Geração Automática de Dados e Tratamento de Não Executabilidade no Teste Estrutural de Software

Paulo Marcos Siqueira Bueno *
DCA/FEEC/Unicamp - Campinas
CP: 6101, 13083-970, Brazil
bueno@dca.fee.unicamp.br

Mario Jino
DCA/FEEC/Unicamp - Campinas
CP: 6101, 13083-970, Brazil
jino@dca.fee.unicamp.br

Abstract

A tool and techniques are presented for test data generation and infeasibility identification in structural software testing technique. The tool is based on: the Dynamic Technique*; search using* Genetic Algorithms*; and reuse of solutions through* Case-Based Reasoning*. The objective is to automatically generate input data which execute complete paths in a program and identify path infeasibility when this is the case; this is done through the* Potential Infeasibility Dynamic Identification Heuristic *proposed. An experiment shows the validity of the developed solutions and the benefit of using the tool. Results attained indicate that, despite the general undecidability of the problems, partial solutions can be useful for software testing practice.*

Keywords*: test data generation, dynamic technique, path infeasibility, genetic algorithms, case-based reasoning.*

1 Introdução

Critérios de teste estrutural de software estabelecem componentes estruturais do programa (referidos como elementos requeridos) a serem exercitados pelo testador. Satisfazer um critério significa exercitar todos os componentes estruturais requeridos pela aplicação do critério ao programa em teste. A tarefa de selecionar dados de teste para exercitar componentes estruturais é extremamente complexa, pois requer a cuidadosa análise do código do programa, esforço mental e conhecimentos de aspectos conceituais do critério utilizado. Portanto, a geração de dados de teste é uma atividade de difícil realização e que tende a ter um alto impacto no custo final do teste. A automação desta atividade é altamente desejável; entretanto, é inviável em casos gerais. Conforme destaca Clarke [4], este problema é equivalente ao "problema da parada", reconhecidamente indecidível.

É possível, quando da aplicação de critérios de teste estrutural, caracterizar um conjunto de caminhos completos no programa que, quando executados, fazem com que os elementos requeridos pelos critérios sejam exercitados. Uma estratégia válida, portanto, é dividir o processo de geração de dados de teste em duas etapas: *[i]* seleção de caminhos completos no programa que exercitem todos os elementos requeridos pelo critério de teste; e, *[ii]* geração de dados de entrada que executem todos os caminhos selecionados. Nesta estratégia tem-se a etapa *[i]* dependente do critério, enquanto a *[ii]* é genérica. Isto significa que um gerador automático de dados de teste que executem caminhos completos do programa pode ser utilizado para diversos critérios de teste estrutural.

*Trabalho desenvolvido com o apoio financeiro da FAPESP, processo 97/07004-7.

A seleção de caminhos para satisfação de critérios de teste estrutural (etapa [i]) vem sendo abordada por vários autores [17, 24, 6, 2, 20]. Para a geração de dados de entrada que executem caminhos no programa (etapa [ii]) duas abordagens destacam-se: a execução simbólica [4, 11, 21] e a técnica dinâmica de geração de dados de teste [14, 5, 8].

Os trabalhos que utilizam execução simbólica buscam representar simbolicamente as condições para a execução de um dado caminho em função das variáveis de entrada do programa. Esta representação simbólica é utilizada por algoritmos que tentam buscar soluções as quais, satisfazendo as condições, provoquem a execução do caminho pretendido. Esta abordagem apresenta algumas restrições em relação ao tratamento de laços, variáveis compostas e chamadas de funções ou procedimentos.

A técnica dinâmica é baseada na execução real do programa, em métodos de minimização de funções e análise dinâmica de fluxo de dados. Dados reais são atribuídos às variáveis de entrada e o fluxo de execução do programa é monitorado. Se um ramo indesejado foi tomado (isto é, o fluxo de execução desviou do pretendido), métodos de minimização de funções são utilizados para determinar valores para as variáveis de entrada para os quais o ramo desejado seria tomado.

Uma questão complexa no teste estrutural de software diz respeito à existência de caminhos não executáveis, caminhos para os quais não existe conjunto de valores de entrada do programa que os executem. Determinar automaticamente estes caminhos é uma questão indecidível. Tratamentos encontrados são parciais, garantindo esta identificação apenas em casos particulares [7, 24]. Comumente são utilizadas execução simbólica; prova de teoremas e análise estática de fluxo de dados.

Este trabalho apresenta uma ferramenta e técnicas para automação da geração de dados e identificação de não executabilidade para a técnica estrutural de teste de software. São utilizados: a técnica dinâmica; a busca baseada em Algoritmos Genéticos; e o reuso de soluções através do Raciocínio Baseado em Casos. É proposta uma heurística de natureza dinâmica que, através do monitoramento do progresso da busca realizada pelo Algoritmo Genético, identifica a provável não executabilidade do caminho pretendido. As soluções foram validadas através de um experimento que apresentou indícios consistentes da pertinência das mesmas e da utilidade da ferramenta proposta.

Além da concepção e implementação de uma ferramenta extremamente útil à prática do teste estrutural de software, podem ser destacadas as seguintes contribuições deste trabalho: a utilização de Algoritmos Genéticos para a geração da dados de teste para executar caminhos em programas; a utilização do paradigma do Raciocínio Baseado em Casos visando permitir o reuso de informações passadas na geração de dados de teste e a definição de uma heurística de caráter dinâmico para o tratamento da questão da não executabilidade.

A seção seguinte descreve conceitos básicos necessários ao entendimento do trabalho; a Seção 3 cita trabalhos relacionados existentes na literatura; na Seção 4 tem-se uma visão geral da feramenta e das técnicas propostas; a Seção 5 introduz conceitos associados aos Algoritmos Genéticos e detalha a sua utilização no modelo proposto; a Seção 6 descreve o modelo de instrumentação do programa em teste, necessária ao monitoramento da execução; a Seção 7 aborda o reuso de soluções passadas; a Seção 8 mostra o tratamento dinâmico para a questão da não executabilidade; a Seção 9 descreve os experimentos realizados e sintetiza os resultados obtidos; a Seção 10 apresenta, finalmente, as conclusões e contribuições deste trabalho e aponta desenvolvimentos futuros promissores.

2 Conceitos Básicos

A estrutura de um programa P pode ser representada por um grafo dirigido $G = (N, E, s, e)$, onde N é um conjunto de nós, E um conjunto de arcos, s é o único nó de entrada e e o único nó de saída. Um arco (n_i, n_j) corresponde a uma possível transferência de controle entre os nós i e j [16]. Um arco (n_i, n_j) é chamado de *ramo* se o último comando de n_i é um comando de seleção ou de repetição. A cada *ramo* pode ser associado um predicado denominado *predicado de ramo*.[1]

Um *sub-caminho* é uma seqüência de nós $(n_1, n_2, ..., n_k)$, $k \geq 2$, tal que existe um arco de n_i para n_{i+1}, $1 \leq i \leq (k-1)$ (isto é, $(n_i, n_{i+1}) \in E$).

Um *caminho* ou *caminho completo* é um sub-caminho onde o primeiro nó é o nó de entrada e o último nó é o nó de saída do grafo G [2]. Além disso, um *caminho pretendido* é um caminho o qual deseja-se executar.

Uma *variável de entrada* x_i para o programa P é uma variável que aparece em um comando de entrada; um parâmetro de entrada; ou ainda, uma variável de escopo global utilizada no programa. Variáveis de entrada podem ser dos diferentes tipos tratados pela linguagem.

$I = (x_1, x_2, ..., x_n)$ é o vetor de variáveis de entrada do programa P.

O *domínio* D_{xi} da variável de entrada x_i é o conjunto de todos os valores que x_i pode assumir.

O *domínio de entrada* D do programa P é o produto cartesiano $D = D_{x1} \times D_{x2} \times ... \times D_{xn}$, onde D_{xi} é o domínio da variável de entrada x_i.

Um único ponto x no espaço de entrada n-dimensional D, $x \in D$ é referido como um *dado de entrada* ou *dado de teste*.

Um caminho é *executável* se existe algum dado de entrada $x \in D$ para o qual o caminho é percorrido durante a execução do programa; caso contrário, tal caminho é *não executável*.

Considerando $CP = (n_1, n_2, ..., n_k)$ um caminho pretendido, o objetivo é descobrir algum dado de entrada $x \in D$ que provoque a execução de CP, ou identificar a não executabilidade deste caminho, se for o caso.

3 Trabalhos Relacionados

Trabalhos utilizando a técnica dinâmica de geração de dados de teste fazem uma avaliação real dos predicados usando para tanto a execução do programa. A partir desta avaliação são usados métodos de minimização, buscando os valores que executem o caminho pretendido. Essencialmente, o próprio programa fornece um meio eficiente de avaliar numericamente o erro nos predicados que fazem com que o caminho executado desvie do pretendido [19].

Na abordagem de Miller e outros [19] os comandos de decisão do programa são substituídos por "restrições de caminho" que medem o quão próximo cada decisão está de ser satisfeita. O programa toma a forma de uma única linha de código com uma série de restrições cujos erros devem ser minimizados para que o caminho seja executado. São utilizados algoritmos de busca direta e consideradas variáveis do tipo real.

---

[1] Em diversos trabalhos os termos *arcos* e *ramos* são utilizadados como sinônimos. Adotou-se neste trabalho a nomeclatura definida por Korel [14], na qual *ramos* são *arcos* aos quais estão associados predicados.

[2] Neste trabalho utilizam-se os termos *caminho* e *caminho completo* como sinônimos. Sempre que for necessário fazer menção a um caminho "não completo" será utilizado o termo *sub-caminho*.

Korel [14] divide o problema de executar um caminho em diversos sub-problemas, onde cada um consiste em satisfazer um predicado do caminho. Cada sub-problema é resolvido utilizando-se uma busca direta para minimizar o valor de funções de erro associadas aos predicados a fim de satisfazê-los. Esta técnica foi estendida por Ferguson de forma a incorporar a seleção de caminhos no processo de geração de dados, utilizando para tanto informações sobre dependência de dados [5].

Gallagher e outros [8] criaram uma função que reflete o erro que provoca o desvio do caminho pretendido considerando conjuntamente todos os predicados já atingidos. É empregada uma técnica de otimização "quasi-Newton" para minimizar esta função visando executar o caminho.

Gupta e outros [10] sugerem um método no qual se o caminho desejado não é executado; a entrada é refinada de forma iterativa. São computadas duas representações de cada predicado: o "slice", conjunto de comandos que influenciam o predicado através do caminho desejado; e a representação linear aritmética do predicado em função das variáveis de entrada. Os "slices" são executados e é computada a representação linear aritmética respectiva. Estas duas representações são utilizadas para refinar iterativamente os valores de entrada iniciais a fim de obter a entrada desejada.

Em nenhuma das abordagens é garantido que os dados de entrada serão obtidos mesmo que o caminho seja executável. A não executabilidade de caminhos é tratada apenas em [10]; os demais autores que utilizam técnica dinâmica não tratam diretamente esta questão.

4 Técnicas e Ferramenta Propostas: Visão Geral

A ferramenta apresentada neste trabalho é um gerador de dados de entrada para executar caminhos completos no programa em teste. É utilizada a técnica dinâmica [14] e um algoritmo de busca para realizar mudanças nas variáveis de entrada com o objetivo de encontrar os dados que executem o caminho pretendido. É mantida uma "base de soluções" consultada sempre que o testador define um novo caminho a ser executado. Este arquivo contém informações das execuções anteriores de caminhos do programa e pode possibilitar a redução do custo da busca. Descreve-se sucintamente a seguir o funcionamento da ferramenta.

Inicialmente tem-se a instrumentação do programa em teste. São inseridas "pontas de prova" com o objetivo de monitorar a execução do programa para que sejam produzidas informações úteis ao direcionamento da geração de dados.

O testador fornece à ferramenta: os parâmetros de controle da busca, dados sobre a manutenção da base de soluções e sobre a interface do programa em teste (número de variáveis e/ou parâmetros de entrada e respectivos tipos e faixas de valores). Devem ser informados também o nome do arquivo do programa em teste executável (versão instrumentada e compilada), além do nome do arquivo onde estão os *caminhos pretendidos*.

Os caminhos pretendidos são tratados um por vez. Considerando o caminho selecionado, é feita uma busca na base de soluções para verificar se este caminho já foi executado anteriormente; caso positivo, é recuperado o conjunto de valores de entrada que provocou a sua execução, que é a solução desejada. Caso contrário, são recuperados os dados de entrada que executaram caminhos similares ao pretendido. Estes dados são utilizados como "região" inicial para o processo de busca dos dados que executam o caminho pretendido. É então acionado o algoritmo de busca, cuja função é a de aprimorar as soluções iniciais (os dados recuperados que executam caminhos similares), fazendo com que estas caminhem em direção à solução desejada; isto é, provoquem a execução do caminho pretendido.

A busca é baseada em um Algoritmo Genético [9, 23, 25]. Este algoritmo tem, neste contexto, a função de combinar e selecionar iterativamente soluções prévias inadequadas com o objetivo de aprimorá-las. O Algoritmo Genético codifica as variáveis de entrada do programa em uma cadeia de bits de comprimento fixo, referida como "indivíduo". A "população" é um conjunto de indivíduos onde cada um representa uma solução potencial para o problema — referida como *solução candidata*. A busca dá-se em um processo iterativo, no qual as soluções tendem a aproximar-se progressivamente do objetivo, até que ele seja alcançado: algum dado de entrada que execute o caminho pretendido seja encontrado. A busca pode ser interrompida adicionalmente se um limite pré-definido de custo (número máximo de gerações) for atingido; ou se for identificada a ausência persistente de progresso, provavelmente associada à não executabilidade do caminho pretendido.

A ferramenta de geração de dados de teste está inserida no contexto da ferramenta de teste POKE-TOOL [3, 16], que apóia a aplicação dos critérios de teste Potenciais Usos [16] para o teste de programas escritos em linguagem C, na versão atualmente em uso. Os caminhos pretendidos serão selecionados de forma automática segundo estratégias definidas em [20], visando a cobertura desses critérios.

5 Busca Baseada em Algoritmos Genéticos

Algoritmos Genéticos são algoritmos de busca baseados nos mecanismos da evolução, seleção natural e da genética [9, 23, 25]. Os Algoritmos Genéticos têm sido aplicados em diferentes áreas relacionadas ao aprendizado de máquina e otimização de funções, sendo considerados um método eficiente e robusto de otimização e busca.

Os Algoritmos Genéticos trabalham com populações de potenciais soluções para um problema, referidas como indivíduos. Estas soluções são submetidas a um processo de seleção baseado no mérito de cada uma e são aplicados operadores genéticos. Utilizando estes operadores o algoritmo cria a geração seguinte a partir das soluções da população corrente, combinando-as e induzindo mudanças. A combinação e a seleção de soluções, quando realizadas de forma iterativa, causam uma evolução contínua da população no sentido da solução do problema de otimização.

Existem na literatura trabalhos que utilizam Algoritmos Genéticos para exercitar ramos do programa. Jones e outros [12] e Michael e outros [18] utilizaram um Algoritmo Genético com este propósito. A função de ajuste é baseada no predicado associado a cada ramo. As variáveis de entrada são expressas de forma codificada e concatenada em uma cadeia de bits. São utilizados operadores genéticos de seleção, recombinação e mutação. Nesses trabalhos Algoritmos Genéticos são utilizados para minimizar erros associados aos predicados do programa que fazem com que um ramo indesejado seja tomado; em ambos só é possível realizar a minimização quando algum dado de entrada que atinge tal predicado seja encontrado. A busca é aleatória até que este dado seja encontrado.

Algoritmos Genéticos são adequados ao teste de software pois são aplicáveis em problemas não lineares, multimodais e descontínuos. Predicados de programas a serem exercitados podem representar funções complexas das variáveis de entrada, visto que, em última análise, relacionam-se à semântica de programas que são desenvolvidos para tratar problemas complexos e diversificados do mundo real.

Neste trabalho utiliza-se um Algoritmo Genético para manipular os valores de entrada do programa a fim de fazer com que o caminho pretendido seja executado. Este algoritmo trabalha com uma população de indivíduos (referidos também por soluções candidatas) onde cada um representa um dado de entrada para o programa, que é uma possível solução do problema.

Sobre esta população são aplicados os operadores genéticos essenciais (mutação simples

e recombinação de ponto único) e é feita a seleção das melhores soluções (isto é, dos dados de entrada mais próximos de executar o caminho pretendido). Em um processo iterativo, a cada nova geração as combinações de valores para as variáveis de entrada que tendem a executar o caminho pretendido vão sendo exploradas e selecionadas. Tal processo provoca o aumento progressivo da qualidade das soluções, direcionando a busca para sub-regiões do domínio de entrada associadas aos dados que executam o caminho pretendido.

A ferramenta comporta de forma direta variáveis de entrada dos tipos primitivos, isto é; inteiros, reais e caracteres. São também tratados vetores uni-dimensionais destes tipos, o que permite variáveis de entrada do tipo vetor de caracteres (ou "strings"). Estruturas de dados complexas como entrada requerem *Drivers de Compatibilização de Tipos*, cuja função é a de receber as variáveis de entrada decodificadas nos tipos tratados pelo Algoritmo Genético e transferí-las para as estruturas complexas aceitas pelo programa em teste. Isto permite a aplicação da ferramenta no teste de programas que recebam como entrada estruturas do tipo registro, matrizes e vetores, por exemplo.

As variáveis de entrada são representadas de forma concatenada e codificadas em código binário. É calculado um "offset" associado aos indivíduos da população considerando os tipos, as faixas de valores e a precisão dessas variáveis no programa em teste. Este "offset" é utilizado no processo de decodificação dos indivíduos necessária à avaliação dos mesmos. Variáveis do tipo real são convertidas para inteiros considerando a precisão a elas associada; variáveis do tipo caracter são tratadas como inteiros pelos respectivos valores ASCII. Cada elemento de "strings" ou vetores é considerado de forma independente para a codificação.

A avaliação de cada indivíduo da população visa permitir a seleção das melhores soluções, realizada a cada geração do Algoritmo Genético e é feita utilizando as informações sobre a execução do programa instrumentado com o dado de entrada codificado no indivíduo. O resultado da avaliação é a associação de um valor de *ajuste* a cada indivíduo da população.

A função de ajuste selecionada para avaliar cada solução candidata tem a forma:

$$Ft = NC - (\frac{EP}{MEP})$$

sendo:

Ft: Medida de ajuste da solução candidata;

NC:*Métrica de Similaridade de Caminhos* computada considerando o número de nós coincidentes entre o caminho executado e o pretendido, a partir do nó de entrada do grafo até o nó onde o caminho executado passa a diferir do pretendido; este valor pode variar entre 1 e o número de nós do caminho pretendido. No primeiro caso (similaridade = 1) apenas o nó de entrada do grafo é comum aos dois caminhos; no segundo caso (similaridade = número de nós do caminho pretendido) o caminho executado e o pretendido são idênticos;

EP: Módulo da função de predicado associada ao predicado onde houve o desvio do caminho pretendido. Este valor reflete o erro que provocou o desvio do caminho executado em relação ao pretendido;

MEP: Maior valor para a função de predicado apurado dentre os caminhos que executaram o mesmo número de nós corretos.

Tabela 1: Função de predicado segundo o tipo de operador relacional envolvido

Predicado	Função de Predicado	rel
$E1 > E2$	$EP = E2 - E1$	$<$
$E1 \geq E2$	$EP = E2 - E1$	$\leq$
$E1 < E2$	$EP = E1 - E2$	$<$
$E1 \leq E2$	$EP = E1 - E2$	$\leq$
$E1 = E2$	$EP = \|E1 - E2\|$	$=$
$E1 \neq E2$	$EP = \frac{1}{(\|E1-E2\|+k1)}$	$\neq \frac{1}{k1}$
$LG(E1)$	$EP = k2$ se $E1 = 0$	
	$EP = 0$ se $E1 \neq 0$	
$LG(!E1)$	$EP = 0$ se $E1 = 0$	
	$EP = k2$ se $E1 \neq 0$	

A função de ajuste reflete o fato de que uma solução candidata está tanto mais próxima da solução desejada quanto maior for o número de nós corretos executados. Considerando diversas soluções com o mesmo número de nós corretos, será considerada mais adequada aquela que apresentar um menor valor absoluto para a *função de predicado* [14] associada ao ramo onde houve o desvio do caminho pretendido. Esta função diz respeito ao erro que provocou o desvio e mede o quão distante está a solução candidata de executar o ramo correto no predicado onde houve o desvio do caminho pretendido.

A função de predicado EP é obtida por análise dinâmica de fluxo de dados. Cada predicado simples $E1$ *op* $E2$ é transformado na forma EP *rel* 0, onde $rel \in \{<, \leq, =, \neq\}$. Por exemplo, o predicado $a > c$ é transformado em $c - a < 0$. A função EP é positiva quando o predicado é falso e negativa quando o predicado é verdadeiro. EP é na verdade uma função das variáveis de entrada do programa; deste modo, alterações nestas variáveis têm o potencial de influenciar o valor desta função. Portanto, é possível manipular as variáveis de entrada a fim de minimizar o valor de EP de um dado predicado. Minimizar EP significa caminhar no sentido de satisfazê-lo.

A Tabela 1 resume o cálculo de EP. Nesta tabela a coluna *Predicado* mostra os possíveis tipos de predicados considerando os vários operadores relacionais; *Função de Predicado* é a função EP respectiva e *rel* é o operador adequado para EP *rel* 0.

Quando o predicado envolve comparações entre caracteres e "strings" o cálculo de EP é feito considerando-se valores ASCII associados aos caracteres.

A computação da função EP resultante, no caso de predicados compostos com operadores lógicos, considera a soma das funções para condições compostas com operador AND e o menor valor dentre elas para o operador OR [8].

Deve-se notar que o valor de $\frac{EP}{MEP}$ significa um coeficiente de erro da solução candidata perante todas as soluções da população que conseguiram executar o caminho correto até o mesmo predicado de desvio. Este valor é utilizado como uma penalidade para a solução. Deste modo, a dinâmica da busca caracteriza-se pela co-existência de dois objetivos: maximizar o número de nós corretamente executados em relação ao caminho pretendido e minimizar a *função de predicado* dos predicados atingidos.

A função de ajuste de uma dada solução candidata estipula a sua chance de sobrevivência, determinando assim a influência desta nas próximas gerações. Isto direciona

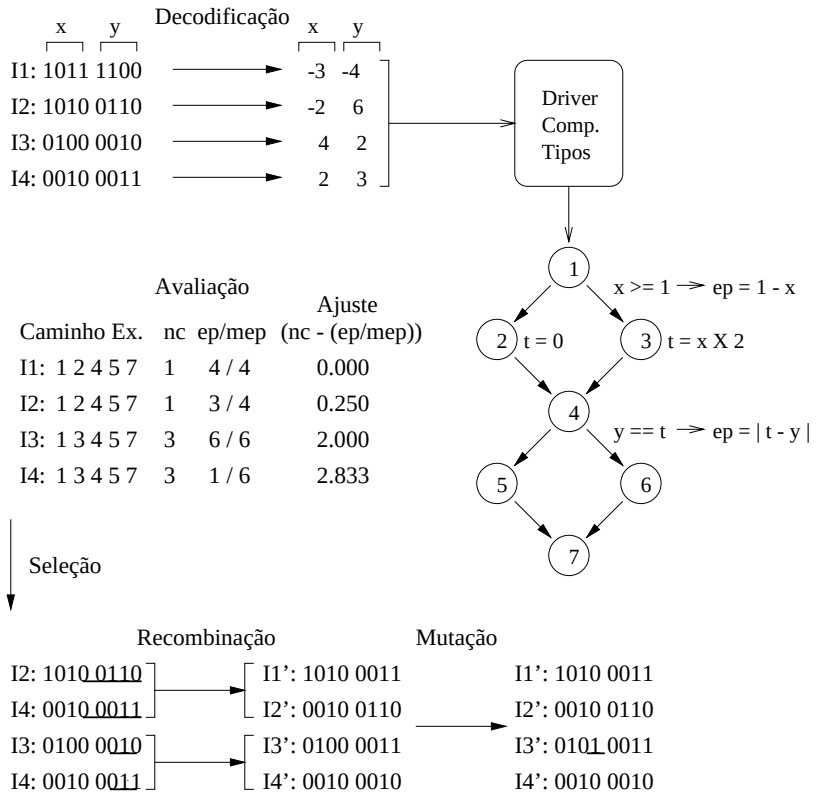

Figura 1: Aplicação do Algoritmo Genético na Ferramenta

a busca realizada pelo Algoritmo Genético, aproximando progressivamente os dados de entrada manipulados da sub-região do domínio de entrada do programa onde estão dados que executam o caminho pretendido.

A Figura 1 ilustra o processo de geração de dados de teste para executar o caminho pretendido 1 3 4 6 7. São destacados os diversos processos relacionados ao ciclo do algoritmo genético. Os indivíduos existentes na população são *decodificados* obtendo-se as variáveis de entrada do programa nos seus tipos originais (inteiros de -7 a +7). Cada dado de entrada, gerado pela decodificação de um indivíduo, passa (se necessário) pelo *driver de compatibilização de tipos* e é utilizado na execução do programa instrumentado em teste. São obtidas então as informações necessárias ao cálculo do ajuste de cada indivíduo, feito na etapa de *Avaliação*. São aplicados os operadores genéticos na população corrente: *Seleção, Recombinação* e *Mutação*. Tem-se assim uma nova população de indivíduos gerada e o início da próxima iteração do ciclo do algoritmo genético.

6 Modelo de Instrumentação

O programa instrumentado para o teste possui "pontas de prova" com o objetivo de monitorar a execução e coletar informações dinâmicas sobre os fluxos de controle e de dados. As pontas de prova são basicamente comandos de escrita em arquivo. Elas são inseridas em pontos específicos do código para que as informações coletadas sejam úteis ao processo de geração de dados de teste. Essencialmente são obtidos: a seqüência de nós executada (caminho executado) e o valor das variáveis utilizadas na avaliação dos predicados percorridos na execução do programa. O fluxo de controle é monitorado

segundo o modelo definido em [3] visando a obtenção do caminho executado. O fluxo de dados é monitorado a fim de apurar dados para o cálculo da *função de predicado*. Essencialmente, quando tem-se alguma decisão no programa, são gravados em arquivo os valores das variáveis ou expressões envolvidas nesta decisão. Nesses casos, pontas de prova são inseridas imediatamente depois dos predicados.

O monitoramento do fluxo de dados é feito através de pontas de prova do tipo $error\_write(N, E1, E2, OR, OL)$. Onde: N é o número do nó no grafo do programa onde a ponta de prova está inserida; $E1$ é a primeira expressão ou variável presente no predicado; $E2$ é a segunda expressão ou variável presente no predicado; OR é o operador relacional envolvido; e OL é o operador lógico envolvido no caso de predicados compostos. A lógica é que esta ponta de prova, presente em um determinado ponto do programa após uma decisão, permite o cálculo da função de predicado associada ao ramo oposto desta decisão. Exemplificando, um comando de seleção seria instrumentado do seguinte modo: $if(i > j)$ $\{error\_write(3, i, j, <=, \#)....\}$ $else$ $\{error\_write(2, i, j, >, \#)....\}$. Onde 3 e 2 são os números dos respectivos nós do grafo e # indica que o predicado é simples.

O modelo proposto é compatível com predicados compostos, considerados através da utilização de pontas de prova associadas a cada condição da decisão. Estruturas do tipo "case" podem ser tratadas através de uma instrumentação "dependente do caminho pretendido" ou de "pontas de provas nulas". Laços infinitos na execução são evitados fazendo com que o programa em teste seja finalizado se o número de nós atravessados for superior a um limite estabelecido. Este modelo encontra-se completamente definido; entretanto, o módulo que instrumenta automaticamente o programa em teste ainda não foi implementado. Esta tarefa é feita manualmente na versão atual.

7 Reuso de Soluções Passadas Baseado em Analogia

O processo de geração de dados de teste proposto requer comumente diversas execuções do programa até que um dado que executa o caminho pretendido seja descoberto. Em cada execução são inseridos em uma *Base de Soluções*: o caminho completo executado no programa e o respectivo conjunto de valores das variáveis de entrada. Esta infomação, gerada como um sub-produto da busca, reflete como ocorre o mapeamento entre os componentes estruturais (caminhos do programa) e conjuntos de valores de entrada que os exercitam, aspecto primordial na geração de dados de teste.

A cada novo caminho pretendido é verificado se este caminho existe na base de soluções; isto é, se em algum momento anterior no processo de geração de dados tal caminho, agora pretendido, foi fortuitamente executado. Caso positivo, o conjunto de valores de entrada respectivo, solução do problema, é recuperado; caso contrário, são recuperados os dados de entrada que executaram caminhos análogos ao pretendido. Esses dados são utilizados para compor a população inicial do Algoritmo Genético. Isto permite utilizar no problema corrente informações geradas previamente durante a busca de dados para executar outros caminhos.

A recuperação de dados associados a caminhos análogos é baseada na *métrica de similaridade* definida (descrita na Seção 5) que quantifica o nível de semelhança entre dois caminhos de programa sob a ótica da geração de dados. São recuperados primeiro os dados de entrada associados aos caminhos com maior nível de similaridade; em seguida, os associados a caminhos com níveis sucessivamente menores. A população inicial do Algoritmo Genético é formada pelos dados recuperados e por dados gerados aleatoriamente, o que garante a diversidade de soluções essencial ao sucesso da busca.

O reuso de soluções passadas análogas permite que a população inicial do Algoritmo Genético contenha dados de entrada próximos dos necessários para a execução do caminho

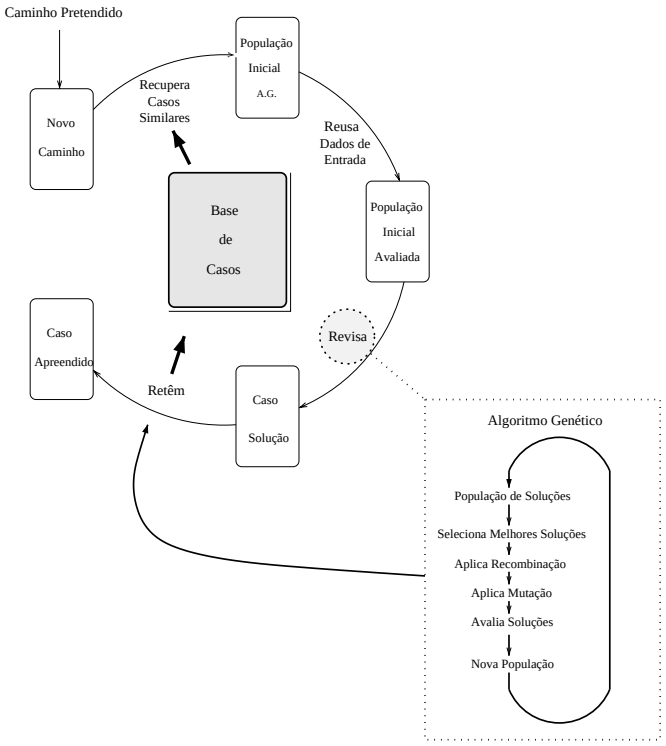

Figura 2: Raciocínio Baseado em Casos no Contexto da Ferramenta

pretendido, o que tende a reduzir o custo da busca.

Existe uma consonância da abordagem de recuperação/adaptação de soluções adotada e o conceito de *Raciocínio Baseado em Casos* (R.B.C.) [13, 1, 15]. O R.B.C. é um paradigma para resolução de problemas em que as soluções são obtidas através da identificação de casos passados similares e do reuso destes casos no novo problema. Os casos são pares problema-solução respectiva e têm o potencial de permitir que um objetivo, ou conjunto de objetivos, seja atingido mais facilmente no futuro. A Figura 2 destaca a utilização do Raciocínio Baseado em Casos no contexto da ferramenta proposta.

A idéia é recuperar raciocínios anteriores ao invés de repetir o esforço anterior, o que é particularmente interessante em problemas complexos, cuja resolução tende a ser custosa e difícil. O Raciocínio Baseado em Casos é também uma abordagem de aprendizado incremental; novas experiências são retidas sempre que um novo problema é resolvido. A fonte de conhecimento consiste de uma memória de casos relacionados a episódios específicos anteriores.

O Raciocínio Baseado em Casos tem uma natureza cíclica e aborda problemas do seguinte modo: dado um novo problema o sistema baseado em casos realiza uma avaliação da situação e gera uma descrição do problema. É feita uma busca na base de casos por problemas anteriores com descrições análogas. A solução do problema com maior nível de similaridade é utilizada como ponto de partida para a geração da solução para o novo problema. A nova solução é guardada para futuro reuso.

A adequação deste paradigma à geração de dados de teste deve-se à alta complexidade deste problema. Encontrar valores de entrada para o programa que satisfaçam as restrições representadas pelos predicados existentes ao longo do caminho pretendido é uma tarefa complexa e dispendiosa do ponto de vista computacional. Neste caso, recuperar dados de entrada executados anteriormente que satisfaçam as restrições referidas pode significar

uma substancial economia em termos de computação e tempo dispendidos.

No contexto deste trabalho os casos consistem das tuplas {*caminho executado, dado de entrada*}. Casos análogos são identificados pela métrica de similaridade de caminhos, permitindo que dados de entrada sejam recuperados da base de casos e componham a população inicial do Algoritmo Genético. Este algoritmo realiza o reparo e a adaptação de soluções necessários à solução do novo problema através da aplicação dos operadores genéticos e da seleção de soluções, realizadas iterativamente. A cada avaliação de uma solução candidata do Algoritmo Genético tem-se a inserção de um novo caso na Base de Soluções. Isto permite um progressivo aumento do número de casos contidos na base que são potencialmente úteis para o teste dos próximos caminhos.

8 Tratamento Dinâmico Para a Questão da Não Executabilidade

A determinação de não executabilidade é estudada por diversos autores. Em [22] tem-se uma abordagem baseada em processamento simbólico que visa detectar cláusulas simples contraditórias nos predicados, associadas a alguns dos caminhos não executáveis. Em [4] também é utilizado o processamento simbólico para a identificação da não executabilidade de caminhos cujos predicados formem uma função linear das variáveis de entrada. Em [7] são utilizadas técnicas de execução simbólica e de análise de fluxo de dados para determinar algumas associações de fluxo de dados não executáveis. Em [10] é utilizada análise numérica para identificar caminhos não executáveis quando os predicados forem lineares em relação às variáveis de entrada. Outros trabalhos visam a previsão da não executabilidade e estudam a influência do número de predicados na executabilidade de um caminho [17, 24].

Neste trabalho o tratamento desta questão é feito pelo monitoramento do progresso da busca realizada pelo Algoritmo Genético. Como destacado na Seção 5 a busca com Algoritmos Genéticos dá-se com uma população de soluções que evoluem seguindo regras probabilísticas; esta evolução pode ser avaliada pelo monitoramento de medidas de desempenho. Esta abordagem é utilizada no contexto da otimização de funções utilizando Algoritmos Genéticos para estudar a adequação de diferentes tratamentos a este problema [9]. No contexto da geração de dados de teste este monitoramento é útil para a identificação de situações de *potencial não executabilidade* do caminho pretendido.

O valor *média de ajuste da população* reflete a média de qualidade das soluções da população e pode ser utilizado para aferir de forma bastante confiável o progresso da busca. Na geração de dados para caminhos executáveis, observa-se um progresso contínuo da média de ajuste, na medida em que os predicados vão sendo alcançados e solucionados pelo Algoritmo Genético. Por outro lado, a tentativa de gerar dados para caminhos não executáveis resulta, invariavelmente, em uma ausência persistente de progresso deste valor; isto ocorre porque, neste caso, algum predicado do caminho é não factível e impossibilita o progresso da busca.

A identificação de ausência de progresso é feita através da seguinte heurística:
Se
i) $aj[i] - aj[i-1] \leq \delta \ \forall \ i,\ j \leq i \leq (j + NL)$ e
ii) $aj[j + NL] - aj[j] \leq \Delta$
então: Situação de *potencial não executabilidade* identificada.

Onde: $aj[i]$ é a média de ajuste da população na geração [3] i; $aj[i-1]$ é a média de ajuste da população na geração $i-1$, imediatamente anterior à i; j é a geração na qual a

[3] Neste contexto o temo *geração* refere-se a um ciclo (ou uma geração) do Algoritmo Genético e não à geração de dados.

relação i foi inicialmente observada; δ é o valor limite abaixo do qual se considera situação de ausência de progresso para duas gerações sucessivas; Δ é o valor limite abaixo do qual se considera situação de ausência de progresso acumulado por NL gerações; e NL é o número de gerações requerido para a satisfação da condição i.

O valor NL determina o quão persistente necessita ser o não progresso da busca para que o testador seja advertido da provável não executabilidade do caminho. Em problemas mais complexos de geração de dados, é desejável associar maiores valores à NL a fim de permitir uma tolerância ao não progresso da busca, aceitável devido a esta maior complexidade. Para tanto foi estabelecida uma equação que modela a complexidade do problema em função do número de variáveis de entrada do programa e do número de nós do caminho pretendido.

$$NL = K1 + K2 \times (Np + Nv)$$

Onde: NL é o número de gerações requerido para a satisfação da condição [i]; $K1$ é um valor constante; Np é o número de predicados do caminho pretendido; Nv é o número de variáveis de entrada do programa (numero de parâmetros + número de variáveis recebidas pelo teclado + número de variáveis globais); $K2$ é o fator de controle da influência de Np e Nv em NL.

Esta equação é utilizada para computar o valor NL e visa ajustar de forma automática a heurística ao problema particular (caminho pretendido), para que um caminho não executável possa ser distinguido de um caminho "difícil" de ser executado.

Atualmente utilizam-se os seguintes parâmetros: $\delta = 0.50$; $\Delta = 0.30$; $K1 = 30$ e $K2 = 3$. Caso sejam satisfeitas as regras i e ii a busca é provisoriamente interrompida e tem-se a identificação da potencial não executabilidade do caminho pretendido. Neste caso o testador é informado do predicado suspeito de provocar a não executabilidade do caminho e dos valores: δ, Δ e NL.

Este enfoque dinâmico é aplicável indistintamente, independentemente do tamanho do programa, de tipos de variáveis envolvidas e de pré-requisitos sobre equações de caminhos tratáveis, limitações inerentes às abordagens anteriores. Entretanto, não é possível classificar o caminho como não executável com 100% de certeza; isto porque, na técnica dinâmica utilizada, sempre existirá a dúvida: o conjunto de valores de entrada adequado não foi encontrado por deficiência da busca ou o conjunto não existe, situação na qual o caminho é não executável. É importante observar que esta identificação, feita de forma manual pelo testador, também é passível de erros.

De certo modo, as abordagens estáticas e a dinâmica possuem características complementares. Uma estratégia interessante é aplicar as primeiras para identificar a não executabilidade de alguns elementos e, então, utilizar a ferramenta na geração de dados para exercitar os restantes. Caso identificada a potencial não executabilidade de algum caminho, o testador pode decidir se confia nesta identificação, ou se a confirma analisando o código; tomando como base, neste caso, o predicado suspeito de causar a não executabilidade, fornecido pela ferramenta.

9 Experimento e Resultados

Foi realizado um experimento inicial para avaliar o desempenho da ferramenta na geração de dados de teste; avaliou-se também a influência do reuso de soluções passadas na geração de dados e a eficácia da heurística de identificação dinâmica de potencial não executabilidade.

Foram utilizados 4 programas com características diferentes de tipos de variáveis tratadas ("strings", caracteres, reais, inteiros e vetores); predicados (simples, compostos, envolvendo variáveis lógicas, vetores e "strings") e estruturas de controle (repetição e seleção), incluindo os programas *quotient* [8] e *find* [7].

Foram selecionados ao todo 24 caminhos executáveis nesses programas. Foram definidos três modos de execução: utilizando o Algoritmo Genético com o reuso de soluções passadas; o Algoritmo Genético sem este reuso e uma busca aleatória. Para cada caminho e modo de execução foram feitas diversas tentativas de geração de dados de teste (um total de 600 tentativas). Em cada tentativa foi registrado o número de execuções do programa necessárias para que algum dado de entrada que executa o caminho fosse obtido (um total de 4.274.917 execuções). Caso fosse atingido um limite superior de número de execuções a busca era encerrada, era constatada a falha da busca e computado este limite para o cálculo de valores médios.

9.1 Custo e Eficácia na Geração Automática de Dados de Teste

Utilizando o *Algoritmo Genético e o Reuso de Soluções Passadas* foi obtido 100% de sucesso. Para cada tentativa foram requeridas em média 2.831, 49 execuções dos programas, equivalentes a aproximadamente 8 minutos de busca [4].

Utilizando *apenas o Algoritmo Genético* foi obtido 97, 92% de sucesso. Para cada tentativa foram requeridas em média 3.180, 11 execuções dos programas, equivalentes a aproximadamente 9 minutos de busca.

Utilizando a *Busca Aleatória* foi obtido 70, 83% de sucesso. Para cada tentativa foram requeridas em média 23.601, 11 execuções dos programas, equivalentes a aproximadamente 67 minutos de busca.

Foi possível observar que a geração aleatória de dados é eficaz para alguns caminhos selecionados. Tais caminhos caracterizam-se por estarem associados a grandes sub-regiões do domínio de entrada do programa e possuirem predicados que estabelecem restrições de fácil satisfação. Nestas situações é possível encontrar, após algumas tentativas, os valores de entrada pertencentes a estas sub-regiões, fazendo com que a geração aleatória seja aplicável.

Em outros casos os predicados existentes ao longo do caminho estabelecem restrições severas no domínio de entrada, fazendo com que apenas dados com características extremamente específicas os executem. Nestes casos a chance de que sejam gerados aleatoriamente valores para execução dos caminhos é pequena, mesmo com um grande número de tentativas. Para estes caminhos o custo da geração aleatória de dados é proibitivo e o índice de sucesso muito baixo. Por outro lado, a utilização da ferramenta permitiu um índice de sucesso de 100% e um custo aceitável para a geração de dados (em torno de 160 vezes menor comparado à geração aleatória).

A análise dos resultados permitiu identificar características em programas e caminhos que tendem a aumentar o custo da geração de dados de teste: programas com um grande número de variáveis de entrada; caminhos com um grande número de predicados; caminhos contendo predicados compostos com o operador lógico AND; caminhos com predicados envolvendo operador relacional de igualdade (exemplo: *if* $(x == y)$) e caminhos com predicados que fazem a comparação de "strings" (exemplo: *if(!strcmp(nome,"teste"))*). Quando estas características estão presentes o benefício da utilização da ferramenta (em termos de aumento do nível de sucesso e redução do custo computacional) torna-se mais

[4] Utilizando uma estação de trabalho Sun Ultra-1; 256Mb RAM; clock 143 MHz; sistema operacional SunOS 5.5.1.

expressivo em relação à geração aleatória de dados.

A comparação da geração de dados utilizando a ferramenta com a geração manual de dados não foi conduzida de forma sistemática; no entanto, a análise dos programas e caminhos utilizados no experimento leva à suposição de que seriam requeridos tempos consideravelmente superiores para a geração manual desses dados, em boa parte dos casos. Além disso, para alguns caminhos pretendidos (sobretudo alguns com diversas iterações em laços e com cerca de 100 predicados) a geração manual de dados seria uma tarefa inviável devido à sua complexidade.

9.2 Eficácia da Heurística de Identificação Dinâmica de Potencial Não Executabilidade

A *heurística de identificação dinâmica de potencial não executabilidade* gerou erros de avaliação em $1,25\%$ dos casos. Nestas situações, caminhos executáveis foram identificados erroneamente como não executáveis. Isto ocorreu porque predicados difíceis de serem satisfeitos (por estabelecerem restrições severas no domínio de entrada do programa) causaram uma ausência persistente de progresso da busca, ocasionando a avaliação incorreta de não executabilidade. Este problema pode ser minimizado pela consideração dos tipos de predicados existentes nos caminhos, para a estimação da complexidade do problema da geração de dados (valor NL, descrito na Seção 8).

Deve-se notar que o tratamento proposto neste trabalho para a questão da não executabilidade foi possível devido à alta confiabilidade do Algoritmo Genético como uma ferramenta de otimização aplicada ao problema da geração de dados de teste. A alta eficácia deste algoritmo permitiu a verificação de uma forte correlação entre a ausência de progresso da busca e a não executabilidade do caminho pretendido. O correto ajuste dos parâmetros da heurística (δ, Δ, $K1$ e $K2$) permite a deteção segura desta ausência de progresso.

As tentativas de geração de dados para os caminhos não executáveis selecionados resultaram, em todos os casos, na correta identificação desta condição, feita pela heurística proposta.

9.3 Impacto do Reuso de Soluções Passadas no Custo e na Eficácia da Geração de Dados de Teste

O *Reuso de Soluções Passadas*, relacionado ao Raciocínio Baseado em Casos permitiu uma redução média de 12% no número de execuções do programa. Um aspecto significativo foi que conseguiu-se 100% de sucesso com o reuso de soluções e o Algoritmo Genético, contra $97,92\%$ de sucesso utilizando apenas o Algoritmo Genético. Isto ocorreu porque o fato de a população inicial do Algoritmo Genético conter soluções "próximas" das buscadas tende a reduzir o número de execuções necessárias para que o dado de entrada que executa o caminho pretendido seja descoberto, aumentado assim a chance de sucesso antes do limite superior estabelecido para este número no experimento.

10 Conclusão e Trabalhos Futuros

Neste trabalho foram abordadas a automação das atividades de geração de dados de teste e de identificação de não executabilidade de caminhos no teste estrutural de software; são tarefas comumente tratadas de forma manual, que exigem do testador a cuidadosa análise do código do programa, grande esforço mental e conhecimentos específicos dos critérios de teste utilizados.

A solução elaborada e implementada integra conceitos consolidados na literatura:

técnica dinâmica de geração de dados, otimização utilizando Algoritmos Genéticos e resolução de problemas através do Raciocínio Baseado em Casos. O objetivo de gerar dados para executar caminhos do programa permite a aplicação desta solução para o teste utilizando diversos critérios estruturais. A compatibilidade com programas que tratam diferentes tipos de variável e com as diversas estruturas presentes na lingugem C torna a ferramenta aplicável a problemas reais.

Um experimento inicial conduzido dá indícios consistentes da validade das técnicas utilizadas e do benefício da utilização da ferramenta, cujo uso, quando comparado à geração aleatória de dados, permite um custo expressivamente menor na geração de dados de teste.

Contribuições deste trabalho incluem: a aplicação de Algoritmos Genéticos para o teste de caminhos do programa; a utilização de Raciocínico Baseado em Casos no contexto do teste de software e a definição de uma heurística dinâmica para o tratamento da não executabilidade.

Foram identificadas diversas extensões futuras promissoras para este trabalho; entre elas: a definição de um modelo genérico para a geração automática de dados de teste; geração automática de dados para o teste de integração e para o teste de regressão, além de aprimoramentos nos modelos desenvolvidos.

Referências

[1] A. Aamodt and Plaza E. Case-based reasoning: Foundational issues, methodological variations, and system approaches. *AICom — Artificial Intelligence Communications*, Vol. 7(1):39–59, 1994.

[2] R. Bodík, R. Gupta, and M. L. Soffa. Refining data flow information using infeasible paths. *Software Engineering Notes*, Vol. No 22(6):361–377, Novembro 1997.

[3] M.L. Chaim. *POKE-TOOL - Uma Ferramenta para Suporte ao Teste Estrutural de Programas Baseado em Análise de Fluxo de Dados.* Dissertação de Mestrado, DCA/FEEC/Unicamp, Campinas - SP, Brazil, Abril 1991.

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

[5] R. Ferguson and B. Korel. The chaining approach for software test data generation. *ACM Transactions on Software Engineering and Methodology*, 5(1):63–86, Janeiro 1996.

[6] I. Forgács and A. Bertolino. Feasible teste path selection by principal slicing. *Software Engineering Notes*, Vol. No 22(6):378–394, Novembro 1997.

[7] F.G. Frankl. *The use of Data Flow Information for the Selection and Evaluation of Software Test Data.* PhD Thesis, Department of Computer Science, New York University, New York, U.S.A., Outubro 1987.

[8] M. J. Gallagher and V. N. Narasimham. Adtest: A test data generation suite for ada software systems. *IEEE Transactions on Software Engineering*, Vol. 23(8):473–484, Agosto 1997.

[9] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley Publishing Company, Inc, University of Alabama, 1989.

[10] N. Gupta, A. P. Mathur, and M. L. Soffa. Automated test data generation using an iterative relaxation method. *Foundations of Software Engineering*, Vol. No 11:231–244, Novembro 1998.

[11] W.E. Howden. Symbolic testing and dissect symbolic evaluation system. *IEEE Transactions on Software Engineering*, Vol. SE-3(4):266–278, Julho 1977.

[12] B. F. Jones, H. H. Sthamer, and D. E. Eyres. Automatic structural testing using genetic algorithms. *Software Engineering Journal*, pages 299–306, Setembro 1996.

[13] J. L. Kolondner. *Case-Based Reasoning*. Morgan Kaufmann Publishers, 1993.

[14] B. Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, Vol. SE-16(8):870–879, Agosto 1990.

[15] D. B. Leake. *Case-Based Reasoning Experiences, Leassons and Future Directions*. AAAI Press — The MIT Press, Menlo Park, CA 94025, 1996. Collection: 17 articles.

[16] J.C. Maldonado. *Critérios Potenciais Usos: Uma Contribuição ao Teste Estrutural de Software*. Tese de Doutorado, DCA/FEEC/Unicamp, Campinas - SP, Brazil, Julho 1991.

[17] N. Malevris, D.F. Yates, and A. Veevers. Predictive metric for likely feasibility of program paths. *Information and Software Technology*, Vol. 32(2):115–118, Março 1990.

[18] C. C. Michael, G. E. McGraw, M. A. Schatz, and Walton C. C. *Genetic Algorithms for Dynamic Test-Data Generation*. Technical Report RSTR-003-97-11, RST Corporation: Suite 250, 21515 Ridgetop Circle Sterling, CA 20166, Maio 1997. (at http://www.rstcorp.com/paper-subject.html).

[19] W. Miller and D. L. Spooner. Automatic generation of floating-point test data. *IEEE Transactions on Software Engineering*, Vol. SE-2(3):223–226, Setembro 1976.

[20] L.M. Peres. *Estudo de Estratégias de Seleção de Caminhos para Satisfação de Critérios de Teste Estrutural*. Dissertação de Mestrado, DCA/FEEC/Unicamp, Campinas - SP, 1997. (em elaboração).

[21] A.M.A. Price and J.S. Herbert. Estratégia de geração de dados de teste baseada na análise simbólica e dinâmica do programa. In *XI Simpósio Brasileiro de Engenharia de Software*, pages 397–411. Fortaleza , Brazil, Novembro 1997.

[22] C.V. Ramamoorthy, F. H. Siu-Bun, and W.T. Chen. On automated generation of program test data. *IEEE Transactions on Software Engineering*, Vol. SE-2(4):293–300, Dezembro 1976.

[23] M. Srinivas and Patnaik L. M. Genetic algorithms: A survey. *IEEE Computer*, 27(6):17–26, Junho 1994.

[24] S.R. Vergilio. *Caminhos Não Executáveis: Caracterização, Previsão e Determinação para Suporte ao Teste de Programas*. Dissertação de Mestrado - DCA/FEEC/Unicamp, Campinas - SP, Brazil, Janeiro 1992.

[25] Michalewicz Zbigniew. *Genetic Algorithms + Data Strutures = Evolution Programs*. Springer Verlag, North Carolina - USA, 3rd edition, 1996.