of at the function level. The authors believe that this higher level integration test may also apply Interface Mutation in a similar way as described in this paper. It is the authors' intent to explore these ideas,

extending Interface Mutation to other levels of interface definition. This includes the interaction between programs as for instance in client/server applications.

We now identify aspects of our study that pose threats to the validity of the conclusions we have drawn from the experimental results. Only one program has been used in our experiment. Certainly, there exists the possibility of obtaining different results and conclusions when Interface Mutation is applied to programs that differ from *Sort* in various ways such as size, application domain, language, etc. It is also possible that the selection and injection of faults could have significant effects on the experimental results. Further, though the faults we injected satisfy our definition of "integration errors," they might be found during unit testing and, hence, render Interface Mutation useless. The cost of Interface Mutation was measured solely by the number of mutants generated and test cases required to kill them. This is not necessarily the best measure of cost. Factors such as the time to find equivalent mutants, time to analyze a mutant to generate a test that will distinguish it from the program under test, etc., might change one's belief that Interface Mutation is indeed a viable alternative. Certainly, additional experimental studies are needed to evaluate the effectiveness of utility of Interface Mutation in a development environment.

ACKNOWLEDGMENTS

The authors would like to thank the Computer Science Department at Purdue University where Professor Márcio E. Delamaro and Professor José C. Maldonado have been visiting scholars. Also, Márcio Delamaro's research was supported by a grant from CAPES, José Maldonado's by CAPES and CNPq, and Aditya Mathur's in part by an award from the Center for Advanced Studies, IBM Toronto Laboratories, and a National Science Foundation award CCR-9102331.

REFERENCES

[1] H. Agrawal, R.A. DeMillo, R. Hataway, W. Hsu, W. Hsu, E. Krauser, R.J. Martin, A.P. Mathur, and E.H. Spafford, "Design of Mutant Operators for C Programming Language," Technical Report SERC-TR41-P, Software Eng. Research Center, Purdue Univ. Mar. 1989.

[2] T.A. Budd and D. Angluin, "Two Notions of Correctness and their Relation toTesting," *Acta Informatica,* vol. 18, no. 1, pp. 31–45, Nov. 1982.

[3] B.J. Choi, R.A. DeMillo, E.W. Krauser, A.P. Mathur, R.J. Martin, A.J. Offutt, H. Pan, and E.H. Spafford, "The Mothra Toolset," *Proc. 22nd Ann. Hawaii Int'l Conf. System Sciences,* Jan. 1989.

[4] M.E. Delamaro, "Mutação de Interface: Um Critério de Adequação Inter-procedimental para o Teste de Integração," Doctoral dissertation, Physics Inst. São Carlos—University of São Paulo, São Carlos, SP, June 1997.

[5] M.E. Delamaro and J.C. Maldonado, "Proteum—A Tool for the Assesment of Test Adequacy for C, Programs," *Proc. Conf. Performability in Computing Systems (PCS 96),* pp. 79–95, July 1996.

[6] M.E. Delamaro, J.C. Maldonado, and A.P. Mathur, "Integration Testing Using Interface Mutation," *Proc. VII Int'l Symp. Software Reliability Engineering (ISSRE '96),* pp. 112–121, Nov. 1996.

[7] M.E. Delamaro, J.C. Maldonado, A. Pasquini, and A.P. Mathur, "Interface Mutation Test Adequacy Criterion: An Empirical Evaluation," Technical Report 83, ICMSC–USP, São Carlos—SP, Jan. 1999.

[8] R.A. DeMillo, R.J. Lipton, and F.G. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer," *IEEE Computer,* vol. 11, no. 4, Apr. 1978.

[9] R.A. DeMillo and A.P. Mathur, "A Grammar Based Fault Classification Scheme and its Application to the Classification of the Errors of TEX," Technical Report SERC-TR165-P, Software Eng. Research Center, Purdue Univ. Sept. 1995.

[10] R.A. DeMillo and A.J. Offutt, "Constraint Based Automatic Test Data Generation," *IEEE Trans. Software Eng.,* vol. 17, no. 9, pp. 900–910, Sept. 1991.

[11] S.C.P.F. Fabbri, J.C. Maldonado, P.C. Masiero, and M.E. Delamaro, "Mutation Analysis Testing for Finite State Machines," *Proc. Fifth Int'l Symp. Software Reliability Eng (ISSRE),* pp. 220–229, Nov. 1994.

[12] S.C.P.F. Fabbri, J.C. Maldonado, P.C. Masiero, and M.E. Delamaro, "Mutation Analisys Applied to Validate Specifications Based on Petri Nets," *Proc. Eighth IFIP Conf. Formal Descriptions Techniques for Distribute Systems and Communication Protocols,* pp. 329–337, Oct. 1995.

[13] P.G. Frankl and E.J. Weyuker, "A Formal Analysis of the Fault-Detecting Ability of Testing Methods," *IEEE Trans. Software Eng.,* vol. 19, no. 3, pp. 202–213, Mar. 1993.

[14] A. Haley and S. Zweben, "Development and Application of a White Box Approach to Integration Testing," *The J. Systems and Software,* vol. 4, pp. 309–315, 1984.

[15] R.G. Hamlet, "Testing Programs with the Aid of a Compiler," *IEEE Trans. Software Eng.,* vol. 3, no. 4, July 1977.

[16] M.J. Harrold and M.L. Soffa, "Selecting and Using Data for Integration Test," *IEEE Software,* vol. 8, no. 2, pp. 58–65, Mar. 1991.

[17] Z. Jin and A.J. Offut, "Integration Testing Based on Software Couplings," *Proc. 10th Ann. Conf. Computer Assurance (COMPASS '95),* pp. 13–23, Jan. 1995.

[18] U. Linnenkugel and M. Müllerburg, "Test Data Selection Criteria for (Software) IntegrationTesting," *Proc. First Int'l Conf. Systems Integration,* pp. 709–717, Apr. 1990.

[19] A.J. Offut, J. Pan, K. Tewary, and T. Zhang, "An Experimental Evaluation of Data Flow and Mutation Testing," *Software Practice and Experience,* vol. 26, no. 2, pp. 165–176, Feb. 1996.

[20] A.J. Offutt, A. Lee, G. Rothermel, R.H. Untch, and C. Zapf, "An Experimental Determination of Sufficient Mutant Operators," *ACM Trans. Software Eng. Methodology,* vol. 5, no. 2, pp. 99–118, 1996.

[21] A.J. Offutt, G. Rothermel, and C. Zapf, "An Experimental Evaluation of Selective Mutation," *Proc. 15th Int'l Conf. Software Eng.,* pp. 100–107 May 1993.

[22] S. Rapps and E.J. Weyuker, "Selecting Software Test Data Using Data Flow Information," *IEEE Trans. Software Eng.,* vol. 11, no. 4, pp. 367–375, Apr. 1985.

[23] W.E. Wong, "On Mutation and Data Flow," PhD dissertation, Dept. of Computer Science, Purdue Univ., W. Lafayette, IN, Dec. 1993.

[24] W.E. Wong, J.R. Horgan, S. London, and A.P. Mathur, "Effect of Test Set Size and Block Coverage on Fault Detection Effectiveness," *Proc. Fifth IEEE Int'l Symp. Software Reliability Eng.,* pp. 230–238, Nov. 1994.

[25] W.E. Wong, J.C. Maldonado, M.E. Delamaro, and A.P. Mathur, "Constrained Mutation in C Programs," *Proc. Eighth Brazilian Symp. Software Eng.,* pp. 439–452, Oct. 1994.

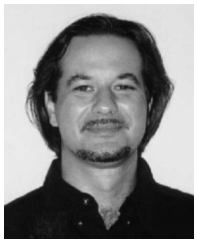
[26] W.E. Wong and A.P. Mathur, "How Strong is Constrained Mutation in Fault Detection?" *Proc. Int'l Computer Symp.,* pp. 515–520, Dec. 1994.

[27] W.E. Wong and A.P. Mathur, "Fault Detection Effectiveness of Mutation and Data Flow Testing," *Software Quality J.,* vol. 4, no. 1, pp. 69–83, Mar. 1995.

[28] W.E. Wong and A.P. Mathur, "Reducing the Cost of Mutation Testing: An Empirical Study," *The J. Systems and Software,* vol. 31, no. 3, pp. 185–196, Dec. 1995.

[29] W.E. Wong, A.P. Mathur, and J.C. Maldonado, "Mutation Versus All-uses: An Empirical Evaluation of Cost, Strength, and Effectiveness," *Proc. Int'l Conf. Software Quality and Productivity,* pp. 258–265, Dec. 1994.

[30] M.R. Woodward, "Mutation Testing—its Origin and Evolution," *Information and Software Technology,* vol. 35, no. 3, pp. 163–169, Mar. 1993.



Márcio E. Delamaro received the BS degree in computer science in 1985 from the Campinas State University (UNICAMP), Brazil, the MS degree in computer science in 1993 from the University of São Paulo (USP), Brazil, and the DS degree in computational physics in 1997 also from USP. From January 1995 to June 1996 he was at Purdue University, as a visiting scholar. Currently, he is a professor of computer science at the Maringá State University (UEM). His areas of interest are software testing, formal methods, and programming languages. He is a member of the Brazilian Computer Society (SBC) and the IEEE Computer Society.

José C. Maldonado received the BS degree in electrical engineering/electronics in 1978 from the University of São Paulo (USP), Brazil, the MS degree in telecommunications/digital systems in 1983 from the National Space Research Institute (INPE), Brazil, and the DS degree in electrical engineering/automation and control in 1991 from the University of Campinas (UNICAMP), Brazil. He worked at INPE from 1979 up to 1985 as a researcher. In 1985, he joined the Computer Science Department of the Institute of Mathematics and Computer at the University of São Paulo (ICMC-USP) where he is currently a faculty member. He has been a visiting scholar at the Technical University of Denmark, and at Purdue University, supported by the Fulbright Program and the Brazilian funding agencies CAPES and CNPq. His research interest are in the areas of software quality, software testing, and reliability. He has published more than 50 research papers and written one book. He is a member of the Brazilian Computer Society (SBC). He is a member of the IEEE Computer Society.

Aditya P. Mathur received the BS, MS, and PhD degrees in 1970, 1972, and 1977, respectively, from the Birla Institute of Technology and Science, Pilani, India. He is currently a professor of computer science at Purdue University, director of the Software Engineering Research Center, and associate head of the Computer Science Department. His research interests lie in software testing and reliability, program auralization, and music composition. He has published more than 75 research papers, written two books, and composed over 25 pieces in a variety of genres. He is a member of the IEEE.