



MARCOS PAULO MAIA NICOLAU

**IMPLEMENTAÇÃO DE FERRAMENTA COM
ABORDAGEM MULTI-OBJETIVA UTILIZANDO
ALGORITMOS EVOLUTIVOS**

LAVRAS - MG

2012

MARCOS PAULO MAIA NICOLAU

**IMPLEMENTAÇÃO DE FERRAMENTA COM ABORDAGEM
MULTI-OBJETIVA UTILIZANDO ALGORITMOS EVOLUTIVOS**

Monografia apresentada ao Colegiado do
Curso de Ciência da Computação, para a
obtenção do título de Bacharel em Ciên-
cia da Computação.

Orientador

Prof.Ms. Juliana Galvani Greggi

LAVRAS - MG

2012

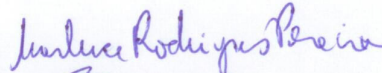
MARCOS PAULO MAIA NICOLAU

**IMPLEMENTAÇÃO DE FERRAMENTA COM ABORDAGEM
MULTI-OBJETIVA UTILIZANDO ALGORITMOS EVOLUTIVOS**

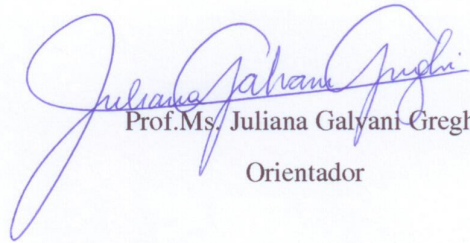
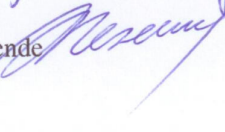
Monografia apresentada ao Colegiado do
Curso de Ciência da Computação, para a
obtenção do título de Bacharel em Ciên-
cia da Computação.

Aprovada em 23 de Outubro de 2012

Profa.Dra. Marluce Rodrigues Pereira



Prof.Dr. Antônio Maria Pereira de Resende



Prof.Ms. Juliana Galyani Greggi

Orientador

LAVRAS - MG

2012

Dedico este trabalho primeiramente à Deus. Depois, aos meus pais e ao meu irmão, meus maiores professores. Aos meus amigos da UFLA, que foram muito companheiros nessa jornada. Aos meus professores, em especial à professora Juliana pela orientação, pelos conselhos e apoio durante toda a graduação.

RESUMO

A Engenharia de Software tem se mostrado cada vez mais importante para o desenvolvimento de software de qualidade. Ferramentas e soluções que visam praticidade e confiabilidade para os sistemas em desenvolvimento, têm muito a acrescentar nesse cenário. Sendo assim, a automatização de processos de Teste de Software se torna muito importante, tanto para agilizar os processos de Teste, quanto para agregar valor aos resultados gerados, visto que a execução manual é muito propensa a erros. O presente trabalho implementa algoritmos para a geração automática de dados de teste, baseado em máquinas de estados finitas estendidas, combinando testes de caixa branca, a partir das heurísticas evolutivas M-GEOvsl e MOST.

Palavras-chave: Teste de Software; Otimização; Heurísticas; Evolutiva

ABSTRACT

Software Engineering has an important role in software development. Tools and solutions aiming practicality and reliability for systems under development, have much to add in this scenario. The Software Testing is one of the areas of Software Engineering which aims to help quality software development. Thus, the automation of software testing becomes very important, both to streamline the test, as to add value to the generated results, since the manual execution is very error-prone. This work implements algorithms for the automatic generation of test data, based on Extended Finite State Machines (EFSM), using evolutionary heuristics M-GEO_{vsI} e MOST.

Keywords: Software Testing ; Heuristics; Evolution

LISTA DE FIGURAS

Figura 1	Grafo de controle de fluxo do algoritmo Busca Binária (BUZZO <i>et al.</i> , 2010)	20
Figura 2	Processo de teste baseado em modelo (YANO, 2011)	23
Figura 3	Exemplo de caminho inactivável em uma MEFE: conflito de dados na ação de t_1 com a guarda de t_3 (YANO, 2011)	24
Figura 4	Modelo geral de Algoritmo Evolutivo (BUZZO <i>et al.</i> , 2010) (MICHALEWICZ; SCHOENAUER, 1996).....	29
Figura 5	Equação da probabilidade de uma perturbação de tamanho s ocorrer (BUZZO <i>et al.</i> , 2010)	37
Figura 6	Espécies no modelo Bak-Sneppen (BUZZO <i>et al.</i> , 2010).....	38
Figura 7	Equação de combinação de funções objetivo (YANO, 2011)	42
Figura 8	Fronteira de Pareto (YANO, 2011).....	44
Figura 9	Comparação entre Populações do M-GEO e M-GEO _{vsl} (YANO, 2011)	45
Figura 10	Exemplos de mutações no M-GEO _{vsl} em partes diferentes da população (YANO, 2011)	47
Figura 11	Fluxograma comparativo entre M-GEO e M-GEO _{vsl} (YANO, 2011)	48
Figura 12	Minimização de um vetor de inteiros de tamanho n para diferentes funções objetivo e um inteiro y (YANO, 2011)	51
Figura 13	Processo Evolutivo do M-GEO _{vsl} para MOST (YANO, 2011)	60
Figura 14	População no M-GEO _{vsl} para geração de dados de teste baseado em MEFE (YANO, 2011)	61
Figura 15	Sistema de caixa eletrônico representado em uma MEFE (BUZZO <i>et al.</i> , 2010)(YANO, 2011)	67
Figura 16	Código exemplo em Java para geração de números aleatórios.	72
Figura 17	Arquivo exemplo de modelagem de uma MEFE.	73

Figura 18	Exemplo para a modelagem da MEFÉ exemplificada na Figura 17 (BUZZO <i>et al.</i> , 2010).	74
Figura 19	Pseudocódigo para verificar caminhos inactíveis.	75
Figura 20	Exemplo de população inicial do M-GEO _{vsl}	75
Figura 21	Vetor Ordem de Aptidão da população do M-GEO _{vsl} para a função objetivo 2 do exemplo.	76
Figura 22	Exemplo de resultado da execução do MOST.	80
Figura 23	Exemplo de saída parcial da execução do MOST.....	81
Figura 24	MEFÉ Exemplo para execução do MOST.	82
Figura 25	Gráfico da quantidade de saídas aplicáveis de uma MEFÉ exemplo na execução do MOST, com visualização do eixo Y de 100 saídas.	83

LISTA DE TABELAS

Tabela 1	Trabalhos para geração de casos de teste baseados em MEFE (YANO, 2011)	26
Tabela 2	Trabalhos sobre testes evolutivos (BUZZO <i>et al.</i> , 2010).....	34
Tabela 3	Exemplo numérico dos passos do M-GEO _{vsl} (YANO, 2011).....	53

SUMÁRIO

1	INTRODUÇÃO	11
1.1	Motivação	15
1.2	Objetivos	16
1.3	Estrutura do Texto	17
2	REFERENCIAL TEÓRICO	18
2.1	Teste de Software	18
2.1.1	Testes de Caixa Branca	19
2.1.2	Teste baseado em modelos	21
2.2	Algoritmos Evolutivos	25
2.2.1	Exemplos da aplicação de AE	35
2.2.2	Otimização Extrema Generalizada (GEO)	36
2.2.3	Otimização Extrema Generalizada com Codificação Real (GEO-real)	40
2.2.4	Otimização Multi-objetiva	41
2.2.5	Otimização Extrema Generalizada Multi-objetiva (M-GEO) ...	43
2.2.6	M-GEO_{vsl}	44
2.2.7	Codificação da Solução	45
2.2.8	Operador de Mutação	46
2.2.9	Processo Evolutivo	47
2.2.10	Os valores de RAN, τ e o número de reinicializações	52
2.2.11	Ótimo de Pareto	55
2.3	MOST	55
2.3.1	Modelo Executável	57
2.3.2	Dependência de MEFE	59
2.3.3	Gerador de casos de teste	59

2.3.4	Codificação da Solução	60
2.3.5	Função Objetivo.....	62
2.4	Máquinas Finitas de Estados	63
2.4.1	Máquina de Estados Finita - MEF	63
2.4.2	MEFE - Máquina de Estados Finita Estendida.....	64
3	METODOLOGIA	69
3.1	Tipo de Pesquisa	69
3.2	Materiais e Métodos	69
3.3	Etapas desenvolvidas	69
4	RESULTADOS	71
4.1	Implementação do M-GEO _{vsl} e MOST	71
4.1.1	O problema dos números aleatórios	71
4.1.2	A modelagem em MEFE	72
4.1.3	Validação das sequências geradas	74
4.2	Resultados do M-GEO _{vsl}	74
4.2.1	Caminhos infactíveis gerados.....	78
4.3	Resultados do MOST	78
5	CONCLUSÃO	84
6	TRABALHOS FUTUROS	85

1 INTRODUÇÃO

Geração de dados de teste em geral, é uma atividade que consome muito tempo e, apesar de os primeiros artigos da área terem sido lançados na década de 60 (BUZZO *et al.*, 2010), segundo (YANO *et al.*, 2011), essa atividade ainda é predominantemente manual. (MANTERE; ALANDER, 2005) criticam o cenário de grande crescimento da complexidade e criticalidade dos softwares enquanto o processo de teste de software em muitas empresas é imaturo. Testar um software ajuda na identificação de erros e, um benefício direto é a demonstração de que as funções de software estão, aparentemente, de acordo com as especificações (BUZZO *et al.*, 2010).

A abordagem evolutiva estudada neste trabalho propõe a geração automática de casos de teste baseado em modelos, utilizando técnicas de caixa branca. Um *modelo executável*¹ de Máquina de Estados Finita Estendida (MEFE) é utilizado para lidar com o problema de geração de caminhos inviáveis. É possível assim, segundo (YANO *et al.*, 2011), encontrar caminhos viáveis, ao invés de efetuar análises de acessibilidade caminho a caminho. Utilizando esses modelos, o sistema pode ser avaliado logo no início do desenvolvimento (BUZZO *et al.*, 2010).

Máquinas de Estados Finitas (MEF) são comumente usadas para representar o comportamento do sistema e têm sido utilizadas para a geração de casos de teste (BOCHMANN; PETRENKO, 1994). Em particular, máquinas de estados finitas estendidas (MEFE) permitem a representação de controle e também dos aspectos do comportamento do sistema, e podem ser usadas para representar uma grande variedade de sistemas (YANO *et al.*, 2011).

¹Modelos executáveis para especificação de requisitos são usados para produzir um protótipo de um sistema na sua fase inicial de desenvolvimento.

Os primeiros trabalhos estudando MEF são da década de 50, e atividades dos anos 60 e início dos 70 foram motivados, principalmente, pela teoria dos autômatos e teste de circuitos sequenciais (LEE; YANNAKAKIS, 1996). Na metade da década de 90, o estudo de MEF foi motivado pelas aplicações em testes de conformidade de comunicação de protocolo (LEE; YANNAKAKIS, 1996). A primeira tese de Ph.D. na área foi de (STHAMER, 1995), que estudou o uso de Algoritmos Genéticos (AG) para geração de dados de testes estruturais (LEE; YANNAKAKIS, 1996). "AG mostraram bons resultados na busca do domínio de entrada para os requisitos de teste, mas outras heurísticas têm de tudo para serem, talvez, melhores soluções", frase que foi escrita por (STHAMER, 1995), abrindo caminho para a implementação de outras estratégias para a automação da geração de dados de teste.

A geração automática de teste pode ser abordada como um problema de *otimização* (BUZZO *et al.*, 2010). Otimização é uma técnica que visa procurar o conjunto de parâmetros para os quais uma função objetivo tem um valor máximo ou mínimo (CLARKE *et al.*, 2003). Ao invés de uma enumeração exaustiva do espaço de estados ², softwares de *testes baseados em busca* ³ utilizam meta-heurísticas para encontrar dados de teste (YANO *et al.*, 2011). Determinar os dados de teste é um problema indecidível, e utilizar meta-heurística é uma maneira de lidar com esse problema. (BUZZO *et al.*, 2010) enfatiza o uso de uma meta-heurística para encontrar os dados, dizendo que a utilização de dados aleatórios pode não explorar completamente o comportamento do software. A não utilização de dados

²A enumeração exaustiva do espaço de estados seria um problema se o domínio de entrada fosse muito grande, tornando o teste exaustivo não prático.

³ Meta-heurísticas que utilizam técnicas de busca são um conjunto de algoritmos genéricos que estão preocupados com a busca de soluções (ótimas ou próximas de ótimas) para um determinado problema com um grande espaço de busca *multimodal* (quando há mais de um ponto máximo para uma determinada função (ZANUSSO, 2001)) (CLARKE *et al.*, 2003).

aleatórios para teste de software já é trabalhada desde o início dos estudos em geração automática de dados de teste, e avaliações da utilização de dados aleatórios para teste de software podem ser encontradas em (DURAN; NTAFOSS, 1984).

Infelizmente não é possível abordar a geração de dados de teste baseada em teste de caixa branca, de forma determinística utilizando, por exemplo, programação linear, pois esses algoritmos frequentemente possuem restrições e funções complexas (BUZZO *et al.*, 2010). Um exemplo é a geração automática de casos de teste para *Systems Under Test* (SUT): o uso do critério que visa cobrir todos os caminhos é inviável, devido a possível presença de *loop*, que levaria a um número infinito de caminhos (BUZZO *et al.*, 2010). Ainda assim, a geração automática de dados de teste possui características que permitem tratá-la como um problema de otimização, como observado, por exemplo, nos trabalhos de (GROSS, 2000)(AGUILAR-RUIZ *et al.*, 2001)(YANO *et al.*, 2011)(BUZZO *et al.*, 2010).

Alguns trabalhos utilizam algoritmos evolutivos (AE) como meta-heurística para geração de dados de teste. (HARMAN; JONES, 2001) afirma que a otimização baseada em algoritmos evolutivos é ideal para a Engenharia de Software (ES) e, segundo (MANTERE; ALANDER, 2005), que faz uma revisão de trabalhos da área de automação de teste utilizando AG e AE até o ano de 2005, não há nenhum trabalho que contradiz essa afirmação. (CLARKE *et al.*, 2003) incentiva o uso de meta-heurísticas na ES, dizendo que se existisse apenas uma solução para um conjunto típico de restrições da ES, então a ES não poderia ser considerada uma "Engenharia" como um todo. (CLARKE *et al.*, 2003) ainda complementa dizendo que os engenheiros de software encaram problemas que consistem em não somente encontrar a solução, mas sim encontrar uma solução aceitável ou perto de ótima para um grande número de alternativas.

A técnica de teste de software evolutivo permite que dados de teste sejam gerados automaticamente através do uso de algoritmos de busca (MCMINN; HOLCOMBE, 2003). Segundo (CLARKE *et al.*, 2003), as chaves para qualquer problema de otimização baseada em busca são:

- a escolha da representação do problema;
- a definição da *função de fitness*⁴.

Dentre os algoritmos evolutivos, uma das técnicas recentes é o *Generalized Extremal Optimization* (GEO) e suas implementações. Essa meta-heurística tem demonstrado ser simples por possuir apenas uma variável de ajuste para configurar seu algoritmo ao problema proposto (BUZZO *et al.*, 2010).

(YANO *et al.*, 2011) trata em seu trabalho apenas da *análise de dependência*⁵, utilizando uma abordagem multi-objetiva chamada MOST, que é uma generalização do GEO. MOST utiliza uma versão executável de um modelo de MEFÉ para desenvolver a solução obtida a partir do uso da meta-heurística (YANO *et al.*, 2011). No trabalho de (YANO *et al.*, 2011), o objetivo é cobrir um determinado propósito de teste que, no caso do MOST, representa uma determinada transição dentro da MEFÉ. Muitas abordagens que para análise de dependência são baseadas em grafos de controle de fluxo, que satisfazem um único nó terminal, o que não é totalmente aplicável à MEFÉ, que pode ter múltiplos ou nenhum nó terminal (YANO *et al.*, 2011).

⁴A Função de Fitness mede o quanto um indivíduo está adaptado ao meio (ou ambiente) onde ele está inserido.

⁵A análise de dependência é usada para lidar com o problema da perda de informações, e com a falta de controle das variáveis internas (ou variáveis de contexto) do MEFÉ utilizando o algoritmo de busca.

O Modelo MOST utiliza uma otimização multi-objetiva, que busca um balanceamento entre o menor *caminho*⁶ da sequência de entrada e a cobertura de teste proposta (YANO *et al.*, 2011). No trabalho de (YANO *et al.*, 2011), o objetivo não é somente satisfazer um determinado critério de teste, mas também para permitir usuários selecionarem casos de teste menores, quando pretende-se reduzir os custos de execução. O MOST não encontra somente o caminho que caracteriza o caso de teste, mas também os dados que marcam esse caminho.

1.1 Motivação

As principais abordagens para a geração automática de casos de teste são: aleatória, simbólica e dinâmica (MICHAEL *et al.*, 2001) (YANO, 2011). Na *geração aleatória*, a probabilidade de selecionar dados de entrada que pertençam a uma pequena porcentagem do domínio de entrada é *muito baixa* (EDVARDSSON, 1999), mas por ser uma abordagem simples e de fácil implementação (em comparação com as outras abordagens citadas), é geralmente usada na literatura como medida de comparação (YANO, 2011). A *execução simbólica* funciona atribuindo valores simbólicos às variáveis reais. Dessa forma, essa execução permite que o problema de geração de dados de teste possa ser reduzido à resolução de expressões algébricas, que envolvem um conjunto de restrições de *igualdade* e *desigualdade* sobre esses valores simbólicos (YANO, 2011). Técnicas de otimização usadas na geração de dados de teste funcionam otimizando uma entrada de teste de acordo com o *critério de teste* expresso em uma *função objetivo* (YANO, 2011). Quando um critério de teste não é satisfeito, técnicas de otimização de funções são utilizadas para buscar dados que mais se aproximam de satisfazer o requisito. Com

⁶Um caminho é uma sequência de nós que, para cada par de nós consecutivos (n_j, n_{j+1}) no caminho há uma borda de n_j para n_{j+1} .

base nessas informações, os dados de teste são incrementalmente modificados até que um deles satisfaça o requisito. Dessa forma, a geração de dados de teste é reformulada como um problema de otimização. Um exemplo seria a busca de dados para a cobertura de instruções, em que uma função objetivo é definida para informar se o dado gerado está próximo ou não de executar a instrução desejada.

"Foram propostos trabalhos SBST multiobjetivo no contexto de teste que focam diferentes propósitos, mas a geração de casos de teste a partir de MEFE ainda não foi abordado"(YANO, 2011). Dessa forma, estudar e implementar uma ferramenta para a execução do processo da geração de dados de teste utilizando abordagem evolutiva e baseada em MEFE é um trabalho que soma informações de um novo método da área de teste de software, e abrange o processo de teste de software a problemas variados. A modelagem em MEFE consegue abranger problemas reais, que vão de problemas da engenharia à problemas de desenvolvimento de software, onde se precisa especificar, por exemplo, um algoritmo e seus passos. Dessa forma, uma ferramenta que considera as informações de uma MEFE e consegue gerar dados de teste para esse modelo, utilizando uma técnica nova (heurística evolutiva, considerando a multi-objetividade e baseada em MEFE), proposta em Julho de 2011, é muito interessante. Se um problema pode ser modelado em MEFE, então a implementação realizada neste trabalho pode ser utilizada para esse problema, e gerar dados de teste para o mesmo.

1.2 Objetivos

O presente trabalho visa o estudo de um novo processo, proposto por (YANO, 2011), para a geração automática de dados de teste, e a implementação de algoritmos, também propostos por (YANO, 2011), baseados em MEFE e fundamenta-

dos em uma heurística evolutiva. Foram implementados os algoritmos M-GEO_{vst} e MOST, para geração de dados de teste, baseando-se em uma heurística evolutiva e tratando a multi-objetividade, preocupando-se com a geração de dados de tamanhos variados, com a flexibilidade (poderá ser usado em problemas que contenham variáveis contínuas, reais e inteiras) e com a qualidade dos dados de teste, que são gerados a partir de um sistema ou problema modelado em MEFÉ. Depois, foram analisados os dados gerados a partir dos algoritmos implementados, e uma comparação com os dados do trabalho de (YANO, 2011) foi realizada. O trabalho futuro *Especificar e desenvolver uma ferramenta que ofereça apoio a abordagem MOST*, proposto por (YANO, 2011) foi estudado, e sua viabilidade foi avaliada.

1.3 Estrutura do Texto

A seção Referencial Teórico (seção 2) explica os conceitos e algoritmos presentes neste trabalho. Está presente nessa seção o referencial para Teste de Software (Seção 2.1), em especial Testes de Caixa Branca (seção 2.1.1). Também são mostradas características de Algoritmos Evolutivos (seção 2.2), e apresentados os algoritmos implementados e estudados neste trabalho (seções 2.2.6 e 2.3). Ainda na seção Referencial Teórico, são explicadas as estruturas MFE (seção 2.4.1) e MEFÉ (seção 2.4.2). A seção Resultados (seção 4) explica a implementação dos algoritmos realizada neste trabalho, apresenta dados de execuções e expõe resultados encontrados e suas explicações. No final, o trabalho é comentado e concluído com a seção Conclusão (seção 5) e, depois, trabalhos futuros são propostos na seção Trabalhos futuros (seção 6).

2 REFERENCIAL TEÓRICO

2.1 Teste de Software

Verificação e Validação (V & V) são processos empregados no desenvolvimento de um software (BUZZO *et al.*, 2010). O primeiro diz respeito às atividades que certificam que um software implementa corretamente uma determinada função. Já a validação refere-se a atividades que confirmam se o software atende os requisitos do usuário. Essas duas atividades são, geralmente, contrastadas com uma outra atividade, chamada *avaliação* que, segundo (WAINER, 2007), é o processo de verificar para que serve e quanto serve um sistema. Ainda segundo (WAINER, 2007), a diferença entre os termos surge por causa da existência de diferenças, mesmo que pequenas, entre o problema que o sistema resolve e o problema real, ou então pelo surgimento de novos problemas a partir da resolução do problema inicial. Uma especificação incompleta ou errada do problema para o qual o sistema foi implementado também é fator de diferenciação para os termos verificação, validação e avaliação. Esses processos constituem a área de teste, e têm como objetivo revelar falhas do SUT, em especial aquelas que impedem que requisitos do sistema sejam atendidos (SOMMERVILLE, 2003).

No processo de desenvolvimento de software, durante a fase de teste, faz-se o uso de várias técnicas. A técnica de caixa branca é utilizada em testes de unidades ⁷ do sistema, ou seja, há o acesso direto ao código fonte, permitindo assim, o exercício dos caminhos de controle e de estrutura (INFOTECH, 1979).

⁷Uma unidade é a menor parte testável de um programa de computador. Em programação procedural, uma unidade pode ser uma função individual ou um procedimento. Idealmente, cada teste de unidade é independente dos demais, o que possibilita ao programador testar cada módulo isoladamente.

Para o teste de integração ⁸ são utilizadas técnicas de caixa preta (HOWDEN, 1976), focadas na construção da arquitetura do software. É também possível a utilização de técnicas de caixa branca para a validação das estruturas de controle (PRESSMAN; MARIA, 2006). PRESSMAN e MARIA (2006) dizem ainda que para os testes de validação, as técnicas de caixa preta são as mais utilizadas.

2.1.1 Testes de Caixa Branca

Os testes de caixa branca são aqueles em que se tem acesso à implementação e à estrutura do software (BUZZO *et al.*, 2010). A partir destas informações é possível derivar conjuntos de testes de forma a exercitar estas estruturas. O objetivo é testar a estrutura interna do software e desenvolver casos de teste baseados na estrutura lógica interna do software (PEDRYCZ; PETERS, 2001), fazendo com que cada caminho lógico seja testado pelo menos uma vez (JACOBSON *et al.*, 1997). Por esse motivo estes tipos de testes geralmente são aplicados a pequenos módulos ou funções do software.

Uma forma de representar a estrutura do módulo para que o projetista possa derivar testes é através de *grafos de controle de fluxos* ⁹. Estes grafos direcionais permitem uma visualização estrutural da lógica do trecho de código (BUZZO *et al.*, 2010). Um exemplo de grafo direcional é mostrado na Figura 1.

(CLARKE *et al.*, 2003) explica três tipos de cobertura de grafos de controle de fluxo:

⁸Teste de integração é a fase do teste de software em que módulos são combinados e testados em grupo. Ela sucede o teste de unidade, em que os módulos são testados individualmente, e antecede o teste de sistema, em que o sistema completo (integrado) é testado num ambiente que simula o ambiente de produção.

⁹Grafo de controle de fluxo é uma representação que usa notação de grafo para descrever todos os *caminhos* que podem ser executados por um programa de computador (BUZZO *et al.*, 2010)

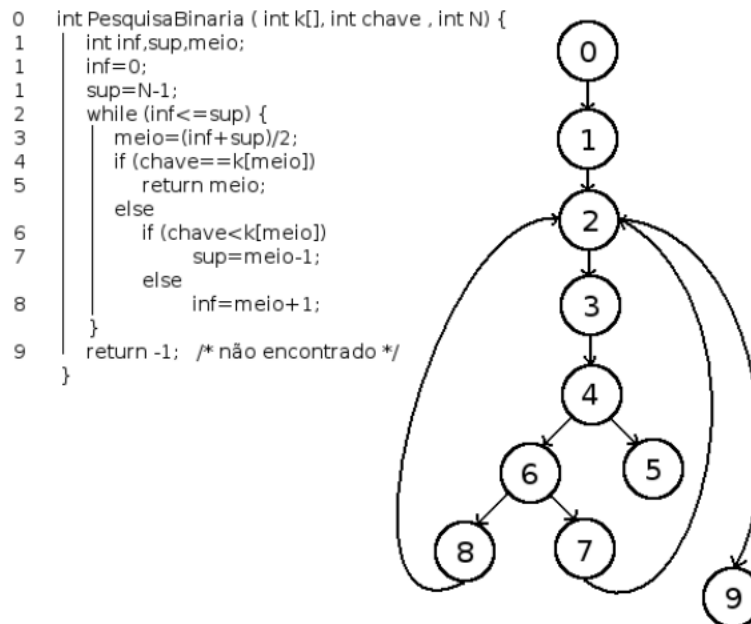


Figura 1: Grafo de controle de fluxo do algoritmo Busca Binária (BUZZO *et al.*, 2010)

- Cobertura de alcance¹⁰: a proporção das afirmações alcançáveis que são alcançadas durante o processo de teste;
- Cobertura de ramo¹¹: a proporção dos ramos executáveis, que são alcançados durante o processo de teste;
- Cobertura de caminho¹²: a proporção de caminhos viáveis, que são cobertos durante o processo de teste.

Testes de caixa branca podem ser aplicados desde o início do processo de desenvolvimento do software. São exemplos de testes de caixa branca, segundo (FRANZEN; BELLINI, 2005):

¹⁰Mesmo no Brasil, o termo mais usado para definir esse tipo de cobertura é *statement coverage*

¹¹Mesmo no Brasil, o termo mais usado para definir esse tipo de cobertura é *branch coverage*

¹²Mesmo no Brasil, o termo mais usado para definir esse tipo de cobertura é *path coverage*

- teste de caminho básico ou abrangência de instrução;
- abrangência de ramificação/condição;
- abrangência de caminho;
- teste de caminho em grafos com loop;
- e teste de estrutura de controle.

(PRESSMAN; MARIA, 2006) apresenta três variações de testes de estrutura de controle:

- teste de condição;
- teste de fluxo de dados;
- e teste de laços.

Técnicas de caixa branca normalmente consideram elementos específicos e medem a proporção desses elementos dentro do código fonte, tudo durante o teste (CLARKE *et al.*, 2003). A essa proporção, é dada o nome de *cobertura*.

A lógica desses métodos é que fica impossível detectar uma falha em alguma parte do código se essa parte nunca é executada (OSTRAND, 2011). Logo, um teste que executa todos os tipos de cobertura e consegue cobrir a maior parte de um SUT, será um teste bem realizado.

2.1.2 Teste baseado em modelos

O processo de teste baseado em modelos envolve as seguintes atividades, que são ilustradas na Figura 2 (YANO, 2011) (UTTING; LEGEARD, 2007) :

1. Construir um modelo genérico que represente o comportamento do sistema em teste a partir dos documentos de especificação. Deve-se focar nos principais aspectos a serem testados, omitindo-se os detalhes do sistema em teste. É importante também validar o modelo, uma vez que a geração de teste tem como base o próprio modelo.
2. Gerar os casos de teste abstratos a partir do modelo. O critério de teste pode ser relacionado a funcionalidades do sistema ou estrutura do modelo, tais como cobertura de estados e transições.
3. Transformar os casos de teste abstratos em executáveis. O objetivo desse passo é obter casos de teste em nível de implementação do sistema, adicionando detalhes específicos, por exemplo, da linguagem de programação que não são abordados no modelo. Ao mudar as regras de transformação do **adaptor**, pode-se reusar o mesmo conjunto de casos de teste em diferentes plataformas de execução.
4. Executar os casos de teste no sistema em teste e atribuir um veredito: passou, não passou ou inconclusivo. Um teste passa quando se obtém a saída esperada, caso contrário, é revelada a presença de falha. O teste é inconclusivo quando a decisão sobre o teste não pode ser tomada.
5. Analisar os resultados dos testes e tomar as ações corretivas. Quando o teste não passa, é preciso verificar se a falha encontra-se no sistema em teste ou no próprio caso de teste, visto que pode-se ter falhas no adaptador, no modelo ou até mesmo na documentação.

A abordagem MBT permite que a equipe de testes utilize os mesmos modelos criados pelos desenvolvedores para a criação de casos de teste que cubram varia-

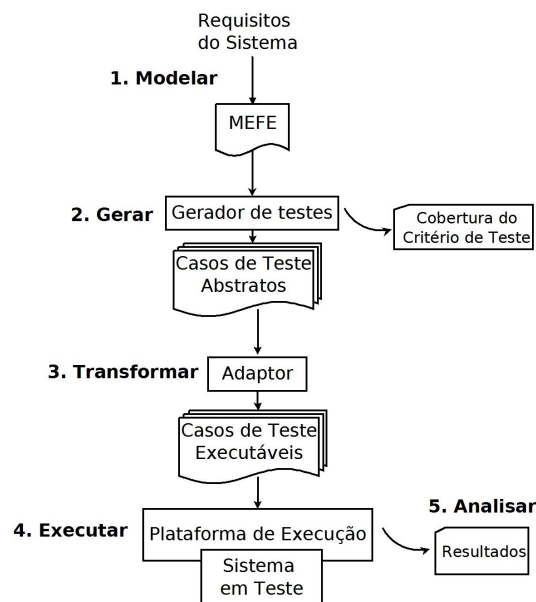


Figura 2: Processo de teste baseado em modelo (YANO, 2011)

dos cenários de uso da aplicação. Além disso, os casos de teste podem ser obtidos mesmo quando o código fonte ainda não está, ou não é disponível. Uma outra vantagem é que os testes podem ser produzidos uma única vez e serem utilizados em diferentes linguagens de programação e plataformas. Em caso de modificações no comportamento do sistema em teste, basta atualizar o modelo e gerar novamente os casos de teste de maneira automática, facilitando a manutenção dos testes (ROBINSON, 2000) (UTTING; LEGEARD, 2007) (UTTING *et al.*, 2011).

Uma das limitações de MBT é que requer habilidade para modelar o sistema, implicando em custos de treinamento e uma curva inicial de aprendizagem. O modelo precisa ser mais simples do que o sistema em teste, senão o esforço de validar o modelo poderia ser equivalente ao de testar diretamente o sistema. Por outro lado, o modelo precisa ser suficientemente preciso para poder gerar casos de teste a partir do mesmo. Há ainda a necessidade de mecanismos de transformar os

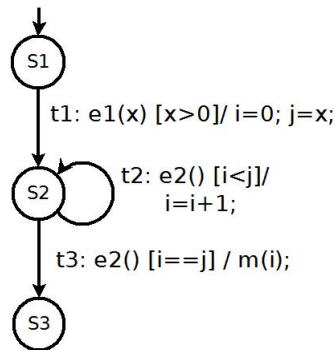


Figura 3: Exemplo de caminho inactível em uma MEFE: conflito de dados na ação de t_1 com a guarda de t_3 (YANO, 2011)

casos de teste abstratos para o nível de implementação da plataforma de execução do sistema. A geração de casos de teste também não abrange as partes omitidas na abstração do modelo (UTTING; LEGEARD, 2007) (YANO, 2011).

Em particular, no teste baseado em MEFE, a geração de casos de teste consiste em buscar caminhos de transições, assim como os dados para exercitá-los. Devido aos dados envolvidos nas ações e guardas das transições das MEFEs, um dos problemas possíveis é a geração de *Caminhos inactíveis*. Um caminho pode ser sintaticamente possível mas semanticamente impossível. Como uma transição precisa de sua guarda para ser disparada, a habilitação de t é restringida pelos valores atuais das variáveis e parâmetros de entrada. (YANO, 2011) mostra um exemplo de conflito de dados, mostrado na Figura 3, que mostra da ação de t_1 com a guarda de t_3 . Devido aos valores de i , x , e j , o caminho t_1t_3 é inactível. Note que t_3 é habilitada apenas após x execuções de t_2 . Assim, a obtenção de caminhos de transições factíveis depende não apenas das interações de eventos de entrada, mas também do histórico de transições executadas anteriormente (WANG, 1990).

A geração de dados de teste baseado em modelos não exige que o código fonte do sistema em teste seja disponível, e também é independente de linguagem de programação e plataforma (YANO, 2011).

A Tabela 1 (YANO, 2011) descreve uma visão geral de algumas soluções baseadas em MEFE, verificando se abordam o fluxo de controle e de dados das MEFES e o problema da geração de caminhos inactíveis. Na abordagem MOST, proposta por (YANO, 2011), uma técnica de otimização baseada em busca é utilizada para gerar os casos de teste a partir de MEFE, assim como os dados envolvidos nos caminhos. A fim de evitar caminhos inactíveis, é utilizado um modelo executável da MEFE que produz apenas caminhos factíveis, pelo menos no nível de modelo.

Neste trabalho, o foco é dado à geração automática de dados de teste, sendo que as posteriores etapas do sistema MBT estão fora do escopo.

2.2 Algoritmos Evolutivos

"Um importante precursor da aplicação de meta-heurísticas na ES é a habilidade de caracterizar a ES como um problema de busca"(CLARKE *et al.*, 2003). (CLARKE *et al.*, 2003) diz ainda que o sucesso da geração automática de dados de teste se deve, em parte, ao fato de que esse problema tem uma grande e natural característica de busca, e possui uma *função de fitness* que pode ser facilmente definida.

Algoritmos evolutivos, ou computação evolutiva, constituem um conjunto de meta-heurísticas baseados nos conceitos de evolução natural e genética (BUZZO *et al.*, 2010) (EIBEN; SMITH, 2003) (MICHALEWICZ; SCHOENAUER, 1996). Uma de suas maiores aplicações é como modelo de otimização global (BUZZO

Abordagem	Fluxo de controle	Fluxo de dados	Caminhos factíveis
(SARIKAYA <i>et al.</i> , 1987)	✓	✓	✗
(URAL; YANG, 1991)	✗	✓	✗
(CHANSON; ZHU, 1994)	✓	✓	✗
(MARTINS <i>et al.</i> , 1999)	✓	✓	✗
(BOURHFIR <i>et al.</i> , 1996)	✓	✓	✓
(DUALE; UYAR, 2004)	✓	✓	✓
(HIERONS <i>et al.</i> , 2004)	✓	✓	✓
(RAMALINGOM <i>et al.</i> , 2003)	✓	✓	✓
(LI <i>et al.</i> , 1999)	✓	✗	✓
(CAVALLI <i>et al.</i> , 1999)	✓	✓	✓
MOST	✓	✓	✓

Tabela 1: Trabalhos para geração de casos de teste baseados em MEFE (YANO, 2011)

et al., 2010). AE podem tratar qualitativamente diferentes tipos de variáveis, tais como variáveis discretas, contínuas, reais e inteiras (AMORIM *et al.*, 2009). Essa característica torna AE uma técnica interessante para ser usada em várias áreas das ciências e engenharias no tratamento de problemas de otimização.

A maioria dos problemas pertencentes ao mundo real é intratável, pois "são problemas para os quais é improvável que se consiga desenvolver métodos analíticos que possam preservar as características físicas e detalhes dos problemas do mundo real"(AMORIM *et al.*, 2009). Sendo assim, tratando-se de otimização, deve ser aplicado a um determinado problema uma técnica de solução que considere a qualidade e precisão da solução que se necessita, bem como a sofisticação do modelo do problema do mundo real (AMORIM *et al.*, 2009).

Segundo (ZITZLER *et al.*, 2000), depois dos estudos em otimização evolutiva lançados na década de 80 (SCHAFFER, 1985) (FOURMAN, 1985), diferentes implementações de AE foram propostas no início da década de 90 (KURSAWE, 1991) (HAJELA; LIN, 1992) (FONSECA *et al.*, 1993) (HORN *et al.*, 1994) (SRINIVAS; DEB, 1994). Depois, essas abordagens e suas variações foram aplicadas à vários problemas de otimização multi-objetiva (ISHIBUCHI; MURATA, 1996) (CUNHA *et al.*, 1997) (VALENZUELA-RENDÓN *et al.*, 1997) (FONSECA; FLEMING, 1998) (PARKS; MILLER, 1998).

Mais tarde, alguns pesquisadores investigaram algumas particularidades da busca evolutiva multi-objetiva, por exemplo, a convergência do *Ótimo de Pareto front* (VELDHUIZEN; LAMONT, 1998) (RUDOLPH, 1998), enquanto outros pesquisadores se concentraram no desenvolvimento de novas técnicas evolutivas (LAUMANNNS *et al.*, 1998) (ZITZLER; THIELE, 1999).

Dentre os subgrupos dos algoritmos evolutivos disponíveis na literatura, existem as Estratégias Evolutivas (EE), Programação Evolutiva (PE), Programação Genética (PG) e os Algoritmos Genéticos (EIBEN; SMITH, 2003). Todos possuem características comuns: *evoluir* uma população em direção a um objetivo, enquanto os indivíduos sofrem *seleção, recombinação e mutação* (BUZZO *et al.*, 2010). Os indivíduos que estão melhor adaptados ao ambiente são favorecidos, podendo sobreviver e evoluir. Segundo (ABREU, 2006), as principais diferenças dos trabalhos que utilizam AG na literatura são:

- implementações diferentes do AG, sendo que cada processo interno (seleção, crossover e mutação) pode ser implementado utilizando estratégias diferentes;
- critério de teste, que é normalmente avaliado pelo conjunto de dados gerados na análise de cobertura de uma determinada característica do programa (instruções, caminhos, decisões);
- e a *função objetivo*, que avalia a aptidão (ou qualidade) de cada indivíduo (ou candidato) à solução, sendo considerada a guia para o AE na geração de dados de teste.

(BUZZO *et al.*, 2010) mostra um modelo geral de um algoritmo evolutivo, modificado de (MICHALEWICZ; SCHOENAUER, 1996), na Figura 4.

(CLARKE *et al.*, 2003) explica os algoritmos evolutivos como sendo uma busca por soluções ótimas através da amostragem de um espaço de busca de forma aleatória e criando um conjunto de soluções candidatas, chamado de *população*.

Um algoritmo evolutivo mantém uma População(t) = { x_1^t, \dots, x_n^t } de indivíduos por iteração (ou geração) (BUZZO *et al.*, 2010). Cada indivíduo é uma potencial

```

t ← 0
init Pop(t)
avale Pop(t)
while condição de parada não satisfeita do
  t ← (t+1)
  selecione Pop(t) a partir de Pop(t-1)
  altere Pop(t)
  avale Pop(t)
end while

```

Figura 4: Modelo geral de Algoritmo Evolutivo (BUZZO *et al.*, 2010) (MICHALEWICZ; SCHOENAUER, 1996)

solução para o problema trabalhado, sendo que sua estrutura e codificação são dependentes do tipo do algoritmo evolutivo implementado (BUZZO *et al.*, 2010). De acordo com o processo de evolução *Neo-Darwinista*, os mais aptos (possíveis melhores soluções) sobrevivem, podendo passar suas características para a nova geração. Cada indivíduo possui uma medida de adaptação, indicando sua potencialidade para solucionar o problema (BUZZO *et al.*, 2010), que é a o valor da medida da função objetivo do problema em questão. Através de diversas interações, que são terminadas a partir de critérios de parada ¹³, uma nova população é criada ao se executar *operadores genéticos* sobre a antiga população (BUZZO *et al.*, 2010). Segundo (BUZZO *et al.*, 2010), esses operadores incluem:

- **seleção:** através de um determinado mecanismo, alguns indivíduos são selecionados (pode-se privilegiar os mais adaptados).
- **alteração:** pode ser uma modificação pequena em um atributo do indivíduo (mutação) ou através da combinação de dois ou mais indivíduos (recombinação).

¹³Os critérios de parada podem ser variados. Dentre os vários critérios, pode-se parar ao encontrar uma solução que atenda aos requisitos ou parar ao atingir o número máximo de iterações permitidas.

Características novas podem aparecer entre os indivíduos durante cada iteração, produzindo diversidade na população (LIBRALÃO; DELBEM, 2005).

A evolução é um processo que conduz, probabilisticamente, populações em direção a picos da superfície¹⁴, enquanto a seleção elimina variantes menos apropriadas (ZUBEN, 2000), que não serão, provavelmente, soluções viáveis.

Os principais aspectos que diferenciam os tipos de algoritmos evolutivos são (BUZZO *et al.*, 2010):

- tipo de codificação dos indivíduos;
- processo de seleção dos indivíduos;
- como e quando os operadores genéticos são empregados;
- e a maneira como a população inicial é criada.

A utilização de algoritmos evolutivos precisa ser feita de maneira cuidadosa. São muitos fatores impactantes, tais como qual algoritmo utilizar e quais operadores genéticos escolher. Deve-se escolher os parâmetros e probabilidades do algoritmo para que ele encontre, de forma eficiente, a solução (BUZZO *et al.*, 2010). Autores mostram em seus trabalhos que a escolha desses parâmetros impactam drasticamente no desempenho do algoritmo e, até mesmo, na qualidade do resultado (MICHALEWICZ; SCHOENAUER, 1996) (GOLDBERG, 1989) (BUZZO *et al.*, 2010).

(OLIVEIRA, 2004) cita alguns méritos de AE, justificando o uso crescente dessa meta-heurística:

¹⁴Um pico da superfície é um máximo local. Essa nomenclatura é utilizada por (ZUBEN, 2000) para explicar a computação evolutiva utilizando os conceitos de *genótipo* e *fenótipo*, que não serão tratados por fugirem do escopo deste trabalho.

- são facilmente implementáveis;
- são modulares e facilmente adaptáveis a qualquer tipo de problema, mesmo os mais complexos;
- são métodos de otimização global, mais robustos a ótimos locais;
- necessitam de pouca informação sobre o problema, basicamente, custo ou ganho de cada possível solução;
- podem encontrar soluções aproximadas, de boa qualidade e, até mesmo, ótimas;
- podem otimizar um grande número de parâmetros discretos, contínuos ou combinações deles;
- realizam buscas simultâneas em várias regiões do espaço de busca (paralelismo inerente);
- podem ser adaptados para encontrar várias soluções ótimas;
- podem ser eficientemente combinados com outras heurísticas.

Alguns problemas não são muito bem aplicáveis à AE. Dentre as principais de AE, tem-se (OLIVEIRA, 2004):

- podem ser bem mais lentos que outros métodos por trabalharem com uma população de soluções;
- dificuldade para se definir um mecanismo de codificação e avaliação de indivíduos que seja apropriado;

- dificuldade para se ajustar os valores de seus parâmetros de desempenho adequadamente;
- fracos mecanismos de intensificação de busca, ocasionando baixas taxas de convergência.

Se comparada à outras heurísticas de busca que trabalham com melhoramentos sucessivos em uma única solução (métodos não populacionais), como o *simulated annealing*¹⁵ e a busca tabu (*tabu search*), AE pode ser um método mais lento (OLIVEIRA, 2004). Ainda segundo (OLIVEIRA, 2004), parte dessa má fama dos AE se parte de sua utilização em problemas que poderiam ser resolvidos satisfatoriamente por métodos exatos ou por meio de heurísticas computacionalmente mais leves. "Velocidade de execução costuma ser o preço que se paga por uma maior robustez"(OLIVEIRA, 2004).

Um dos parâmetros mais importantes na computação evolutiva é a codificação (YANO *et al.*, 2011) (BUZZO *et al.*, 2010) (OLIVEIRA, 2004) e, por isso tem sido uma relevante área de pesquisa dentro da computação evolutiva (OLIVEIRA, 2004).

Trabalhos anteriores (BRUNO T. ; MARTINS, 2005) (HARMAN, 2007) (SY; DEVILLE, 2001) demonstraram utilizar meta-heurísticas para gerar dados de teste automaticamente é mais confiável do que utilizar processos manuais ou valores aleatórios (BUZZO *et al.*, 2010). Baseando nesses resultados, a implementação de uma ferramenta utilizando uma abordagem evolutiva, segundo esses trabalhos, será mais confiável. Essa foi a proposta de (YANO *et al.*, 2011): abordagem evolutiva na geração automática de dados de teste, baseado em MEFÉ.

¹⁵Será usada a expressão em inglês, por falta de uso dessa expressão na lingua portuguesa.

No contexto de otimização, o desempenho do algoritmo está diretamente relacionado à escolha da função objetivo e do seu domínio de busca (BUZZO *et al.*, 2010). A utilização de algoritmos evolutivos para a geração automática de dados de testes é uma área passível de pesquisa por novas maneiras de melhorar o desempenho em determinados tipos de problemas. Alguns trabalhos sobre testes utilizando abordagens evolutivas são mostrados na Tabela 2.

Um problema geral de otimização multi-objetiva pode ser descrito como um vetor de funções f que mapeia uma tupla de m parâmetros (variáveis de decisão) para uma tupla de n objetivos (ZITZLER; THIELE, 1998). Formalmente, segundo (ZITZLER; THIELE, 1998):

$$\min \setminus \max y = f(x) = (f_1(x), f_2(x), \dots, f_n(x))$$

sujeito à

$$x = (x_1, x_2, \dots, x_m) \in X$$

$$y = (y_1, y_2, \dots, y_n) \in Y$$

onde x é chamado de *vetor de decisão*, X representa os *parâmetros do espaço de busca*, y é o *vetor objetivo* e Y é o *espaço objetivo* de busca.

Um problema de otimização multi-objetiva é caracterizado pela otimização simultânea de várias funções objetivo com diferentes soluções ótimas (GOLDBERG, 1989). Geralmente, as funções objetivo de um problema de otimização multi-objetivo são não-comensuráveis e conflitantes entre si e por esta razão não existe uma única solução que seja simultaneamente ótima para todos os objetivos, e sim um conjunto de soluções denominado conjunto eficiente ou Ótimo de Pareto (AMORIM *et al.*, 2009).

Método	Método de Otimização Principal	Função Objetivo	Critério de Teste
(PARGAS <i>et al.</i> , 1999)	AG	Avaliação da semelhança entre predicados cobertos	Instruções e Predicados
(BUENO; JINO, 2002)	AG	Semelhança de caminhos e penalidade	Caminhos
(WEGENER; BÜHLER, 2004)	AG	Área/distância entre carro e área de colisão	Caminhos
(ABREU, 2006)	GEO	Semelhança de caminhos	Caminhos
(WATKINS; HUF-NAGEL, 2006)	AG	Probabilidade inversa do caminho	Caminhos
(MANSOUR; SALAME, 2004)	AG	Semelhança de caminhos e penalidade associada aos predicados	Caminhos
(LAKHOTIA <i>et al.</i> , 2007)	AG	Multi-objetivo - distância e nível de aproximação	Predicados
(ALBA; CHICANO, 2006)	EE	Avaliação de funções associadas aos predicados	Condicionais
(WINDISCH <i>et al.</i> , 2007)	PSO	Avaliação de funções associadas aos predicados	Predicados
(MICHAEL <i>et al.</i> , 2001)	AG	Avaliação de funções associadas aos predicados	Condições e decisões

Tabela 2: Trabalhos sobre testes evolutivos (BUZZO *et al.*, 2010)

A abordagem multi-objetiva do M-GEO_{vsl} permite o ajuste automático do tamanho da solução para cobrir o propósito de teste (YANO *et al.*, 2011). Essa é uma característica que difere muito essa abordagem de outras abordagens, pois nesses trabalhos o tamanho deve ser dado pelo usuário antes da execução e, em alguns casos, o algoritmo não consegue achar solução para o tamanho passado (YANO *et al.*, 2011).

O conjunto de soluções de um problema de otimização multi-objetiva é composto por todos os vetores de decisão para os quais seus correspondentes vetores de objetivo não podem ser melhorados em uma determinada dimensão κ , sem ser degradado em outra dimensão κ' (ZITZLER; THIELE, 1998).

2.2.1 Exemplos da aplicação de AE

Há diferentes abordagens utilizando AE. A tese de doutorado de (GROSS, 2000) concentra os estudos em testes temporais para sistemas de tempo real. O autor tenta medir a qualidade de testes evolutivos estudando a relação entre a complexidade dos objetos de teste e a qualidade do resultado produzido através dos testes evolutivos. Ele relata que não é muito diferente a maneira como os algoritmos evolutivos e os humanos enxergam a complexidade de uma rotina, ou seja, programas que são complexos para a análise humana, também serão complexos para testes evolutivos.

(AGUILAR-RUIZ *et al.*, 2001) utiliza algoritmos evolutivos para estimar dados de projetos de desenvolvimento de software através do uso de um simulador. Um Banco de Dados (BD) do projeto é gerado, e AE é usado para produzir regras que vão direcionar o desenvolvimento do software. O objetivo principal do trabalho de (AGUILAR-RUIZ *et al.*, 2001) é gerar regras que irão ajudar o pro-

jetista de software a manter o desenvolvimento do software dentro do tempo e do orçamento e, no final da implementação, ter um software de qualidade.

(ZITZLER; THIELE, 1998) apresenta um exemplo da presença da multi-objetividade na resolução de um problema real. O autor apresenta uma situação de construção de um sistema hardware/software complexo. Um design ótimo deve ser um design que minimiza custo e consumo de energia, enquanto maximiza a performance do sistema. São situações conflitantes, já que uma arquitetura alcança alta performance com altos custos, e uma arquitetura alternativa mais barata pode consumir muito mais energia. O problema então é descobrir o que é mais importante nesse projeto. (ZITZLER; THIELE, 1998) finaliza propondo uma ferramenta que explora o espaço de busca e encontra soluções Pareto-ótimas utilizando o ótimo de Pareto para design de software, em um tempo razoável.

2.2.2 Otimização Extrema Generalizada (GEO)

O algoritmo de Otimização Extrema Generalizada (*Generalized Extremal Optimization* - GEO)¹⁶ é uma meta-heurística que também se baseia no conceito de seleção natural (BUZZO *et al.*, 2010). Este algoritmo é uma generalização do método de Otimização Extrema, proposto por (BOETTCHER; PERCUS, 2001). Ambos os modelos tiveram seus processos internos inspirados na pesquisa de (BAK; SNEPPEN, 1993) que propõe a presença de *Criticalidade Auto Organizada* (*Self-Organized Criticality* - CAO) em certos ecossistemas (BUZZO *et al.*, 2010). A teoria CAO propõe que sistemas grandes e complexos tendem a evoluir para um *estado crítico*, onde uma mudança em único elemento pode produzir perturbações que atinjam qualquer número de elementos do sistema. Uma lei de potência des-

¹⁶Será chamado de GEO - Sigla para o termo em inglês - para manter o padrão com os trabalhos da área.

$$P(s) \approx s^{-\tau} \quad (1)$$

Figura 5: Equação da probabilidade de uma perturbação de tamanho s ocorrer (BUZZO *et al.*, 2010)

creve a probabilidade de uma perturbação de tamanho s ocorrer (BUZZO *et al.*, 2010):

onde τ é um número positivo. Uma análise dessa função mostra que há mais chance de ocorrer perturbações pequenas do que perturbações maiores, que poderiam afetar todo o sistema.

O modelo de Bak-Sneppen apresenta as espécies de um ecossistema alinhadas lado a lado de forma circular, mantendo uma relação de vizinhança umas com as outras (BUZZO *et al.*, 2010), como mostrado na Figura 6.

Cada espécie recebe um *valor arbitrário* do intervalo $[0,1]$, tal que valores mais baixos indicam espécies menos adaptadas. O sistema evolui forçando a evolução das espécies menos adaptadas do sistema (BUZZO *et al.*, 2010). A evolução dessas espécies menos adaptadas reflete nas espécies vizinhas, que também tenderão a evoluir, devido à presença de um novo competidor local. Este ciclo se repete de maneira que a aptidão média do ecossistema aumenta gradualmente (BUZZO *et al.*, 2010), até que, em um certo momento, todas as espécies possuirão aptidão superior a um *valor crítico*. Logo, tanto espécies bem adaptadas ao sistema quanto as poucas adaptadas são forçadas a mudar e, mesmo que não seja o ideal, algumas podem ficar abaixo do nível crítico. Essa perturbação no equilíbrio do sistema é chamada de *avalanche*, e segue a regra de potência descrita na equação (1) (BUZZO *et al.*, 2010). Este modelo possibilita a construção de uma heurística de otimização que pode encontrar soluções rapidamente, mudando sistematicamente

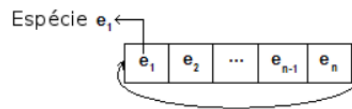


Figura 6: Espécies no modelo Bak-Sneppen (BUZZO *et al.*, 2010)

as espécies menos adaptadas e escapando de *mínimos locais* através das avalanches (BUZZO *et al.*, 2010). Essa característica de evitar mínimos locais do GEO, torna esse algoritmo uma boa escolha para se trabalhar com multi-objetividade, como será descrito na seção 2.2.5.

O algoritmo GEO foi criado para ser utilizado em grande parte de problemas com qualquer tipo de variável, seja contínua, discreta ou uma mistura delas (BUZZO *et al.*, 2010). No algoritmo GEO uma população é representada por uma sequência(*string*) de bits, onde cada bit é uma espécie (BUZZO *et al.*, 2010), o que difere esse algoritmo de alguns algoritmos genéticos, onde cada espécie é uma possível solução do problema, representada por uma sequência de bits.

(BUZZO *et al.*, 2010) mostra uma resumida descrição dos passos de um algoritmo GEO:

1. Inicialize aleatoriamente uma *string* de bits de tamanho L que codifica N variáveis do projeto.
2. Mude o bit k dentro da população e calcule o valor da função objetivo para esta nova população. Atribua ao *bit* um índice de adaptabilidade de acordo com o ganho ou perda adquirida com esta mutação em relação ao melhor valor da função objetivo encontrada até o momento (início do passo 2). Retorne o *bit* ao valor original.
3. Repita o passo 2 para todos os *bits* da população.

4. Ordene os *bits* de acordo com seus índices de adaptabilidade, de $k=1$ para o *bit* menos adaptado até $k=L$ para o mais adaptado. Ordene os *bits* aleatoriamente com distribuição uniforme caso dois ou mais tenham um índice de adaptabilidade igual.
5. Escolha com probabilidade uniforme um *bit* candidato para sofrer mutação. Gere um número aleatório RAN com probabilidade uniforme entre $[0,1]$. Calcule $P_i = k(-\tau)$, sendo k a posição de i e τ um valor ajustável. Caso P_i seja igual ou maior que RAN mude o (*bit*). Senão, repita o processo até que um *bit* seja modificado.
6. Repita os passos de 2 a 4 até que um critério de parada seja satisfeito.
7. Retorne a melhor solução (configuração de *bits*) encontrada.

A variável τ deve ser definida antes de se implementar o algoritmo GEO. Com apenas esta variável ajustável, o algoritmo pode se tornar tanto uma busca aleatória quanto uma busca local (BUZZO *et al.*, 2010). Quando $\tau \rightarrow 0$, qualquer candidato escolhido sofrerá mutação, caracterizando uma busca aleatória, mas quando $\tau \rightarrow \infty$, somente o primeiro bit sofrerá mutação, caracterizando uma busca local (BUZZO *et al.*, 2010). Com apenas uma variável de ajuste o GEO se destaca em relação a diversos outros algoritmos (BUZZO *et al.*, 2010), como por exemplo o SGA clássico (SMITH *et al.*, 1991), que necessita que o tamanho da população, a probabilidade de recombinação e a probabilidade de mutação sejam ajustados antes da execução do algoritmo. Trabalhos anteriores mostraram que o valor de τ que traz os melhores resultados geralmente está entre $[0,10]$ (BUZZO *et al.*, 2010).

Existem variações do algoritmo GEO, como por exemplo o GEOvar, onde os *bits* são ordenados separadamente para cada codificação de cada uma das variáveis do projeto. Existe também a variação chamada de GEOreal.

2.2.3 Otimização Extrema Generalizada com Codificação Real (GEOreal)

O GEOreal busca contornar uma limitação do algoritmo GEO, que ao utilizar a codificação binária obriga o implementador a definir a precisão (número de bits) das variáveis (BUZZO *et al.*, 2010). O algoritmo GEOreal segue praticamente o mesmo princípio básico do GEO, a principal mudança é que agora o valor da variável é atualizada criando-se uma *perturbação* através de um número aleatório com distribuição gaussiana (BUZZO *et al.*, 2010).

(BUZZO *et al.*, 2010) mostra os passos do GEOreal:

1. Inicialize aleatoriamente uma população de N espécies onde cada variável de projeto é representada por uma espécie.
2. Uma por vez, altere cada variável segundo a equação (2). x'_i é o novo valor da i -ésima variável e $N(0, s)$ é um número aleatório com distribuição gaussiana de média zero e desvio padrão s . Calcule V_i (valor da avaliação de i) e gere o par ordenado (i, V_i) . Armazene este par e retorne o valor de x_i para antes da perturbação.

$$x'_i = x_i + N(0, \sigma)x_i \quad (2)$$

3. Ordene as variáveis, de 1 a N , de acordo com V_i atribuindo um rank k para cada variável. A espécie menos adaptada recebe $k = 1$ e a melhor adaptada recebe $k = N$.
4. Escolha aleatoriamente com probabilidade $k^{-\tau}$ uma espécie. Esta será substituída pelo seu valor x'_i calculado no passo 3. Atualize a nova população formada pelas espécies anteriores e a espécie substituída.
5. Repita os passos 2 a 4 até que algum critério de parada seja atingido.
6. Retorne a melhor função objetivo encontrada e o conjunto de melhores variáveis.

2.2.4 Otimização Multi-objetiva

Uma das direções apontadas como trabalhos futuros para testes baseados em heurísticas, é a otimização multi-objetiva, visto que diversos problemas da engenharia de software envolvem múltiplos objetivos (HARMAN *et al.*, 2009). O problema multiobjetivo é apresentado formalmente como:

Um problema multi-objetivo minimiza $F(\vec{x}) = [F_1(\vec{x}), \dots, F_{nf}(\vec{x})]$ sujeito a $g_i(\vec{x}) \leq 0, i = 1, \dots, m$, onde $(\vec{x} = x_1, \dots, x_n)$ é um vetor n -dimensional de variáveis de projeto de algum universo Ω e nf é o número de funções objetivo.

Para tentar simplificar o problema, tenta-se combinar todos os objetivos em somente um, processo esse que é denominado *agregação de funções* (YANO, 2011). Pode-se combinar um problema com nf funções objetivo $F_1(x), \dots, F_{nf}(x)$ sobre algum vetor x de acordo com um conjunto de coeficientes c_1, \dots, c_{nf} :

Dessa maneira, o problema é tratado como um problema de *otimização mono-objetiva*, ou um problema de otimização simples. "Em geral, não é possível de-

$$F(\vec{x}) = \sum_{i=1}^{nf} c_i F_i(\vec{x}) \quad (3)$$

Figura 7: Equação de combinação de funções objetivo (YANO, 2011)

terminar precisamente os valores dos coeficientes" (BUZZO *et al.*, 2010). Assim, o ajuste dos objetivos com os coeficientes é subjetivo, depende do problema em questão e está propenso a erros, visto que o usuário deverá informar os valores, de forma empírica ou não (BUZZO *et al.*, 2010) (YANO, 2011) (ZITZLER *et al.*, 2000).

Para evitar o problema da *determinação dos coeficientes*, pode-se encontrar um conjunto de soluções ótimas que procure balancear todos os objetivos, utilizando a *otimalidade de Pareto*. Neste caso, as soluções de um problema multiobjetivo constituem o *ótimo de Pareto*, que leva em consideração todos os objetivos do problema em questão, e os valores correspondentes das funções objetivo formam a *fronteira de Pareto*, assim como será explicado na seção 2.2.11. Segundo (YANO, 2011), cada solução é *não dominada* por outras soluções, uma vez que a solução não pode ser melhorada em qualquer objetivo sem causar uma degradação em no mínimo um outro objetivo.

As seguintes definições formais foram retiradas do trabalho de (YANO, 2011):

Um vetor $\vec{u} = u_1, \dots, u_{nf}$ é dito dominar $\vec{v} = v_1, \dots, v_{nf}$ (denotado por $\vec{u} \prec \vec{v}$) se e somente se u é parcialmente menor que v , i.e., $\forall i \in 1, \dots, nf, u_i \leq v_i \wedge \exists i \in \{1, \dots, nf\} : u_i < v_i$.

Para um determinado problema multiobjetivo $F(\vec{x})$, o conjunto ótimo de Pareto $(P^*)P^* = \{\vec{x} \in \Omega \mid \neg \exists \vec{x}' \in \Omega : F(\vec{x}') \prec F(\vec{x})\}$.

Para um determinado problema multiobjetivo $F(\vec{x})$ e conjunto ótimo de Pareto P^* , a fronteira ótima de Pareto (PF^*) é $PF^* = \{\vec{u} = F(\vec{x}) | \vec{x} \in P^*\}$

A Figura 8 ilustra a computação da *otimalidade de Pareto*¹⁷ para a minimização de duas funções objetivos quaisquer, F_1 e F_2 . As soluções representadas pelos pontos p_1, p_2, p_3 na Figura pertencem à fronteira de Pareto. Já p_4 representa uma solução dominada por p_2 , visto que os valores de F_1 para p_2 e p_4 são os mesmos, mas p_2 possui um valor melhor em relação a F_2 do que p_4 . Portanto p_4 não faz parte do conjunto de Pareto. Sob o ponto de vista de um problema de minimização, a solução p_1 possui o melhor valor para F_1 , mas o pior valor para F_2 . Analogamente, a solução p_3 possui o melhor valor para F_2 , mas o pior para F_1 . Como muitas soluções podem ser obtidas no conjunto de Pareto, o tomador de decisão é responsável por selecionar as soluções desejadas. Por exemplo, em um problema em que é dada prioridade para F_1 , a solução p_1 poderia ser selecionada entre as soluções do conjunto de Pareto.

Gerar o conjunto de Pareto é computacionalmente caro (YANO, 2011), mas algoritmos evolutivos podem ajudar a encontrar uma solução aproximada. As seções 2.2.5 e 2.2.6 descrevem melhor alguns algoritmos utilizados neste trabalho.

2.2.5 Otimização Extrema Generalizada Multi-objetiva (M-GEO)

M-GEO é a sigla para GEO multi-objetivo, chamado assim por tratar várias funções objetivo. (BUZZO *et al.*, 2010) utiliza o M-GEO para gerar sequências de eventos que conduzam uma MEFÉ a um determinado estado desejado. (BUZZO *et al.*, 2010) diz ainda que a MEFÉ é responsável por representar os possíveis

¹⁷O termo *otimalidade*, no contexto em que foi empregado, refere-se à soluções ótimas para funções objetivo diferentes, em um único momento ou para um determinado parâmetro. Ou seja, extrai de todas as funções objetivo os melhores resultados delas para um único determinado contexto.

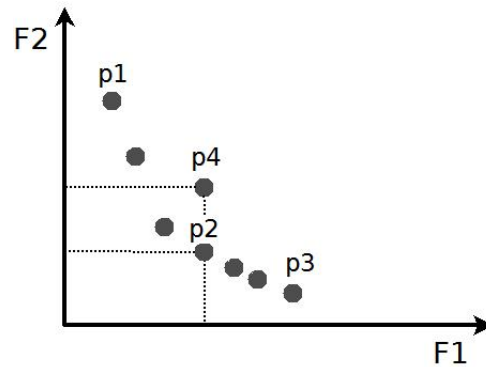


Figura 8: Fronteira de Pareto (YANO, 2011)

comportamentos de um sistema real, e o estado que se deseja alcançar é gerado pelo M-GEO, que é baseado no conceito de *Ótimo de Pareto*, tratado na seção 2.2.11.

Nos métodos GEO, as espécies são dispostas ao longo de uma linha (vetor), e um valor de adaptação é atribuído à cada espécie, analogamente ao modelo de (BAK; SNEPPEN, 1993). A população consiste na cadeia de espécies que codificam as variáveis de projeto (YANO, 2011). A população no M-GEO segue a mesma codificação binária do GEO. Existe também o M-GEO_{vsl} (seção 2.2.6), proposto por (YANO, 2011), e que serve de base para o presente trabalho. Existem ainda outras versões de GEO, mas que não serão aqui tratadas, por fugirem do escopo do problema.

2.2.6 M-GEO_{vsl}

Além de realizar uma otimização multi-objetivo, a principal característica do M-GEO_{vsl} é a capacidade de gerar soluções de diferentes tamanhos (YANO, 2011). Essa abordagem é uma versão do M-GEO e, por isso, é capaz de tratar

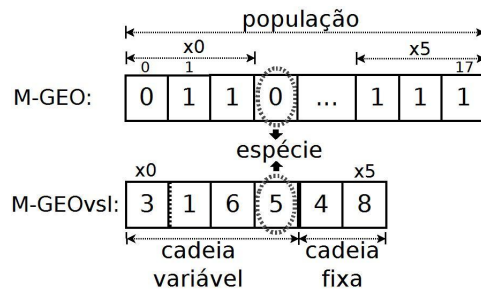


Figura 9: Comparação entre Populações do M-GEO e M-GEO_{vsl} (YANO, 2011)

problemas que contenham variáveis inteiras, reais ou discretas. Em contramão, diferentemente do GEO e do M-GEO, o M-GEO_{vsl} utiliza *codificação discreta* para codificar sua população (YANO, 2011). Cada variável do projeto é tratada como uma espécie nessa abordagem. A Figura 9, mostra uma comparação das codificações das populações entre a abordagem M-GEO e M-GEO_{vsl}: enquanto há dezoito espécies no M-GEO (considerando que cada espécie consuma três bits), na abordagem M-GEO_{vsl} há apenas seis.

2.2.7 Codificação da Solução

Assim como o M-GEO, no M-GEO_{vsl} a representação das espécies se assemelha ao modelo de (BAK; SNEPPEN, 1993). A população do M-GEO_{vsl} é dividida em duas partes: a parte com tamanho que pode variar ao longo do processo e a parte de tamanho fixa (YANO, 2011). A primeira espécie da população é a que dita o tamanho da parte variável. Como essa parte variável da população pode variar de tamanho ao longo do processamento evolutivo do método, essa primeira espécie é forçada a alterar seu valor sempre que necessário. Cada elemento da cadeia variável está dentro do mesmo domínio de valores. Já as espécies da parte fixa da população, podem ter domínios de valores diferentes e são opcionais, pois repre-

sentam variáveis do projeto do problema, que não precisam de uma representação dinâmica de tamanho (YANO, 2011).

As soluções com diferentes tamanhos são permitidas por conta da possibilidade de variação de tamanho da população ¹⁸ no M-GEO_{vsl}.

2.2.8 Operador de Mutação

A mutação de elementos no M-GEO_{vsl} é diferente e mais complexa do que as mutações do GEO, M-GEO e suas outras variações. Para modificar as espécies no GEO, existe apenas o operador de mutação, e cada espécie é mutada de acordo com o seu domínio (podem assumir apenas os valores 0 e 1). Já no M-GEO_{vsl}, a espécie pode ser codificada por números inteiros ou reais, além de outros fatores que agregam complexidade à mutação de espécies dessa solução (YANO, 2011).

No M-GEO_{vsl}, um caso especial é a mutação da primeira espécie da parte variável da população. Duas situações devem ser consideradas quando essa espécie é mutada: seu valor pode aumentar ou diminuir, causando mudanças na cadeia variável. No primeiro caso (o valor da primeira espécie aumenta), é necessário aumentar o número de espécies da cadeia variável, adicionando espécies de valor variável (ou randômico) no final dessa cadeia. No segundo caso (o valor da primeira espécie diminui), as espécies sobressalentes da parte variável da população são ignoradas. Logo, é através da mutação que o M-GEO_{vsl} consegue populações com tamanhos diferentes (YANO, 2011).

(YANO, 2011) apresenta um exemplo da mutação de espécies em diferentes partes da população, utilizando o operador de mutação do M-GEO_{vsl} na Figura 10. Na parte *a*, a primeira espécie da população sofre mutação, passando de 3 para 4,

¹⁸A população consiste da cadeia de espécies que codificam as variáveis de projeto.

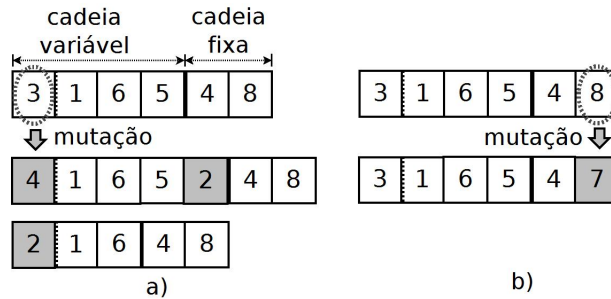


Figura 10: Exemplos de mutações no M-GEO_{vsl} em partes diferentes da população (YANO, 2011)

o que força uma nova espécie no final da cadeia variável (a espécie com valor 2 foi adicionada nesse exemplo). Quando a primeira espécie da população muda de 3 para 2, a espécie de valor 5 é ignorada para que seja mantida a coerência para o algoritmo, ou seja, a cadeia variável deve possuir o mesmo número de elementos descritos na primeira espécie da população¹⁹. Na parte *b* ocorre uma mutação em uma espécie da parte fixa, e o valor da espécie muda de 8 para 7.

2.2.9 Processo Evolutivo

As maiores diferenças entre M-GEO e M-GEO_{vsl} estão na maneira como essas abordagens tratam a população, mas em geral, os passos para os dois algoritmos são quase os mesmos. Uma visão geral das duas abordagens é apresentada na Figura 11 através de um fluxograma.

(YANO, 2011) descreve os passos do M-GEO_{vsl}, como mostrado:

1. "Inicialize a população: para a parte variável da população, gere um valor inteiro aleatório R para a primeira espécie e inicialize aleatoriamente uma

¹⁹A primeira espécie da população é o elemento R, gerado nos passos anteriores do algoritmo M-GEO_{vsl}

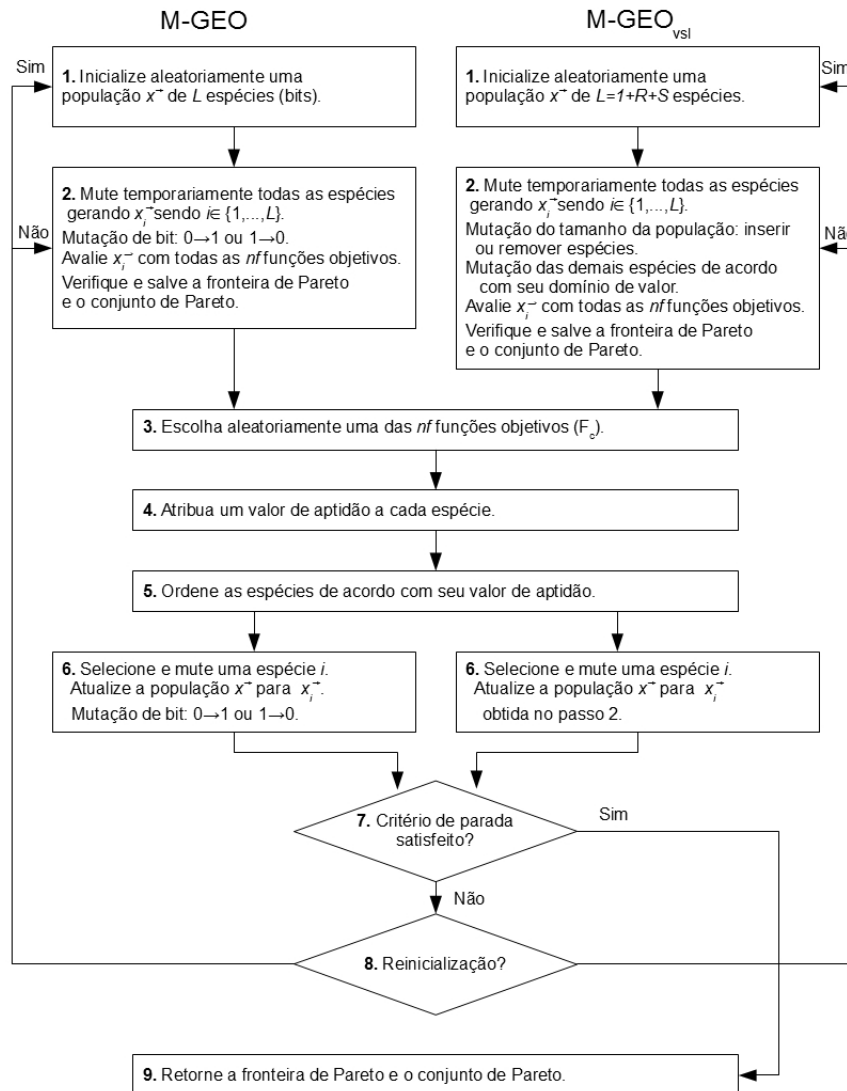


Figura 11: Fluxograma comparativo entre M-GEO e M-GEO_{vst} (YANO, 2011)

cadeia de tamanho R ; e para a parte fixa, se existir, inicialize aleatoriamente uma cadeia de tamanho S . Assim, a população \vec{x} consiste de $R + 1$ espécies da parte variável, mais S espécies da parte fixa: $\vec{x} = x_0, x_1, \dots, x_R, x_{R+1}, \dots, x_{R+S}$ ";

2. "Mute cada espécie i para outro valor mut_i de seu domínio, uma a cada momento, gerando \vec{x}_i . Todas as funções objetivos avaliam \vec{x}_i , calculando $F(\vec{x}_i) = [F_1(\vec{x}_i), \dots, F_{nf}(\vec{x}_i)]$, onde nf é o número de funções objetivo. Após essa avaliação, o valor da espécie i retorna ao seu valor original. Esse processo é repetido para todas as espécies. Desse modo, todas as nf funções objetivo avaliam todas \vec{x}_i , onde $i \in \{0, 1, \dots, R + S\}$. A aproximação da fronteira de Pareto é verificada e atualizada com \vec{x}_i , conforme necessário para conter apenas soluções não-dominadas, e é mantido um arquivo separado".
3. "Escolha aleatoriamente uma função objetivo F_c , onde $c \in \{1, \dots, nf\}$ e nf é o número de funções objetivo";
4. "Associe um valor de aptidão a cada espécie i como $\Delta_i = F_c(\vec{x}_i) - ref$, onde ref é um valor de referência. A aptidão indica o ganho (ou perda) relativo à mutação da espécie, comparado com um valor de referência (e.g., zero)";
5. "Ordene as espécies de acordo com seu valor de aptidão. A primeira posição ($k = 1$) da ordenação indica a espécie menos adaptada. Para um problema de minimização, os menores valores de Δ_i estão nas menores posições da ordenação. Se duas ou mais espécies possuem a mesma aptidão, ordene-os aleatoriamente com distribuição uniforme";
6. "Escolha com probabilidade uniforme uma espécie candidata i para mutar. Gere um número aleatório RAN com distribuição uniforme no intervalo de $[0, 1]$. Se $P_i(k) = k^{-\tau}$ é igual ou menor que RAN , a espécie é confirmada

para mutar. Tau (τ) é um parâmetro livre configurado pelo usuário que controla o determinismo da busca. Se $\tau = 0$, qualquer variável possui a mesma probabilidade para ser mutada. Por outro lado, quanto maior o valor de τ , mais determinística a busca se torna para mutar a variável menos adaptada. Na prática, os valores usados para τ que maximizam a eficiência do algoritmo têm sido usado entre [1,5]. Uma nova espécie é escolhida até que uma espécie é confirmada para mutar. Mude a população atual \vec{x} para \vec{x}_i obtida no Passo 2";

7. "Verifique se o critério de parada é alcançado. A condição de parada é o número máximo de avaliações de todas as funções objetivo";
8. "Verifique se uma reinicialização da população deve ser iniciada. A condição para uma reinicialização é $numAval = int(maxNumAval / numReInit)$, onde $numAval$ é o número máximo de avaliações das funções objetivos, e $numReInit$ é o número de reinicializações desejado. Em caso afirmativo, o algoritmo retorna ao Passo 1 e uma nova população é gerada aleatoriamente, mantendo-se as soluções da fronteira de Pareto. Caso contrário, o algoritmo retorna ao Passo 2";
9. "Retorne o conjunto de Pareto e a fronteira de Pareto".

Para ilustrar o funcionamento da multi-objetividade no M-GEO_{vsl}, é mostrado na Figura 12 um exemplo de minimização em relação a um vetor de inteiros \vec{x} de tamanho n para as funções objetivo F_1 e F_2 e um inteiro y .

Um exemplo numérico para esse problema é mostrado na Tabela 3. A Tabela 3 foi baseada em um exemplo retirado de (YANO, 2011). Nesse caso, a população representa um vetor \vec{x} (cadeia variável) e o número y (cadeia fixa). Primeiramente,

$$\begin{aligned}
\text{Minimizar } \vec{x} : F_1(\vec{x}, y) &= \sum_{j=1}^n x_j^y \\
F_2(\vec{x}, y) &= \sum_{j=1}^n (x_j - 2)^y \\
\text{Sujeito a : } 0 &\leq \vec{x} \leq 100 \\
\text{Onde : } \vec{x} &= \{x_1, x_2, \dots, x_n\}
\end{aligned}$$

Figura 12: Minimização de um vetor de inteiros de tamanho n para diferentes funções objetivo e um inteiro y (YANO, 2011)

um vetor de inteiros (\vec{x}) de tamanho 4 e um valor inteiro (y) são aleatoriamente gerados. Assim, a população possui 6 espécies nesse momento. No segundo passo, todas as espécies são mudadas temporariamente, uma de cada vez, para ordenar a aptidão das espécies. Note que a primeira espécie, que representa o tamanho do vetor, é mutado de 4 para 6. Assim, dois valores inteiros são escolhidos aleatoriamente para completar o vetor. É importante observar que antes da próxima espécie mutar, a espécie anterior retorna à sua condição original. Cada nova configuração é avaliada por ambas funções objetivos, obtendo-se um ponto candidato p à fronteira de Pareto. Se p é uma solução não-dominada, então p é incluído na aproximação da fronteira de Pareto e as soluções dominadas por p são removidas. No exemplo, a função objetivo F_2 é escolhida no passo 3. No próximo passo, a aptidão das espécies é determinada, calculando-se Δ_i com $ref = 0$. A ref é um valor que servirá de base para o cálculo da aptidão das espécies de acordo com a função objetivo em questão (neste exemplo, a função objetivo 2, escolhida no passo 3). Portanto, não é uma regra que o valor de ref seja 0. Pode-se utilizar, por exemplo, o valor de ref como sendo o valor da função objetivo da população primária, gerada no passo 1. Assim, saberia-se se a mutação de uma espécie foi vantajosa ou não para a população, de acordo com a função objetivo que está sendo comparada. Em seguida, todas as espécies são ordenadas pelos seus valores de aptidão em relação à função objetivo escolhida F_2 . Note que a posição de ordenação da primeira espécie é $k = 6$ e a última é $k = 1$. Isso indica que a primeira espécie é melhor adaptada que a úl-

tima, uma vez que ao mutar a primeira espécie, F_2 melhora menos que mutando-se a última espécie.

No passo 6, uma espécie é selecionada para mutar de acordo com o valor de τ sobre a ordenação. No exemplo, $\tau = 1.0$ é utilizado, e a terceira espécie com posição $k = 3$ (não é o menos adaptado) é escolhido para mutar. Desse modo, o tamanho da população permanece com 6 espécies. No entanto, se a primeira espécie fosse selecionada para mutar, o tamanho da população aumentaria para 8 espécies. Quando o número máximo de avaliações de todas as funções objetivos é alcançado, o conjunto de Pareto e a fronteira de Pareto são retornados. Caso contrário, verifica-se se uma reinicialização deve ser iniciada. Re-inicialização é um recurso comum em abordagens multi-objetivo para evitar ótimos locais (YANO, 2011). Observe que o M-GEO_{vsl} possui apenas dois parâmetros livres que devem ser ajustados a cada problema: τ é o número de reinicializações. É possível obter, com apenas 2 parâmetros informados pelo usuário, soluções de tamanhos variáveis utilizando abordagem multi-objetiva para a busca.

Em relação aos diversos algoritmos multi-objetivos, o M-GEO_{vsl} não utiliza os conceitos de dominância como critério para guiar a busca, tais como em algoritmos multi-objetivos: NSGA-II, NPGA, MOGA e SPEA (YANO, 2011).

2.2.10 Os valores de RAN , τ e o número de reinicializações

Os valores de RAN , τ e do número de reinicializações são de suma importância para os resultados e sua qualidade (YANO, 2011). Assim como explicado no passo 6 da seção 2.2.9, se o valor de τ for próximo de 1, então qualquer espécie terá grandes chances de mutar, porém quanto maior o valor de tau, mais a mutação é restringida às espécies menos adaptadas (em relação à uma determinada

Passo	População	F_1	F_2	F_c	Δ_i	k
1	Inicialize a população					
	6 — 3 8 10 7 9 1 2	304	176	-	-	-
2	Calcule F_1 e F_2 para cada mutação de espécie					
	6 — 3 8 10 7 9 1 2	304	176	-	-	-
	4 — 1 8 10 7 2	214	126	-	-	-
	6 — 3 3 10 7 9 1 2	167	91	-	-	-
	6 — 3 8 2 7 9 1 2	126	62	-	-	-
	6 — 3 8 10 5 9 1 2	198	110	-	-	-
	6 — 3 8 10 7 1 1 2	193	95	-	-	-
	6 — 3 8 10 7 9 2 2	188	101	-	-	-
	6 — 3 8 10 7 9 1 2	28	20	-	-	-
3	Selecione uma função objetivo					
	6 — 3 8 10 7 9 1	222	126	F_2	-	-
4	Encontre a aptidão de cada espécie					
	6 — 3 8 10 7 9 1 2	304	176	F_2	176	-
	...					
	4 — 3 8 10 5 1	28	20	F_2	20	-
5	Ordene as espécies de acordo com sua aptidão					
	4 — 3 8 10 5 1	28	20	F_2	20	1
	...					
	6 — 3 8 10 7 9 1 2	304	176	F_2	176	6
6	Escolha uma espécie candidata à mutação					
	6 — 3 3 10 7 9 1 2	167	91	F_2	91	3
7	Verifique se o critério de término é alcançado					
8	Verifique se é necessário uma reinicialização					
9	Retorne o conjunto de Pareto e a fronteira de Pareto					

Tabela 3: Exemplo numérico dos passos do M-GEO_{vst} (YANO, 2011)

função objetivo) da população. Além das consequências do valor de τ , o valor de *RAN*, que é escolhido randomicamente no passo 6 do processo evolutivo (seção 2.2.9), é de suma importância, pois se for um valor muito baixo (o valor mais baixo considerado para *RAN* no M-GEO_{vsI} é 0), dará condições para todas as espécies mutarem. Se for *RAN* for zero, todas as espécies podem mutar, mas se o valor de *RAN* for diferente de 0, mas muito próximo desse valor, a execução do M-GEO_{vsI} só irá depender do valor de τ para validar a mutação das espécies. Porém, se *RAN* for um valor elevado (o maior valor para *RAN* é 1), somente as espécies menos adaptadas terão condições de mutar. Por exemplo, se *RAN* for 1, não importa o valor de τ , somente a primeira espécie do vetor de aptidão, gerado no passo 7 do processo evolutivo (seção 2.2.9) terá condições de mutar, e como o valor de *RAN* é escolhido randomicamente, deve-se atentar bastante para esse parâmetro, que tem o poder de transformar a execução do M-GEO_{vsI} em uma execução determinística. Para contornar o problema do *RAN*, à cada execução dos passos, seu valor é atualizado.

O número de reinicializações possibilita uma variedade maior de resultados para o conjunto de Pareto (YANO, 2011). Como grande parte dos dados de execução do M-GEO_{vsI} são escolhidos randomicamente em tempo de execução, não se pode garantir, por exemplo, que a primeira espécie da população seja mutada em algum momento (o que confere soluções de tamanhos diferentes). Sendo assim, as reinicializações da população são de extrema importância para que seja encontrado um novo valor para a primeira espécie da população e, conseqüentemente, se consiga populações com tamanhos diferentes para o conjunto e para a fronteira de Pareto.

2.2.11 Ótimo de Pareto

Algoritmos evolutivos multi-objetivos são implementados para resolver problemas quando a solução precisa considerar alguns objetivos conflitantes simultaneamente e não existe nenhuma solução ótima (YANO *et al.*, 2011). Em outras palavras, não há uma única solução que simultaneamente otimize todos os objetivos. A solução é *balancear* o resultado entre todos os objetivos. Esse balanceamento é conhecido como *Conjunto ótimo de Pareto*, e suas funções objetivo correspondentes formam o *Pareto front*, ou fronteira de Pareto (YANO *et al.*, 2011) (ZITZLER; THIELE, 1998).

2.3 MOST

²⁰ (*Multi-objective Search-based Testing approach from EFSM*) é uma abordagem de teste proposta por (YANO, 2011), que combina o conceito de teste baseado em modelo e teste baseado em busca 2.1.2.

Segundo (YANO, 2011), os principais passos da abordagem MOST são:

1. Modelar uma MEFE M a partir da especificação;
2. Obter o modelo executável de M ;
3. Validar M ;
4. Instrumentar o modelo executável de M ;
5. Analisar as dependências de M ;

²⁰MOST é abreviação de um termo em inglês. Será chamado de MOST para manter a compatibilidade com outros trabalhos da área.

6. Gerar sequências de entrada com o algoritmo evolutivo $MGEO_{vsl}$ e avaliar o caminho disparado;
7. Obter os casos de teste;

Uma MEFÉ M , que é modelada de acordo com a seção 2.4.2, representa o comportamento do sistema e é manualmente obtida através da especificação do sistema (YANO, 2011). Na abordagem MOST, ao invés de considerar somente a estrutura estática da MEFÉ, um *modelo executável* é utilizado para gerar os casos de teste. Utilizar um modelo executável de MEFÉ significa implementar o modelo em uma linguagem de programação, e esse modelo pode ser usado para produzir dinamicamente o *caminho de transições*, que é disparado por uma sequência de eventos de entrada. No trabalho de (YANO, 2011), a execução do modelo executável de MEFÉ considera tanto o *aspecto de controle* quanto os *dados do modelo*. É importante *validar* o modelo, ou seja, executá-lo para verificar se a MEFÉ realmente representa o comportamento do sistema. No presente trabalho, não é dado um enfoque muito grande às considerações dos aspectos de controle e dos dados de modelo para a geração de dados de teste, mesmo porque as funções objetivo utilizadas para teste das implementações não necessitam de tais dados. Trabalhos que implementem esses conceitos junto à sua solução são propostos na seção 6.

O modelo pode ou não ser validado, mas a validação pode conferir maior credibilidade aos resultados gerados. Logo, o modelo é preparado para a geração de casos de teste, instrumentando o código do modelo executável para produzir o caminho de transições, disparado por uma sequência de eventos de entrada (YANO, 2011). Nesse momento entra em cena o algoritmo $M-GEO_{vsl}$ (seção 2.2.6) para gerar as sequências de entrada e os valores dos parâmetros envolvidos no modelo.

Depois, o M-GEO_{vsl} avalia o caminho produzido pelo modelo executável. Para guiar a busca, as informações geradas na análise de dependência 2.3.2 são utilizadas pelo algoritmo M-GEO_{vsl}. (YANO, 2011) chama a atenção para o fato de que as dependências são geradas em um passo anterior à geração de casos de teste. Na abordagem MOST, os casos de teste obtidos levam em consideração a cobertura da *transição alvo* e o tamanho da *sequência de entrada* gerada (YANO, 2011). Além desses dois parâmetros propostos por (YANO, 2011), o valor das funções objetivo, que são considerados na etapa de execução do M-GEO_{vsl}, também são considerados, e são de suma importância para os dados de teste gerados. Imagine uma situação de uma MEFÉ, em que um dos parâmetros presentes em suas transições sejam valores ou pesos. Um resultado interessante seria encontrar um caminho que chegue à uma determinada transição alvo (função objetivo utilizada por (YANO, 2011)), que contenha a maior ou a menor soma de pesos das transições pelas quais a execução de teste passou. Esses pesos poderiam ser, por exemplo, valores de pedágio de uma rodovia, ou o número de dados a serem considerados para resolver um determinado problema de engenharia.

2.3.1 Modelo Executável

"O principal objetivo de utilizar um modelo executável é obter de forma dinâmica o caminho de transições disparado por uma sequência de eventos de entrada durante a execução do modelo"(YANO, 2011). Técnicas tradicionais de geração de casos de teste baseado em MEFÉ realizam apenas uma análise estática do modelo, técnicas essas que ficam expostas ao possível problema da *explosão de estados* 2.4.2 devido aos domínios dos dados do modelo (YANO, 2011). A ideia de se utilizar um modelo executável de MEFÉ é o controle sobre as transições do modelo,

de acordo com os valores à cada momento da execução. (YANO, 2011) explica o controle sobre as transições do modelo executável:

"Uma transição t_x é apenas disparada quando o evento de entrada correspondente i_x é recebido e a guarda g_x é satisfeita, considerando-se que o estado atual é q_x^s . Como resposta, a ação a_x é executada e o estado atual muda para q_x^t . A guarda utiliza os valores das variáveis e parâmetros de acordo com o histórico das transições que foram executadas anteriormente. Os valores desses dados podem ser modificados pelas ações das transições, por exemplo, em um comando de atribuição. O modelo permanece no mesmo estado e uma saída nula é produzida quando a guarda g_x é falsa ou um evento inesperado é recebido".

A possibilidade de validar o modelo antes da geração dos casos de teste para verificar se o modelo representa o comportamento do sistema é uma grande vantagem da abordagem MOST. "Visto que os casos de teste são derivados a partir da MEFÉ, é importante a validação do modelo para verificar se o modelo está de acordo com a especificação do sistema em teste"(YANO, 2011). Como nessa abordagem os aspectos de controle e de dados do modelo são considerados para sensibilizar o caminho, o problema da geração de *caminhos inactíveis* é evitado, o que confere outra grande vantagem para a abordagem MOST.

Como a modelagem de uma MEFÉ é realizada manualmente para a utilização na abordagem MOST (YANO, 2011), é muito importante a fase de validação da MEFÉ para seguir com os passos do MOST. Para o presente trabalho, todas as MEFÉ utilizadas para a geração de dados são validadas ²¹.

²¹As MEFÉ utilizadas no presente trabalho foram retiradas de trabalhos publicados anteriormente (YANO, 2011) (BUZZO *et al.*, 2010) (CLARKE *et al.*, 2003) (JACOBSON *et al.*, 1997).

2.3.2 Dependência de MEFÉ

Segundo (YANO, 2011), a *Análise de Dependências* tem sido utilizada em muitas atividades da Engenharia de software, tais como manutenção de software e teste. A dependência existe em softwares, e definem o relacionamento entre os comandos, considerando o fluxo de controle e de dados. As estruturas de controle dizem respeito à estrutura de controle do programa, e as dependências de dados focam a relação de definição e uso de variáveis (YANO, 2011).

Os conceitos de dependência utilizados neste trabalho seguem a idéia de (ANDROUSOPOULOS *et al.*, 2009), assim como proposto por (YANO, 2011). (ANDROUSOPOULOS *et al.*, 2009) definem as dependências em termos de transições de uma MEFÉ, ao invés de nós em dependência de programa (BUZZO *et al.*, 2010). Uma noção geral de dependência de controle entre transições é mostrada por (YANO, 2011), através da função *CAMINHO*. A função caminho pode representar diferentes tipos de caminho, mas não serão mostrados aqui por fugirem do escopo do trabalho. Maiores detalhes sobre os conceitos de dependência podem ser encontrados nos trabalhos (ANDROUSOPOULOS *et al.*, 2009) (YANO, 2011).

2.3.3 Gerador de casos de teste

Na abordagem MOST, a geração de casos de teste é reformulada como um problema de *otimização*. Os casos de teste são gerados automaticamente a partir do algoritmo M-GEO_{vst}. O algoritmo gera uma sequência de entrada, que consiste de eventos de entrada e dados envolvidos nos eventos. O modelo executável recebe essa sequência como entrada e produz o caminho de transições percorrido pela

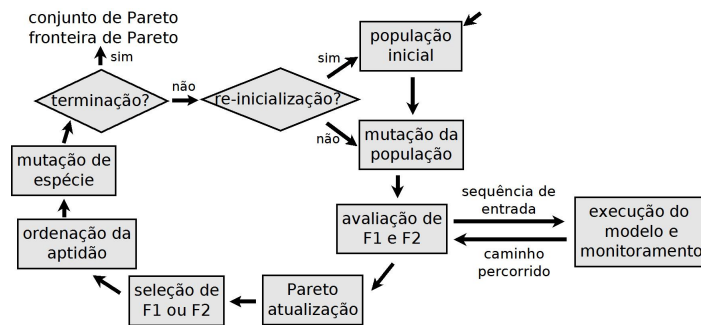


Figura 13: Processo Evolutivo do M-GEO_{vsl} para MOST (YANO, 2011)

sequência gerada, assim como mostrado na Figura 13. As sequências de entrada são avaliadas para verificar se os critérios de funções objetivo são satisfeitos, como por exemplo, encontrar uma transição alvo, ou obter um caminho mínimo com um número de transições percorridas. Baseado nos valores das funções objetivo, a sequência de entrada é modificada para tentar cobrir o critério de teste (YANO, 2011).

Na execução do M-GEO_{vsl}, são geradas as sequências de entrada, que são analisadas em tempo de execução de acordo com as funções objetivo envolvidas e, posteriormente são avaliadas, para eliminar a presença de soluções com caminhos infactíveis.

2.3.4 Codificação da Solução

Como mostrado na seção 2.2.6, a população em M-GEO_{vsl} consiste de uma parte fixa e outra variável. A parte variável consiste dos eventos de entrada, que constitui o aspecto de controle do caso de teste. A primeira espécie dessa população determina o tamanho da parte variável ou, simplesmente, o número de eventos de entrada. Apenas para relembrar, a mutação dessa primeira espécie que confere

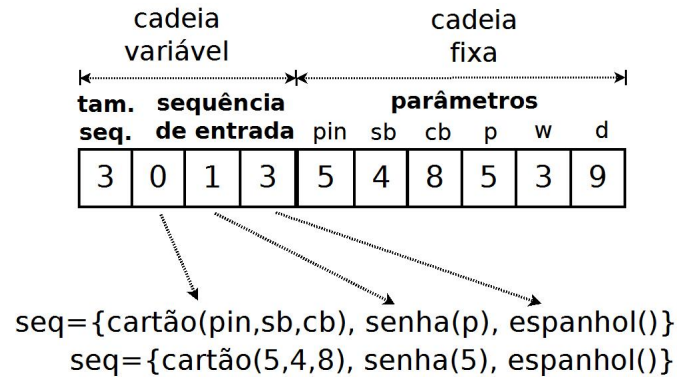


Figura 14: População no $M-GEO_{vst}$ para geração de dados de teste baseado em MEFE (YANO, 2011)

tamanhos diferentes para os dados de saída do $M-GEO_{vst}$. Sendo assim, o tamanho da população é variável em relação ao tamanho da sequência de entrada. A parte fixa contém todos os parâmetros de todos os eventos de entrada do modelo, representando os dados do caso de teste. Cada parâmetro envolvido no modelo é uma espécie na população. (YANO, 2011) mostra um exemplo de uma população para uma MEFE de um caixa eletrônico através da Figura 14. A MEFE utilizada por (YANO, 2011) para servir de exemplo para a Figura 14, é a Figura 15.

A primeira espécie da população (3), mostra que a cadeia variável da população contém 3 elementos, logo, 3 eventos de entrada. Cada evento de entrada necessita de uma quantidade de parâmetros. Por exemplo, o evento de entrada 0, necessita de 3 parâmetros que estão contidos na cadeia fixa, e no caso desta Figura, são representados pelas espécies 5, 4 e 8. A senha, evento de entrada representado pela espécie 1, necessita somente de um dado, que é representado na cadeia fixa pela espécie 5. Como o outro evento de entrada não necessita de parâmetros, as espécies 3 e 9 estão obsoletas para esse caso de teste representado. Se por algum motivo fosse gerado um caso de teste que possuísse x eventos de entrada, e esses eventos de entrada necessitassem de um número y de parâmetros, e a população

não oferecesse, mesmo depois de todas as mutações do M-GEO_{vst}, quantidade suficiente de espécies para servirem de parâmetro para os eventos de entrada, essa população iria resultar em uma sequência de dados de teste ineficaz para a MEFE em questão e, portanto, seria desconsiderada.

2.3.5 Função Objetivo

A função objetivo é responsável por guiar a meta-heurística na geração automática de dados de teste. (CLARKE *et al.*, 2003) diz que as técnicas de geração de dados de teste são baseadas em uma noção da cobertura ou especificação do código do programa, e essa cobertura pode ser medida e incorporada em uma função objetivo.

A função objetivo vai depender de cada problema, de cada solução desejada e a maneira como se trata o problema. Se o problema for um problema crítico ²², então a função objetivo deve considerar todas as variáveis que circundam o problema (mesmo que seja muito difícil considerar todas as variáveis de um problema real).

O trabalho de (YANO *et al.*, 2011) considera resolver um problema considerando várias funções objetivo. Porém, por se tratar de problemas diferentes, cada função objetivo de um determinado escopo do problema a ser tratado terá um objetivo diferente, direcionando a solução do problema para determinada resposta. Para contornar esse problema, o Ótimo de Pareto foi utilizado para *balancear* os resultados das funções objetivo presentes no problema.

²²Problema crítico na Engenharia de Software é um problema relacionado diretamente com pessoas. Quanto mais ligado à pessoas, mais crítico é o problema. Um exemplo de sistema crítico é o software que comanda um avião: se o software falhar, os passageiros podem sofrer acidentes.

2.4 Máquinas Finitas de Estados

2.4.1 Máquina de Estados Finita - MEF

Segundo (DELAMARO *et al.*, 2007), uma máquina de estado finito (MEF) é um modelo de máquina hipotética composta por estados e transições. Cada transição liga um estado a a um estado b , sendo que a e b podem ser o mesmo estado²³. A máquina pode estar, a cada momento, em apenas um estado, chamado *estado atual*. Em resposta a um evento de entrada, a máquina gera um evento de saída e, assim, muda para um novo estado. "Tanto o evento de saída gerado quanto o novo estado são definidos unicamente em função do estado atual e do evento de entrada"(DELAMARO *et al.*, 2007).

De acordo com (LEE; YANNAKAKIS, 1996), uma MEF é definida como uma 5-tupla (X, Z, S, s_0, f_z, f_s) , sendo:

- X - um conjunto não vazio de símbolos de entrada;
- Z - um conjunto finito de símbolos de saída;
- S - um conjunto finito não-vazio de estados;
- s_0 - o estado inicial;
- $f_z - (S \times X) : z$ é a função de saída;
- $f_s - (S \times X) : s$ é a função de próximo estado.

Um dos primeiros trabalhos a utilizar MEF para teste de software foi o trabalho de (CHOW, 1978), que testava o design de softwares utilizando MEF. Porém,

²³Quando um estado a é ligado ao mesmo estado a por uma transição t , t é chamada auto-transição.

à medida que MEF foi sendo utilizada para problemas mais complexos, os engenheiros de software encontraram limites na estrutura do modelo. Uma limitação de uma MEF tradicional, é que ela apenas representa os *fluxos de controle*, e não fornece mecanismos para modelar um importante aspecto comportamental do sistema, como o *fluxo de dados*. Assim, uma MEF tradicional tende a ter o problema de *explosão de estados* (ou *estados excessivos*) ao representar dados do sistema. Uma solução foi *estender* a MEF, criando a Máquina de Estados Finitos Estendidos (MEFE).

2.4.2 MEFE - Máquina de Estados Finita Estendida

A MEFE estende a MEF através da inclusão de variáveis de contexto, predicados e ações (MARTINS *et al.*, 1999). Uma transição de uma MEFE é também caracterizada por predicados e ações, somadas às atribuições de uma MEF: estado de origem, estado de destino e interação de entrada. (LEE; YANNAKAKIS, 1996) caracteriza uma MEFE como sendo uma 8-tupla $(S, s_0, I, O, V, P, A, g)$, onde:

- S é o conjunto não vazio de estados;
- s_0 é o estado inicial;
- I é o conjunto finito de estados;
- O é o conjunto finito de saídas;
- V é o conjunto de variáveis;
- P é o conjunto de predicados que operam sobre as variáveis;
- g é a função de transição de estado definida como $g : S \times I \times P(V) \rightarrow S \times O \times A(V)$.

Em uma MEF tradicional, tanto o evento de saída gerado quanto o novo estado são definidos unicamente em função do estado atual e do evento de entrada. Já a MEFE estende uma MEF no sentido de permitir interações com parâmetros, uso de variáveis, transições associadas com predicados, que dependem dos parâmetros do evento de entrada recebido e dos valores atuais das variáveis, e associadas com ações, que podem atualizar valores de variáveis (DSSOULI *et al.*, 1999) (YANO, 2011).

Em 1985 (JR, 1985) utilizou MEFE através de uma técnica denominada *estelle*, utilizada para descrever sistemas de uma maneira formal. (CHENG; KRISHNAKUMAR, 1993) e (KITA *et al.*, 1995) utilizaram MEFE para gerar dados de teste em 1993 e 1995, respectivamente. Três anos depois, (CHENG; KRISHNAKUMAR, 1996) utilizou MEFE para gerar automaticamente vetores funcionais.

(LEE; YANNAKAKIS, 1996) caracteriza uma MEFE como uma quintupla

$$M = (I, O, S, \vec{x}, T) \quad (4)$$

onde I, O, S, \vec{x} e T são conjuntos finitos dos símbolos de entrada, símbolos de saída, estados, variáveis e transições, respectivamente. Cada transição t do conjunto T é uma sextupla (LEE; YANNAKAKIS, 1996):

$$t = (s_t, q_t, a_t, o_t, P_t, A_t) \quad (5)$$

onde s_t, q_t, a_t e o_t são estado inicial (ou estado atual), estado final (ou próximo estado), entrada e saída, respectivamente. $P_t(\vec{x})$ é um predicado sobre os valores

atuais das variáveis e $A_t(\vec{x})$ é uma ação sobre os valores das variáveis (LEE; YANNAKAKIS, 1996).

Inicialmente a máquina está em um determinado estado $s_1 \in S$ com os valores iniciais de variável \vec{x}_{init} . Suponha que em um determinado estado s , os valores de variáveis sejam \vec{x} . Para uma entrada a , a máquina segue uma transição $t = (s, q, a, o, P, A)$ se \vec{x} é válido para $P : P(\vec{x}) = TRUE$. Nesse caso, a máquina tem como saída o , muda os valores atuais de variável através da ação $\vec{x} := A(\vec{x})$, e move-se para o estado q .

Para cada estado $s \in S$ e entrada $a \in I$, todas as transições com estado inicial s e entrada a seguem a regra:

$$t_i = (s, q_i, a, o_i, P_i, A_i), 1 \leq i \leq r. \quad (6)$$

Em uma MEFE *determinística*, os conjuntos de valores de variáveis válidas dos predicados r são mutualmente disjuntas:

$$X_{P_i} \cap X_{P_j} = \emptyset, 1 \leq i \neq j \leq r. \quad (7)$$

Caso contrário, a MEFE é *não determinística*.

Em uma MEFE determinística há somente uma transição possível para seguir, sendo que para cada estado e entrada, as transições associadas têm valores diferentes para cada um de seus predicados. Em uma MEFE não determinística há mais de uma transição possível para seguir.

(YANO, 2011) mostra uma MEFE modelada para especificar o comportamento de um caixa eletrônico em um grafo direcional (Figura 15). A explicação de (YANO, 2011) é dada como segue:

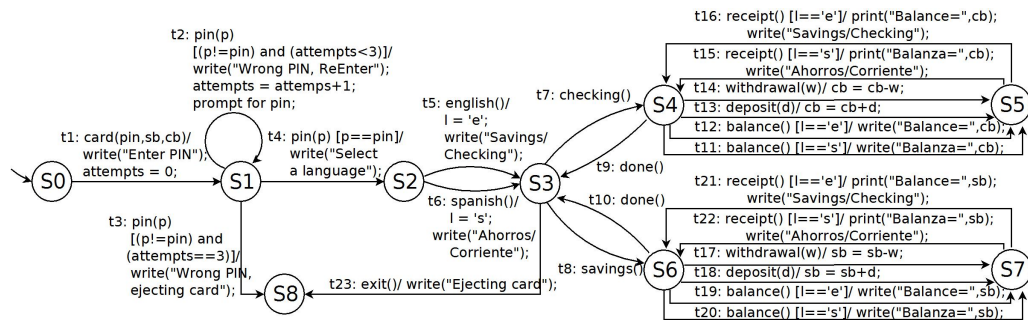


Figura 15: Sistema de caixa eletrônico representado em uma MEFE (BUZZO *et al.*, 2010)(YANO, 2011)

"Os dados do cartão e a senha devem ser fornecidos para a utilização das funcionalidades do caixa eletrônico, que inclui verificar saldo, fazer depósito e saque da conta corrente e poupança, e também escolher a linguagem do sistema entre inglês e espanhol. Cada transição t_x é descrita por: $i_x[g_x]/a_x$. Para disparar t_x , a máquina deve receber a entrada i_x com o estado atual em q_x^s e satisfazer g_x . Se g_x é verdadeiro, a_x é executado, podendo mudar valores de variáveis do modelo."

Uma sequência de entrada é composta por eventos de entrada e os valores de seus parâmetros. Por exemplo, uma MEFE, ao receber a sequência de entrada $seq1 = [card(442, 686, 12), pin(442), spanish(), savings(), done(), savings(), deposit(654), receipt()]$ percorre o seguinte caminho de transições $caminho1 = \{t_1 t_4 t_6 t_8 t_{10} t_8 t_{18} t_{22}\}$ (YANO, 2011). Essa sequência possui apenas *eventos de entrada nominais* que são eventos especificados em transições ao longo do caminho. Já os *eventos de entrada inoportunos* (ou inesperados) são aqueles *não especificados* em transições em um determinado estado. Assim como (YANO, 2011), assume-se que a máquina permaneça no estado atual ao receber um evento de entrada inoportuno e gere uma saída *nula* como resposta. Caso a sequência de entrada seja $seq2 = [card(442, 686, 12), exit()_d, pin(442), withdrawal(924)_d, spanish(), savings(), english()_d, done(), savings(), deposit(654), receipt()]$, sendo os even-

tos inoportunos especificados pelo símbolo $*^d$, o caminho de transições disparado seria o mesmo de *seq1*.

3 METODOLOGIA

3.1 Tipo de Pesquisa

O presente trabalho se baseia em pesquisa teórica, exploratória e experimental. Contém estudos da área de softwares evolutivos, multi-objetividade na resolução de problemas de otimização, modelagem de sistemas e geração de dados de teste baseada em modelos. Um estudo sobre a proposta de (YANO, 2011) também foi feito, e a implementação dessa proposta foi realizada, gerando assim, os dados de teste baseando-se em MEFE.

3.2 Materiais e Métodos

A implementação da ferramenta foi realizada utilizando somente softwares livres. A implementação foi na linguagem C, e espera-se que a ferramenta seja utilizada, à priori, pela Universidade Estadual de Campinas (UNICAMP), local onde o trabalho de (YANO, 2011) foi desenvolvido. Buscou-se implementar o algoritmo utilizando estruturas de dados abstratas, como pilhas e filas, tentando sempre otimizar o tempo de execução do algoritmo. Os códigos estão comentados e organizados, para que seja possível realizar melhorias posteriores no código mais facilmente.

3.3 Etapas desenvolvidas

Primeiramente foram realizados estudos das áreas de geração de dados de teste, focando principalmente nos trabalhos que utilizam meta-heurística evolutiva. Os conceitos envolvidos no escopo do problema foram explicados e, os mais im-

portantes, exemplificados. Depois de bastante conhecimento adquirido, o trabalho de (YANO, 2011) foi detalhadamente estudado. As informações mais importantes do trabalho de (YANO, 2011) foram mostradas, analisadas, comentadas e, algumas questões questionadas. Logo, os algoritmos propostos por (YANO, 2011) foram implementados. A maioria dos testes realizados no trabalho de (YANO, 2011) foram implementados utilizando a linguagem C e, para aproveitar alguns dados para comparação e somar dados em uma mesma linguagem para a linha de pesquisa da proposta, a implementação realizada nesse trabalho foi também utilizando a linguagem C. Os dados gerados foram analisados, e uma pequena comparação com os dados do trabalho de (YANO, 2011) e outros trabalhos da área foi realizada.

4 RESULTADOS

4.1 Implementação do M-GEO_{vsl} e MOST

O M-GEO_{vsl} e o MOST foram implementados na linguagem C, utilizando o IDE Eclipse. A implementação foi baseada na proposta de (YANO, 2011) (explicada na seção 2.2.6). Foram utilizados conceitos de implementação aprendidos no curso de Ciência da Computação para tentar organizar o código de maneira correta, de modo que o código ficasse modularizado, legível e bem documentado. A importância dessa organização torna-se ainda mais evidente com a possibilidade de uso, em outras pesquisas, do trabalho aqui implementado. No total, foram criados 6 arquivos, sendo 2 arquivos de extensão *.c* (*main.c* e *opVet.c*) e 4 *headers* (*funcoesObjetivo.h*, *opArquivo.h*, *opMath.h* e *opVet.h*), todos distribuídos na mesma pasta. Para executar o programa, basta compilar todos, seguindo por exemplo o comando *gcc* no linux: *gcc *.c -g -o main -Wall -Wextra*.

Como o algoritmo M-GEO_{vsl} precisa a todo momento de muitos números aleatórios, para eliminar o problema de *sementes* para aleatoriedade da função *rand* no C, os números aleatórios foram gerados em um programa em Java, para um arquivo texto que é lido para o *main.c*, que contém o programa principal do M-GEO_{vsl}. A geração de números aleatórios em Java é melhor explicada na seção 4.1.1.

4.1.1 O problema dos números aleatórios

A utilização de números randômicos na execução do M-GEO_{vsl} e do MOST é muito grande. Logo, a todo momento são gerados números randômicos, buscando


```
public class randomNumbersJava {  
    public static void main(String[] args) {  
        for(int i = 1; i < 1000000; i++) {  
            a = (int) (Math.random() * 10 + 1);  
            System.out.println(a);  
        }  
    }  
}
```

Figura 16: Código exemplo em Java para geração de números aleatórios.

resultados cada vez mais abrangentes, e que não ficam presos a uma determinada faixa de valores. Logo, a utilização da função `Math.rand()` da linguagem C se mostrou um pouco debilitada no sentido de gerar números aleatórios. Mesmo com a utilização de *sementes* para guiar a geração de números randômicos, essa função não gerou números com a variedade necessária para a resolução dos problemas propostos. Essa situação já foi discutida em (TAUSWORTHE, 1965).

Para contornar tal problema, foram gerados alguns números randômicos através da linguagem **Java** e armazenados em um arquivo texto. No momento da execução, quando necessário um número randômico, esse arquivo texto é lido, e a execução em C continua normalmente. A geração dos números randômicos se deu através do código mostrado na Figura 16.

4.1.2 A modelagem em MEFE

Todas as MEFE utilizadas neste trabalho foram retiradas de exemplos de MEFE de trabalhos anteriores (YANO, 2011) (BUZZO *et al.*, 2010). Logo, como mostrado na seção 2.3, a validação das MEFE para sequente aplicação do $M-GEO_{vsl}$ não se torna necessário, ainda mais pelo fato de que a validação de MEFE é realizada, segundo (YANO, 2011), de forma manual, diretamente pelo testador

```

6
0
3 4 5
7
0 1 5 2
0 2 0 1
2 1 3 4
1 3 4 7
2 4 1 3
3 4 5 8
4 5 1 1
1 5 8 9

```

Figura 17: Arquivo exemplo de modelagem de uma MEFÉ.

²⁴. Todas as MEFÉ foram modeladas em arquivos texto, e em momento de execução, uma determinada MEFÉ é lida do algoritmo para os passos serem executados. A modelagem da MEFÉ segue o padrão mostrado na Figura 17.

A primeira linha contém o número de estados (6). A segunda linha contém o estado inicial (0). A terceira linha contém todos os estados finais (3, 4 e 5). A quarta linha contém o número de transições, e as linhas subsequentes (7 linhas) são as transições, sendo a primeira coluna o estado do qual sai a transição, a segunda coluna o estado ao qual ela chega, a terceira coluna o valor de entrada e a quarta coluna o valor de saída. O exemplo acima é resultado da MEFÉ mostrada na Figura 18. O estado 0 contém uma seta que não sai de nenhum outro estado, indicando que o estado 0 é o estado inicial. Os estados 3, 4 e 5 contém dois círculos, o que indica que são estados finais.

²⁴Testador é a pessoa responsável por guiar os teste.

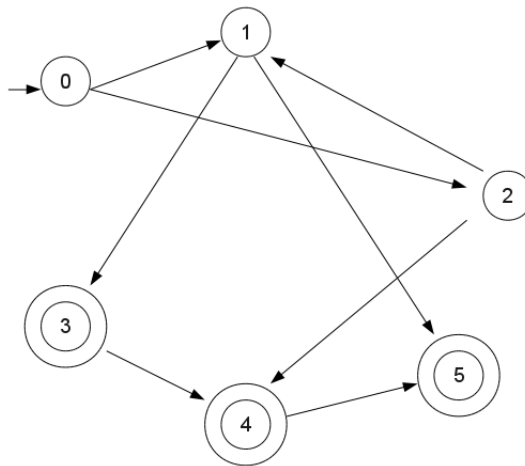


Figura 18: Exemplo para a modelagem da MEFE exemplificada na Figura 17 (BUZZO *et al.*, 2010).

4.1.3 Validação das sequências geradas

Após geradas as sequências de entrada e seus parâmetros, um algoritmo é executado para validar os casos de teste gerados, buscando por caminhos infactíveis. Esse procedimento tenta garantir que os caminhos presentes no resultado da execução do MOST são caminhos factíveis, eliminando casos de teste que tomariam tempo do testador e não resultariam em resultados factíveis.

O pseudo-algoritmo para verificar a factibilidade dos caminhos gerados é mostrado na Figura 19. Esse algoritmo é aplicado à cada população gerada pelo $M\text{-GEO}_{vsl}$.

4.2 Resultados do $M\text{-GEO}_{vsl}$

O $M\text{-GEO}_{vsl}$ foi implementado de acordo com a seção 2.2.6. As *partes* do algoritmo citadas nesta seção se referem às partes mostradas na explicação do

```

Para cada elemento da populacao (linha de saida da execucao) :
  Se o valor e aplicado e um estado e alcancado faca :
    continuar execucao
  Se nao :
    Se o estado atual representa estadofinal faca :
      retorna VERDADEIRO
      pare a execucao
    Se nao :
      retorna FALSO
      pare a execucao
    fim Se
  fim Se
  retorna VERDADEIRO
fim Para

```

Figura 19: Pseudocódigo para verificar caminhos inactíveis.

```

X = 2 | 2 10 | 5 2 7 10 1 4 1 9 9

```

Figura 20: Exemplo de população inicial do M-GEO_{vsf}.

algoritmo M-GEO_{vsf}, presente na seção 2.2.6. Os valores de R (primeiro elemento do vetor) e o valor de S (número de elementos da parte fixa) foram encontrados de forma aleatória, como explicado na seção 4.1.1.

Para mostrar o resultado gerado pelo M-GEO_{vsf}, um exemplo de execução será mostrado a seguir:

Primeiramente, os valores de R e S são aleatoriamente encontrados:

```

Valor de R:2
Valor de S:9

```

É então preenchida a população com valores aleatórios, de acordo com o passo 1 do M-GEO_{vsf} (seção 2.2.6). Para a parte variável, são encontradas 2 espécies, e para a parte fixa, 9 espécies. A população preenchida contém n espécies, sendo que $n = 2 + 9 + 1$ espécies, como mostrado na Figura 20.

$$OA_2 = 11, 10, 2, 5, 8, 6, 4, 1, 3, 7, 9, 0$$

Figura 21: Vetor Ordem de Aptidão da população do M-GEO_{vst} para a função objetivo 2 do exemplo.

Essa é a população inicial para a primeira repetição do M-GEO_{vst}. Como R tem valor 2, os valores 2 e 10 caracterizam a parte variável da população, e os outros 9 valores ($S = 9$), caracterizam a parte fixa.

O passo 2 do algoritmo pede para que cada espécie seja mutada, o que irá gerar n novas populações. Depois, é calculado o valor da função objetivo para cada população com uma única espécie mutada, e esse valor é armazenado em um vetor $FOM(n)$ (Valor da função objetivo para a população mutada). Depois de todas as espécies terem sido mutadas, uma das funções objetivo f_x é escolhida, e é gerado um vetor $OA(n)$ (*Ordem de aptidão*, que ordena todas as populações mutadas de acordo com a função objetivo escolhida). O valor de referência ref para o vetor OA é o valor da mesma função objetivo para a população inicial X , o que é explicado no passo 4 do algoritmo. Para o nosso exemplo, foi escolhida aleatoriamente a função objetivo f_2 , e de acordo com essa função, o vetor OA ficou como mostrado na Figura 21. A população com a espécie 11 mutada é a menos adaptada de acordo com f_2 , e a população com a espécie 0 mutada é a melhor adaptada para a f_2 , como mostrado no passo 5 do M-GEO_{vst}.

O próximo passo é encontrar aleatoriamente os valores de τ e RAN :

Valor de RAN : 0.050000

Valor de τ : 2

Logo, é escolhida aleatoriamente uma espécie α da população X para ser candidata a mutar. Após escolhida, se o valor de $Pi(k) = k^{-\tau}$ for menor ou igual a

RAN , então essa espécie é confirmada para mutar, sendo k a posição de α no vetor OA . No exemplo a seguir, algumas espécies foram testadas:

```

Especie Encontrada:2
PosicaoNoVetorAptidao:3
Valor de Pi(k):0.111111
Especie Encontrada:6
PosicaoNoVetorAptidao:6
Valor de Pi(k):0.027778

```

A espécie 2 foi encontrada, mas seu valor de Pi não foi menor ou igual ao valor de RAN . A espécie 2, posição 3 no vetor OA possui $Pi(k) = 0.111111$, que é maior do que $\tau = 0.05$, e portanto, a espécie 2 do vetor *Ordem de Aptidão* não foi confirmada para mutar. No exemplo, a espécie 2 corresponde ao número 10, posição 3 do vetor. Depois, a espécie 6, posição 6 no vetor OA foi encontrada, e o valor de $Pi(k)$ foi menor ou igual a RAN . Logo, a espécie 6 foi confirmada para mutar. O vetor antigo e o novo vetor são mostrados a seguir:

Vetor Antigo:

2 | 2 10 | 5 2 7 **10** | 1 4 1 9 9

Novo Vetor Mutado:

2 | 2 10 | 5 2 7 **8** | 1 4 1 9 9

Todo esse processo para esse mesmo vetor X é repetido 1000 vezes. Depois, por 10 vezes é voltado ao passo 1, e novos valores são atribuídos à R e S , garantindo ainda mais a aleatoriedade de resultados e diferentes tamanhos de populações²⁵.

²⁵Diferentes tamanhos de população é uma característica de destaque no trabalho de (YANO, 2011).

No momento da escolha da espécie α a ser mutada, se os valores de τ e RAN forem muito grandes, isso irá excluir a escolha de qualquer espécie, e irá limitar essa escolha somente à espécie que ocupar o pior lugar do vetor OA .

4.2.1 Caminhos inactíveis gerados

A proposta deste presente trabalho é gerar dados de teste para MEFÉ, utilizando abordagem evolutiva e considerando a multi-objetividade. Os dados são alcançados, porém, na execução do M-GEO_{vsI}, caminhos inactíveis também são gerados. É trabalho do algoritmo MOST, explicado na seção 2.3, tratar os caminhos, eliminando os caminhos inactíveis encontrados pelo M-GEO_{vsI}.

4.3 Resultados do MOST

O MOST utiliza os resultados gerados do M-GEO_{vsI} em sua execução. A utilização da MEFÉ é feita de acordo com a seção 4.1, que mostra a Figura 17, a qual contém a visualização de um arquivo de exemplo mostrando como a MEFÉ é lida para dentro da execução do MOST.

O MOST pode gerar um resultado grande ou pequeno, o que depende da sensibilidade do M-GEO_{vsI} para *evoluir* as suas populações para resultados satisfatórios da MEFÉ, e também da MEFÉ, que pode possuir ou não uma grande variedade de caminhos possíveis. Um exemplo de resultados da execução do MOST é mostrado na Figura 22. Nesse exemplo, são mostrados 5 blocos de resultados, e os três pontos entre os blocos indicam que há mais resultados entre os resultados mostrados. No primeiro bloco, o primeiro elemento (R) gerado foi o 8, o que resultou em 8 elementos aleatórios para a população. O valor de S é 9, o que gerou 9 elemen-

tos fixos. Cada linha representa uma população dominante *evoluida* da população anterior (ou linha anterior). Por exemplo, da linha 1 para a linha 2, o elemento evoluído foi o da terceira posição (da esquerda para a direita) da população, que evoluiu de 9 para 1. O elemento 1 é mostrado na segunda linha do resultado. Todas as linhas do resultado seguem esse padrão. É possível observar que os primeiros elementos dos blocos são diferentes, e isso acontece devido à volta ao passo 1 da execução do MOST, explicado na seção 2.3.

A Figura 24 apresenta uma MEFÉ com 2 parâmetros para cada transição. O primeiro parâmetro representa o dado de entrada para a transição, e o segundo representa a saída na execução da transição. Para essa MEFÉ de exemplo, a saída do MOST em uma determinada execução²⁶ é mostrada na Figura 23. A Figura 23 é um outro exemplo de saída da execução do MOST, e possui as mesmas características da Figura 22.

Ou seja, para o primeiro resultado (linha 1), a MEFÉ da Figura 24 seria executada, saindo do estado 0, indo para o estado 1 e terminando no estado 5. A saída dessa execução seria [2,9]. A execução do primeiro resultado contendo o primeiro elemento da população igual a 4 (4 | 5 4 5 1 | 0 0 3 1 2 4 5 6 7 3), percorreria os estados 0, 1, 3, 4, 5, nessa ordem, e teria como resultado [2,7,8,1]. Todas essas execuções foram geradas pela execução do MOST, utilizando o algoritmo evolutivo e multi-objetivo M-GEO_{vsl}.

O M-GEO_{vsl} gera muitos resultados ($n_r = 10 * 1000$), e nem todos esses resultados são aplicáveis à uma determinada MEFÉ. A saída do MOST apresenta somente saídas que seriam, teoricamente, aplicáveis à MEFÉ em execução. Um fator importante é: se a parte variável da população de saída do M-GEO_{vsl} con-

²⁶É preciso deixar bem claro que as saídas do MOST variam de execução para execução, pois se trata de um algoritmo evolutivo multi-objetivo, o que torna a execução do MOST não determinística.

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

Figura 22: Exemplo de resultado da execução do MOST.

```

2 | 5 8 | 10 1 1 2 6 4 7
2 | 5 8 | 9 1 1 2 6 4 7
2 | 5 8 | 9 1 4 2 6 4 7
2 | 5 8 | 9 10 4 2 6 4 7
...
2 | 5 8 | 9 10 5 2 6 4 7
2 | 5 4 | 9 10 5 2 6 4 7
2 | 5 4 | 6 10 5 2 6 4 7
2 | 5 4 | 6 3 5 2 6 4 7
2 | 5 4 | 9 3 0 2 6 4 7
...
4 | 5 4 5 1 | 0 0 3 1 2 4 5 6 7 3
4 | 5 4 5 1 | 2 0 3 1 2 4 5 6 7 3
4 | 5 4 5 1 | 2 9 3 1 2 4 5 6 7 3
4 | 5 4 5 1 | 7 0 3 1 2 4 5 6 7 3
4 | 5 4 5 1 | 7 0 3 10 2 4 5 6 7 3
...
3 | 5 4 5 | 8 2 1
3 | 5 4 5 | 4 2 1
...
3 | 0 1 1 | 3 3 7
3 | 0 1 1 | 5 3 7
3 | 0 1 1 | 9 3 7
...
3 | 0 1 6 | 3 3 7
3 | 0 1 3 | 3 3 7

```

Figura 23: Exemplo de saída parcial da execução do MOST.

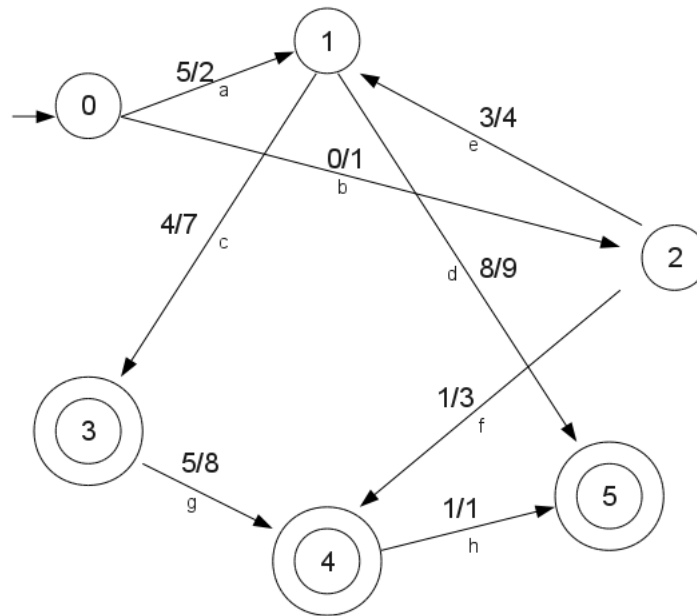


Figura 24: MEFE Exemplo para execução do MOST.

ter 5 entradas, mas se na terceira entrada desse grupo, a MEFE já estiver em um estado final e a quarta entrada da população não for executável, o MOST aceita essa população, pelo fato de que na terceira entrada da população a MEFE já foi executada com sucesso. Se a MEFE já tiver sido executada com sucesso, entradas a mais não devem atrapalhar, e devem ser ignorada pelo testador.

Para ilustrar a quantidade de resultados aplicáveis que são gerados no MOST, a MEFE da Figura 24 foi executada 12 vezes, e a quantidade de populações aplicáveis é mostrada no gráfico da Figura 25. A Figura 25 mostra a quantidade de populações geradas que são aplicáveis à MEFE em questão. Cada execução contém 1000 possíveis resultados, mas somente algumas são aplicáveis, e essa quantidade é mostrada no gráfico. A visualização do eixo X está no intervalo (0-100), para melhor visualização do resultado.

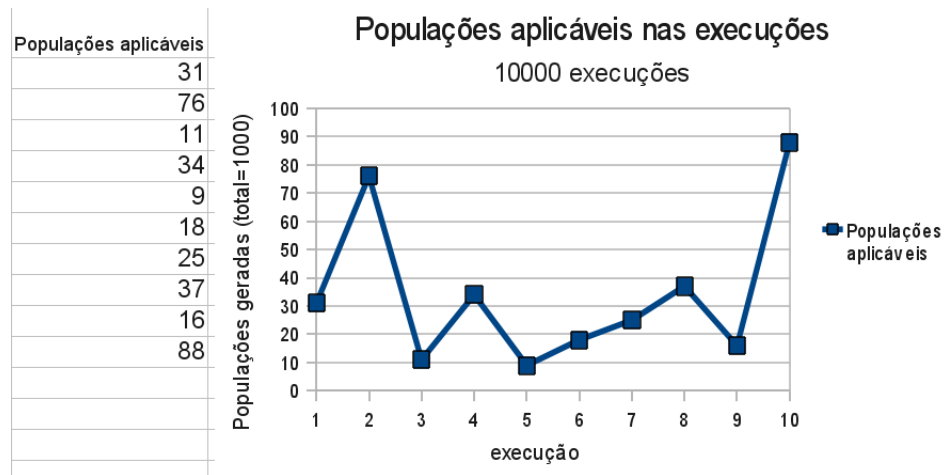


Figura 25: Gráfico da quantidade de saídas aplicáveis de uma MEFE exemplo na execução do MOST, com visualização do eixo Y de 100 saídas.

É possível perceber que a quantidade de saídas aplicáveis que o MOST apresenta é pequena em relação à quantidade de saídas geradas pelo $M-GEO_{vsl}$, porém, o MOST conseguiu encontrar soluções de tamanhos variados automaticamente. O objetivo não é otimizar o gasto computacional, mas sim gerar dados de teste utilizando uma heurística evolutiva e combinando a multi-objetividade. Além disso, sabe-se que a utilização de um algoritmo evolutivo pode ter um gasto computacional maior, entretanto, a utilização de uma heurística é considerada mais confiável do que processos manuais para geração dos dados de teste.

5 CONCLUSÃO

Buscando aplicar a proposta de (YANO, 2011), os algoritmos M-GEO_{vst} e MOST foram implementados. A implementação do problema proposto permitiu obter uma visão mais prática da solução do problema.

Conclui-se que a abordagem implementada é eficiente para a geração de dados de teste. A implementação do problema não é trivial e precisa de refinamentos. Alguns desses refinamentos são propostos nesse trabalho. O desenvolvimento do trabalho colaborou bastante para a ampliação dos conhecimentos na área de Testes de Software, especialmente na área de geração de dados para teste, baseado em modelos.

6 TRABALHOS FUTUROS

Como trabalhos futuros ficam as seguintes propostas:

- Adequação dos algoritmos implementados para abranger problemas que não sejam somente da geração de dados de teste;
- Implementar solução para MOST que considere aspectos de controle e de dados de uma MEFE.
- Melhorar o tratamento para caminhos inactíveis nos dados de teste gerados para uma determinada MEFE.
- Comparação mais detalhada dos resultados deste trabalho com os resultados do trabalho de (YANO, 2011).

Referências

ABREU, B. *Uma abordagem evolutiva para a geração automática de dados de teste*. Tese (Doutorado) — Master's thesis, IC/Unicamp, Campinas, SP, 2006.

AGUILAR-RUIZ, J.; RAMOS, I.; RIQUELME, J.; TORO, M. An evolutionary approach to estimating software development projects* 1. *Information and Software Technology*, Elsevier, v. 43, n. 14, p. 875–882, 2001.

ALBA, E.; CHICANO, J. Software testing with evolutionary strategies. *Rapid Integration of Software Engineering Techniques*, Springer, p. 50–65, 2006.

AMORIM, E.; ROMERO, R.; MANTOVANI, J. Fluxo de potência ótimo descentralizado utilizando algoritmos evolutivos multiobjetivo. *Sba: Controle*

& Automação Sociedade Brasileira de Automatica, SciELO Brasil, v. 20, n. 2, p. 217–232, 2009.

ANDROUTSOPOULOS, K.; GOLD, N.; HARMAN, M.; LI, Z.; TRATT, L. A theoretical and empirical study of efsm dependence. In: IEEE. *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*. [S.l.], 2009. p. 287–296.

BAK, P.; SNEPPEN, K. Punctuated equilibrium and criticality in a simple model of evolution. *Physical Review Letters*, APS, v. 71, n. 24, p. 4083–4086, 1993.

BOCHMANN, G.; PETRENKO, A. Protocol testing: review of methods and relevance for software testing. In: ACM. *Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis*. [S.l.], 1994. p. 109–124.

BOETTCHER, S.; PERCUS, A. Optimization with extremal dynamics. *Physical Review Letters*, APS, v. 86, n. 23, p. 5211–5214, 2001.

BOURHFIR, C.; DSSOULI, R. *et al.* Automatic test generation for efsm-based systems. Citeseer, 1996.

BRUNO T. ; MARTINS, E. . S. F. Automatic test data generation for path testing using a new stochastic algorithm. In: UBERLANDIA, MG, BRASIL. *XIX Simposio Brasileiro de Engenharia de Software*. [S.l.], 2005.

BUENO, P.; JINO, M. Automatic test data generation for program paths using genetic algorithms. *International Journal of Software Engineering and Knowledge Engineering*, Singapore; New Jersey: World Scientific, c1991-, v. 12, n. 6, p. 691–709, 2002.

BUZZO, A.; MARTINS, E.; SOUSA, F. Uso do algoritmo georeal para abordar a geração automática de dados de teste. p. 33, 2010.

CAVALLI, A.; LEE, D.; RINDERKNECHT, C.; ZAÏDI, F. Hit-or-jump: An algorithm for embedded testing with applications to in services. *Formal Methods for Protocol Engineering And Distributed Systems*, p. 41–56, 1999.

CHANSON, S.; ZHU, J. Automatic protocol test suite derivation. In: IEEE. *INFOCOM'94. Networking for Global Communications., 13th Proceedings IEEE*. [S.l.], 1994. p. 792–799.

CHENG, K.; KRISHNAKUMAR, A. Automatic functional test generation using the extended finite state machine model. In: ACM. *Proceedings of the 30th international Design Automation Conference*. [S.l.], 1993. p. 86–91.

CHENG, K.; KRISHNAKUMAR, A. Automatic generation of functional vectors using the extended finite state machine model. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, ACM, v. 1, n. 1, p. 57–79, 1996.

CHOW, T. Testing software design modeled by finite-state machines. *Software Engineering, IEEE Transactions on*, IEEE, n. 3, p. 178–187, 1978.

CLARKE, J.; DOLADO, J.; HARMAN, M.; HIERONS, R.; JONES, B.; LUMKIN, M.; MITCHELL, B.; MANCORIDIS, S.; REES, K.; ROPER, M. *et al.* Reformulating software engineering as a search problem. In: IET. *Software, IEE Proceedings-*. [S.l.], 2003. v. 150, n. 3, p. 161–175.

CUNHA, A.; OLIVEIRA, P.; COVAS, J. Use of genetic algorithms in multicriteria optimization to solve industrial problems. In: *Proceedings of the Seventh International Conference on Genetic Algorithms*. [S.l.: s.n.], 1997. p. 682–688.

DELAMARO, M.; MALDONADO, J.; JINO, M. Introdução ao teste de software. *Editora Campus*, 2007.

DSSOULI, R.; SALEH, K.; ABOULHAMID, E.; EN-NOUAARY, A.; BOURHFIR, C. Test development for communication protocols: towards automation. *Computer Networks*, Elsevier, v. 31, n. 17, p. 1835–1872, 1999.

DUALE, A.; UYAR, M. A method enabling feasible conformance test sequence generation for efsm models. *Computers, IEEE Transactions on, IEEE*, v. 53, n. 5, p. 614–627, 2004.

DURAN, J.; NTAFOSS, S. An evaluation of random testing. *Software Engineering, IEEE Transactions on, IEEE*, n. 4, p. 438–444, 1984.

EDVARDSSON, J. A survey on automatic test data generation. In: *Proceedings of the 2nd Conference on Computer Science and Engineering*. [S.l.: s.n.], 1999. p. 21–28.

EIBEN, A.; SMITH, J. *Introduction to evolutionary computing*. [S.l.]: Springer Verlag, 2003.

FONSECA, C.; FLEMING, P. Multiobjective optimization and multiple constraint handling with evolutionary algorithms. ii. application example. *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on, IEEE*, v. 28, n. 1, p. 38–47, 1998.

FONSECA, C.; FLEMING, P. *et al.* Genetic algorithms for multiobjective optimization: Formulation, discussion and generalization. In: CITESSEER. *Proceedings of the fifth international conference on genetic algorithms*. [S.l.], 1993. v. 423, p. 416–423.

FOURMAN, M. Compaction of symbolic layout using genetic algorithms. In: L. ERLBAUM ASSOCIATES INC. *Proceedings of the 1st International Conference on Genetic Algorithms*. [S.l.], 1985. p. 141–153.

FRANZEN, M.; BELLINI, C. Arte ou prática em teste de software. *REAd*, v. 11, n. 3, 2005.

GOLDBERG, D. *Genetic algorithms in search, optimization, and machine learning*. [S.l.]: Addison-wesley, 1989.

GROSS, H. Measuring evolutionary testability of real-time software. 2000.

HAJELA, P.; LIN, C. Genetic search strategies in multicriterion optimal design. *Structural and Multidisciplinary Optimization*, Springer, v. 4, n. 2, p. 99–107, 1992.

HARMAN, M. Automated test data generation using search based software engineering. In: IEEE COMPUTER SOCIETY. *Proceedings of the Second International Workshop on Automation of Software Test*. [S.l.], 2007. p. 2.

HARMAN, M.; JONES, B. Search-based software engineering. *Information and Software Technology*, Elsevier, v. 43, n. 14, p. 833–839, 2001.

HARMAN, M.; MANSOURI, S.; ZHANG, Y. Search based software engineering: A comprehensive analysis and review of trends techniques and applications. *Department of Computer Science, King's College London, Tech. Rep. TR-09-03*, 2009.

HIERONS, R.; KIM, T.; URAL, H. On the testability of sdl specifications. *Computer Networks*, Elsevier, v. 44, n. 5, p. 681–700, 2004.

HORN, J.; NAFPLIOTIS, N.; GOLDBERG, D. A niched pareto genetic algorithm for multiobjective optimization. In: IEEE. *Evolutionary Computation, 1994. IEEE World Congress on Computational Intelligence., Proceedings of the First IEEE Conference on*. [S.l.], 1994. p. 82–87.

HOWDEN, W. Reliability of the path analysis testing strategy. *IEEE Transactions on Software Engineering*, IEEE, p. 208–215, 1976.

INFOTECH, S. o. A. R. Infotech state-of-the-art report. *Software Testing*, Infotech International, v. 1, 1979.

ISHIBUCHI, H.; MURATA, T. Multi-objective genetic local search algorithm. In: IEEE. *Evolutionary Computation, 1996., Proceedings of IEEE International Conference on*. [S.l.], 1996. p. 119–124.

JACOBSON, I.; GRISS, M.; JONSSON, P. Software reuse: architecture, process and organization for business success. Addison-Wesley Professional, 1997.

JR, R. L. The features and facilities of estelle: a formal description technique based upon an extended finite state machine model. In: NORTH-HOLLAND PUBLISHING CO. *Proceedings of the IFIP WG6. 1 Fifth International Conference on Protocol Specification, Testing and Verification V*. [S.l.], 1985. p. 271–296.

KITA, R.; TREMBLAY, S.; LYNCH, T. *Method and apparatus for generating tests for structures expressed as extended finite state machines*. [S.l.]: Google Patents, fev. 28 1995. US Patent 5,394,347.

KURSAWE, F. A variant of evolution strategies for vector optimization. *Parallel Problem Solving from Nature*, Springer, p. 193–197, 1991.

LAKHOTIA, K.; HARMAN, M.; MCMINN, P. A multi-objective approach to search-based test data generation. In: ACM. *Proceedings of the 9th annual conference on Genetic and evolutionary computation*. [S.l.], 2007. p. 1098–1105.

LAUMANN, M.; RUDOLPH, G.; SCHWEFEL, H. A spatial predator-prey approach to multi-objective optimization: A preliminary study. In: SPRINGER. *Parallel Problem Solving from Nature—PPSN V*. [S.l.], 1998. p. 241–249.

LEE, D.; YANNAKAKIS, M. Principles and methods of testing finite state machines—a survey. *Proceedings of the IEEE*, IEEE, v. 84, n. 8, p. 1090–1123, 1996.

LI, X.; HIGASHINO, T.; HIGUCHI, M.; TANIGUCHI, K. Automatic test case derivation for communication protocols in an extended fsm model. *Electronics and Communications in Japan (Part I: Communications)*, Wiley Online Library, v. 82, n. 10, p. 50–60, 1999.

LIBRALÃO, G.; DELBEM, A. Codificação nó-profundidade com operador de recombinação para algoritmos evolutivos. XXV Congresso SBC, 2005.

MANSOUR, N.; SALAME, M. Data generation for path testing. *Software Quality Journal*, Springer, v. 12, n. 2, p. 121–136, 2004.

MANTERE, T.; ALANDER, J. Evolutionary software engineering, a review. *Applied Soft Computing*, Elsevier, v. 5, n. 3, p. 315–331, 2005.

MARTINS, E.; SABIÃO, S.; AMBROSIO, A. Condata: a tool for automating specification-based test case generation for communication systems. *Software Quality Journal*, Springer, v. 8, n. 4, p. 303–320, 1999.

- MCMINN, P.; HOLCOMBE, M. The state problem for evolutionary testing. In: SPRINGER. *Genetic and Evolutionary Computation—GECCO 2003*. [S.l.], 2003. p. 214–214.
- MICHAEL, C.; MCGRAW, G.; SCHATZ, M. Generating software test data by evolution. *Software Engineering, IEEE Transactions on*, IEEE, v. 27, n. 12, p. 1085–1110, 2001.
- MICHALEWICZ, Z.; SCHOENAUER, M. Evolutionary algorithms for constrained parameter optimization problems. *Evolutionary computation*, MIT Press, v. 4, n. 1, p. 1–32, 1996.
- OLIVEIRA, A. Algoritmos evolutivos híbridos com detecção de regiões promissoras em espaços de busca contínuos e discretos. *Algoritmos evolutivos híbridos com detecção de regiões promissoras em espaços de busca contínuos e discretos*, 2004.
- OSTRAND, T. White-box testing. *Encyclopedia of Software Engineering*, 2011.
- PARGAS, R.; HARROLD, M.; PECK, R. Test-data generation using genetic algorithms. *Software Testing Verification and Reliability*, Chichester, Sussex, England: J. Wiley, c1992-, v. 9, n. 4, p. 263–282, 1999.
- PARKS, G.; MILLER, I. Selective breeding in a multiobjective genetic algorithm. In: SPRINGER. *Parallel Problem Solving From Nature—PPSN V*. [S.l.], 1998. p. 250–259.
- PEDRYCZ, W.; PETERS, J. Engenharia de software: teoria e prática. *Rio de Janeiro: Campus*, 2001.
- PRESSMAN, R.; MARIA, G. M. *Engenharia de software*. [S.l.]: McGraw-Hill, 2006.

RAMALINGOM, T.; THULASIRAMAN, K.; DAS, A. Context independent unique state identification sequences for testing communication protocols modelled as extended finite state machines. *Computer Communications*, Elsevier, v. 26, n. 14, p. 1622–1633, 2003.

ROBINSON, H. Intelligent test automation. *Software Testing and Quality Engineering*, SQE, v. 2, p. 24–33, 2000.

RUDOLPH, G. On a multi-objective evolutionary algorithm and its convergence to the pareto set. In: IEEE. *Evolutionary Computation Proceedings, 1998. IEEE World Congress on Computational Intelligence., The 1998 IEEE International Conference on*. [S.l.], 1998. p. 511–516.

SARIKAYA, B.; BOCHMANN, G.; CERNY, E. A test design methodology for protocol testing. *Software Engineering, IEEE Transactions on*, IEEE, n. 5, p. 518–531, 1987.

SCHAFFER, J. Multiple objective optimization with vector evaluated genetic algorithms. In: L. ERLBAUM ASSOCIATES INC. *Proceedings of the 1st international Conference on Genetic Algorithms*. [S.l.], 1985. p. 93–100.

SMITH, R.; GOLDBERG, D.; EARICKSON, J. Sga-c: A c-language implementation of a simple genetic algorithm. Citeseer, 1991.

SOMMERVILLE, I. *Engenharia de Software, 6ª edição*. [S.l.: s.n.], 2003.

SRINIVAS, N.; DEB, K. Multiobjective optimization using nondominated sorting in genetic algorithms. *Evolutionary computation*, MIT Press, v. 2, n. 3, p. 221–248, 1994.

STHAMER, H. *The automatic generation of software test data using genetic algorithms*. Tese (Doutorado) — Citeseer, 1995.

SY, N.; DEVILLE, Y. Automatic test data generation for programs with integer and float variables. In: PUBLISHED BY THE IEEE COMPUTER SOCIETY. *ase*. [S.l.], 2001. p. 13.

TAUSWORTHE, R. Random numbers generated by linear recurrence modulo two. *Mathematics of Computation*, v. 19, n. 90, p. 201–209, 1965.

URAL, H.; YANG, B. A test sequence selection method for protocol testing. *Communications, IEEE Transactions on*, IEEE, v. 39, n. 4, p. 514–523, 1991.

UTTING, M.; LEGEARD, B. *Practical model-based testing: a tools approach*. [S.l.]: Morgan Kaufmann, 2007.

UTTING, M.; PRETSCHNER, A.; LEGEARD, B. A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability*, Wiley Online Library, 2011.

VALENZUELA-RENDÓN, M.; URESTI-CHARRE, E.; MONTERREY, I. A non-generational genetic algorithm for multiobjective optimization. In: CITeseer. *in Proceedings of the Seventh International Conference on Genetic Algorithms*. [S.l.], 1997.

VELDHUIZEN, D. V.; LAMONT, G. Evolutionary computation and convergence to a pareto front. In: CITeseer. *Late Breaking Papers at the Genetic Programming 1998 Conference*. [S.l.], 1998. p. 221–228.

WAINER, J. Métodos de pesquisa quantitativa e qualitativa para a ciência da computação. *Atualização em Informática*. Org: Tomasz Kowaltowski; Karin Breitman. Rio de Janeiro: Ed. PUC-Rio; Porto Alegre: Sociedade Brasileira de Computação, 2007.

WANG, Y. *Finding executable paths in protocol conformance testing*. Tese (Doutorado) — Simon Fraser University, 1990.

WATKINS, A.; HUFNAGEL, E. Evolutionary test data generation: a comparison of fitness functions. *Software: Practice and Experience*, Wiley Online Library, v. 36, n. 1, p. 95–116, 2006.

WEGENER, J.; BÜHLER, O. Evaluation of different fitness functions for the evolutionary testing of an autonomous parking system. In: SPRINGER. *Genetic and Evolutionary Computation—GECCO 2004*. [S.l.], 2004. p. 1400–1412.

WINDISCH, A.; WAPPLER, S.; WEGENER, J. Applying particle swarm optimization to software testing. In: ACM. *proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*. [S.l.], 2007. p. 1121–1128.

YANO, T. Uma abordagem evolutiva multiobjetivo para geracao automatica de casos de teste a partir de maquinas de estados. In: UNIVERSIDADE ESTADUAL DE CAMPINAS. [S.l.], 2011.

YANO, T.; MARTINS, E.; SOUSA, F. de. Most: a multi-objective search-based testing from efsm. In: IEEE. *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*. [S.l.], 2011. p. 164–173.

ZANUSSO, M. Fundamentos dos algoritmos geneticos. 2001. Disponível em: <<http://www.dct.ufms.br/~mzanusso/mestrado/AlgGen.doc>>.

ZITZLER, E.; DEB, K.; THIELE, L. Comparison of multiobjective evolutionary algorithms: Empirical results. *Evolutionary computation*, MIT Press, v. 8, n. 2, p. 173–195, 2000.

ZITZLER, E.; THIELE, L. Multiobjective optimization using evolutionary algorithms—a comparative case study. In: SPRINGER. *Parallel problem solving from nature—PPSN V*. [S.l.], 1998. p. 292–301.

ZITZLER, E.; THIELE, L. Multiobjective evolutionary algorithms: A comparative case study and the strength pareto approach. *Evolutionary Computation, IEEE Transactions on*, IEEE, v. 3, n. 4, p. 257–271, 1999.

ZUBEN, F. V. Computação evolutiva: uma abordagem pragmática. *Anais da I Jornada de Estudos em Computação de Piracicaba e Região (1a JECOMP)*, v. 1, p. 25–45, 2000.