Software Testing Optimization Models

Chapter · January 2006

4 authors:

Crescenzio Gallo

Università degli studi di Foggia

100 PUBLICATIONS **100** CITATIONS

SEE PROFILE

Giancarlo Stasio

Università degli studi di Foggia

7 PUBLICATIONS **3** CITATIONS

SEE PROFILE

Cristina Di Letizia

Università degli studi di Foggia

3 PUBLICATIONS **3** CITATIONS

SEE PROFILE

A. Di Francesco

Università degli studi di Foggia

1 PUBLICATION **0** CITATIONS

SEE PROFILE

SOFTWARE TESTING OPTIMIZATION MODELS

Crescenzio Gallo, Giancarlo De Stasio
{c.gallo, g.destasio}@unifg.it

Dip.to di Scienze Economiche, Matematiche e Statistiche - Università di Foggia
Largo Papa Giovanni Paolo II, 1 - 71100 Foggia (Italy)

Abstract

Software errors can be a serious problem, because of possible damages (and related costs) and the burden of the needed corrections. Software testing, whose aim is to discover the errors in software products, requires a lot of resources and from it derives the overall quality (i.e. reliability) of the software product. Testing is a decisive factor in attempting to discover as soon as possible, and so reduce, the presence of errors; it involves a lot of resources against an error discovery expectation, and is thus susceptible of optimization: i.e., the best equilibrium between the number of tests to make and the global expected value of discovered errors has actually to be achieved. In fact, it is not economically feasible to proceed with testing over a given limit as well as to execute too few tests, running in the risk of having too heavy expenses because of too residual errors. In the present work we propose some models of software testing optimization, making use of an integer linear programming approach solved with a "branch & bound" algorithm.

1. Introduction

The presence of errors in system and application software causes remarkable costs, both because of the damages provoked and the burden of the needed corrections. Testing, an activity whose aim is to discover the (inevitable) errors embedded in software products, requires a lot of resources and costs during the development and maintenance phases.

The overall quality of the software product, especially in critical application domains such as the medical one [14], is therefore mostly related to the presence (or absence) of errors, that is to reliability that can be defined as the probability that it works properly (without errors) within a given period of time [10], [12], [7], [9]. Even though noting that error is not an intrinsic characteristic of software (because of its being tied to use conditions and user expectations) it is undoubtable that testing is a decisive factor in attempting to discover as soon as possible, and so reduce, the presence of errors [3]. Testing takes up a lot of (money and time) resources against an error discovery expectation, and is thus susceptible of optimization: i.e., the best equilibrium between the number of tests to make and the global expected value of discovered errors has actually to be achieved. In fact, it is not economically feasible to proceed with testing over a given limit as well as to execute too few tests, running in the risk of having too heavy expenses because of too residual errors [13]. Of course, given the intrinsic probabilistic nature of error discovery, the value of a test-case execution may be assigned only with probabilistic (or statistical) criteria, or with a-priori (even rough) estimates [8], [15].

2. Software Testing

Testing is an activity that receives in input the software product and outputs a quality report. Validation is performed against requirements (which expose the software functional nature, i.e. “what it does”) and project (which expose the software logical nature, i.e. “how it does”) specifications [10], [3], [13], [2].

2.1. Testing approaches

Philosophies upon which testing methods are based are essentially two: Path Testing and Functional Testing. Path testing aims to “exercise” program graph paths. Functional testing aims to verify functional requirements, depending on input-output software relationships [11], [4]. Such approaches are enforced by several studies revealing a strong correlation between software complexity (given by its structural properties) and error characteristics (number, type, discovery and correction time). The program structure influences the number of the possible paths (e.g., a program with a backward “go to” in its execution flow has potentially infinite paths) [8]. Being, however, an exhaustive testing (of all the possible paths defined by program logic and functionality) impossible, one my want to select an optimal (in economic sense) test case set.

2.2. Testing strategies

There are “static” testing strategies, which do not imply program execution. They substantially lead themselves to a deep code inspection and reach, however, preliminary and limited outcomes [3], [1], [9]. There are several “dynamic” strategies (based upon program execution), from which the most significant seem to be [3], [1], [9].

Path testing

It tends to execute every program graph path at least once; given the huge number of possible paths (in a nontrivial program) this strategy can be applied only with drastic reductions. Besides, it may be observed that the candidate errors to be revealed by path testing are those of logic and computational type.

Branch testing

It requires each branch in a program to be tested at least once; it is one of the simplest and best known strategies, but obviously less effective than path testing because error depends more upon branches combination than upon a single one.

Functional testing

It aims to verify software functionalities independently from its internal structure (i.e. implementation logic). It requires test-case selection to be made upon functional relationships between input and output domain values. Being of course all the possible I/O relationships near to infinity, here too there is the problem of extracting a suitable test case set.

Structured testing

It approximates path testing, and requires the software system under test to be broken down into a hierarchy of functional modules, in which the single modules are first tested (functional testing) and then their integration inside the whole software system is verified (integrated path testing). Structured testing increases branch testing reliability because it allows verifying branch combinations.

No one testing strategy is complete and reliable, however, in the sense that no one of them guarantees software correctness.

2.3. Approaches for testing optimization models

The behavior of tests effectiveness with respect to allocated resources (work, money, time, computation) is asymptotic (see fig. 1). This means that, among all possible tests, only a part of them is economically useful [13], [2], [15]. Testing optimization is a particular resource allocation problem. Let us assign to each program's runnable path its test execution cost (e.g. related to execution time) and its test execution value related to possible found errors (e.g. related to instructions number and/or program blocks executed). The tests' execution total value is the sum of selected paths' tests execution, and the tests' execution total cost is the sum of their corresponding costs.

Two possible approaches to the definition of models of testing optimization are:

- (M_1) Limited resources model. To maximize the total tests' execution value, given a maximum total execution cost.

- (M_2) Given quality model. To minimize the total tests' execution cost, given a minimum total execution value.

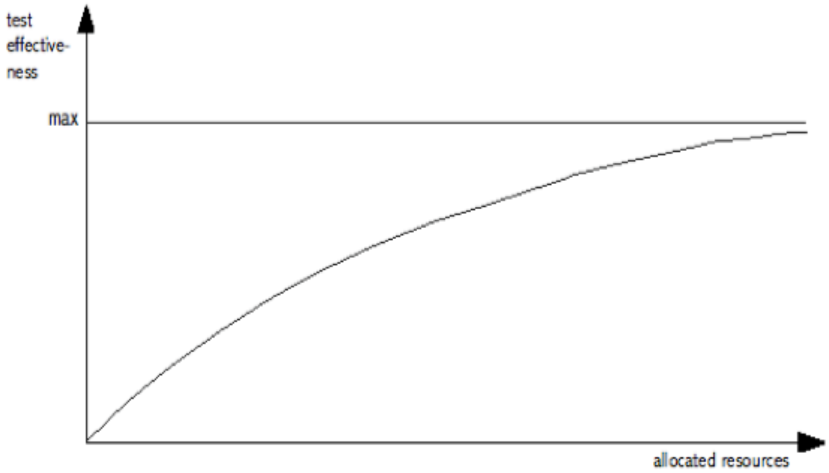


Figure 1: *Test effectiveness related to resources*

3. Testing optimization models

3.1. Model variables

Let us suppose to identify, inside a program, M blocks b_i (i = 1..M). Let N the number of runnable paths p_j (j = 1..N) determined through functional tests. Let:

COMPOSITION MATRIX: a matrix A = $\{a_{i,j}\}$ defined as $a_{i,j} = 1$ if the block b_i belongs to the runnable path p_j, 0 otherwise (for i=1..M and j=1..N). Its columns characterize the b_i blocks of which each p_j path is made up, while its rows characterize the p_j paths containing each b_i block.

COVERAGE DEGREE: the coverage degree of a runnable path is the ratio between the number of its distinct blocks and the number of all program blocks, i.e.:

$$T_j = 1/M \bullet \Sigma_{i=1..M}\, a_{i,j} \quad (j=1..N)$$

EXECUTION MATRIX: a matrix E = $\{e_{i,j}\}$, where $e_{i,j}$ represents the number of times the block b_i is executed within the runnable path p_j (i=1..M; j=1..N).

BLOCK LENGTH: let l_i the length of a block expressed in number of statements.

Let us finally associate to each runnable path p_j a bivalent integer variable $x_j = 1$ if p_j is selected for testing, 0 otherwise (j = 1..N).

3.2. The objective function and the constraints

Let c_j the test execution cost of p_j, and v_j its execution value. E.g., the cost can be initially considered proportional to only the execution time, and not being dependent on error debug and fix times. Test execution value can then be considered equal to the number of statements (code lines) involved by the control flow during p_j execution:

$$v_j = \Sigma_{i=1..M}\, e_{i,j} l_i \qquad (i = 1..N)$$

To hold account of the number of blocks covered by each path p_j, we correct its value through the coverage degree T_j, so its execution test value becomes:

$$v_j = T_j\, \Sigma_{i=1..M}\, e_{i,j} l_i \qquad (i = 1..N)$$

The objective function z_1 of the testing optimization model M_1 (representing the total tests' execution value) is:

$$z_1(x) = \Sigma_{j=1..N}\, v_j x_j, \text{ to be maximized.}$$

For model M_2 the objective function, representing the total tests' execution cost, is:

$$z_2(x) = \Sigma_{j=1..N}\, c_j x_j, \text{ to be minimized.}$$

In the M_1 approach, the overall value's maximization must be done for a given maximum total test execution cost c_{max}, i.e.:

$$\Sigma_{j=1..N}\, c_j x_j \leq c_{max},\ x_j = \{0,1\} \quad (j = 1..N)$$

The M_2 model requires the total test execution cost to be minimized for a given minimum total test execution value v_{min}, i.e.:

$$\Sigma_{j=1..N}\, v_j x_j \geq v_{min},\ x_j = \{0,1\} \quad (j = 1..N)$$

Both are integer bivalent linear programming models.

3.3. Block redundancy effect

The previous base models do not allow for program's topological coverage factor, definable as the ratio between the number of distinct blocks within the chosen paths, and the total number of blocks in program. In fact, because M_1's objective function and M_2's constraint does take into account the total number of executed - but not necessarily distinct - blocks, it may happen that the paths selected in an optimal solution of M_1 or M_2 cover a number of distinct blocks lower than those coverable within another nonoptimal (but obviously better) solution: i.e., the redundancy effect of blocks within the various test-case paths (measuring the correlation degree between paths in terms of common blocks) is not considered. The REDUNDANCY EFFECT can be defined, for each block b_i, as the number of paths in which the block is present respect to the total number of paths:

$$RE_i = 1/N \cdot \Sigma_{j=1..N}\, a_{i,j}$$

The GLOBAL REDUNDANCY EFFECT OF Blocks for all blocks is given by all RE_i's average value, i.e.:

$$GREB = 1/(MN) \cdot \Sigma_{i=1..M}\, \Sigma_{j=1..N}\, a_{i,j}$$

3.4. Changing M_1 and M_2 models

In order to allow for block redundancy effect, a path execution test value may be corrected according to the value of the remaining selected paths and their blocks. This corresponds to associate to each block N partial values (one for each path), representative of the contribution given by the block to the test value of each path. Let $w_{i,j}$ the partial value associated to block b_i in path p_j. If we assume $w_{i,j}$ to be positive only for those paths it belongs to, and that the sum of the partial values of the blocks be the path's overall value v_j, we obtain:

$$w_{i,j} = e_{i,j}\, l_i\, T_j \qquad (i = 1..M;\ j = 1..N)$$

To obtain the best overall value, it will be necessary to hypothesize the inclusion, in the solution, of the blocks in those paths in which they have the best (maximum) partial value $w_{i,j}$; from that, the total test execution value allowing for the blocks' redundancy effect is given by the total sum of the maximum partial values of each block. From above said, the new formulation of model M_1 becomes:

$$\max z_1(x) = \Sigma_{\iota=1..M} \max_{j=1..N} \{w_{i,j} x_j\}, \text{ subject to}$$

$\Sigma_{j=1..N}\ c_j x_j \le c_{max},\ x_j = \{0,1\}\ \ (j = 1..N)$

Model M_2 becomes:

$\min z_2(x) = \Sigma_{j=1..N}\ c_j x_j$, subject to

$\Sigma_{\iota=1..M}\ \max_{j=1..N}\{w_{i,j}x_j\} \le c_{max},\ x_j = \{0,1\}\quad (j = 1..N)$

A completeness index of the test done is given, for both models, by the ratio between the obtained total test value and the desired total test value, i.e.:

$R = \Sigma_{\iota=1..M}\ \max_{j=1..N}\{w_{i,j}x_j\} / \Sigma_{\iota=1..M}\ \max_{j=1..N}\{w_{i,j}\}$

4. Solution algorithm for the proposed optimization model

The two proposed optimization models are similar; we develop model M_1 because it appears more suitable to real situations. Being M_1 a bivalent integer linear programming model, the most efficient (from the application simplicity and convergence speed to optimum solution point of view) algorithms are those based on “branch & bound” techniques. In particular the Land and Doig (1960) algorithm has been chosen, for the “knapsack” suitably modified problem. This algorithm has some valuable advantages, such as:

- it is simple to apply;
- the maximum number of steps for it to converge is about 2^N (where N is the number of p_j chosen runnable paths);
- it is of combinatorial type and then speedier than an enumerative-like algorithm.

Moving forward in algorithm’s application, the initial problem is transformed into more and more simple ones, expressed similarly to the model, but with an ever lowering variables’ number. Land and Doig’s algorithm’s formulation for the model’s solution is:

$\max z(x) = \Sigma_{\iota=1..M}\ \max_{j=1..N}\{w_{i,j}x_j\}$, subject to

$\Sigma_{j=1..N}\ c_j x_j \le c_{max},\ x_j = \{0,1\}\ \ (j = 1..N)$

5. Application of the model in software maintenance

In this section it will be examined the application of the proposed model of testing optimization to the software being modified in the maintenance phase.

5.1. Software maintenance

The maintenance process objectives are ([10], [4], [12], [7]):

• correction of errors missed in the testing phase and discovered during use;
• introduction of new functionalities;
• elimination of no more required functionalities;
• adaptation of software to new hardware configurations;
• possible optimization.

Whichever the objective is, the maintenance operation requires the following steps:

1. determining the detailed lists of the maintenance operation;

2. understanding of the software module(s) on which to act;
3. finding the acting points;
4. finding the elements upon which the adopted changes' effect spreads;
5. final testing of the changes.

Given the high algorithmic content and architectural complexity of the system software, one thinks that this is the more interesting area in the application of the techniques exposed in the present work. The more interesting is step 4 in which, after applying the required corrections to software after the specific operation, one passes to examine all the instructions (or sets of instructions, such as e.g. blocks, routines, functional modules) which has to possibly be corrected because of the adopted changes.

Above all, step 5 is particularly important, in which we aim to apply the optimization model only to those paths being influenced by changes.

5.2. Changes testing

After a program modification a very careful testing has to be made, in order to verify that the carried out changes do not degrade (instead to improve) software quality.

In the maintenance phase it turns also out that it is usually impossible to get an exhaustive testing, and the testing optimization models aim to select a certain subset of paths to be tested (*selective testing*, [15]).

A valid help to our task may come from the "influence matrix" IB=$\{i_{k,p}\}$ among the blocks of a program, so defined: $i_{k,p}$ = 1 if block k influences block p (directly or not), 0 otherwise, for k, p = 1..M, being M the number of distinct blocks in the program.

Starting with the matrices A (see 3.1) and IB, the paths influenced by the changes can be identified as those containing at least one block influenced by a change. Through these premises it is possible to develop an algorithm that allows to obtain the set of the paths (runnable before the modification) influenced by the change of the generic block b_k.

Let BJ the indexes set of the program's blocks influenced by the change. If J is the indexes set of the runnable paths *before* the change, and A_j is the j−th column of the composition matrix A of the runnable paths, then PBJ = $\{j \in J \mid A_j \cap BJ \neq \emptyset\}$ is the indexes set of the paths influenced by the change.

5.3. Application of the testing optimization model

After making some changes to a program, one may consider to apply the testing optimization model to select one or more subsets of runnable paths. The testing of the chosen paths must guarantee that there has been no degradation of software quality after change application ([13], [12], [2]).

For the testing optimization model to be applied the runnable paths need to be defined, among those the solution model's algorithm must select the paths to test.

Let J^* the indexes set of all the paths not influenced by the changes; the paths in J^* are not therefore considered in the application of the model.

The paths influenced by the changes are represented by the set PBJ (see 5.2). Holding into account that every path identifies a program functionality, for each path in PBJ it can happen that:

- the functionality carried out by that path has been removed because no more required: the path is then no more runnable and will no longer be considered in model's application;

• the functionality carried out by that path is executed in a wrong manner: some changes have been adopted, the old path identified by a PBJ's index has been removed and a new one (not belonging to J^*) is defined, which must be considered in the model solution algorithm's application;

• new functionalities and then new paths have been introduced, which has to be considered in model's application.

Starting with the detailed lists of changes it is then possible to point out a set of runnable paths.

Let J' = {1, 2, ..., N'} the set of runnable paths within the modified program. Usually, program's modifications imply the changes of some program's particular characteristics such as:

• total number of code lines;

• number of blocks M;

• matrices A, E, IB, W;

• values v_j for the paths in the set J^*.

Let us suppose the changes do not influence the paths identified by J^*:

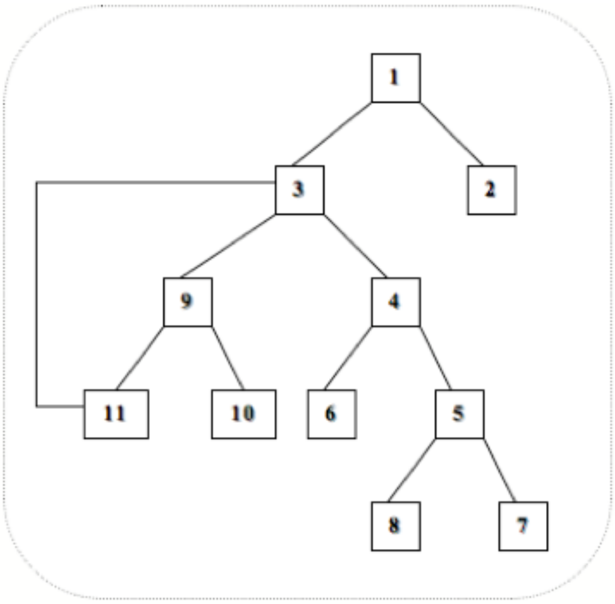


Figure 2: *A sample program's graph*

therefore such paths, being error-free and not modified, will not be considered in the model's application.

Let then c_j the test execution cost of the j−th path, and v_j its test execution value. Let M' the number of blocks and W' = $w'_{i,j}$ (i=1..M', j=1..N') the partial values matrix of the modified program.

The optimization model in its application to the modified program then becomes:

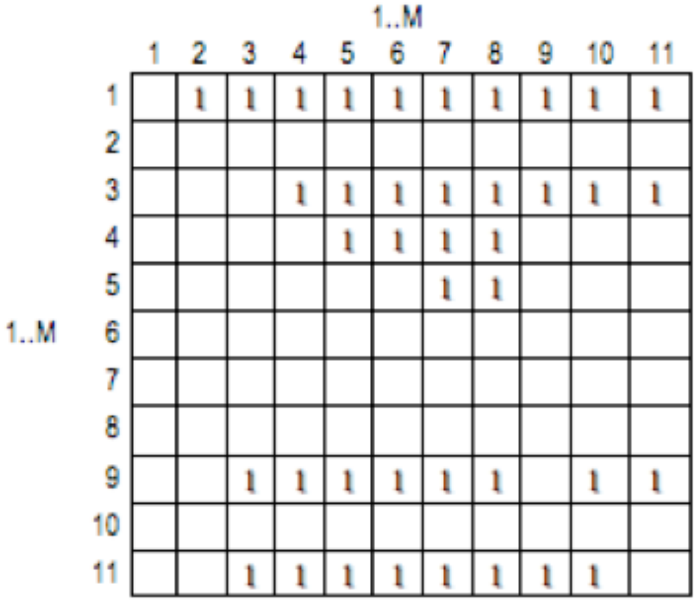
max z (x) = $\sum_{i=1..M} \max_{j=1..N} \{w_{i,j}\, x_i\}$, subject to

$\sum_{j=1..N} c_j x_j \leq c_{max}$, x_j={0,1} (j=1..N).

In conclusion, it is opportune to consider the importance that covers the blocks' effect of redundancy upon the cardinality of the set PBJ. It is easy to understand that the cardinality of such a set is proportional to the redundancy degree RE_i

(see 3.3) of the blocks b_i modified or influenced by the changes.

In particular, if we suppose to modify an instruction of the block b_1 (the first block in the program, common to all the runnable paths; see figures 2 and 3) we have $RE_1=1$, i.e. the block belongs to all paths and so they all have been influenced by the change; therefore, card(PBJ) = N.



1..M

		1	2	3	4	5	6	7	8	9	10	11
	1		1	1	1	1	1	1	1	1	1	1
	2											
	3				1	1	1	1	1	1	1	1
	4					1	1	1	1			
	5							1	1			
1..M	6											
	7											
	8											
	9			1	1	1	1	1	1		1	1
	10											
	11			1	1	1	1	1	1	1	1	

Figure 3: *Influences matrix IB of sample program*

6. Conclusions

In order to solve (if it is ever possible to identify all the functional paths of a program) the problem of the optimal choice of paths for software testing (that is the identification of optimal test data) testing optimization models are definable, based on factors such as:

- the program structure (defining the path's test execution value, correlated with the number of executed instructions);
- the available resources for the program testing (defining the path's test execution cost, correlated with the execution time).

The proposed software testing optimization model is of bivalent integer linear programming type:

$$\max z(x) = \Sigma_{i=1..M} \max_{j=1..N} \{w_{i,j}x_j\}, \text{ subject to}$$

$$\Sigma_{j=1..N}\, c_j x_j \le c_{max},\ x_j=\{0,1\} \qquad (j=1..N)$$

The problems that can arise in the application of such optimization model are:

- the practical impossibility to identify all the runnable paths inside a program (or software system);

- a greater memory requirement to store the composition matrix of runnable paths, the execution matrix of runnable paths, the matrix of partial values, the representative tree of the "branch & bound" algorithm's application for the solution of the model;

- a greater execution time of the solution algorithm.

A real application of the testing optimization models is the optimal choice of paths to test after a program change during the software maintenance phase. From a strictly practical point of view, it is just this last application that can turn out much effective to reach a high level of software quality. The proposed testing optimization model does not provide any suggestion (not even methodological) for estimating the number of residual errors in a program, after completing the testing activity. We think this is the direction toward which to address research, in order to achieve a deeper knowledge about software products reliability.

References

[1] Basili V.R., Belby R.W., 1987, *Comparing the effectiveness of software strategies*, IEEE Trans. on Soft. Eng., vol. SE-13, n. 12.

[2] Ceriani M., Marini D., Palmieri L., 1980, *Un esperimento di valutazione del testing di un programma in un ambiente industriale di produzione del software*, in Atti del Congresso AICA, pp. 831–840.

[3] Fairley R.E., 1978, *Tutorial: Static analysis and dinamic testing of computer software*, IEEE Computer, vol. 11, n. 4, pp. 14–23.

[4] Gelperin D., Hetzel B., 1988, *The growth of software testing*, Comm. of the ACM, vol. 31, n. 6.

[7] Hamlet R., 1988, *Special section on software testing*, Comm. of the ACM, vol. 31, n. 6.

[8] Kuo-Chung T., 1980, *Program testing complexity and test criteria*, IEEE Trans. on Soft. Eng., vol. SE-6, n. 6, pp. 531–538.

[8] Land A., Doig A., 1960, *An automatic method of solving discrete programming problems*, Econometrika, 28(3), pp. 497–520.

[9] Miller E., Howden W.E., (Eds.), 1981, *Tutorial: Software Testing and Validation Techniques*, IEEE Computer Society Press, New York.

[10] Myers G.J., 1976, *Software Reliability: Principles and Practices*, John Wiley and Sons, New York.

[11] Ostrand T.J., Balcer M.J., 1988, *The category-partition method for specifying and generating functional tests*, Comm. of the ACM, vol. 31, n. 6.

[12] Ramamoorthy C.V., Bastiani F.B., 1982, *Software reliability: Status and perspectives*, IEEE Trans. on Soft. Eng., vol. SE-8, n. 4.

[13] Sorkowitz A.R., 1979, *Certification testing: A procedure to improve the quality of software testing*, IEEE Computer, vol. 12, n. 8, pp. 20–24.

[14] Weide P., 1994, *Improving medical device safety with automated software testing*, Med. Dev. Diag. Indust., 16(8):6679).

[15] Zeil S.J., White L.J., 1981, *Sufficient test sets for path analysis testing strategies*, in: Proceedings of the 5s1 International Conference on Soft. Eng., IEEE, San Diego, California, pp. 184–191.