Available online at www.sciencedirect.com

The Journal of Systems and Software 79 (2006) 756–771

The Journal of Systems and Software

www.elsevier.com/locate/jss

Constraint based structural testing criteria

Silvia Regina Vergilio [a,*], José Carlos Maldonado [b], Mario Jino [c], Inali Wisniewski Soares [d]

[a] *DInf-UFPR, C.P. 19081, 81531970 Curitiba, Brazil*
[b] *ICMSC-USP, C.P. 668, 13560-970 São Carlos, Brazil*
[c] *DCA-FEEC-UNICAMP, C.P. 6101, 13083-970 Campinas, Brazil*
[d] *UNICENTRO, C.P. 3010, 85010-990 Guarapuava, Brazil*

Received 25 June 2004; received in revised form 7 June 2005; accepted 11 June 2005
Available online 27 July 2005

Abstract

Structural criteria generally divide the input domain of the program under test and require the execution of at least one point from each derived sub-domain without addressing the most relevant question: "Which points from each sub-domain should be selected?". This question is related to data-sensitive faults which lead to one of the drawbacks of the testing activity. The constraints and conditions used by fault-based data generation techniques describe faults related to the boundaries of these sub-domains. Our conjecture is that we would improve the efficacy of the adequate test case sets by associating those constraints and conditions to the elements required by a criterion. With this goal, this work presents Constraint Based Criteria (CBC) that associate a constraint *C* to an element *E*, required by a structural criterion. CBC allow to combine the fundamentals of different testing generation techniques with structural testing, increasing the probability of revealing faults described by *C*. We also discuss complexity, inclusion relation and automation aspects of CBC. Results from three experiments of CBC evaluation using the factors cost, efficacy and strength provide evidence that our objectives have been achieved. We also present some results from the evaluation of random test sets.
© 2005 Elsevier Inc. All rights reserved.

Keywords: Data flow based criteria; Constraint based testing; Mutation analysis

1. Introduction

One of the goals of testing is to reveal as many faults as possible. It is a fundamental activity for software quality assurance. The testing activity aims at developing "good test cases". A good test case is one that has a high probability of finding an unrevealed fault. Testing techniques and criteria have been proposed to achieve the testing goals with minimal effort and costs. Testing criteria can be seen as predicates to be satisfied and are usually derived by applying one of the following techniques: functional, structural (control and data-flow based criteria) and fault-based (mutation analysis criterion). The efficacy of these criteria may depend on the strategy used for generating test data sets adequate to each criterion.

In principle, any testing criterion *C* can be used, with its associated advantages and disadvantages, as a mechanism either to assess or to select test case sets. Generally speaking, these criteria require program elements to be exercised. Each element partitions the program input domain into sub-domains and any point selected from the related sub-domain is considered adequate to exercise the corresponding required element. Most of these criteria, specifically the structural testing criteria, do not address the most relevant question: "Which

[*] Corresponding author. Tel.: +55 41 33613411; fax: +55 41 33613205.
E-mail addresses: silvia@inf.ufpr.br (S.R. Vergilio), jcmaldon@icmsc.sc.usp.br (J.C. Maldonado), jino@dca.fee.unicamp.br (M. Jino), inali@unicentro.br (I.W. Soares).

0164-1212/$ - see front matter © 2005 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2005.06.012

points from each sub-domain should be selected?" (Frankl and Weyuker, 1993). This question is related to data-sensitive faults which lead to one of the drawbacks of the testing activity.

The selection of "good test cases" has been addressed by test data generation techniques, using different principles driven by fault classification and models. These techniques can be classified into: (1) random generation (Duran and Ntafos, 1984); (2) generation based on symbolic execution (Clarke, 1976; Howden, 1977; Ramamoorthy et al., 1976); (3) generation based on dynamic execution (Korel, 1990); (4) generation based on genetic algorithms (Bueno and Jino, 2001; Michael et al., 2001) and (5) generation based on fault-sensitive techniques. Fault-sensitive techniques lead to the selection of points from the input domain of a program related to specific kinds of faults. The points selected are expected to have a high probability of revealing the corresponding fault. Examples of these techniques are: Domain Testing (White and Cohen, 1980), Constraint Based Testing (De Millo and Offutt, 1991), (BOR/BRO) Boolean and Relational Operator Testing, BRE(ε) Boolean and Relational Expression Testing with parameter ε (Tai, 1993). The fault-sensitive techniques establish constraints and conditions to choose the test data based on the assumption that the faults are related to the boundaries of the sub-domains. It is assumed that a given sub-domain or program element has already been identified.

Our conjecture is that by associating to the required elements the constraints and conditions of the test data generation techniques, we would improve the efficacy of the adequate test case sets, i.e., we increase the number of revealed faults.

Based on a study using simple programs from the literature and exploring fault-sensitive techniques, we introduced the Constraint Based Criteria (CBC) by associating a constraint to the required elements of a given criterion, and we conducted a preliminary evaluation, named Experiment I (Vergilio et al., 1997). A required element of the CBC is of the form (E, C); a test case exercises element (E, C) if it exercises E and satisfies constraint C. Constraint C is related to kinds of faults; it is a condition that describes faults. More than one constraint can be associated to an element. The test sets generated to satisfy the structural criterion have a high probability of revealing the fault described by C. Results from Experiment I, have encouraged the implementation of a framework to support CBC criteria. Afterwards, another study, Experiment II, using the same Unix programs previously used by Wong et al. (1994), was carried out (Vergilio et al., 2001; Soares and Vergilio, 2004).

Although in both experiments we observed an increase in the efficacy, in Experiments I and II, we adopted criterion based strategies to satisfy the criteria, that is, all the adequate test sets were manually generated in an "ad hoc" way to cover a particular element. The application of a criterion based strategy is not always possible because it can be very expensive. In general, random test sets are utilized, because this kind of generation seems to be more practical (Duran and Ntafos, 1984). However, random test sets do not assure the satisfaction of a criterion and the elements that are not covered can imply a lower efficacy. Some questions related to these aspects can be posed: "What is the mean coverage obtained with random sets?" or "Do the results related to strength, cost and efficacy and, obtained in Experiments I and II remain valid with random generation?".

This work synthesizes Experiments I and II and describes results of another experiment, Experiment III conducted with randomly generated test sets previously used by Wong et al. (1994). Experiment III, has the goal of evaluating the random strategy with CBC, by answering the above questions. The results of these three experiments provide a good basis to devise a strategy for application of the studied testing criteria.

This paper is organized as follows. Section 2 provides an overview of the testing activity: techniques and criteria. A short description of the most relevant fault-sensitive data generation techniques is also presented. In Section 3, we summarize the motivation to introduce CBC and define some criteria based on the most known structural testing criteria. In addition, we present CBC property analysis: complexity, inclusion relation and implementation aspects. In Section 4, Experiments I and II are revisited and an analysis comparing the results from both experiments is done. Section 5 describes Experiment III with random generation and presents steps of a testing strategy that combines random and incremental criterion based strategies. In Section 6, we summarize the main results and achievements and discuss future work.

2. Background: testing criteria

Structural testing criteria consider a particular implementation to derive the testing requirements. They can be classified as control flow, complexity and data flow based criteria.

For the control flow based criteria, the basic abstraction used to establish the required elements is the Control Flow Graph (CFG). Basically, they require the execution of components of the program under test in terms of corresponding elements of the CFG: nodes, edges and paths. A node characterizes a block of statements executed sequentially. Edges are associated with possible transfers of control between nodes. A path is determined by a sequence of nodes. The complexity based criterion requires the execution of all independent paths of the program; it is based on McCabe's complexity concept (McCabe, 1976).

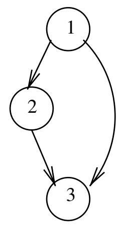
Essentially, the control flow and complexity based criteria would not favor revealing computation faults since the computations in the statements do not necessarily affect the CFG used to establish the required elements. Data flow testing criteria (Herman, 1976; Laski and Korel, 1983; Maldonado et al., 1992; Ntafos, 1984; Rapps and Weyuker, 1985; Ural and Yang, 1988) attempt to overcome this limitation by requiring data flow associations to be exercised: the interactions between variables definitions and subsequent references (uses, either predicative or computational) to those definitions.

Consider, for instance, program *max* (De Millo and Offutt, 1991) that prints the largest of its two inputs, and its CFG (Fig. 1). Table 1 shows examples of control and data flow required elements. To satisfy a structural criterion C it is necessary to cover all the elements required by the criterion; in this case we say that 100% coverage is obtained and that the corresponding test case set T is C-adequate. Infeasible paths are one of the problems to applying the structural criteria. A path of a program is infeasible if there is no set of values for the input variables, global variables and parameters of the program that cause the path to be executed. In particular, an association (or any required element) can be covered by a set of different paths of the program; if all the paths in that set are infeasible the element is infeasible. In the presence of infeasible elements it is not possible to obtain 100% coverage. The tester is responsible for determining the path feasibility, since this question is, in general, undecidable (Frankl and Weyuker, 1988).

All-uses criterion is one of the Rapps and Weyuker's family of data flow testing criteria (Rapps and Weyuker, 1985). A variant family of Rapps and Weyuker's testing criteria, named Potential Uses Criteria family, is based on the concept of Potential Use (Maldonado et al., 1992). The Potential Use criteria do not require an explicit use to occur, as all of the data flow based criteria do; simply a definition clear path from nodes where a definition occurs to another node is required in order to characterize an association, in this case a potential-

```
    void max()
1   {  int mx, m, n;
1      scanf("%d %d", &m,&n);
1      mx = m;
1      if (n>=m)
2        mx = n;
3      printf("%f", mx);
3   }
```

a



b

Fig. 1. Program *max*. (a) Source code; (b) CFG.

Table 1
Some control flow and data flow required elements

Criterion	Examples of required elements
All-nodes	1, 2, 3
All-edges	$(1,2)$, $(2,3)$, $(1,3)$
All-paths	$(1,2,3)$, $(1,2)$, $(2,3)$, $(1,3)$
All-uses	$(1,(1,2),\{n,m\})$, $(1,(1,3),\{n,m\})$, $(1,2,\{n\})\ldots$
All-potential-uses	$(1,(1,2),\{n,m\})$, $(1,(1,3),\{n,m\})$, …

association. Another representation for an association is introduced, such as $(1,(1,2),\{n,m\})$ (last rows of Table 1). This notation represents two associations $(1,(1,2),n)$ and $(1,(1,2),m)$ and can be used because there is a path in the graph that does not redefine all variables in the association; in this case, n and m.

Concerning the Potential Uses criteria, to derive the required elements we only need to associate variable definitions to each node in the CFG. The tool Poketool (Chaim, 1991) generates a list of required elements and does an adequacy analysis of a given test data set, determining coverage measures for the Rapps and Weyuker's and Potential Uses criteria.

Although the main motivation of the testing criteria is to reveal faults, structural testing criteria address the faults somewhat indirectly in order to establish the required elements. On the other hand, fault based criteria deal with the kinds of faults more directly. For instance, one of the key points of mutation testing, a fault based criterion (De Millo et al., 1978), is the mutation operators used to generate mutant programs. These operators can be related to a fault model that reflects typical faults committed by a developer. All mutants are executed using a given input test case set T. If a mutant M presents different behavior from P, it is said to be dead; otherwise, it is said to be alive. In this case, either there is no test case in T that is capable of distinguishing M from P or M and P are equivalent. Determining equivalence between programs is, in general, undecidable (similar to the infeasibility problem). The objective is to find a test case set able to kill all non-equivalent mutants; such a test case set T is considered adequate to test P. The Mutation Score (MS), obtained by the relation between the number of mutants killed and the total number of non-equivalent mutants generated, is used to assess the adequacy of a given test case set.

In the experiment reported herein we use Proteum (PROgram TEsting Using Mutation) (Delamaro and Maldonado, 1996), that supports Mutation Testing for C programs. Table 2 shows examples of mutation operators available in Proteum, illustrated using program *max* (Fig. 1).

All the testing criteria present advantages and weaknesses; hence, studies comparing them and addressing their complementary aspects are very important (Frankl and Weyuker, 1993; Maldonado et al., 1992; Weyuker, 1988; Weyuker, 1990; Wong et al., 1994). These studies,

Table 2
Examples of mutations

Operator	Mutation
Relational operator by	/* 1 /* if ($n > = m$)
Relational operator	/* 1 */ if ($n > m$)
Variable by	/* 1 */ $mx = n$
Other variable	/* 1 */ $mx = m$

either theoretical or empirical, are usually based on three factors (Wong et al., 1994): (1) cost: the number of test cases required to satisfy the criterion; (2) efficacy: the ability to reveal faults; (3) strength: the difficulty of satisfying a criterion, given that another one has been satisfied. This last factor, from the theoretical point of view, is related to the inclusion relation amongst criteria (Rapps and Weyuker, 1985). A criterion C_1 includes a criterion C_2 (denoted by $C_1 \rightarrow C_2$) if, for every program, every test data set that satisfies C_1 also satisfies C_2. If neither C_1 includes C_1 nor C_2 includes C_1, C_1 and C_2 are incomparable. The inclusion relation is often used to compare criteria considering Factors 1 and 3; however, the fact that C_1 includes C_2 does not guarantee that C_1 is better at revealing faults than C_2 (Frankl and Weyuker, 1993).

Under certain conditions, it has been shown that the data flow based criteria include the control flow based criteria. Moreover, some empirical studies show that Mutation Analysis is more efficacious and expensive than the structural criteria (Wong et al., 1994). Maldonado et al. (1992) show that the criteria of the PU family include the criteria proposed by Rapps and Weyuker and that no other data flow criterion includes PU criteria. Moreover, some of the PU criteria bridge the gap between all-edges and all-paths criteria even in the presence of infeasible paths. Structural criteria and mutation testing are theoretically incomparable and only empirical studies can point out the relationship among these criteria, in practice (Wong et al., 1994).

The structural and mutation based criteria have been proposed and used, at least at the beginning, to address the coverage issue. Nevertheless, as any testing criterion, they can be seen as a mechanism either to assess or to select a test case set. In the latter case, a test data generation technique should be considered. Since we are interested in aggregating these two perspectives, next we focus on the most relevant fault-sensitive data generation techniques.

Domain Testing (DT). It is based on a geometrical analysis of the domain boundary. Points are chosen on or near the domain borders that are most sensitive to domain faults. The predicates through a path determine the borders of the path domain. A closed border is formed by predicates with $\geqslant$, $\leqslant$ or $=$, and an open border is formed with $<$, $>$ and $\neq$. Ideally, Domain Testing should be applied to all paths in a program, however the number of paths may be practically infinite. Therefore, the DT technique should be used with a path based criterion, for example, a structural criterion, that determines the set of paths to be executed. The basic Domain Testing technique, named $N \times 1$ technique, was proposed by White and Cohen (1980), under a set of assumptions. It selects N ON points and 1 OFF point to test a given border of the path domain, where N is the domain dimension, given by the number of variables in the predicate that forms the border being tested. ON points are on the border being tested. OFF points are at a small distance ε, in the open side of the given border.

Constraint Based Testing (CBT). It was proposed by De Millo and Offutt (1991). Algebraic constraints are used to describe a specific type of fault. CBT aims at generating test cases with the goal of killing the mutants. To kill a mutant a test case has to satisfy the following conditions: (1) reachability condition—the test case executes a path that reaches the changed statement; (2) necessity condition—the state of the program after execution of the mutated statement is incorrect; (3) sufficiency condition—the final state of the program is incorrect. For example, consider the program *max*, (Fig. 1) and the last mutant of Table 2. The necessity condition to produce an incorrect intermediate state in the program is $m \neq n$ but it is not sufficient to produce an incorrect final result. The sufficiency condition is $m \neq n$ and $n < m$. DeMillo and Offut observe that determining reachability and sufficiency conditions is not always possible; it is an undecidable problem. They assume that if a test case meets the necessity condition it will usually meet the sufficiency condition and will contribute to kill one or more mutants.

Predicate Based Testing. It was proposed by Tai (1993) to generate a test set that guarantees detection of boolean and relational operator faults. They are: Boolean Operator (BOR) testing, Boolean and Relational Operator (BRO) testing, and Boolean and Relational Expression testing with parameter ε (or BRE(ε) testing). Tai defines a set of constraints X to a predicate C; X can be viewed as an input of C and a constraint of X can be: t (true), f (false), $>$ (>0), $=$ ($=0$), or $<$ (<0). Combinations of these constraints are established for composed predicates according to its boolean operators. These constraints must be satisfied by the test cases. The idea of BRE is to replace occurrences of $>$ and $<$ by $+\varepsilon$ and $-\varepsilon$, respectively, where ε is the smallest number greater than 0, using fundamentals of DT.

3. CBC: definition and property analysis

Prior to Experiments I and II, we had accomplished some theoretical and practical studies with the fault-sensitive techniques mentioned in the last section. The goal was to obtain a strategy for satisfying structural testing criteria and increasing its efficacy. The CBT technique was applied considering the Proteum tool operators. DT was applied to paths to cover the associations required by the all-potential-uses criterion supported by Poketool. During the experiment, the efficacy, number of required test cases, and the limitations of each technique were studied. We have observed some interesting results that are motivations to the definition of the CBC; they are listed below.

- *Necessary conditions were not always sufficient, mainly for computational faults; determining sufficient conditions for this kind of fault was more difficult than for domain faults*. In this work, we follow Howden's classification (Howden, 1976): domain and computation faults. Domain faults correspond to an incorrect association of a path in the program and its domain. Computation faults occur when a specific input leads to the correct path, but a fault in one statement causes a wrong function to be computed. We also distinguish faults in data structures, which are related to incorrect declarations of data structures in the program.
- *The combination of Control and Data Flow Testing with DT is very advantageous.* Ideally, the DT technique should be applied to all the paths in the program, but this is not always possible due to loops. The criterion used to choose the set of paths for DT application influences its efficacy. A stronger path based criterion can select a set of paths with a high probability of containing a path that can reveal the fault. But this is not always sufficient and, in this case, the combination of structural criteria with DT is very advantageous.
- *In many cases the sufficient condition is the condition for executing a given path in the program*. In many cases, we observed that the combinations of necessary conditions, such as the ones of CBT, with path conditions have a high probability of meeting sufficient conditions. This result is similar to the one presented by Murril et al. (2002). Our conjecture is that this aggregation will lead to more effective adequate test case sets.

3.1. Definition of Constraint Based Criteria

We define CBC by associating constraints to an element required by a structural testing criterion. Thus, Constraint Based Criteria require pairs (E, C) (structural element, constraint), named constrained elements. When a constraint C is associated to an element E, the input domain of E is constrained. A constrained element will be covered if C is satisfied during the execution of a path that covers E. We can use any structural testing criterion to derive a constraint based criterion. Constraint Based Criteria using the most known structural testing criteria are presented below.

- *Constrained Control Flow Based Criteria*. All-constrained-nodes and all-constrained-edges criteria are defined.
  - *All-constrained-nodes*. Every constrained node (i, C) must be exercised. A constrained node is exercised if C is satisfied during the execution of a path that covers i.
  - *All-constrained-edges*. Every constrained edge $((i,j), C)$ must be exercised. A constrained edge is exercised if C is satisfied during the execution of a path that covers (i,j).
- *Constrained Data Flow Based Criteria*. All constrained-uses and all-constrained-potential-uses criteria are defined.
  - *All-constrained-uses*. Every constrained association $((i, k, x), C)$ or $((i, (j, k), x), C)$ must be exercised. A constrained association is exercised if a path that covers the association is executed and C is satisfied during that execution.
  - *All-constrained-potential-uses*. Every constrained potential-association $((i, k, x), C)$ or $((i, (j, k), x), C)$ must be exercised. A constrained potential-association is exercised if a path that covers the potential-association is executed and C is satisfied during that execution.

Selecting the constraints C is a very important task. The efficacy of test data can depend on it. To improve the efficacy of the test sets C should describe faults revealed by functional and fault-based techniques, generally not directly addressed by the structural criteria. In this way, C could describe faults in data structures, mutation faults, etc. Table 3 shows constraints that can be used to derive constraint based criteria. The constraints are derived by using fundamentals of Mutation and Predicate Based Testing (BOR/BRO). Other techniques can be used such as Boundary Value Analysis. Note that only necessary conditions are derived. These conditions, in addition to the conditions implied by the paths that cover the required structural elements, can also meet sufficient conditions.

To select the constraints, we consider aspects such as information about the program under test, about the program domain, infeasibility, and so on. The last one is related to the conditions of the paths that cover E. The path condition can be inconsistent with the constraint C resulting in an infeasible constrained element. Determining infeasible elements is an undecidable question; hence, reducing the number of infeasible elements is desirable and this aspect should be properly addressed. Other aspects are the desired reliability and CBC's complexity. We suggest to apply CBC using dif-

Table 3
Constraint levels

Level	Related technique	Necessary condition
1 Necessary information (i,j,k,x)	Variable mutation	$X=0,\ X<0,\ X>0$ $A[e_1]>0,\ A[e_1]<0,\ A[e_1]=0$ $S\cdot c=0,\ S\cdot c>0,\ S\cdot c<0$ $p=null$
2 Necessary information a set as: (i,j,x_1), (i,j,x_2), …, (i,j,x_n)	Variable mutation	$A[e_1]\neq B[e_1]$ $p\neq q$ $A[e_1]\neq X$ $S\cdot c\neq X$ $(*p)\neq X$ $Y\neq X$ $S\neq T$ $S\cdot c\neq T\cdot c$ $S\cdot c_1\neq S\cdot c_2$
3 Necessary information $(i,(j,k),x)$ and predicate in (j,k)	BRO/BRE	
	C_1 or C_2	$F(C)=S_{1f}\%S_{2f}$ $T(C)=\{S_{2t}*\{f_2\}\}\$\{\{f_1\}*S_{2t}\}$ $f_1\in S_{1f}ef_2\in S_{2f}e(f_1,f_2)\in F(C)$
	C_1 and C_2	$T(C)=S_{1t}\%S_{2t}$ $F(C)=\{S_{1f}*\{t_2\}\}\$\{\{t_1\}*S_{2f}\}$ $t_1\in S_{1t}et_2\in S_{2t}e(t_1,t_2)\in T(C)$

ferent levels of constraints to reduce costs and the number of infeasible elements, as illustrated in Table 3. The tester can choose one or more levels of constraints to derive the constrained elements. The other levels may be not considered.

Table 4 shows examples of constraints that can be used at each level for program *max*. For example, if the tester chooses Levels 1 and 2 necessary constraints that describe the fault of changing variables in the program will be selected. This option derives 6 constrained-associations for Association 1. If the tester choose Level 3 only, BOR and BRO conditions only are used and 3 constrained-associations are required for Association 1. If Levels 1, 2 and 3 are used, 9 constrained-associations are derived.

Table 4
Constrained-potential-associations for program *max* from Fig. 1

Association	Level 1	Level 2	Level 3
1. $(1,(1,2),n)$	$n>0$ $n<0$ $n=0$	$n\neq m$ $n\neq mx$ $mx\neq m$	$n>m$ $n=m$ $n=m$
2. $(1,(1,2),m)$	$m>0$ $m<0$ $m=0$	$n\neq m$ $n\neq mx$ $mx\neq m$	$n>m$ $n=m$
3. $(1,(1,2),mx)$	$mx>0$ $mx<0$ $mx=0$	$n\neq m$ $n\neq mx$ $mx\neq m$	$n>m$ $n=m$
4. $(1,(1,3),n)$	$n>0$ $n<0$ $n=0$	$n\neq m$ $n\neq mx$ $mx\neq m$	$n<m$
5. $(1,(1,3),m)$	$m>0$ $m<0$ $m=0$	$n\neq m$ $n\neq mx$ $mx\neq m$	$n<m$
6. $(1,(1,3),mx)$	$mx>0$ $mx<0$ $mx=0$	$n\neq m$ $n\neq mx$ $mx\neq m$	$n<m$
7. $(2,(2,3),mx)$	$mx>0$ $mx<0$ $mx=0$		

3.2. Inclusion relation and complexity aspects

A constraint based criterion includes its correspondent structural criterion, even in the presence of infeasible paths, if we associate to every required element E the constraint "$C=$ true" so that every possible pair (E,"true") is required by the constraint criterion for each required element E.

We preserve the inclusion relation using the same set of constraints to derive all the constraint based criteria. For example, if we use the same set of constraints to derive all-constrained-nodes and all-constrained-uses, we preserve the inclusion relation: as all-uses includes all-edges, all constrained-uses will include all-constrained-edges (Fig. 2a). In the presence of infeasible paths, all-uses does not include all-edges; hence, all-constrained-uses criterion does not include all-constrained-edges, even using the same set of constraints. If different sets of constraints are used, constraint based criteria may be incomparable (Fig. 2b).

The complexity of a constraint-based criterion depends on the set of constraints used and on the complexity of the correspondent structural criterion. For example, the complexity of data flow based criteria is

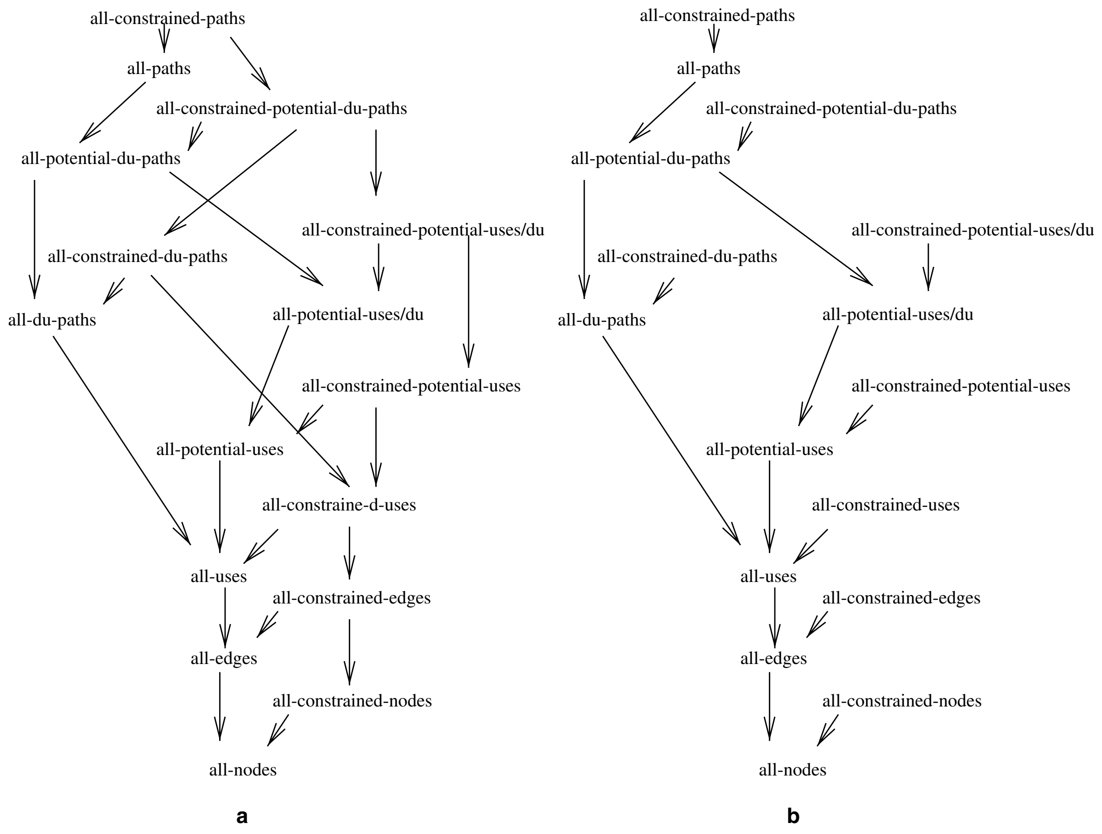


Fig. 2. Inclusion relations among criteria: (a) using the same set of constraints and (b) using different sets of constraints.

exponential with respect to the number of decisions statements in the program; however, the empirical complexity of data flow based criteria is linear, observed in experiments (Maldonado et al., 1992; Weyuker, 1988). The complexity of the constrained data flow based criterion is also exponential; some evidences of its empirical complexity are given in Section 4.

3.3. Implementation aspects

An extension to Poketool was implemented to support CBC. Basically two modules were proposed, *CB-Kernel* and *CB-Eval*. The relationship among these modules and the Poketool modules is presented in Fig. 3. The CBC application starts with *PokeIL* which

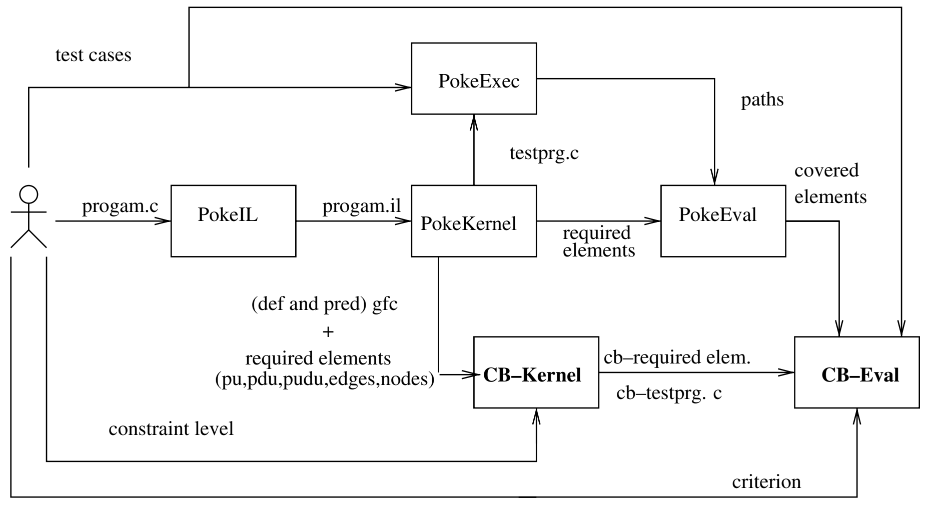


Fig. 3. Poketool extension.

transforms the program being tested (*program.c*) into an intermediate language program (*program.il*). *PokeKernel* module is responsible for generating all static information on the program. It generates three graphs: the CFG, the def-graph and the pred-graph. The last two contain, respectively, the variables definitions and the predicates associated to the CFG elements. After this, it determines the elements required by the supported structural criteria. It also generates the instrumented program (*testprg.c*), which produces a program trace when executed. *CB-Kernel* uses the information generated by *PokeKernel* and the level of constraints given by the user to determine the constrained elements and another instrumented program (*cb-testprg.c*). In this initial version, *CB-Kernel* generates constraints only applying the BRO technique (Level 3 in Table 3).

In a second step, the user submits the test cases to the *PokeExec* module which executes *testprg.c* and produces the executed paths. These paths are used by the *Poke-Eval* module to determine the covered structural elements. With this information the *CB-Eval* module checks the constraints that were satisfied by executing *cb-testprg.c* and, finally, provides a test set adequacy coverage with respect to the chosen constrained criterion. This criterion can be: all-constrained-potential-uses, all-constrained-potential-uses/du, all-constrained-potential-du paths, all-constrained-edges or all-constrained-nodes.

4. Applying Constraint Based Criteria

In this section, we report on two previous experiments conducted to evaluate Constraint Based Structural Criteria. We used all-potential-uses (PU) criterion, all-constrained-potential-uses (C-PU) criterion and mutation analysis (MA). The goals were to evaluate the applicability and the efficacy improvement, by comparing with other structural and fault-based criteria.

The first experiment, named *Experiment I* (Vergilio et al., 1997), used very simple programs and the CBC application was manually. The results encouraged us to implement the extension to Poketool, described in the last section. Then, we conducted another experiment, named *Experiment II* (Vergilio et al., 2001; Soares and Vergilio, 2004), using Unix programs that had been used by Wong et al. (1994) to evaluate the Mutation Analysis cost. In addition to this, Experiment II used an incremental approach to satisfy the criteria, by considering the order given by the inclusion relation.

In this section, we summarize those experiments; by considering the two works we reached new conclusions that are at the end of this section. Next, we present the description of the experiments and the analysis of the results with respect to three factors: cost, efficacy and strength. The strength analysis is accomplished by comparing PU and CPU, PU and MA, and, CPU and MA.

To make easy the comparison, we consider in this paper two measures:

- Coverage*

$$C^*(P,T) = \frac{E_c(P,T)}{E(P) - I_e(P)} \quad \text{and}$$

- Mutation Score

$$MS(P,T) = \frac{M_d(P,T)}{M(P) - M_e(P)}, \quad \text{where}$$

– P: program under test;
– T: a given test data set;
– $C^*(P,T)$: obtained coverage;[1]
– $E_c(P,T)$: number of covered elements;
– $E(P)$: number of required elements;
– $I_e(P)$: number of infeasible elements;
– $MS(P,T)$: mutation score;
– $M_d(P,T)$: number of dead mutants;
– $M(P)$: number of non anomalous generated mutants;
– $M_e(P)$: number of equivalent mutants.

In both experiments we manually determined infeasible elements and equivalent mutants.

4.1. Experiment I: literature programs

4.1.1. Description

(1) *Programs*. We used programs found in the literature: *max* (De Millo and Offutt, 1991), *whi80* (White and Cohen, 1980), *change*, *sum*, *triangle* (Ramamoorthy et al., 1976), and *voas* (Voas et al., 1991).
(2) *Generation of required elements*. The potential-associations were generated using Poketool. The constrained elements were manually generated using all the levels of Table 3.
(3) *Adequate test case set generation and determination of infeasible elements*. Two different testers carried out this task. One generated a test set to satisfy the PU criterion and the other a test set to satisfy the C-PU criterion. Table 5 shows the results from this step for both criteria. This table presents the

[1] We denoted *coverage** the coverage obtained considering C^* criteria, that eliminate the infeasible elements from the required ones (Frankl and Weyuker, 1986; Maldonado et al., 1992). For those criteria, it is always possible to achieve 100% coverage. Because of this, the information here is presented in a different way from one previously presented (Vergilio et al., 2001; Soares and Vergilio, 2004).

Table 5
Results from Experiment I

Program	Criterion	E(P)	I_e	C^*	No. test cases	Efficacy (%)	MS	Effec./no. test cases/
change	PU	4	0	1	2	0/1 (0)	0.9	2/2
	C-PU	27	4	1	8	1/1 (100)	0.98	7/8
max	PU	3	0	1	2	2/3 (66.6)	0.85	2/2
	C-PU	23	0	1	7	3/3 (100)	1	7/7
sum	PU	8	0	1	3	2/2 (100)	0.97	3/3
	C-PU	55	23	1	5	2/2 (100)	0.99	5/5
whi80	PU	20	1	1	6	1/1 (100)	0.97	6/6
	C-PU	150	17	1	16	1/1 (100)	1	14/16
Total	PU	35	1	1	13	5/7 (71.43)	0.9225	13/13
	C-PU	255	44	1	36	7/7 (100)	0.9925	33/36

number of required and infeasible elements and the coverage obtained for each program. For example, for *change*, the PU criterion requires 4 elements, all of them feasible and 100% coverage was obtained with 2 test cases. Using C-PU we obtained 27 required elements, 4 infeasible, and 8 test cases necessary for 100% coverage.For all programs $C^* = 1$, given that all the criteria were satisfied. This strategy can be named as criterion-based strategy. The set of test cases necessary to satisfy a criteria is named a *C*-adequate test set. Thus, only effective test cases are counted, that is, test cases that really kill a mutant or cover a required element.

(4) *Execution of incorrect versions using both sets*. The faults of the programs are described in Table 6. Each version has only one fault corresponding to a simple mutation; for instance, program *max* has three incorrect versions *max-1*, *max-2*, *max-3*—three faults were considered independently. The obtained efficacy is shown in Table 5. For *change* and *max*, the PU criterion did not reveal all the faults.

(5) *Comparison with MA*. To do this, we submitted all the versions to Proteum and calculated the MS, presented in Table 5. All the available operators of Proteum were used. For example, for program *change* the PU-adequate test set obtained a MS of 0.9, that is, it did not kill 10% of the non-equivalent mutants. All the test cases in the PU-adequate set were effective in Proteum, that is, each one killed at least one mutant.

Table 6
Fault descriptions of Experiment I

Program	Fault
max-1	$\max = m$ by $\max = abs(m)$
max-2	$n >= m$ by $n <= m$
max-3	$\max = m$ by $\max = n$
whi80-1	$if(k >= (i+2))$ by $if(k >= (i+1))$
change-1	$j = j+1$ by $j = j+l$
sum-1	$if(i < n)$ by $if(i <= n)$
sum-2	$pow(2,i)$ by $(pow(i,2))$
triangle-1	$if(c < 0 \| a > (b+c))$ by $if(c < 0 \| a > (b+c))$
voas-1	$d = b*b - 4*a*c$ by $d = b*b - 5*a*c$

4.1.2. Cost analysis

Table 5 shows that the C-PU criterion requires 7.28 times more elements than the PU criterion. However, the number of test cases required is only 2.76 times greater. It seems that the number of required test cases does not grow at the same rate of the number of required elements, due to the number of infeasible elements.

4.1.3. Efficacy analysis

The C-PU criterion revealed all the faults with an improvement of 28.57% over the PU criterion. This fact shows that the used constraints really describe the faults of the incorrect versions.

4.1.4. Comparison with MA

Comparing the PU and C-PU criterion against mutation analysis we observe that all the test data in the PU-adequate test sets are effective, that is, really contribute to kill mutants. However, a MS of 1 for the PU adequate test sets was not obtained for any of the programs. The C-PU criterion determines higher MSs than the ones determined by the PU adequate sets. The average of the C-PU adequate sets is 0.9925 and, for two programs, a MS equal to 1 was obtained. For all programs the MS for the C-PU adequate set is greater than the MS for the PU adequate set. The most significant improvement was for program *max*, a 15% increase in the MS. The smaller improvement was for program *sum*, from 0.97 to 0.99, a 2% increase; even though a quite high MS (0.97) had already been achieved.

We can also observe that for *change* and *whi80* some test cases in the C-PU adequate sets are not effective.

The fact that for *whi80* C-PU has a MS of 1, with 7 test cases out of 8, illustrates that it is possible to kill all the Proteum mutants, that is, to satisfy Mutation Analysis without satisfying the C-PU criterion.

The results provide evidence that it is worthwhile to further investigate CBC, since significant improvement in the efficacy is observed with an acceptable increase in the cost.

4.2. Experiment II: Unix programs

4.2.1. Description

(1) *Programs. cal, checkeq, comm, look, spline, tr, uniq*, Unix programs used by Wong et al. (1994) to evaluate the Mutation Analysis cost.
(2) *Generation of required elements.* The potential-associations were generated by the extension to Poketool. Because of this, only BRO constraints were used to derive the constrained elements (Level 3 of Table 3).
(3) *Generation of a PU adequate test case set and determination of infeasible elements.* We first generated test sets with the goal of covering the elements required by the PU criterion.
(4) *C-PU coverage analysis using the PU-adequate test sets and determination of infeasible elements.* The test sets generated in Step 3 were evaluated against the C-PU criterion obtaining constrained associations not yet covered. Additional test cases were generated and infeasible constrained-associations were determined. In this step the PU-adequate sets were used, given that the goal was to evaluate the strength of the PU and C-PU criteria. Because of this, this strategy is called incremental criterion-based; it considers the ordering of the criteria given by the inclusion relation.Table 7 presents the coverage obtained for both criteria for each program. For program *cal*, the PU criterion requires 242 associations (required elements), but 69 are infeasible; 12 test cases were needed to satisfy the PU criterion. The other row in the table shows the results for the C-PU criterion. To satisfy the C-PU criterion we had to add 5 test cases to the PU-adequate test case set (column No. Test Cases of Table 7).
(5) *Execution of the incorrect versions using both sets.* The same incorrect versions derived by Wong et al. (1994) were used, each one with only a fault. The PU and C-PU adequate test sets from the criterion-based generation were used to execute the incorrect versions of each program. Table 7 shows the results of the efficacy analysis. For program *cal*, the PU and C-PU criteria revealed, respectively, 19 and 20 faults, out of 20.
(6) *Comparison with MA*. The PU and C-PU adequate test sets were submitted to Proteum. Table 7 gives the obtained MSs and the number of test cases in the PU and C-PU adequate test sets that are MA adequate, that is, that really killed a mutant.

4.2.2. Cost analysis

Table 7 shows that the C-PU criterion requires 2.17 times more elements than the PU criterion; however only 1.39 times more test cases are required. These results show the applicability of the all-constrained potential-uses criterion and are similar to the results of

Table 7
Results from Experiment II

Program	Criterion	$E(P)$	I_e	C^*	No. test cases	Efficacy	MS	Effec./No. test cases
cal	PU	242	69	1	12	19/20	0.9641	12/12
	C-PU	488	179	1	17	20/20	0.9877	17/17
checkeq	PU	582	114	1	76	20/22	0.9440	42/76
	C-PU	1539	471	1	164	20/22	0.9685	54/164
comm	PU	427	150	1	68	19/19	0.9257	30/68
	C-PU	884	383	1	74	19/19	0.9523	33/74
look	PU	524	87	1	34	20/20	0.9321	21/34
	C-PU	1045	247	1	40	20/20	0.9349	22/40
spline	PU	999	206	1	57	16/20	0.9554	44/57
	C-PU	2352	956	1	63	16/20	0.9577	47/63
tr	PU	1527	933	1	30	19/19	0.7824	24/30
	C-PU	3040	2014	1	35	19/19	0.8626	36/35
uniq	PU	262	32	1	36	18/19	0.9808	24/36
	C-PU	552	136	1	42	18/19	0.9856	27/42
Total PU		4563	1591	100	313	146/157 (92)	0.9298	173/327
Total C-PU		9900	4386	100	435	148/157 (94.27)	0.9501	226/454

Experiment I: the number of test cases does not grow at the same rate of the number of required associations.

Experiment II used the incremental criterion based strategy. First, we generated the PU-adequate sets and, after, the same tester added test cases to satisfy the C-PU criterion. Because of this, another aspect, related to strength analysis, could be explored in this experiment: the difficulty of satisfying C-PU given that the PU criterion was satisfied.

We observe that for some programs the number of additional test cases is very low. For *cal* and *checkeq*, this number is about 50% greater. This fact is related to the kind of constraints used (BRO); *checkeq* has a great number of composite predicates.

Once the tester had generated test cases to satisfy the PU criterion and identified the infeasible associations little additional effort was necessary to generate test cases for the C-PU criterion and identify infeasible constrained-associations. This happens because the tester had already acquired knowledge about the program. Another question related to the applicability of the C-PU criterion is time during the adequacy analysis, performed by Poketool extension. In average, three times more hours were spent with the C-PU criterion.

4.2.3. Efficacy analysis

The C-PU criterion's efficacy was greater than the PU criterion's efficacy (Table 7). The C-PU criterion revealed 2.27% more faults than the PU criterion; 3.13% more when we consider only domain faults and no increase when we consider data structure faults. This fact shows the influence of the constraints used to derive the constrained elements. BRO constraints describe faults in predicates, which are associated to domain faults. The probability of revealing this kind of faults was greater. However, constraints describing data structure faults were not used and there is no difference between the PU and C-PU criteria when we consider this kind of faults.

4.2.4. Comparison with MA

Concerning the strength analysis using mutation testing, the C-PU criterion achieved a mean increase of 3%. The most significant improvement was for *tr*; An increase in the MS of 7% was achieved with only five additional test cases. Program *tr* has also the greatest percentage of infeasible elements and equivalent mutants. This characteristic should be further explored. Although the C-PU test sets kill a greater number of mutants than the PU test set for all programs, there is no significant difference in the MS for programs *look, spline* and *uniq*. A possible explanation for this is the kind of constraints used (BRO). Consider the program *uniq*, the C-PU set killed only seven mutants more than the PU set, meaning a very small difference in the score. However, if we analyze only the mutants generated by the operator ORRN (Relational Operator Replacement) we observe an increase of 5%. The C-PU set killed all mutants in this mutation class (see Table 8). The remaining alive mutants belong to other classes of mutation, not described by the BRO constraints.

Table 8
MS of PU and C-PU test sets for relation operator mutation

Criterion	Number of mutants			MS
	Total	Infeasible	Dead	
PU	95	13	77	0.95
C-PU	95	13	82	1

Another important point is observed considering the number of effective test cases. In spite of high PU and C-PU scores, only 50% of the PU and C-PU adequate test cases are effective. This fact points out the difficulty of satisfying these criteria.

4.3. Concluding remarks—Experiments I and II

In both experiments we observe the applicability of CBC. The number of required test cases does not grow at the same rate of the number of required elements. The cost, given by the number of test cases, and efficacy are clearly influenced by the number of constraints used. In Experiment I, using 3 levels of constraints, the number of required elements and test cases required by the C-PU criterion are, respectively, 7 and 2 times greater than the required by the PU criteria. The number of required elements and test cases are 2 and 1.4 times greater.

The experiments show an increase in the efficacy: in both experiments the efficacy of the C-PU set is greater. This difference is 28.57% in Experiment I and 2.27% in Experiment II. These differences in the number of test cases and in the efficacy are due to the used constraints. In Experiment II, we used only one level of constraints and in Experiment I, we used three.

It should also be noticed the importance of using different levels of constraints; this can reduce the number of required elements and can be associated to the characteristics and desired reliability of the program under test. Another point to be considered is that the incremental approach can reduce the number of necessary test cases. This should be explored in future experiments.

Concerning the strength analysis, the C-PU adequate test sets obtained a MS greater than the PU-adequate sets did, with an improvement around 3%. We notice that the kind of constraints used also influences the score and the class of dead mutants. We have evidences that it is easier to satisfy Mutation Analysis given that the C-PU criterion is satisfied than that the PU criterion is. We view CBC as intermediate criteria between the data flow based criteria and mutation testing in terms of cost and efficacy, although they are incomparable.

5. Experiment III

This section describes Experiment III. It has the goal of evaluating the use of random test cases, instead of criterion-based ones. Because random generation is more practical and requires a low effort, it is interesting to evaluate the strength and efficacy of the C-PU and PU criteria using this strategy.

(1) *Programs*. The same programs of Experiment II.
(2) *Generation of required elements*. To allow comparison with the criterion-based strategy, this step was conducted exactly as in Experiment II, because we want a comparison with the incremental strategy.
(3) *Submission of test pools, randomly generated by* Wong et al. (1994). The test pools available for each program were submitted to Poketool and its extension and the coverage for the PU and CPU criteria was obtained. During this step, we did not generate additional test cases; hence, except for program *cal*, the criteria were not satisfied. Table 9 provides the PU and C-PU adequacy of the pools, the number of effective test cases (i.e., the ones that really contribute to increase the coverage, in a given execution order) and the pool size for each program. For instance, for program *uniq*, coverages of 0.934 and 0.8877 were obtained, respectively, for the PU and C-PU criteria. Only 31 and 32 test cases, out of 431, were effective, respectively, for PU and C-PU. It should also be observed that the number of random effective test cases was smaller than the size of the criterion-based adequate sets.
(4) *Execution of the incorrect versions using both random sets*. The same incorrect versions from Experiment II were used. The incorrect versions were executed only with effective test cases from the random pools. Results from efficacy are shown in Table 9.
(5) *Submission of the random pools to Proteum*. Only the effective random test cases are used to perform the strength analysis, using all the operators of Proteum. The obtained MS, as well the number of test cases necessary to kill the mutants are also in Table 9.

5.1. Cost analysis

Comparing the criterion-based and random test sets, we observe in Table 7, that only for program *cal* the random test set satisfied the criteria, that is, all the feasible associations were covered. For all other programs the criteria were not satisfied. For the PU criterion, 2402 associations were covered by random tests (80.82%) and 570 (19.18%) of feasible associations were not covered. Similar results were obtained for C-PU: respectively 4328 (78.49%) and 1186 (21.51%). If we had used the number of random test cases equal to the size of those adequate test sets, instead of using the complete pools, the results obtained with random test cases would be worse than the ones presented herein.

An explanation for the low score obtained by the random pools is that some of the elements required by the

Table 9
Results from Experiment III

Program	Criterion	C*	No. test cases	Efficacy	MS	Effec./pool size/
cal	PU	1	7/162	18/20	89.11	7/7
	CPU	1	17/162	19/20	98.18	17/17
checkeq	PU	0.7372	13/166	19/22	84.93	10/13
	CPU	0.6142	13/166	19/22	84.93	10/13
comm	PU	0.9314	58/754	19/19	94.26	31/58
	CPU	0.9262	59/754	19/19	94.26	31/59
look	PU	0.8897	29/193	20/20	89.08	21/29
	CPU	0.8905	29/193	20/20	89.08	21/29
spline	PU	0.8310	28/700	13/20	89.18	19/28
	CPU	0.8497	36/700	13/20	89.31	21/36
tr	PU	0.6143	9/870	19/19	68.28	7/8
	CPU	0.6109	11/870	19/19	68.36	9/11
uniq	PU	0.9347	31/431	17/19	90.08	17/31
	CPU	0.8877	32/431	17/19	90.08	17/32
Total PU		0.8082	175/3062	139/157 (88.53)	0.8622	112/179
Total C-PU		0.7843	194/3062	141/157 (89.80)	0.8765	126/198

Table 10
Coverage analysis for *main* function of *comm*

Criterion	E (*main*)	I_e	Criterion-based		Random	
			E_c (*main*)	No. tests	E_c (*main*)	No. tests
PU	396	147	249	67	230	56
C-PU	820	372	448	71	416	57

PU criterion and mainly for the C-PU criterion are not easily covered. In other words, the probability of generating randomly a test case to exercise those elements is low, since there would be few test cases (in many cases only one) in the partitions related to those elements. For instance, consider the function *main* of the *comm* program (Table 10). For this function, the PU criterion requires 396 elements and 249 (62%) are feasible. A set of 56 random test cases covers 230 elements. However, to execute the remaining elements (19 elements) it is necessary to execute 14 more criterion-based test cases; 11 of the 19 remaining elements require very specific test cases. At the end, a total of 70 test cases are necessary, similar to the criterion-based strategy. This fact should be considered in the use of a random strategy.

5.2. Efficacy analysis

Observe in Tables 7 and 9 that the criterion-based generation strategy revealed 7 more faults than the random strategy for both criteria. The C-PU's efficacy when using random strategy is also greater than the PU's efficacy. Constraints contribute to increase the efficacy of the structural criterion independently of the test generation strategy. In both cases, the C-PU criterion revealed 2 more faults than the PU criterion. It should be observed that even though the adequacy of the pools were not very high, being smaller for all the programs, except for *cal*, the random test case sets achieved the same efficacy achieved by the criterion-based strategy for 4 of the programs. This is a good result for the random strategy.

5.3. Comparison with MA

When we consider the random pools, the C-PU sets obtained an increase of 1.43% and there is no difference between PU and C-PU scores for the programs *checkeq, comm, look* and *uniq*. In addition to, if we compare with criterion-based sets, we observe that, for all programs, the random MS is smaller, even for program *cal* that was satisfied by the random set. Maybe, this is the reason that for *cal*, the random C-PU set presents the most significant improvement in the score (8%). It should be pointed out that the random sets are not adequate to the criteria. This fact can also explain the same PU and C-PU scores for most programs.

5.4. Application of the testing criteria

During the experiments described in this paper we observe some testing aspects that are now being considered for application of the studied criteria and strategies that can be accomplished by using the following steps.

(1) Generate a set of random test data T_r. This set can be used as the initial test set for the application of the criteria.
(2) Select the criteria. The application of all criteria mentioned in this paper is not necessary. The selection of the criteria should be done considering the costs and desired reliability. Soares and Vergilio (2004) propose the use of an incremental strategy, that considers the inclusion relation of criteria, such as the one applied in Experiment II. In spite of structural and MA being incomparable, we observed in our experiments that there is an empirical relation between them. MA is a stronger and more expensive criterion. It is easier to satisfy MA given that the C-PU criterion was satisfied than the PU criterion was. In this way, we apply the criteria in the order given by the inclusion relation:
    - Control Flow Based Criteria: all-node, all-edges,...;
    - Constrained Control Flow Based Criteria: all-constrained-nodes, all-constrained-edges,...;
    - Data Flow Based Criteria: all-uses, all-potential-uses,...;
    - Constrained Data Flow Based Criteria: all-constrained-uses, all-constrained-potential-uses,...;
(3) Obtain the coverage or score for T_r. The evaluation of the adequacy of the test sets must be done by using a testing tool, such as Poketool or Proteum.
(4) If the coverage (or score) is compatible with the desired reliability level and chosen criterion, we can stop testing.
(5) Apply the criterion based strategy, by generating test data set T_a capable of covering the remaining elements. We know, by our experience, that the effort of increasing coverage above 80% is very high and, as explained in the last section, this can make the difference in the efficacy, because parts of the program that are not usually reached can be tested. However, there are in the literature works (and tools) that can be used to aim at the automation of this step (Bueno and Jino, 2001; Korel, 1990; Michael et al., 2001; Peres et al., 2000) and, consequently, to decrease effort and costs. In spite of this, the tester needs to determine infeasible elements or equivalent mutants in many cases.

(6) Obtain the final coverage using T, such as $T = T_i + T_a$. If the testing requirements are satisfied, stop testing.
(7) If we are applying a set of criteria and using the incremental approach, we take T as the new initial test set and it will be easier to satisfy the next criterion in the hierarchy.

6. Conclusions

Constrained Based Criteria consider positive aspects from different testing generation strategies. They associate a constraint to an element required by a structural criterion describing necessary conditions to reveal typical faults. It requires constrained elements (E, C) that are covered by a test case that executes a path including the structural element E and that satisfies the constraint C. They are motivated by an experiment with different test case generation strategies. In most cases, necessary conditions are not sufficient to reveal the faults and the sufficient ones are, in general, conditions to execute a particular path in the program.

CBC's aspects of complexity and inclusion relation amongst criteria, as well, of implementation aspects are discussed. An extension to Poketool enabled the evaluation of a test set according to the all-constrained-potential-uses criterion. We also revisited results from two experiments with the all-potential-uses and all-constrained potential-uses criteria and added an experiment with random strategy.

Experiment I used simple programs and three levels of constraints. A criterion-based strategy to satisfy the criteria was used. Experiment II uses more complex programs and the constraints were automatic generated using only one level (BRO constraints). An incremental criterion-based strategy was used to allow analysis on the PU and C-PU criteria strengths.

Similar results were obtained with both experiments. They show that CBC is applicable. The number of test cases does not grow at the same rate of the number of required elements. In both experiments the C-PU's efficacy was greater. When we compare the results with MA, we obtain an example that shows that structural testing criteria and MA are incomparable. However, in both experiments and for most programs, we observe that MA is more difficult to satisfy, and that it is easier to satisfy MA given that the C-PU criterion was satisfied than if the PU criterion was.

This points out that CBC are intermediate criteria between the correspondent structural criteria and MA. Satisfying the C-PU criterion takes a longer time than that for satisfying the the PU criterion. With respect to difficulty of satisfying and determining infeasible elements, we observed, also in Experiment II that, once the tester had generated test cases to satisfy the PU criterion and had identified the feasible associations, little additional effort was necessary to generate test cases for the C-PU criterion and to identify infeasible constrained associations. That is because the tester had already acquired knowledge about the program when he or she applied the PU criterion and determined some infeasible patterns. This fact is considered by the testing strategy presented in Section 5.

In Experiment II, the C-PU criterion required a lower number of required elements and necessary test cases than in Experiment I. The efficacy and MS was also lower. The reason for this is the kind of constraints used in both experiments or in the used strategy. These facts point out the relevance of selecting constraints to be associated to an element. This task is very important, because they will influence the efficacy and the costs. For example, in Experiment II, we used constraints that describe faults in predicates. This influenced the efficacy. There is no difference in the PU and C-PU efficacies when we consider faults in data structures.

Constraints must be chosen describing faults complementary to those revealed by structural testing, such as faults in data structures. In addition to this, other factors must be considered: number of infeasible elements or a selective Constraint Based Criteria. Applying selective Constraint Based Criteria means that only specific constraints are used. Factors as the domain application and required reliability should determine the constraint selection. This may reduce costs, by decreasing the number of required elements and the number of test cases.

Experiment III shows that criterion-based strategy is more expensive and requires a lot of effort for determining infeasible elements. But, as a side benefit, the knowledge acquired during this phase can be used during maintenance and debugging. On the other hand, random generation of test cases does not guarantee the criterion's satisfaction. We observe in our experiments that only for one program the random test data sets satisfied the criterion (the mean coverage obtained was around 80%). Exercising the remaining elements may result a greater efficacy.

However, the cost and efficacy, as well the obtained mean MS, point out that the use of random strategy does not invalidate the results with respect to the C-PU and PU criteria obtained in Experiment I and II. Therefore, the combination of random and incremental criterion-based strategy, such as the strategy introduced in Section 5, is very promising. It can reduce effort in the testing activity. Additional test cases would be derived after its application to guarantee the criterion satisfaction. In systems where a high reliability is required, a criterion must be satisfied. In these cases, stronger criteria must be applied and the combination of both strategies seems to be very advantageous. This idea should be explored in further work.

References

Bueno, P., Jino, M., 2001. Automatic test data generation for program paths using genetic algorithms. In: 13th International Conference on Software Engineering and Knowledge Engineering—SEKE, pp. 2–9.

Chaim, M., 1991. POKE-TOOL—Uma Ferramenta para Suporte ao Teste Estrutural de Programas Baseado em Análise de Fluxo de Dados. Master Thesis, DCA/FEEC/Unicamp, Campinas—SP, Brazil (in Portuguese).

Clarke, L., 1976. A system to generate test data and symbolically execute programs. IEEE Transactions on Software Engineering SE-2 (3), 215–222.

De Millo, R., Lipton, R., Sayward, F., 1978. Hints on test data selection: help for the practicing programmer. IEEE Computer C-11 (April), 34–41.

De Millo, R., Offutt, A., 1991. Constraint-based automatic test data generation. IEEE Transactions on Software Engineering SE-17 (9), 900–910.

Delamaro, M.E., Maldonado, J. 1996. A tool for the assessment of test adequacy for C programs. In: Proceedings of the Conference on Performability in Computing Systems, East Brunswick, New Jersey, USA, pp. 79–95

Duran, J., Ntafos, S., 1984. An evaluation of random testing. IEEE Transactions on Software Engineering SE-10 (4), 438–444.

Frankl, F., Weyuker, E., 1986. Data flow testing in the presence of unexecutable paths. In: Proceedings of the Workshop on Software Testing'. Computer Science Press, Banff, Canada, pp. 4–13.

Frankl, F., Weyuker, E., 1988. An applicable family of data flow testing criteria. IEEE Transactions on Software Engineering SE-14 (10), 1483–1498.

Frankl, P., Weyuker, E., 1993. A formal analysis of the fault-detecting ability of testing methods. IEEE Transactions on Software Engineering SE-19 (3), 202–213.

Herman, W., 1976. Flow analysis approach to program testing. The Australian Computer Journal 8 (3), 259–266.

Howden, W., 1976. Reliability of the path analysis testing strategies. IEEE Transactions on Software Engineering SE-2 (3), 208–215.

Howden, W., 1977. Symbolic testing and dissect symbolic evaluation system. IEEE Transactions on Software Engineering SE-3 (4), 266–278.

Korel, B., 1990. Automated software test data generation. IEEE Transactions on Software Engineering SE-16 (8), 870–879.

Laski, J., Korel, B., 1983. A data flow oriented program testing strategy. IEEE Transactions on Software Engineering SE-9 (3), 347–354.

Maldonado, J., Chaim, M., Jino, M., 1992. Bridging the gap in the presence of infeasible paths: potential uses testing criteria. In: XII International Conference of the Chilean Computer Science Society, Santiago, Chile, pp. 323–340.

McCabe, T., 1976. A software complexity measure. IEEE Transactions on Software Engineering SE-2 (4), 308–320.

Michael, C., McGraw, G., Schatz, M., 2001. Generating software test data by evolution. IEEE Transactions on Software Engineering 27 (12), 1085–1110.

Murril, B., Morell, L., Olimpiew, E., 2002. A perturbation-based testing strategy. In: Eighth International Conference on Engineering of Complex Systems. IEEE Press, pp. 145–152.

Ntafos, S., 1984. On required element testing. IEEE Transactions on Software Engineering SE-10 (6), 795–803.

Peres, L., Vergilio, S., Jino, M., Maldonado, J., 2000. Path selection strategies in the context of software testing criteria. In: 1st IEEE Latin American Test Workshop, Rio de Janeiro, pp. 222–227.

Ramamoorthy, C., Siu-Bun, F.H., Chen, W., 1976. On automated generation of program test data. IEEE Transactions on Software Engineering SE-2 (4), 293–300.

Rapps, S., Weyuker, E., 1985. Selecting software test data using data flow information. IEEE Transactions on Software Engineering SE-11 (4), 367–375.

Soares, I., Vergilio, S., 2004. Mutation analysis and constraint based criteria: results from an empirical evaluation in the context of software testing. Journal of Electronic Testing: Theory and Applications-JETTA (20), 439–445.

Tai, K., 1993. Predicate-based test generation for computer programs. In: Proceedings of International Conference on Software Engineering. IEEE Press, pp. 267–276.

Ural, H., Yang, B., 1988. A structural test selection criterion. Information Processing Letters 28 (3), 157–163.

Vergilio, S., Maldonado, J., Jino, M., 1997. Constraint based selection of test sets to satisfy structural software testing criteria. In: XVII International Conference of the Chilean Computer Science Society. IEEE-Press, Los Alamitos, CA.

Vergilio, S., Maldonado, J., Jino, M., 2001. Constraint based criteria: An approach for test case selection in the structural testing. Journal of Electronic Testing: Theory and Applications—JETTA 17 (2), 175–183.

Voas, J., Morell, L., Miller, K., 1991. Predicting where faults can hide from testing. IEEE Software, 41–47.

Weyuker, E., 1988. An empirical study of the complexity of data flow testing. In: Proceedings of the Second Workshop on Software Testing, Verification and Analysis. Computer Science Press, Banff, Canada, pp. 188–195.

Weyuker, E., 1990. The cost of data flow testing: An empirical study. IEEE Transactions on Software Engineering SE-16 (2), 121–128.

White, L., Cohen, E., 1980. A domain strategy for computer program testing. IEEE Transactions on Software Engineering SE-6 (3), 247–257.

Wong, W., Mathur, A., Maldonado, J., 1994. Mutation versus all-uses: an empirical evaluation of cost, strength and effectiveness. In: Software Quality and Productivity—Theory, Practice, Education and Training, Hong Kong.

Silvia Regina Vergilio received the the MS (1991) and DS (1997) degrees from University of Campinas, UNICAMP, Brazil. She is currently at the Computer Science Department at the Federal University of Paraná, where she has been a faculty member since 1993. She has been involved in several projects and her research interests are in the areas of Genetic Programming and Software Engineering as software testing, software quality and software metrics.

José Carlos Maldonado received his BS in Electrical Engineering/Electronics in 1978 from the University of São Paulo (USP), Brazil, his MS in Telecommunications/Digital Systems in 1983 from the National Space Research Institute (INPE), Brazil, and his D.S. 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 Mathematical Sciences and Computing at the University of São Paulo (ICMC-USP) where he is currently a faculty member. His research interests are in the areas of Software Quality and Software Testing and Reliability. He is a member of the SBC (Brazilian Computer Society) and of the IEEE Computer Society.

Mario Jino received the B.S. degree in electronic engineering from Instituto Tecnológico de Aeronáutica (ITA), Brazil, in 1967, the MSc in electrical engineering from the State University of Campinas (UNICAMP), Brazil, in 1974, and the Ph.D. degree in computer sciences from the University of Illinois, Urbana-Champaign, Illinois, USA, in 1978. From 1967 to 1971 he was a researcher at the Brazilian Space Agency (INPE), where he was involved in the Remote Sensing

Project. Since 1971 he is a teacher at UNICAMP where he is presently an associate professor in the Department of Computer Engineering and Industrial Automation of the School of Electrical and Computer Engineering. His current research interests include: software quality; software testing, debugging and maintenance; object orientation; and software metrics. He has been involved in several technological development projects with government and private entities, and has served as technical advisor/referee for Brazilian research funding agencies as well as in several scientific conferences and symposiums. He is a member of the IEEE Computer Society, the IEEE, the ACM, SBA (the Brazilian Society of Automatic Control), and SBC (the Brazilian Computer Society).

Inali W. Soares received her BS (1998) from State University of Paraná (Unicentro) and the MS (2000) from Federal University of Paraná (UFPR). She is currently at the Computer Science Department at State University of Paraná, (Unicentro), Brazil. She has research interests in the areas of Software Engineering, specifically in software testing.