Uma estratégia para geração de seqüências de verificação para máquinas de estados finitos

Paulo Henrique Ribeiro

SERVIÇO DE PÓS-GRADUAÇÃO DO ICMC-USP

Data de Depósito: 26 de outubro de 2010

Assinatura: ______________________

Uma estratégia para geração de seqüências de verificação para máquinas de estados finitos

Paulo Henrique Ribeiro

Orientador: ***Prof. Dr. Adenilso da Silva Simão***

Dissertação apresentada ao Instituto de Ciências Matemáticas e de Computação — ICMC/USP, como parte dos requisitos para obtenção do título de Mestre em Ciências de Computação e Matemática Computacional.

USP - São Carlos
Outubro/2010

Aos meus pais, Paulo e Claudete.

Agradecimentos

A Deus, por me proporcionar vários momentos de alegria em minha vida e por me oferecer forças para ultrapassar os obstáculos que existiram em meu caminho, colocando ao meu lado pessoas especiais que me ajudaram em todos os momentos.

Aos meus pais, Paulo e Claudete, por me incentivarem em minhas decisões em todos os momentos. Agradeço pelos bons princípios que me ensinaram e que irei levar sempre comigo. Agradeço meu irmão e sua esposa, André e Renata, minha irmã, Elaine, e meu sobrinho, João Vitor, pela amizade e por participar dos melhores momentos de minha vida. Agradeço a Kelly, pelo amor, compreensão e amizade. Aos meus avós, tios e primos, pelos bons momentos que tive junto a eles e por todo incentivo que ofereceram a mim.

Ao meu professor, orientador e amigo Adenilso, pela confiança demonstrada neste trabalho. Agradeço pelos ensinamentos e conselhos ao longo desses 5 anos de trabalho, que me auxiliaram em diversos momentos, e pela paciência por todo esse período.

Aos amigos do LabES, por tudo que me proporcionaram nesse período: Cutigi, Endo, Dusse, Gondim, Rodolfo, Rafael, Fabiano, Rodrigo, Vinícius, Frotinha, Cabeção, Marcão, Marcelo, Draylson, Vânia, Abe, Otávio, KLB, Leandro, Delinha, Gambi, Maria, Kátia, Erika, Marllos, Vanessa e Alex. Também aos professores do LabES que me auxiliaram no decorrer deste trabalho: Adenilso, Simone, Ellen e Masiero .

A todos meus amigos de Barretos pelos bons momentos que compartilhei junto a eles: amigos do ensino fundamental, do ensino médio, do cursinho, da mocidade espírita e da rodoviária. Aos amigos que fiz na Ícaro, Hominiss e Próxima Prime, que me proporcionaram novos conhecimentos e bons momentos durante o período que trabalhei junto a eles. Aos meus amigos de Ribeirão Preto e da Intelinet, pelos ensinamentos e momentos descontraídos: Cutigi, Gustavo, Homero, Ribeirão, Renan, Mário, Daniel, Roger, Adriana e Eliza.

Aos amigos da Info04 por tudo que vivemos durante a graduação: Cutigi, Andrezão, Teteco, Camisa, Nagao, WC, Fabi, Ki-Suco, Ribeirão, Carrara, Tio, Santana, Caneca, Danilo, Carpinelli, Togo, Xandão, Astro, Rubinho, Ramon, Nany, Alê, Tati, Gê, Bruno, Marco, Josi e ao meu grande amigo Mineiro.

Aos amigos que ajudaram na revisão desta dissertação: Adenilso, Cutigi, Dusse, Endo e Frotinha.

Aos professores do ICMC, do mestrado e da graduação, e a todos os professores que já tive, por todo o ensinamento passado, pela ajuda, pela amizade e pela confiança. Aos funcionários do ICMC, pelo auxílio prestado, e aos funcionários da segurança, pelas conversas e momentos de descontração.

A todas as pessoas que contribuíram de alguma forma para a realização deste trabalho.

Obrigado a todos.

Resumo

O Teste baseado em modelos tem como objetivo auxiliar a atividade de testes, gerando conjuntos de casos de teste a partir de modelos, como Máquinas de Estados Finitos (MEFs). Diversos métodos de geração de conjuntos de caso de teste têm sido propostos ao longo das últimas décadas, com algumas contribuições recentes. Dentre esses trabalhos, há os que geram seqüências de verificação, que são conjuntos de caso de teste formados por uma única seqüência e que são capazes de detectar os defeitos de uma implementação cujo comportamento pode ser modelado a partir de uma MEF. Neste trabalho é proposto um algoritmo de geração de seqüências de verificação que tem a finalidade de gerar seqüências menores que as seqüências geradas pelos métodos existentes. O algoritmo, que é baseado na técnica de algoritmos genéticos e nas condições de suficiência para a completude de casos de teste, consiste basicamente em criar novas seqüências a partir de seqüências menores. Por meio de mutações, novas seqüências são geradas pelo algoritmo. As condições de suficiência são utilizadas para determinar quais seqüências geradas são seqüências de verificação. Também são apresentados neste trabalho os estudos experimentais realizados para determinar o comportamento do algoritmo diante de diferentes contextos.

Abstract

Model-based testing aims at aiding the testing activity, generating test cases from models such as Finite State Machines (FSM). Several test cases generation methods have been proposed along the last decades, with some recent contributions. Among these works, there are those that generate checking sequences, which are test cases formed by a single sequence and which are capable of detecting faults in an implementation whose behavior can be modeled as an FSM. This work proposes a checking sequences generation algorithm which aims at generating sequences smaller than the sequences generated by existing methods. The algorithm, which is based on the genetic algorithms technique and sufficient conditions for completeness of test cases, basically consists of creating new sequences from small sequences. Through mutations, new sequences are generated by the algorithm. The sufficient conditions are used to determine which sequences are checking sequences. Experimental studies are presented in this work to determine the behavior of the algorithm on different contexts.

Sumário

Abstract **iii**

1 Introdução **1**
1.1 Contextualização 1
1.2 Motivação 4
1.3 Objetivo 5
1.4 Organização 5

2 Teste Baseado em Máquinas de Estados Finitos **7**
2.1 Considerações Iniciais 7
2.2 Fundamentos do Teste de Software 8
2.2.1 Conceitos e Definições 9
2.2.2 Fases de Teste 9
2.2.3 Técnicas de Teste 11
2.3 Teste Baseado em Modelos 12
2.3.1 Máquinas de Estados Finitos 13
2.4 Seqüências Básicas 15
2.5 Teste Baseado em Máquinas de Estados Finitos 17
2.5.1 Condições de Suficiência para Completude 19
2.5.2 Métodos de Geração de Seqüências de Verificação 22
2.5.2.1 Método de Gonenc (1970) 23
2.5.2.2 Método de Ural et al. (1997) 26
2.5.2.3 Método de Chen et al. (2005) 27
2.5.2.4 Método de Hierons e Ural (2002) e Hierons e Ural (2006) . 28
2.5.2.5 Método de Yalcin e Yenigun (2006) 30
2.5.2.6 Método de Simão e Petrenko (2008) 31
2.5.3 Confirmação de Seqüências de Verificação Baseada em Condições de Suficiência 33
2.6 Considerações Finais 37

3 Método de Geração de Seqüências de Verificação **39**
3.1 Considerações Iniciais 39
3.2 Algoritmo de Geração 40

3.3 Passos do Algoritmo de Geração . . . 41
3.4 Exemplo . . . 44
3.5 Aspectos da Implementação . . . 48
3.5.1 Gerador de Conjuntos de Distinção . . . 50
3.5.2 Verificador das Condições de Suficiência . . . 50
3.5.3 *Gerador de Seqüências de Verificação* . . . 52
3.6 Considerações Finais . . . 56

4 Avaliação Experimental **57**
4.1 Considerações Iniciais . . . 57
4.2 Estudo Experimental 1 . . . 58
4.2.1 Variação da Quantidade de Estados . . . 59
4.2.2 Variação da Quantidade de Entradas . . . 59
4.2.3 Variação da Quantidade de Saídas . . . 60
4.2.4 Variação da Quantidade de Transições . . . 61
4.3 Estudo Experimental 2 . . . 62
4.3.1 Comparação entre Parâmetros de Configuração . . . 62
4.3.2 Comparação entre Condições de Suficiências . . . 64
4.4 Estudo Experimental 3 . . . 66
4.4.1 Comparação entre Tempos de Execução . . . 66
4.4.2 Comparação entre Parâmetros de Configuração . . . 67
4.5 Considerações Finais . . . 68

5 Conclusões **71**
5.1 Contribuições . . . 72
5.2 Limitações . . . 72
5.3 Trabalhos Futuros . . . 73

Referências **74**

Lista de Figuras

2.1 Exemplo de MEF extraído de (Gonenc, 1970). 13
2.2 Exemplo de MEF extraído de Dorofeeva et al. (2005a). 24
2.3 Grafo-X_d. 24
2.4 Grafo-β. 25
2.5 Grafo-β reduzido. 26
2.6 MEF extraída de (Ural et al., 1997). 27
2.7 MEF extraída de (Chen et al., 2005). 28
2.8 Grafo G_D para a MEF da Figura 2.7. 30
2.9 MEF extraída de (Yalcin e Yenigun, 2006). 31
2.10 Árvore de teste. 34
2.11 Grafo de distinção construído a partir da árvore de teste. 35

3.1 Passos do algoritmo genético. 41
3.2 Exemplo de adição de seqüência de transferência e de remoção de entradas sobrepostas em uma *seqüência* χ. 42
3.3 Interface desenvolvida para o **CheSCon**. 52

4.1 Comparação entre os métodos variando a quantidade de estados. 59
4.2 Comparação entre os métodos variando a quantidade de entradas. 60
4.3 Comparação entre os métodos variando a quantidade de saídas. 61
4.4 Comparação entre os métodos variando a quantidade de transições. 62
4.5 Tamanho das seqüências de verificação e o tempo de execução variando do tamanho do conjunto *Fittest*. 63
4.6 Tamanho das seqüências de verificação e o tempo de execução variando do tamanho da população. 64
4.7 Comparação entre as condições de suficiência de Simão e Petrenko (2010) com as condições propostas por Ural et al. (1997). 65
4.8 Tempo de execução do **GGCS** e do **GGCS Distributed**. 67
4.9 Taxa de redução obtida pelo **GGCS Distributed** em relação ao **GGCS**. 68

Lista de Tabelas

2.1 Tabela representativa da MEF ilustrada na Figura 2.1. 14
2.2 Confirmação de nós do grafo de distinção ilustrado na Figura 2.11. 38
2.3 Confirmação da cobertura das transições da MEF a partir das seqüências confirmadas. 38

3.1 Passo 1 - Genes gerados. 45
3.2 Passo 2 - Cromossomos da população inicial. 45
3.3 Passo 3 - Cromossomos da população atual com suas respectivas seqüências χ. 46
3.4 Passo 4 - Cromossomos da população atual ordenados crescentemente a partir dos seus respectivos $fitness$. 47
3.5 Passo 5 - Novos cromossomos gerados. 48
3.6 Passo 6 - Nova População. 48
3.7 Cromossomos mais aptos em cada iteração. 49
3.8 Formato de entrada (MEF) e saída (Conjunto de Distinção) do Gerador de Conjuntos de Distinção. 50
3.9 Formato conjunto de casos de teste do Verificador das Condições de Suficiência. 51

Capítulo

1

Introdução

1.1 Contextualização

O crescente uso da Tecnologia da Informação (TI) em diversas atividades da sociedade está ocasionando ultimamente uma evolução dos recursos computacionais e um aumento da dependência de sistemas de software em vários ambientes, inclusive em atividades consideradas críticas. Entretanto, juntamente com a evolução dos recursos computacionais, ocorreu o aumento da complexidade dos sistemas utilizados, o que resultou na necessidade de verificar se os sistemas não possuem erros que possam prejudicar significativamente os seus usuários.

Desenvolver um software de qualidade contudo é uma tarefa árdua e a execução de testes em um programa torna-se uma etapa fundamental para que a aplicação atinja a qualidade desejável. Atividades de testes contribuem para a detecção de erros em um software, que podem ser causados por vários motivos, como um requisito implementado incorretamente ou um defeito existente no código fonte da aplicação.

A criação de modelos para representar uma implementação pode auxiliar na etapa de testes, pois permite obter um maior conhecimento sobre o software a ser testado. Um modelo pode ser definido como uma representação de uma aplicação, descrevendo o seu comportamento. Outro auxílio significativo que um modelo pode oferecer refere-se à criação de casos de teste, considerando que uma das maiores dificuldades do teste de

software é definir uma seleção de casos de teste que são válidos para o programa a ser testado e que tenham alta probabilidade de revelar defeitos no software.

Modelos de software não são considerados uma técnica nova, visto que os responsáveis por testar aplicações freqüentemente desenvolvem modelos, porém, muitas vezes de maneira informal, minimizando o auxílio que os modelos podem oferecer. Algumas técnicas permitem a criação de modelos formais a partir de grafos. Uma das maneiras de criar modelos de sistemas computacionais é feita com o uso de Máquinas de Transição de Estado, tais como Máquinas de Estados Finitos (MEFs), *Statecharts* e Redes de Petri.

Por ser considerada uma atividade difícil de ser realizada, a geração de conjuntos de casos de teste eficientes tem sido o tema de estudos de pesquisadores da área da computação. Vários métodos de geração de seqüências de testes foram propostos a partir do uso de Máquinas de Estados Finitos. Dada uma especificação de uma MEF e um software cuja implementação é baseada nessa especificação, é possível definir um conjunto de testes capaz de verificar se o software comporta-se de acordo com sua especificação, identificando assim os possíveis erros existentes.

Durante décadas vários pesquisadores implementaram diversos métodos para a geração de casos de teste utilizando MEF, como, por exemplo, o método de Hennie (1964), os métodos DS (Gonenc, 1970), W (Chow, 1978), UIO (Sabnani e Dahbura, 1988), UIOv (Vuong et al., 1989), Wp (Fujiwara et al., 1991), HSI (Petrenko et al., 1993), H (Dorofeeva et al., 2005b) e *State Counting* (Petrenko e Yevtushenko, 2005).

O ponto crucial desses métodos está na forma de garantir que uma implementação está em um estado conhecido depois da aplicação de alguma seqüência de entrada. Esse problema é normalmente simplificado quando a especificação da MEF tem uma seqüência de distinção (*distinguishing sequence*), que é uma seqüência de entrada que para cada estado diferente da MEF produz saídas diferentes. Entretanto, nem todas MEFs possuem uma seqüência de distinção. Adicionalmente, um conjunto de distinção é um conjunto de seqüências de entrada, uma para cada estado da MEF, tal que para cada par de estados distintos, sejam produzidas saídas diferentes. Um conjunto de distinção pode ser obtido a partir de uma seqüência de distinção. Entretanto, existem MEFs que possuem um conjunto de distinção, mas não uma seqüência de distinção (Boute, 1974).

Nesse contexto, alguns pesquisadores desenvolveram métodos para a geração de seqüência de verificação. Uma seqüência de verificação é um conjunto de casos de teste formado por apenas uma seqüência de entrada que pode ser usada para verificar se uma implementação está correta, dadas certas hipóteses de teste. Vários métodos têm como objetivo a geração de seqüência de verificação após a identificação de uma seqüência de distinção, como, por exemplo, os métodos propostos por Hennie (1964), Gonenc (1970),

Ural et al. (1997), Hierons e Ural (2002), Chen et al. (2005), Ural e Zhang (2006), Hierons e Ural (2006) e Simão e Petrenko (2008).

Hennie (1964) desenvolveu a base para os métodos de geração existentes. O método proposto pelo pesquisador utiliza seqüências de distinção e seqüências de transferência, que conduzem a MEF de um estado a outro, para gerar a seqüência de verificação. Gonenc (1970) propôs um método de geração de seqüência de verificação baseado em grafos, mas assim como o método de Hennie (1964), também faz uso de seqüências de distinção. Boute (1974) demonstrou como gerar uma seqüência de verificação para MEFs que podem não ter seqüências de distinção, mas possuem conjuntos de distinção.

A minimização dos conjuntos de casos de teste gerados a partir dos métodos desenvolvidos recentemente baseia-se em algumas condições para manter a efetividade do conjunto gerado em relação à detecção de erros. Essas condições, chamadas de *condições de suficiência*, têm o objetivo de garantir a completude de um conjunto de casos de teste, indicando que o conjunto é eficiente na detecção de erros. Dessa maneira, quando um conjunto de casos de teste satisfaz as condições de suficiência, pode-se garantir que o conjunto é n-completo, sendo n o número de estados da MEF. Um conjunto de casos de teste é considerado n-completo quando possui a propriedade de revelar a existência de diferenças entre uma MEF de n estados e uma implementação desenvolvida com base nessa MEF e que tenha no máximo n estados.

Há dois motivos principais para investigar condições de suficiência. O primeiro é baseado no fato que novas condições de suficiência podem aprimorar os métodos para geração de casos de teste, reduzindo o tamanho do conjunto sem perder a eficiência na detecção de erros. O segundo é que condições de suficiência menos exigentes podem facilitar a comprovação da completude de muitos métodos de geração de casos de teste existentes, além da possibilidade de aprimorá-los.

Ural et al. (1997) propuseram um teorema que define condições de suficiência para uma seqüência ser considerada uma seqüência de verificação, sendo que esse método exige que a MEF seja completa e possua uma seqüência de distinção. Um método também é sugerido por Ural et al. (1997), o qual aborda o problema de encontrar uma seqüência de verificação como o *Rural Chinese Postman Problem* (RCPP). Esse trabalho foi aperfeiçoado posteriormente por Hierons e Ural (2002), Chen et al. (2005), Hierons e Ural (2006) e Yalcin e Yenigun (2006), que utilizaram as condições de suficiência propostas por Ural et al. (1997). Entretanto, esses métodos trabalham somente com MEFs completas. Além disso, quando a implementação a ser testada tem a funcionalidade de *reset*, esses métodos não tentam utilizar a entrada *reset* para reduzir a seqüência de verificação. A operação *reset* é uma operação que "reinicia" corretamente a MEF, ou seja, leva a implementação ao seu estado inicial.

Simão e Petrenko (2010) apresentaram um conjunto de condições de suficiência para conjuntos de casos de teste n-completos que são menos exigentes do que os existentes atualmente na literatura. Um algoritmo também foi apresentado pelos autores, cujo propósito é determinar a completude de conjuntos de casos de teste. Simão e Petrenko (2008) propuseram um método para geração de seqüências de verificação que realiza a melhor escolha local em cada passo. Essa abordagem diverge dos métodos propostos em trabalhos recentes, como Hierons e Ural (2002), Chen et al. (2005), Ural e Zhang (2006) e Hierons e Ural (2006), os quais utilizam modelagem teórica de grafos para minimizar o tamanho da seqüência de verificação. Outra característica do método proposto por Simão e Petrenko (2008) é que, ao contrário dos demais métodos de geração de seqüências de verificação, ele pode ser utilizado para MEFs parciais.

1.2 Motivação

O teste de um sistema computacional é uma etapa importante no processo de desenvolvimento, pois é necessário elaborar e executar testes precisos para garantir que o programa desenvolvido seja confiável. O uso de MEFs para apoiar o teste de um software pode fornecer benefícios a todo o projeto, uma vez que fornece um modelo da aplicação implementada de uma forma mais clara e legível. Outro motivo para a utilização de MEF é a geração de casos de teste, que em muitas ocasiões é uma tarefa complexa. Entretanto, muitos métodos de geração de seqüências de testes retornam um conjunto com uma quantidade elevada de entradas, fato esse que torna a execução dessa seqüência na implementação um processo de alto custo computacional.

A redução da seqüência de verificação gerada por um método pode fornecer vantagens ao projeto, reduzindo o tempo de execução da seqüência na implementação que é desenvolvida a partir de um modelo. Apesar de ser uma área investigada há décadas, visto que a maioria dos métodos de geração de seqüências de verificação citados neste trabalho têm como objetivo a geração de seqüências cada vez menores, pesquisas ainda se fazem necessárias nesse contexto, pois os conjuntos de testes gerados pelos métodos existentes ainda podem ser reduzidos.

Os métodos mais eficientes de geração de seqüências de verificação, tais como os propostos por Hierons e Ural (2002), Chen et al. (2005), Ural e Zhang (2006) e Hierons e Ural (2006), são baseados nas condições de suficiência propostas por Ural et al. (1997). Investigar novas condições de suficiência faz com que novos métodos sejam propostos. As novas condições propostas por Simão e Petrenko (2010), que generalizam as condições elaboradas por Ural et al. (1997), podem ser exploradas para o desenvolvimento de métodos

de geração de seqüências de verificação, gerando seqüências menores do que as seqüências dos métodos atuais.

1.3 Objetivo

Este trabalho de mestrado tem como objetivo a investigação de um novo algoritmo para a geração de seqüências de verificação para MEFs, visando a geração de seqüências menores que as seqüências geradas por outros métodos existentes na literatura, mas garantindo a mesma efetividade na detecção de defeitos. Neste trabalho foi desenvolvido um algoritmo para a geração de seqüências de verificação baseado na técnica de algoritmos genéticos e nas condições de suficiência propostas por Simão e Petrenko (2010). Estudos experimentais também foram realizados para avaliar o comportamento do algoritmo proposto, determinando as vantagens e desvantagens em diferentes contextos.

1.4 Organização

Este trabalho está organizado da seguinte forma. No Capítulo 2 são apresentados os conceitos básicos sobre teste de software e teste baseado em modelos. Posteriormente, neste mesmo capítulo são abordados os principais conceitos relacionados a MEFs e a teste baseado em MEFs, além de métodos de geração de seqüências de verificação. Outro assunto apresentado neste capítulo são as condições de suficiência, base deste trabalho. Em seguida, no Capítulo 3 é apresentado o algoritmo de geração de seqüências de verificação proposto neste trabalho. Os resultados dos estudos experimentais do método proposto são apresentados no Capítulo 4. Por fim, no Capítulo 5 são apresentadas as conclusões, limitações e os trabalhos futuros deste trabalho de mestrado.

Capítulo

2

Teste Baseado em Máquinas de Estados Finitos

2.1 Considerações Iniciais

Sistemas cujas seqüências de ações são definidas em função do seu estado atual e da entrada fornecida podem ser modelados a partir de técnicas de modelagem formal que permitem descrever as ações do sistema em um grafo. O uso de uma técnica de modelagem formal de um software pode proporcionar benefícios na atividade de teste, pois durante a realização dessa atividade, um dos obstáculos é obter as características da implementação a ser testada. Enquanto uma especificação não rigorosa deixa margem a opiniões e especulações, um modelo formal apresenta de forma clara as propriedades da implementação.

A técnica de Teste Baseado em Modelos faz uso de modelos formais para gerar casos de teste, sendo que a geração de casos de testes é uma atividade considerada como um obstáculo durante os testes de um software. Como o modelo representa o comportamento da aplicação, determinando as possíveis ações durante a execução do software, podem-se utilizar métodos que, a partir de um modelo, geram conjuntos de casos de teste eficientes com o objetivo de encontrar defeitos na implementação, tornando menos complexa a atividade de teste.

Existem diversas técnicas de modelagem formal, tais como Máquinas de Estados Finitos (MEFs), Máquinas de Estados Finitos Estendidas (MEFEs), *Statecharts* e Redes de Petri. Devido ao escopo deste trabalho, neste capítulo são abordados os fundamentos sobre teste de software baseado em Máquinas de Estados Finitos. Na Seção 2.2 são apresentados os principais fundamentos relacionados ao teste de software. Teste baseado em modelos e Máquinas de Estados Finitos são apresentados na Seção 2.3. Na Seção 2.4 são descritos as principais seqüências básicas envolvidas com MEFs. O teste baseado em Máquinas de Estados Finitos é o assunto abordado na Seção 2.5, explicando os conceitos sobre Condições de Suficiência e sobre métodos de geração de Seqüências de Verificação.

2.2 Fundamentos do Teste de Software

O desenvolvimento de um software de qualidade é uma tarefa árdua. Apesar do uso de ferramentas e métodos sistemáticos de desenvolvimento, os defeitos permanecem presentes nos sistemas e, desse modo, o teste de um software torna-se uma etapa fundamental para que a aplicação atinja uma qualidade desejável.

As atividades de Verificação, Validação e Teste (VV&T) são consideradas uma das principais para garantir a qualidade de um sistema computacional. Verificação, segundo Pressman (2005), consiste de atividades que garantem que o software implementa corretamente uma função específica. A atividade de validação tem como objetivo garantir que o software desenvolvido esteja de acordo com os requisitos estabelecidos inicialmente. O teste é um conjunto de atividades que devem encontrar os erros em um programa. As atividades de teste são uma etapa importante em um projeto de desenvolvimento de um software, visto que essas atividades buscam identificar os defeitos ainda existentes no software antes que o usuário final inicie a sua utilização.

Segundo Pressman (2005), a execução da atividade de teste é realizada com sucesso se ela conseguir detectar a presença de defeitos na aplicação. O objetivo secundário dessa atividade é fornecer uma indicação de confiabilidade e qualidade do software testado. O fornecimento dessa indicação é plausível porque se a atividade de teste não detectar a presença de defeitos, considerando que o conjunto de casos de teste possui uma alta qualidade na detecção de defeitos, pode-se afirmar que o software está funcionando de acordo com as suas especificações e que os requisitos estipulados em um momento anterior foram corretamente implementados na aplicação. Entretanto, o teste não pode indicar a ausência absoluta de erros, mas apenas revelar que defeitos estão presentes no software.

2.2.1 Conceitos e Definições

Vários termos foram definidos com o objetivo de evitar ambigüidade e facilitar o entendimento dos conceitos relacionados à área de teste de software. Dois institutos internacionais elaboraram dois conjuntos de definições para padronizar alguns conceitos relacionados a teste de software. O *Institute of Electrical And Electronics Engineers* (IEEE) estabeleceu um glossário padrão (IEEE Std 610.12, 1999) e o *British Standards Institution* desenvolveu um vocabulário de teste de software (BS 7925-1, 1998).

Os principais termos e expressões serão descritos sucintamente segundo essas especificações de teste de software:

Sistema em teste: (*system under test* ou SUT) é o sistema, subsistema ou componente a ser testado.

Caso de teste: (*test case*) é o conjunto de dados de entrada, condições de execução e resultados esperados elaborados para atingir um objetivo específico de teste, verificando a conformidade de um determinado requisito.

Conjunto de teste: (*test case suite* ou *test suite*) é a coleção de um ou mais casos de teste de um sistema em teste.

Defeito: (*fault*) elementos incorretos que possam estar presentes em um software, como passos, processos ou definições de dados.

Engano: (*mistake*) ação ou decisão de um indivíduo envolvido no processo de desenvolvimento que tenha levado à inserção de um defeito.

Erro: (*error*) diferença entre um valor produzido por uma implementação e aquele que seria esperado, segundo a especificação.

Falha: (*failure*) exibição de um erro, normalmente provocada pela manifestação de um defeito.

Neste trabalho os termos "erro" e "defeito" serão utilizados como sinônimos para indicar uma causa. O termo "falha" é utilizado para indicar uma manifestação externa de um erro, ou seja, uma conseqüência.

2.2.2 Fases de Teste

Por se tratar de uma atividade crucial no desenvolvimento de um software, a atividade de teste deve ser planejada e executada detalhadamente para que os seus resultados sejam satisfatórios. A divisão dessa atividade em fases proporciona alguns benefícios,

pois permite que diferentes tipos de defeitos do software sejam abordados, estabelecendo estratégias adequadas para a geração de casos de teste confiáveis, o que minimiza a complexidade existente na atividade.

As atividades de teste em diferentes fases do processo de desenvolvimento podem ser classificadas em cinco grupos, conforme descrito a seguir:

Teste de Unidade: explora a menor unidade do projeto procurando por falhas de implementação. Nessa fase é testada a interface para verificar os parâmetros de entrada e saída, as estruturas de dados para avaliar a integridade dos dados armazenados e as condições de limite para garantir se a unidade opera adequadamente nos limites estabelecidos.

Teste de Integração: identifica falhas associadas às interfaces entre módulos de um software. Nessa fase as unidades testadas são integradas. Muitas falhas podem ser identificadas com essa atividade de teste, tais como interfaces incorretas, falta ou conflito de funcionalidades, violação da integridade de arquivos e estruturas de dados globais, seqüência incorreta de unidades, tratamento de erros (exceções) incorreto, problema de configuração e falta de recursos para atender a demanda das unidades.

Teste de Sistema: avalia o software em busca de falhas por meio da sua utilização. Tem como objetivo demonstrar que o sistema implementa os requisitos funcionais e não funcionais. Todos os elementos que compõe o ambiente no qual estará inserido o sistema devem ser considerados, incluindo a plataforma de execução e outros sistemas externos que, de alguma forma, estão relacionados ao sistema em teste.

Teste de Aceitação: são realizados geralmente por um grupo de usuários finais do sistema que irão decidir se aceitam ou não o sistema, verificando se a aplicação contém as funcionalidades acordadas no início do projeto.

Teste de Regressão: consiste em aplicar, a cada nova versão do software, os testes aplicados anteriormente. O objetivo é validar modificações realizadas e demonstrar que estas não afetaram as partes inalteradas.

Apesar da divisão por fases da atividade de teste minimizar a sua complexidade, ainda é importante garantir uma maneira confiável para a execução de cada uma das fases citadas anteriormente. A separação em etapas de cada uma destas fases proporciona uma maior confiabilidade na geração, execução e avaliação dos casos de teste, garantindo assim um teste mais eficaz na detecção de defeitos. Dessa maneira, cada fase de teste deve envolver quatro etapas básicas: **planejamento do teste**, etapa onde são formulados quais os testes que serão realizados; **projeto dos casos de teste**, que consiste na elaboração de

um conjunto de casos de teste que atenda os critérios estabelecidos; **execução do teste**, em que o programa é executado com os casos de teste anteriormente criados; e, por último, a **avaliação dos resultados**, que consiste em avaliar as saídas produzidas pelo teste para que ações posteriores possam ser tomadas.

2.2.3 Técnicas de Teste

O teste exaustivo, que consiste de testar a aplicação com todas as suas entradas possíveis, tem uma alta confiabilidade para a detecção de erros em um software. Entretanto, é inviável devido ao fato de que o conjunto de entradas pode ser infinito. Logo, determinar as entradas a serem utilizadas durante os testes é um fator importante e essa escolha é feita de acordo com as informações existentes sobre o software.

A atividade de teste tem como dependência os dados existentes sobre a aplicação a ser avaliada. A presença ou a ausência de dados irão influenciar diretamente os testes que podem ser realizados em um software, visto que um dos objetivos dessa tarefa é verificar se a implementação está de acordo com os requisitos estabelecidos anteriormente.

Existem várias técnicas de teste que podem ser utilizadas para avaliar um sistema computacional. Cada uma dessas técnicas tem características únicas que as fazem mais apropriadas em determinadas situações. Essas técnicas de teste podem ser agrupadas basicamente em três grupos, descritas a seguir:

Teste Funcional: o objetivo dessa técnica é encontrar discrepâncias entre o comportamento do sistema e o descrito inicialmente em sua especificação. Os requisitos de teste são estabelecidos a partir da especificação do software, não considerando necessariamente a sua estrutura. Segundo Pressman (2005), nesse tipo de teste procura-se descobrir erros relacionados a cinco categorias: funcionalidades incorretas ou omitidas, erros de interface, erros de estrutura de dados ou de acesso a dados externos, erros de comportamento ou desempenho e, por fim, erros de iniciação e término.

Teste Estrutural: tem como principal objetivo testar os detalhes do código fonte de um software. Ao utilizar a técnica estrutural para testar uma aplicação, o testador irá gerar os casos de teste a partir de uma observação da estrutura interna do programa e executá-los para observar a saída retornada para as entradas fornecidas. Segundo Pressman (2005), os defeitos que podem ser descobertos por meio dessa técnica são: defeitos lógicos, pressuposições incorretas, defeitos de projeto e defeitos tipográficos.

Teste Baseado em Erros: possui a finalidade de testar um software com base nos erros típicos e comuns cometidos durante o desenvolvimento do software. Essa técnica

utiliza informações sobre os erros que podem existir em um software para a derivação dos casos de teste. Essas informações geralmente variam devido às características da linguagem de programação, método ou técnica utilizada ao longo do desenvolvimento do software.

A diferença existente entre essas técnicas refere-se à origem da informação utilizada na construção e na avaliação dos conjuntos de casos de teste (Maldonado, 1991). Essas técnicas são complementares e devem ser utilizadas em conjunto, pois revelam classes diferentes de erros (Pressman, 2005).

2.3 Teste Baseado em Modelos

A técnica de Teste Baseado em Modelos faz uso de modelos formais para gerar casos de teste, atividade considerada como obstáculo durante os testes de um software. Como o modelo representa o comportamento da aplicação, determinando as possíveis ações durante a execução do software, podem-se utilizar métodos que, a partir de um modelo, geram casos de teste eficientes com o objetivo de encontrar defeitos na implementação, tornando menos complexa a atividade de teste.

O processo de teste baseado em modelos pode ser dividido em quatro fases, todas descritas resumidamente a seguir:

- **Modelagem do comportamento do sistema:** essa fase tem como objetivo a criação de um modelo formal que representa a implementação a ser testada utilizando alguma das técnicas de modelagem citadas anteriormente, tais como MEFs, MEFEs, *Statecharts* e Redes de Petri.

- **Geração dos casos de teste:** o modelo criado na fase anterior é utilizado nos métodos de geração casos de teste.

- **Execução do teste:** os casos de teste gerados na fase anterior são executados no software.

- **Avaliação do teste:** o comportamento do software durante a execução dos casos de teste é comparado com o comportamento esperado, determinando a existência de erros na aplicação testada.

Apesar de proporcionar benefícios na atividade de testes, o desenvolvimento de modelos formais é uma tarefa que deve ser executada de maneira criteriosa, visto que todo o comportamento do sistema deve ser especificado no modelo. Existem diversas técnicas para modelar formalmente uma aplicação. As técnicas mais conhecidas são Máquinas de

Estados Finitos (MEFs), Máquinas de Estados Finitos Estendida (MEFEs), *Statecharts* e Redes de Petri, todas baseadas em Máquinas de Transições de Estados. Elas diferem entre si em características relativas à forma como certos elementos são explícita ou implicitamente representados. Em geral, as Máquinas de Estados Finitos (MEFs) representam os modelos mais simples, que estão relacionadas diretamente a este trabalho e são descritas a seguir.

2.3.1 Máquinas de Estados Finitos

Uma Máquina de Estados Finitos (MEF) é uma máquina hipotética composta por estados e transições (Gill, 1962). Essa máquina pode estar em apenas um estado em um determinado momento e a interligação dos estados é feita pelas transições. Cada transição liga um estado s_i a um estado s_j (s_i e s_j podem ser o mesmo estado). Ao receber uma entrada, a máquina muda de estado e gera uma saída. Tanto o evento de saída gerado quanto o estado atingido são definidos em função do estado atual e do evento de entrada (Davis, 1988). Uma MEF pode ser representada graficamente, por meio de um grafo direcionado ou por meio de uma tabela de transição. No primeiro caso, os estados são apresentados por nós e as transições por arcos, sendo que cada transição está relacionada a uma entrada e a uma saída produzida em resposta a essa entrada. Na Figura 2.1 tem-se a representação de uma MEF com seis estados, baseada na MEF de Gonenc (1970).

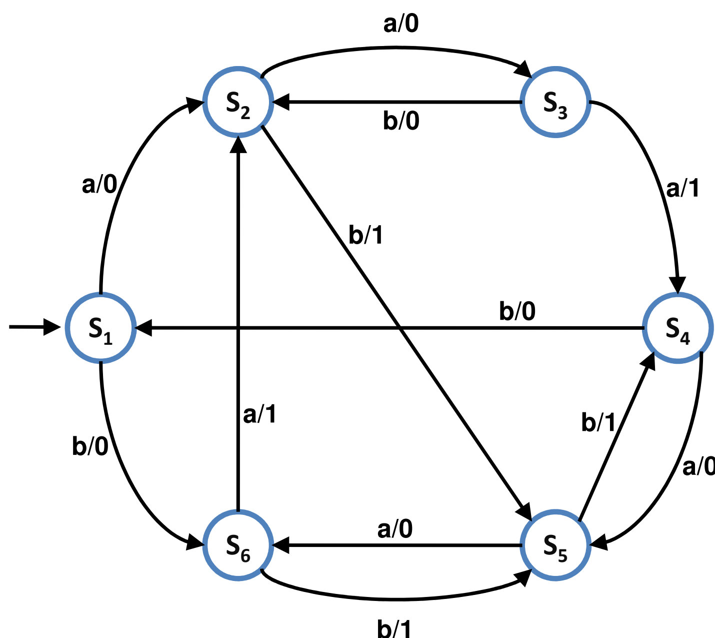


Figura 2.1: Exemplo de MEF extraído de (Gonenc, 1970).

Em uma tabela de transição, os estados são representados por linhas e as entradas, por colunas (Davis, 1988). Por exemplo, a tabela de transição da MEF da Figura 2.1 é apresentada na Tabela 2.1.

Tabela 2.1: Tabela representativa da MEF ilustrada na Figura 2.1.

	Saída		Próximo Estado	
Estado \ Entrada	a	b	a	b
s_1	0	0	s_2	s_6
s_2	0	1	s_3	s_5
s_3	1	0	s_4	s_2
s_4	0	0	s_5	s_1
s_5	0	1	s_6	s_4
s_6	1	1	s_2	s_5

Uma MEF A pode ser representada formalmente por uma tupla $(S, s_0, X, Y, D, \delta, \lambda)$ (Petrenko e Yevtushenko, 2005) , onde:

- S é um conjunto finito de estados, incluindo o estado inicial s_0.
- X é um conjunto finito de entradas.
- Y é o conjunto finito de saídas.
- D é um domínio da especificação, $D \subseteq S \times X$.
- δ é uma função de transição, $\delta : D \rightarrow S$.
- λ é uma função de saída, $\lambda : D \rightarrow Y$.

O conjunto S de estados representa o conjunto de todas as configurações possíveis do sistema em relação aos símbolos de entrada e saída. A MEF da Figura 2.1 possui o conjunto de estados $S = \{s_1, s_2, s_3, s_4, s_5, s_6\}$, sendo s_1 o estado inicial. O conjunto finito de entradas X descreve as variáveis de entrada do sistema. O alfabeto de entrada da MEF ilustrada na Figura 2.1 é $X = \{a, b\}$. O conjunto finito de saída Y descreve as variáveis de saída do sistema. A MEF da Figura 2.1 possui o alfabeto de saída $Y = \{0, 1\}$.

Uma tupla $(s, x) \in D$ é uma transição definida de M. Um seqüência $\alpha = x_1x_2...x_k$ é dito ser uma seqüência de entrada definida no estado $s \in S$ se existe $s_1, s_2, ..., s_{k+1}$, onde $s_1 = s$, tal que $(s_i, x_i) \in D$ e $\delta(s_i, x_i) = s_{i+1}$ para todo $1 < i < k$. Denota-se por $\Omega(s)$ o conjunto de todas as seqüências de entradas definidas no estado s.

A função de transição δ e a função de saída λ são estendidas para seqüência de entradas definida, incluindo a seqüência vazia, que é denotada por ϵ. Tem-se que $\delta(s, \epsilon) = s$ e $\lambda(s, \epsilon) = \epsilon$ para todo $s \in S$. Seja β uma seqüência de entradas e $\delta(s, \beta) = s'$, então, para todo $x \in X$ define-se $\delta(s, \beta x) = \delta(s', x)$ e $\lambda(s, \beta x) = \lambda(s, \beta)\lambda(s', x)$.

A motivação no uso de MEFs na modelagem de software deve-se à existência de uma classe de sistemas que apresentam dois tipos de primitivas: estímulos e operações. Estímulos são entradas do mundo real para o sistema e operações são eventos causados pelo sistema pela ativação de operações como resposta a algum estímulo. Exemplos desses sistemas são os aplicativos de áreas como Analise Léxica, Processamento de Dados, Controle de Processos em Tempo Real, Protocolos de Comunicação e Telecomunicações, além de outros sistemas reativos. Esses tipos de software podem ser modelados por meio de uma MEF, o que proporciona benefícios na atividade de testes, como a geração de casos de teste.

2.4 Seqüências Básicas

Algumas seqüências básicas de entrada são utilizadas freqüentemente por métodos de geração de conjuntos de casos de teste. Essas seqüências muitas vezes são utilizadas para a obtenção de um resultado parcialmente satisfatório na geração de seqüências de teste (Simão, 2007).

O conjunto **state cover** de uma MEF, normalmente referenciado por Q, é um conjunto de seqüências de entrada que faz com que todos os estados de uma MEF sejam atingidos. Para uma MEF com n estados, um conjunto *state cover* contém n seqüências, incluindo a seqüência vazia ϵ. A seqüência ϵ faz com que a MEF fique em seu estado inicial, enquanto que as demais seqüências levam a máquina para os demais estados. Desse modo, para todo estado $s \in S$, existe uma seqüência de entrada $\alpha \in Q$, tal que $\delta(s_0, \alpha) = s$. A MEF da Figura 2.1 possui o conjunto *state cover* $Q = \{\epsilon, a, aa, aaa, ab, b\}$.

Um conjunto de seqüências de entrada é considerado um **transition cover** de uma MEF M se, quando aplicado a M, exercita ao menos uma vez cada uma das transições definidas em M (Naito e Tsunoyama, 1981). O conjunto *transition cover*, normalmente chamado como conjunto P, pode ser obtido a partir de um conjunto *state cover*, concatenando cada uma das seqüências de entrada do *state cover* com as entradas definidas no estado final atingido pela aplicação de uma seqüência do *state cover*. Ou seja, para cada estado $s \in S$ e para cada entrada $x \in X$, existe uma seqüência de entrada $\alpha \in P$ tal que $\delta(s_0, \alpha) = s$ e $\alpha.x \in P$. Considerando a MEF da Figura 2.1, tem-se o conjunto *transition cover* $P = \{\epsilon, a, b, aa, ab, aaa, aab, aaaa, aaab, aba, abb, ba, bb\}$.

Uma **seqüência de transferência** χ de s_i para s_j, é uma seqüência que conduz M do estado s_i para o estado s_j, ou seja, $\delta(s_i, \chi) = s_j$. Uma **seqüência de distinção**, denominada DS (*Distinguishing Sequence*), de uma MEF é uma seqüência de entrada que, para cada estado da MEF, produz saídas diferentes. Ou seja, para $s_i, s_j \in S$ tal que $i \neq j$, $\lambda(s_i, DS) \neq \lambda(s_j, DS)$. Assim, é possível identificar em que estado a MEF estava antes da aplicação da DS. Considerando a MEF da Figura 2.1, tem-se a seqüência de distinção $DS = \{aab\}$.

Um **conjunto de distinção** é um conjunto de seqüências de entrada, uma para cada estado da MEF M, tal que para cada par de estados distintos, as respectivas seqüências de entradas desses estados tem um prefixo comum que produz saídas diferentes (Simão e Petrenko, 2009). Como exemplo, o conjunto de distinção para a MEF da Figura 2.1 é $DSet = \{aab, aab, ab, aab, aab, ab\}$. Um conjunto de distinção pode ser determinado a partir de seqüências de distinção. Entretanto, existem MEFs que não possuem seqüência de distinção, mas possuem conjuntos de distinção (Boute, 1974). Os conjuntos de distinção correspondem à seqüência de distinção adaptativas investigadas por Lee e Yannakakis (1994).

Uma **seqüência única de entrada e saída** (*Unique Input/Output Sequence*), também referenciada como seqüência UIO, de um estado s_i é uma seqüência de entrada de identificação de s_i. Com a aplicação de uma seqüência UIO em uma MEF M, pode-se distinguir s_i de qualquer outro estado de M, pois a saída produzida é específica (única) do estado s_i. Dessa maneira, para $s_i, s_j \in S$ tal que $i \neq j$, $\lambda(s_i, UIO_i) \neq \lambda(s_j, UIO_i)$. Um conjunto de distinção, apresentado anteriormente, é sempre composto por seqüências UIO, porém um conjunto de seqüência UIO pode não ser um conjunto de distinção, uma vez que em um conjunto de distinção é necessária a existência de prefixos comuns entre as seqüências do conjunto, requisito não obrigatório para um conjunto de seqüências UIO. A MEF da Figura 2.1 possui as seqüências $UIO_1 = ba$, $UIO_2 = ab$, $UIO_3 = ab$, $UIO_4 = bb$, $UIO_5 = bb$ e $UIO_6 = ab$. É possível observar que em determinados pares de seqüências UIO, não existe um prefixo comum entre as seqüências de entrada, algo obrigatório em um conjunto de distinção.

Dois estados $s_i, s_j \in S$ são **equivalentes** se $\lambda(s_i, \gamma) = \lambda(s_j, \gamma)$ para todo $\gamma \in \Omega(s_i) \cap \Omega(s_j)$. Esse conceito pode ser aplicado em estados de MEFs diferentes. MEFs são equivalentes se seus estados iniciais são equivalentes. De forma análoga, dois estados, $s_i, s_j \in S$ são **distinguíveis** se $\lambda(s_i, \gamma) \neq \lambda(s_j, \gamma)$ para todo $\gamma \in \Omega(s_i) \cap \Omega(s_j)$. MEFs são distinguíveis se seus estados iniciais são distinguíveis.

Duas seqüências α, β são **distinguíveis** se existe $\alpha\gamma, \beta\gamma$, tal que $\lambda(\delta(s_0, \alpha), \gamma) \neq \lambda(\delta(s_0, \beta), \gamma)$. De forma análoga, duas seqüências α, β são **T-distinguíveis** se existe

$\alpha\gamma, \beta\gamma \in T$, tal que $\lambda(\delta(s_0, \alpha), \gamma) \neq \lambda(\delta(s_0, \beta), \gamma)$. Duas seqüências α, β são **T-equivalentes** se existe $\alpha\gamma, \beta\gamma \in T$, tal que $\lambda(\delta(s_0, \alpha), \gamma) = \lambda(\delta(s_0, \beta), \gamma)$.

Uma **seqüência de teste** (ou **caso de teste**) é uma seqüência de entrada definida na MEF. Um **conjunto de seqüências de teste** é um conjunto finito de seqüências de teste, tal que não existam dois casos de teste α e β em que α é prefixo de β. O tamanho de um conjunto de seqüências de teste é obtido pelo número de símbolos de entrada contido no conjunto adicionado com o número de operações *resets*.

A operação **reset** é uma operação especial que leva a MEF de qualquer estado para o estado inicial e com saída nula. É denotada pela letra r e aparece como primeiro símbolo em uma seqüência de teste. Portanto, o número de *resets* de um conjunto de teste é o número de seqüências que ele possui.

Na geração de testes a partir de MEFs, assume-se que a implementação pode ser modelada como uma MEF contida em um domínio de defeitos. Essa hipótese, conhecida como hipótese de teste, é necessária para que um conjunto finito de testes possa ser gerado (Chow, 1978; Ural et al., 1997; Hierons e Ural, 2006; Hennie, 1964). $\Im(M)$ denota o domínio de defeitos definido pelo conjunto de todas as MEFs com o mesmo alfabeto de entrada e no máximo o mesmo número de estados de M, utilizado por grande parte dos métodos de geração, como por exemplo, os métodos W (Chow, 1978), Wp (Fujiwara et al., 1991), HSI (Petrenko et al., 1993; Luo et al., 1994), H (Dorofeeva et al., 2005b), entre outros. $\Im_T(M)$ denota o domínio de defeitos definido pelo conjunto N de todas as MEFs pertencentes a $Im(M)$, tal que N e M são T-equivalentes.

Um conjunto de seqüências de teste T é **n-completo**, ou simplesmente **completo**, se para cada MEF $N \in \Im(M)$ tal que N e M são distinguíveis, existe uma seqüência pertencente a T que distingue N de M. Ou seja, um conjunto de caso de teste é considerado n-completo quando possui a propriedade de revelar a existência de diferenças entre uma MEF de n estados e uma implementação desenvolvida com base nessa MEF e que tenha no máximo n estados.

2.5 Teste Baseado em Máquinas de Estados Finitos

Sistemas tais como protocolos de comunicação, sistemas de telecomunicações e outros sistemas reativos, apesar de geralmente serem implementados utilizando uma linguagem de programação procedimental ou orientada a objetos, podem ter o seu comportamento modelados por meio de uma MEF. Porém, se uma implementação I foi desenvolvida com base em uma MEF M, assume-se que I pode ser modelada como M, e, dessa maneira, deve-se verificar a conformidade de I com M. A implementação I está em conformidade com a especificação M se e somente se para cada seqüência de entrada no qual comporta-

mento de M seja definido, I comporta-se de maneira idêntica. Diz-se que a implementação é *quasi-equivalente* à especificação (Gill, 1962; Sidhu e Leung, 1989).

O teste baseado em MEF tem como objetivo gerar um conjunto de casos de teste a partir de uma MEF, que representa um modelo do software a ser testado, de tal forma que consiga detectar os defeitos existentes em uma implementação. Caso o conjunto de teste gerado pelo método seja considerado n-completo e não encontre nenhuma falha no sistema implementado, pode-se concluir que a implementação está de acordo com a sua especificação, que nesse caso é a própria MEF.

Dessa forma, o teste baseado em MEF considera que tanto a especificação quanto a implementação sejam modeladas por uma MEF. Assim, a implementação contém uma falha no caso em que possuir um comportamento diferente em relação à especificação. Segundo Chow (1978), os defeitos podem ser dos tipos:

- **Estados Faltantes:** quando os estados existente na implementação precisam ser aumentados para tornar a implementação equivalente à especificação.
- **Estados Extras:** quando os estados existente na implementação precisam ser reduzidos para tornar a implementação equivalente à especificação.
- **Falha de Transferência:** quando o estado atingido por uma transição não é o correto.
- **Falha de Saída:** quando a saída gerada por uma transição não é a correta.

Diversos métodos têm sido propostos para a geração de conjunto de casos de teste a partir de MEFs. Para um conjunto de seqüências de teste gerado a partir de uma MEF, uma questão importante refere-se em como avaliar a efetividade (ou qualidade) desse conjunto, ou seja, avaliar sua cobertura em relação aos erros revelados. O conjunto de teste é n-completo, onde n é o número de estados de M, se conseguir distinguir M de todas as outras MEFs possíveis com no máximo n estados. Aplicando um conjunto de teste n-completo na especificação M e na implementação I, M e I devem fornecer saídas idênticas. Caso contrário, a implementação não está em conformidade com a especificação. Os métodos de geração de casos de teste têm o objetivo de gerar conjuntos n-completos, para assim garantir a efetividade na detecção de defeitos na implementação.

Outra questão importante nesse contexto refere-se às características de uma MEF, uma vez que algumas características são requisitos para a utilização de métodos de geração de casos de teste. A seguir são definidas as principais características relacionadas às MEFs:

- **Completa:** propriedade existente quando todos os estados possuem transições definidas para cada elemento do conjunto de entrada. Caso um ou mais estados não possuam pelo menos um símbolo de entrada, a MEF é considerada parcial.

- **Fortemente conexa:** propriedade atribuída à MEF quando, para qualquer par de estados s_i e s_j, houver uma seqüência que faça a MEF atingir o estado s_j a partir do estado s_i.

- **Determinística:** ocorre quando em cada estado da MEF existe apenas uma transição para uma dada entrada.

- **Mínima**: quando os estados, tomados par-a-par, são distinguíveis, ou seja, a MEF não possui estados equivalentes. Os estados são considerados equivalentes quando produzem as mesmas saídas para uma mesma entrada.

- **Máquina de Mealy:** as ações de entrada e saída são representadas pelas transições.

Segundo Fujiwara et al. (1991), um método de geração de conjuntos de seqüências de teste tem como objetivo oferecer a possibilidade de executar atividades de teste e validação em sistemas modelados de acordo com alguma técnica de modelagem, de forma sistemática, por meio de procedimentos bem definidos para a geração dessas seqüências.

Esses métodos diferem principalmente pela efetividade, pelo tamanho do conjunto de seqüências de teste gerado e pelo custo computacional para gerar esse conjunto. Os métodos para geração de conjuntos de seqüências de teste possuem como característica comum a utilização da funcionalidade *reset*, que reinicia a MEF, levando ao seu estado inicial. Os principais métodos de geração de conjuntos de seqüências de testes são: W (Chow, 1978), Wp (Fujiwara et al., 1991), HSI (Petrenko et al., 1993) e State Counting (Petrenko e Yevtushenko, 2005).

Além dos métodos de geração de conjuntos de seqüências de teste, há também os métodos de geração de seqüências de verificação, que são conjuntos n-completos formados por apenas uma seqüência de entradas e que serão explicadas com mais detalhes na Seção 2.5.2.

2.5.1 Condições de Suficiência para Completude

As condições de suficiência são condições que, uma vez satisfeitas, garantem que o conjunto de casos de teste possui a propriedade de ser n-completo. Desse modo, além de proporcionar um meio para avaliar se um conjunto de casos de teste é eficiente na detecção de erros, as condições de suficiência também fornecem uma base para o desenvolvimento de métodos de geração de casos de teste.

As condições de suficiência apresentadas por Ural et al. (1997), que dependem da existência de uma seqüência de distinção da MEF, fundamentaram o desenvolvimento de métodos de geração de seqüências de verificação. Com base nas condições de suficiência propostas, os autores apresentaram um método que aborda o problema de encontrar uma

seqüência de verificação n-completa para uma MEF. As condições de suficiências propostas por Ural et al. (1997) também foram utilizadas por Chen et al. (2005), Hierons e Ural (2006) e Yalcin e Yenigun (2006) para o desenvolvimento de novos métodos de geração de seqüências de verificação.

Novas condições de suficiência para casos de teste n-completos também foram propostas por Dorofeeva et al. (2005b), que também desenvolveram o método H, considerado uma extensão do método HSI. Um algoritmo para a minimização de conjuntos de casos de teste baseados nas condições de suficiência propostas por Dorofeeva et al. (2005b) é apresentado em (Neto e Simão, 2007), demonstrando assim que novas condições de suficiência permitem a criação de novos métodos e, dessa maneira, reduzem o tamanho de conjuntos de casos de teste. Dado um conjunto de casos de teste T, um conjunto de seqüências de entradas está confirmado se e somente se possuir seqüências de transferência que levam a cada estado existente na MEF M e seqüências que convergem (isto é, seqüências que conduz para o mesmo estado a MEF M), também convergem em qualquer MEF que tem a mesma quantidade de estados de M e que produza a mesma saída de M para o conjunto de casos de teste T.

Simão e Petrenko (2010) apresentaram condições de suficiência para casos de teste n-completos. Essas condições são menos exigentes do que as existentes atualmente, como as condições propostas por Ural et al. (1997) e Dorofeeva et al. (2005b). As condições propostas, que consideram somente MEFs determinísticas, se baseiam no fato de que se existe um conjunto de seqüências confirmadas o qual inclui a seqüência vazia e percorre cada transição, então o caso de teste é n-completo. Em Definição 1 é apresentado o conceito de conjunto de seqüências confirmadas segundo Simão e Petrenko (2010).

Definição 1 *(Simão e Petrenko, 2010) Dado um caso de teste T de uma MEF $M = (S, s_0, X, Y, \delta, \lambda)$ e $K \in \Omega_M(s_0)$. O conjunto K é $\Im_t(M)$-confirmado (ou simplesmente confirmado) se $\delta(s_0, K) = S$ e, para cada $N = (Q, q_o, X, Y', \Delta, \Lambda) \in \Im_t(M)$, para todos $\alpha, \beta \in K, \Delta(q_0, \alpha) = \Delta(q_0, \beta)$ se e somente se $\delta(s_0, \alpha) = \delta(s_0, \beta)$. Caso uma seqüência de entrada esteja contida em um conjunto confirmado, essa seqüência é considerada uma seqüência confirmada.*

O teorema demonstrado por Simão e Petrenko (2010), apresentado abaixo, expressa que, para um caso de teste T ser n-completo, basta que exista um conjunto confirmado K, tal que K contenha uma seqüência vazia e percorra as transições de M.

Teorema 1 *Dado um conjunto de teste T e uma MEF M com n estados. T é n-completo para M se existe um conjunto K de seqüências confirmadas com as seguintes propriedades:*

1. $\epsilon \in K$.

2. *Para todo* $(s, x) \in D$*, existe* α *e* $\alpha x \in K$*, tal que* $\delta(s_o, \alpha) = s$*.*

A seguir são apresentados lemas que indicam maneiras de se construir um conjunto de seqüências confirmadas. O primeiro lema proposto apresenta condições para que um conjunto *state cover* seja um conjunto confirmado.

Lema 1 *Dado um caso de teste* T *de uma MEF* M *e um conjunto state cover* K*, se as seqüências de* K *são* T*-distinguíveis entre si, então* K *é um conjunto confirmado.*

O segundo lema proposto por Simão e Petrenko (2010) demonstra condições para adicionar uma seqüência a um conjunto preservando a propriedade de confirmação desse conjunto. Desse modo, conjuntos confirmados podem ser gerados de forma incremental.

Lema 2 *Dado um conjunto confirmado* K *e uma seqüência de transferência* α *para o estado* s*, se para cada* $s' \in S$*, tal que* $s \neq s'$*, existe uma seqüência* $\beta \in K$ *que é* T*-distinguível da seqüência* α *e que leva a MEF ao estado* s'*, então* $K \cup \{\alpha\}$ *também é um conjunto confirmado.*

Por fim, o terceiro lema de Simão e Petrenko (2010) apresenta uma outra maneira para confirmar seqüências, que deve ser aplicada após a confirmação das seqüências pelas outras condições de suficiência.

Lema 3 *Definindo um conjunto de seqüências confirmadas (*K*) e um prefixo conjunto de teste (*α*), se existir duas seqüências pertencentes ao conjunto* K *que levem ao mesmo estado a partir do estado inicial, ou seja,* β *e* $\chi \in K$*, tal que* $\delta(s_0, \beta) = \delta(s_0, \chi)$*, e uma seqüência* ω*, de modo que* $\beta\omega \in K$ *e* $\chi\omega = \alpha$*, então pode-se afirmar que o conjunto* $K \cup \{a\}$ *é um conjunto confirmado.*

Além de serem menos exigentes que as condições de suficiências propostas anteriormente e permitir a geração de casos de teste menores, outra vantagem das condições de suficiência apresentadas por Simão e Petrenko (2010) é que essas condições podem ser aplicadas tanto para seqüências com a operação *reset* ou para seqüências sem essa operação, o que as tornam mais abrangentes em relação à sua aplicação.

Por serem consideradas a base para o desenvolvimento de novos métodos de geração de casos de teste, o surgimento de novas condições de suficiência sugere a investigação de novos métodos de geração de casos de teste. Assim como ocorreu com as condições propostas por Ural et al. (1997), nas quais novos métodos de geração de seqüências de verificação foram desenvolvidos, tais como Chen et al. (2005), Hierons e Ural (2006) e Yalcin e Yenigun (2006), as condições de suficiência propostas por Simão e Petrenko (2010) podem ser exploradas com o objetivo de desenvolver métodos que geram seqüências de verificação menores que os métodos atuais.

2.5.2 Métodos de Geração de Seqüências de Verificação

Uma seqüência de verificação (*checking sequence*) da MEF M é uma seqüência de entrada que diferencia a classe de MEFs, equivalentes à M, de todas as outras MEFs com no máximo n estados, onde n é o número de estados de M. Ou seja, uma seqüência de verificação, gerada a partir de uma MEF, é uma seqüência de entrada que pode ser utilizada para verificar se uma implementação dessa MEF está correta (Simão e Petrenko, 2008). Entretanto, ao contrário dos conjuntos de seqüências de teste, uma seqüência de verificação não possui a entrada *reset*. A importância das seqüências de verificação se dá ao fato que existem máquinas que não possuem a funcionalidade *reset* e, portanto, seria inviável a utilização de um caso de teste com a entrada *reset* em máquinas dessa classe.

Métodos para a geração de seqüências de verificação foi o assunto de muitos trabalhos nas últimas décadas. Hennie (1964), o precursor dos estudos dessa área, desenvolveu um método para gerar seqüências de verificação a partir de uma seqüência de distinção. O método proposto por Hennie (1964) primeiramente aplica a seqüência de distinção em todos os estados, que devem estar ordenados. Seqüências de transferências são utilizadas para conduzir a MEF para o próximo estado. Posteriormente, cada transição da MEF é verificada e, para isso, a MEF é levada para um estado conhecido após a verificação de uma transição. Hennie não aborda em seu artigo sobre a inicialização da MEF, e, dessa forma, a MEF deve estar no seu início para a geração da seqüência de verificação. Esse requisito pode ser obtido por meio de uma seqüência de sincronização, que faz a especificação e a implementação voltarem para um estado conhecido.

Kohavi e Kohavi (1968) demonstraram que, no lugar da seqüência de distinção, como proposto por Hennie (1964), prefixos dessa seqüência podem ser utilizados para reduzir a seqüência de verificação. Boute (1974) demonstra que seqüências de verificação menores poderiam ser obtidas se, em vez de seqüências de distinção, um conjunto de distinção fosse utilizado, e se a sobreposição entre as seqüências de identificação fosse explorada. Um conjunto de distinção é um conjunto de seqüências de entradas, uma para cada estado da MEF, tal que para cada par de estados distintos, um prefixo comum das respectivas seqüências de entradas produzam saídas diferentes. Um conjunto de distinção pode ser obtido de uma seqüência de distinção, porém, existem MEFs que têm um conjunto de distinção, mas não possuem uma seqüência de distinção. O método proposto de Boute (1974), além de demonstrar como gerar uma seqüência de verificação para MEFs que não possuem seqüência de distinção, mas que possuem conjuntos de distinção, também determina quando as transições são "automaticamente" verificadas, ou seja, quando a verificação de uma transição é uma conseqüência da verificação de outras transições. Nas próximas seções são apresentados os principais métodos de geração de seqüências de verificação.

Dentre os métodos de geração de seqüências de verificação propostos na literatura, há alguns que utilizam grafos no seu processo para elaborar o caso de teste. Dentre esses métodos, pode-se destacar Gonenc (1970), Ural et al. (1997), Chen et al. (2005), Hierons e Ural (2006) e Yalcin e Yenigun (2006). Há também métodos baseados em busca local, como o método proposto por Simão e Petrenko (2008).

2.5.2.1 Método de Gonenc (1970)

O método desenvolvido por Gonenc (1970), denominado neste trabalho como método DS, tem como objetivo gerar seqüências de verificação para MEFs com seqüências de distinção, que deve ser a menor possível para gerar um conjunto menor de casos de teste. O método é composto por três partes: a seqüência inicial, que leva a máquina a ser testada para um estado específico, a seqüência α, que verifica todos os estados da máquina, e a seqüência β, que avalia todas as transições. As seqüências α e β são, então, concatenadas para formar uma seqüência de verificação.

A partir do momento em que a seqüência de verificação é criada para ser aplicada na máquina em um determinado estado (estado inicial), a máquina deve estar nesse estado especificado. A seqüência inicial, a primeira parte do método, tem como objetivo levar a máquina a ser testada para o estado inicial e, a partir disso, a seqüência de verificação pode ser executada.

Na segunda parte, uma seqüência α é gerada, semelhante àquela seqüência de distinção que garantia diferenciar os estados em uma implementação. Em uma seqüência α, a seqüência de distinção é aplicada para cada estado da MEF, usando seqüências de transferência, se necessário. Para gerar a seqüência α, é construído o grafo X_d, onde cada estado da MEF é representado por um nó. A seqüência X_d, nesse contexto, representa a seqüência de distinção escolhida. As ligações entre os nós do grafo representam a aplicação da seqüência X_d sobre um determinado nó da máquina. A seqüência α é gerada percorrendo o grafo sem repetir as arestas.

Na terceira parte, a seqüência β é gerada para que cada transição da MEF seja verificada. Uma seqüência β é a concatenação de transições, seguida pela seqüência de distinção, utilizando seqüências de transferências, se necessário. Dessa maneira, cada transição é verificada por meio da execução da entrada relacionada a ela e depois a seqüência X_d é aplicada para reconhecer o estado atingido, direcionado pela transição. A seqüência β é gerada de forma análoga à seqüência α. Dessa forma, um segundo grafo é construído, denominado grafo β. Porém, nesse grafo as arestas tem o objetivo de representar seqüência do tipo $x.X_d$, para todo $x \in X$, onde X é o conjunto de entradas da MEF. A geração da seqüência β se dá a partir da cobertura de todas as arestas do grafo.

Um exemplo do método DS é demonstrado a seguir a partir da MEF ilustrada na Figura 2.2, cuja a menor seqüência de distinção é $X_d = yyy$. Na primeira parte do método, o grafo-X_d é construído para gerar a seqüência-α. As ligações do grafo representam a aplicação da seqüência de distinção X_d em cada nó. A Figura 2.3 ilustra o grafo-X_d para a MEF utilizada no exemplo.

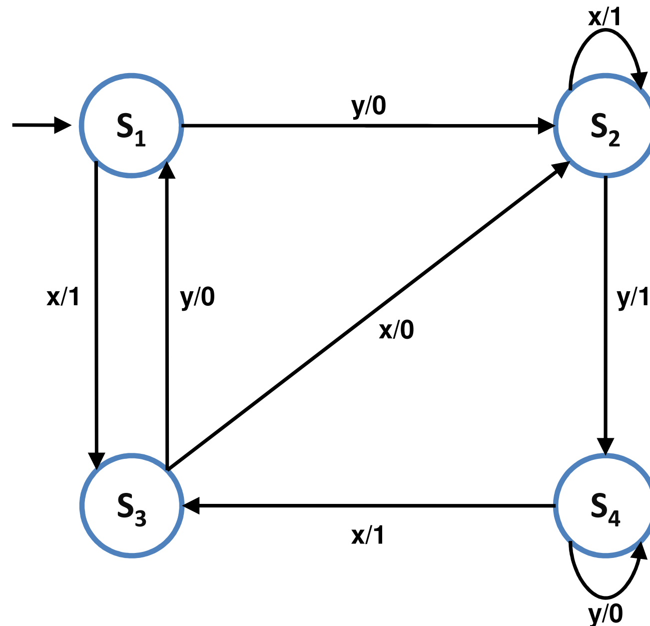


Figura 2.2: Exemplo de MEF extraído de Dorofeeva et al. (2005a).

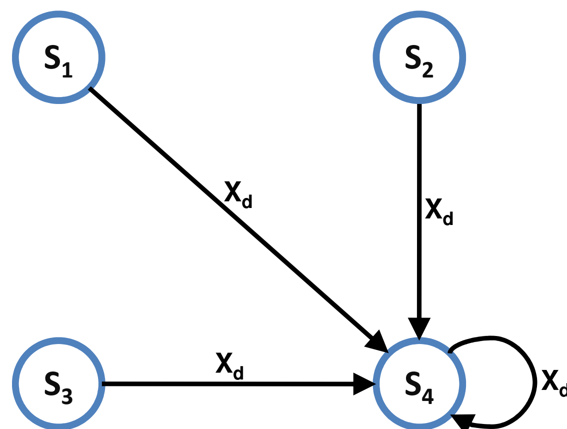


Figura 2.3: Grafo-X_d.

No início da geração da seqüência-α, um estado que não é destino de nenhuma aresta do grafo, intitulado como estado de origem, é escolhido de forma arbitrária. No grafo da Figura 2.3, o estado s_1 é selecionado e marcado como "reconhecido" e é aplicado a

seqüência de distinção X_d, que nesse exemplo leva a MEF ao estado s_4, que assim como o estado s_1, é marcado como reconhecido.

A seqüência X_d é aplicada novamente, o que faz a máquina permanecer no estado s_4, que já é reconhecido, porém, essa seqüência é aplicada novamente para garantir que a MEF irá permanecer no mesmo estado. Após essa confirmação, é necessário selecionar um novo estado de origem, aplicando uma seqüência de transferência, cujo objetivo é atingir um estado não reconhecido da MEF. Nesse exemplo é aplicada a seqüência x no estado s_4, o que leva a máquina a um novo estado de origem, o s_3. A seqüência X_d é aplicada nesse estado, atingindo o estado s_4. Após a confirmação da permanência da MEF no estado s_4, aplicando para isso X_d, é aplicada a seqüência de transferência xx para levar a MEF ao estado s_2, onde X_d é executado e assim o estado é marcado como reconhecido. Por último, X_d é executado novamente no estado s_4 para realizar a verificação citada anteriormente. Estando todos os estados reconhecidos, a seqüência-α gerada é $yyyyyyyyyxyyyyyyxxyyyyyy$.

O próximo passo do método é a geração da seqüência-β. Para isso é criado o grafo-β (Figura 2.4). Para garantir um conjunto de casos teste menor possível, pode ser realizado algumas reduções no grafo. A primeira redução está associada à última transição da seqüência X_d. Um exemplo é a execução dessa seqüência no estado s_1, que faz a MEF passar pelos estados s_2, s_4 e s_4, ocasionando a exclusão da transição y do estado s_4. Analogamente, a transição y do estado s_2 também pode ser excluída do grafo-β.

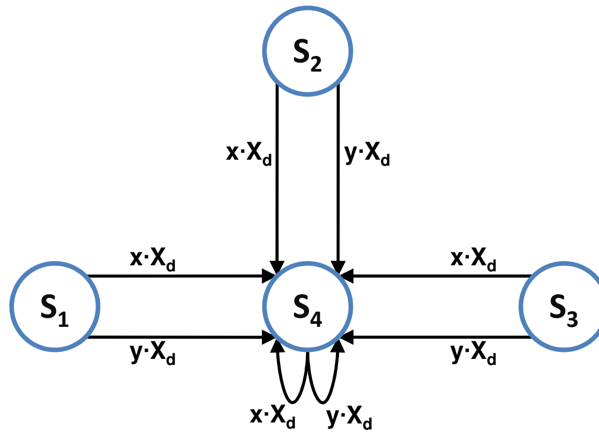


Figura 2.4: Grafo-β.

A outra redução que pode ser realizada está relacionada à seqüência de transferência aplicada na geração da seqüência-α para atingir um estado de origem da MEF. Como o estado inicial e final para toda seqüência de transferência são reconhecidos, as transições utilizadas por essa seqüência podem ser excluídos do grafo-β. Um exemplo é a seqüência x, aplicada no estado s_4 para acessar o estado s_3 com o objetivo de executar a seqüência

de distinção nesse último estado. Como esses dois estados são conhecidos, a transição x do estado s_4 pode ser retirada do grafo-β. De maneira análoga, a seqüência xx, que levou a MEF do estado s_4 ao estado s_2, passando pelo estado s_3, também pode ser excluída do grafo-β. Como todos esses estados são conhecidos, a transição x do estado s_3 também pode ser retirada do grafo. A Figura 2.5 ilustra o grafo-β do exemplo com todas as reduções possíveis. A seqüência-β é gerada percorrendo o grafo-β reduzido. Nesse exemplo, a seqüência-β é $xyyyxyyyyyxxxyyyxyyyy$.

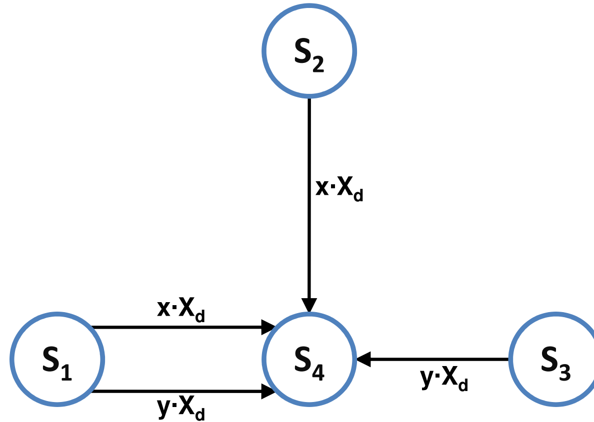


Figura 2.5: Grafo-β reduzido.

O método DS gera o caso de teste a partir da concatenação das seqüências α e β. Nesse caso, é obtida a seqüência de verificação SV_{Gonenc}, de tamanho 45.

$$SV_{Gonenc} = \{yyyyyyyyyxyyyyyyxxyyyyyyxyyyxyyyyyxxxyyyxyyyy\}$$

É importante citar que o número de transições que são verificadas pelo método é reduzido, utilizando o fato que a última transição da seqüência de distinção, quando aplicada no estado, será checada quando todas as outras transições da seqüência de distinção estiverem sido verificadas.

2.5.2.2 Método de Ural et al. (1997)

No trabalho de Ural et al. (1997), um método para a construção de seqüências de verificação é proposto e, da mesma forma que o método DS (Gonenc, 1970), é aplicável somente para MEFs que possuam uma seqüência de distinção. O método proposto objetiva minimizar o tamanho da seqüência de verificação gerada e, para isto, utiliza seqüência de distinção. Novas condições de suficiência para determinar que uma seqüência seja uma seqüência de verificação foram encontradas pelos pesquisadores e posteriormente utilizadas em muitos trabalhos, tais como (Hierons e Ural, 2002), (Chen et al., 2005), (Ural e Zhang, 2006), (Hierons e Ural, 2006) e (Yalcin e Yenigun, 2006).

As seqüências α e β do método de Gonenc (1970) são divididas em duas partes, que são combinadas com seqüências de transferência apropriadas para formar uma seqüência de verificação. O problema de encontrar uma seqüência de verificação de tamanho mínimo é equivalente ao *Rural Chinese Postman Problem* (RCPP). O RCPP é um problema NP-completo cujo objetivo é encontrar um caminho mínimo em um grafo, percorrendo algumas arestas requeridas. Ural et al. (1997) demonstraram como um grafo pode ser definido, tal que o caminho do RCPP satisfaça as condições de suficiência estabelecidas e, dessa maneira, torna-se uma seqüência de verificação. Para a execução do método, um conjunto de seqüências α e um conjunto de seqüências de transferência devem ser fornecidos como parâmetros de entrada. Aprimoramentos desse método é proposto por Hierons e Ural (2002, 2006).

A aplicação do método proposto por Ural et al. (1997) na MEF da Figura 2.6 é ilustrada a seguir. Considerando a seqüência de distinção $D = ab$, o conjunto $\alpha = \{\alpha_1, \alpha_2\}$, onde $\alpha_1 = abab$ e $\alpha_2 = ababab$, e o conjunto de seqüências de transferência $T = \{T_1^1, T_2^2, T_3^2\}$, onde $T_1^1 = ab$, $T_2^2 = ab$ e $T_3^2 = ab$, a seqüência de verificação gerada pelo método é SV_{Ural}, de tamanho 33.

$$SV_{Ural} = \{ababaabaabbabbabaaabababaababaaab\}$$

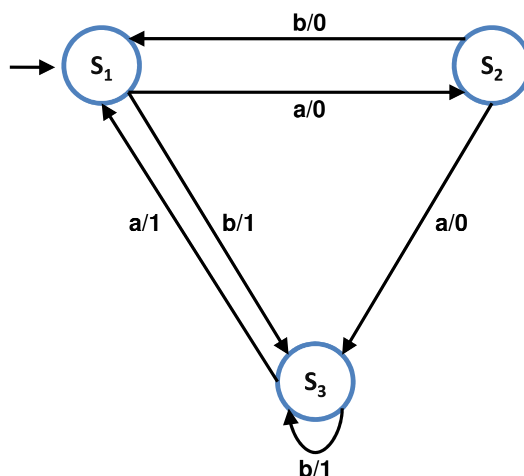


Figura 2.6: MEF extraída de (Ural et al., 1997).

2.5.2.3 Método de Chen et al. (2005)

Chen et al. (2005) apresentaram um método de geração de seqüência de verificação para MEFs que possuem seqüência de distinção. O trabalho demonstrou que o tamanho de uma seqüência de verificação pode ser ainda mais reduzido comparado com o trabalho

de Hierons e Ural (2002). Quando a MEF possui uma seqüência de distinção (seqüência α'), uma maneira eficiente de produzir uma seqüência de verificação é a partir de elementos do conjunto $E_{\alpha'}$ da seqüência α', que verificam subconjuntos de estados, e os elementos de um conjunto E_C, que são subseqüências que têm o objetivo de testar transições de forma individual.

A otimização que os pesquisadores propuseram refere-se à geração de uma seqüência de verificação pequena por meio da associação dos elementos de $E_{\alpha'}$ e E_C. Foi investigado o problema de eliminar subseqüências de E_C para a última transição executada pela seqüência de distinção aplicada para um estado particular.

Um exemplo do método de Chen et al. (2005) é demonstrado a seguir, baseado na MEF ilustrada na Figura 2.7. Uma seqüência de distinção para essa MEF é $D = aba$. Entretanto, prefixos de D são suficientes para distinguir cada estado. Desse modo, têm-se as seguintes seqüências: $D_1 = aba$, $D_2 = aba$, $D_3 = ab$, $D_4 = ab$ e $D_5 = ab$. Utilizando os conjuntos D_i, tem-se o conjunto $E_{\alpha'} = \{\alpha'_1\}$, onde $\alpha'_1 = D_3D_5D_1D_2D_4D_5$.

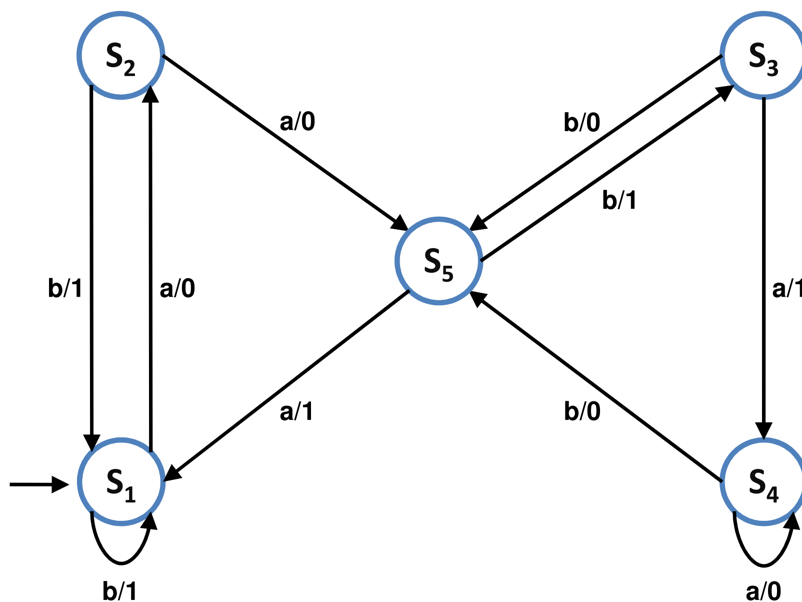


Figura 2.7: MEF extraída de (Chen et al., 2005).

A seqüência de verificação gerada pelo método é SV_{Chen}, de tamanho 44.

$$SV_{Chen} = \{abaabaaabbbabababaaabaababababaabaababaaaaba\}$$

2.5.2.4 Método de Hierons e Ural (2002) e Hierons e Ural (2006)

Recentemente, Hierons e Ural demonstraram que uma seqüência de verificação eficiente podia ser produzida combinando os elementos de um conjunto A, constituído a partir de

seqüências chamadas de seqüências α', e elementos do conjunto γ, que testam transições de maneira individual, usando o conjunto E_C de transições da MEF. No trabalho anterior dos pesquisadores, os conjuntos A e E_C eram predefinidos, ao contrário do conjunto γ, que era definido a partir do conjunto A. Entretanto, nesse trabalho não havia qualquer indicação para determinar os conjuntos A e E_C para gerar uma seqüência de verificação de tamanho reduzido.

Em (Hierons e Ural, 2006) é abordado o problema de geração dos conjuntos A e E_C com o objetivo de construir uma seqüência de verificação com o menor tamanho possível a partir do algoritmo proposto por Hierons e Ural (2002), disponibilizando uma maneira de gerar um conjunto A ótimo, minimizando o tamanho das subseqüências que são combinadas para gerar a seqüência de verificação, e uma otimização na geração do conjunto E_C a partir de um conjunto A. Dessa forma, a geração de seqüências de tamanho reduzido possui duas fases:

- Minimizar o tamanho total das subseqüências que serão combinadas e
- Combinar de maneira otimizada as subseqüências geradas.

Tanto no trabalho atual (Hierons e Ural, 2006) quanto no trabalho anterior (Hierons e Ural, 2002), os pesquisadores utilizaram seqüências α' como base para a geração da seqüência de verificação. Essas seqüências são definidas da seguinte maneira: o primeiro passo é escolher um $V_K \subseteq V(1 \leq k \leq q)$ cuja união é V, que por sua vez representa os nós existentes no grafo de uma determinada MEF, e ordenar os elementos existentes em V_K.

Para cada elemento de V_K, deve-se produzir uma seqüência de distinção. Após a execução dessa seqüência no elemento atual de V_K, aplica-se uma seqüência de transferência que irá levar a máquina ao estado que o próximo elemento de V_K representa. Dessa maneira, a união de todas essas seqüências, chamadas de seqüências α', forma o conjunto α'.

Para gerar o conjunto α', um grafo, chamado de grafo G_D, é criado. A Figura 2.8 representa o grafo G_D para a MEF ilustrada na Figura 2.7. As ligações do grafo representam a aplicação da seqüência de distinção D em cada nó da MEF. Por exemplo, a aplicação da seqüência de distinção no estado s_5 conduz a MEF até o estado s_2. O conjunto α' é constituído de seqüências α', que representam todos os caminhos mínimos do grafo G_D, sendo que no final de cada seqüência α' é aplicada a seqüência de distinção para reconhecer o estado final da seqüência α'.

Considerando a MEF da Figura 2.7 e a seqüência de distinção $D = aba$, tem-se o conjunto $\alpha' = \{\alpha'_1, \alpha'_2\}$, onde α'_1 corresponde pela execução de *abaabaabaabaaba* a partir do estado s_5 e α'_2 corresponde pela execução de *abaaba* a partir do estado s_3.

Figura 2.8: Grafo G_D para a MEF da Figura 2.7.

O método proposto por Hierons e Ural (2006) gera como resultado a seqüência $SV_{Hierons}$, de tamanho 64.

$SV_{Hierons} = \{abababaabaabaaabababaaabaaaabaababaabaabbabaabaabaabaabaaababababa\}$

2.5.2.5 Método de Yalcin e Yenigun (2006)

Muitos métodos de geração de seqüência de verificação existentes na literatura fazem uso de seqüências de distinção para identificar os estados da MEF durante a construção de seqüências de verificação, tais como os métodos propostos por Hennie (1964), Gonenc (1970) e Ural et al. (1997). Entretanto, existem MEFs que não possuem seqüências de distinção, mas que contém seqüências UIO, algo que motivou alguns pesquisadores a desenvolverem métodos de geração de seqüências de verificação que utilizam seqüências UIO ao invés de seqüências de distinção. Por exemplo, Sabnani e Dahbura (1988) e Aho et al. (1991) demonstraram como uma seqüência de verificação pode ser gerada a partir de seqüências UIO para os casos onde a MEF não possui uma seqüência de distinção. Porém, os métodos que utilizam seqüências de distinção geram seqüências de verificação menores comparadas aos métodos que utilizam somente seqüências UIO.

Yalcin e Yenigun (2006) demonstraram que mesmo que uma MEF possua uma seqüência de distinção, a seqüência UIO dessa máquina também pode ser utilizada junto com a seqüência de distinção para gerar uma seqüência de verificação menor. Quando uma entrada é executada na MEF e a saída é observada, deve-se reconhecer não somente o estado antes da aplicação dessa entrada, mas também identificar o estado que é atingido após a aplicação da entrada. A seqüência de distinção tem o objetivo de reconhecer o estado depois da execução de uma transição. Uma seqüência UIO também provê uma informação similar ao da seqüência de distinção. Dessa maneira, para reconhecer um estado após a aplicação de uma entrada, pode-se aplicar a seqüência UIO, observando a saída, que é única para cada estado. Como a seqüência UIO de um estado é menor ou igual à seqüência de distinção de uma MEF, o uso de seqüências UIO em vez da seqüência de distinção reduz o tamanho da seqüência de verificação.

Entretanto, não é sempre que uma seqüência UIO pode ser utilizada no lugar de uma seqüência de distinção. Segundo Yalcin e Yenigun (2006), deve-se usar a seqüência UIO somente quando é garantido que todas as transições dos estados estão contidos na seqüência UIO. Baseado nessa observação, foi proposta uma modificação no método proposto por Hierons e Ural (2002). No novo método, além de seqüências de distinção para reconhecer os estados, também são utilizadas seqüências UIO para a mesma finalidade. Essa modificação resulta na diminuição da seqüência de verificação gerada pelo método, tornando-o mais eficiente do que o método proposto por Hierons e Ural (2002).

Considerando a MEF da Figura 2.9, a seqüência de distinção $D = aa$ e a seqüência UIO $U_3 = b$ para o estado s_3, o método proposto gerou o caso de teste $SV_{Yalcin} = aaaaaaaabaaaaabaaabbaaaaaa$, de tamanho 26, menor que a seqüência gerada pelo método proposto por Hierons e Ural (2002), que gerou uma seqüência para a mesma MEF de tamanho 27.

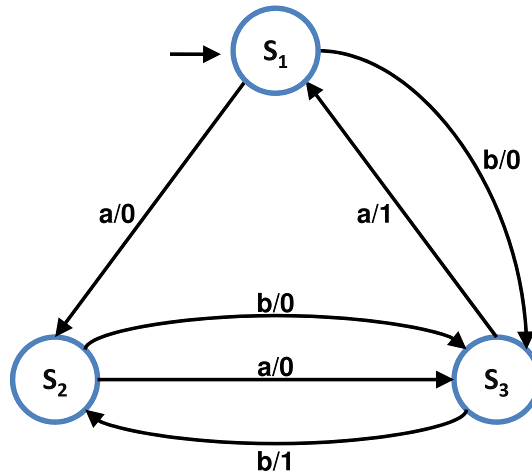


Figura 2.9: MEF extraída de (Yalcin e Yenigun, 2006).

2.5.2.6 Método de Simão e Petrenko (2008)

As limitações existentes nos métodos baseado em grafos, citados nas seções anteriores, como a necessidade da MEF ser completa e a não utilização da entrada *reset* para encurtar a seqüência de verificação, fazem com que esses métodos não sejam utilizados em determinadas situações. Nesse contexto, o método proposto por Simão e Petrenko (2008) não possui as limitações citadas, podendo assim ser utilizado para gerar seqüência de verificação para MEFs parciais com um conjunto de distinção e, possivelmente, com a função de *reset*. O método proposto faz a melhor escolha local em cada passo. Essa abordagem diverge dos métodos propostos em trabalhos recentes (Ural e Zhang, 2006; Hierons e Ural,

2002; Chen et al., 2005; Hierons e Ural, 2006), os quais utilizam modelagem teórica de grafos para minimizar o tamanho da seqüência de verificação.

Em cada passo do método, duas situações podem acontecer: (i) se o estado final da seqüência gerada até aquele ponto é conhecido, escolhe-se uma nova transição para ser verificada, tal que o estado de origem da transição seja alcançado com a menor seqüência de transferência possível; (ii) Se o estado final da seqüência não estiver verificado, então aplica-se a seqüência de identificação adequada. Esse processo é repetido até que a situação (i) aconteça.

Segundo os autores, a abordagem baseada na otimização local tem pelo menos duas vantagens sobre a abordagem teórica de grafos. Primeiramente, métodos teóricos de grafos tentam otimizar globalmente o tamanho da seqüência de verificação, mas somente depois de alguns parâmetros de entrada são configurados, por exemplo, o conjunto da seqüência α e da seqüência de transferência utilizados por Ural e Zhang (2006). Entretanto, o tamanho da seqüência de verificação é influenciado por esses parâmetros e, dessa forma, uma seqüência sub-otimizada pode ser gerada. Em vez de assumir que parâmetros apropriados são fornecidos, o novo método faz escolhas baseando-se na informação disponível até um certo ponto da execução. A segunda vantagem do método está em sua capacidade de extensão, visto que algumas heurísticas, como propostas por Chen et al. (2005) e Yalcin e Yenigun (2006), podem ser integradas ao método, aumentando a sua eficiência.

Com o objetivo de apresentar neste trabalho uma comparação entre o método proposto em (Simão e Petrenko, 2008) e os demais métodos de geração de seqüências de verificação, foram realizados alguns estudos de caso cujos resultados indicaram que o método proposto por Simão e Petrenko (2008) produz seqüências de verificação menores quando comparado aos outros métodos existentes. No primeiro estudo de caso foi realizada uma comparação entre o método proposto por Simão e Petrenko (2008) e o método apresentado em (Gonenc, 1970) com base na MEF ilustrada na Figura 2.2. Enquanto o método proposto por Gonenc (1970), apresentado na Seção 2.5.2.1, gerou a seqüência de verificação SV_{Gonenc}, de tamanho 45, o método proposto por Simão e Petrenko (2008) gerou a seqüência de verificação $SV_{Simao_1} = yyyyyyxyyyxxyxxxyxyxyyy$, de tamanho 23, ocasionando uma redução significativa, de aproximadamente 50%, na seqüência de verificação.

O segundo estudo de caso foi realizado com objetivo de comparar o método apresentado nesta seção com o método proposto em (Ural et al., 1997). Como foi descrito na Seção 2.5.2.2, o método de proposto por Ural et al. (1997) gerou a seqüência de verificação SV_{Ural}, de tamanho 33, para a MEF ilustrada na Figura 2.6. Porém, utilizando o método proposto por Simão e Petrenko (2008), foi gerada uma seqüência de verificação de tamanho 19 ($SV_{Simao_2} = aaaaababaabaababaaa$) para a MEF da Figura 2.6, proporcionando uma redução de aproximadamente 40% no tamanho da seqüência de verificação.

O objetivo do terceiro estudo de caso foi de realizar uma comparação entre o método baseado em busca local de Simão e Petrenko (2008) com os métodos propostos em (Chen et al., 2005) e (Hierons e Ural, 2006), que foram apresentados nas Seções 2.5.2.3 e 2.5.2.4, respectivamente. Utilizando a MEF ilustrada na Figura 2.7, o método proposto por Chen et al. (2005) gerou a seqüência de verificação SV_{Chen}, de tamanho 44, e o método de proposto por Hierons e Ural (2006) gerou a seqüência $SV_{Hierons}$, de tamanho 64. Entretanto, utilizando a mesma MEF, o método proposto por Simão e Petrenko (2008) gerou uma seqüência de tamanho 32 ($SV_{Simao_3} = abaaaabaabababaababbaabbbabaaaaa$), apresentando reduções de 28% e 50% à seqüência gerada pelos métodos de Chen et al. (2005) e Hierons e Ural (2006), respectivamente.

O quarto estudo de caso compartilha o mesmo objetivo dos demais, porém, nesse contexto foi utilizado o método proposto em (Yalcin e Yenigun, 2006). O exemplo do método de Yalcin e Yenigun (2006), apresentado na Seção 2.5.2.5, gerou a seqüência de verificação SV_{Yalcin}, de tamanho 26, para a MEF ilustrada na Figura 2.9. O método proposto por Simão e Petrenko (2008) gerou a seqüência de verificação $SV_{Simao_4} = aaaaabaabaaba$, de tamanho 13, reduzindo pela metade a seqüência de verificação.

2.5.3 Confirmação de Seqüências de Verificação Baseada em Condições de Suficiência

Como discutido anteriormente, as condições de suficiência têm o objetivo de garantir a completude de um caso de teste. Dessa forma, além de fornecer uma base para o aprimoramento de métodos de geração de casos de teste, as condições de suficiência oferecem uma maneira menos complexa de determinar se um conjunto de casos de teste é n-completo.

Para ilustrar as condições de suficiência propostas por Simão e Petrenko (2010), abordadas nas seções anteriores, é apresentado um exemplo que tem o objetivo de comprovar a completude de uma seqüência por meio dessas condições de suficiência. Considerando a MEF ilustrada na Figura 2.6, o método proposto por Ural et al. (1997) gerou a seqüência de verificação SV_{Ural}, de tamanho 33, descrita na Seção 2.5.2.2, e o método proposto por Simão e Petrenko (2008) gerou uma seqüência de verificação de tamanho 19 (SV_{Simao_2}), ilustrada na Seção 2.5.2.6. Entretanto, a seqüência $aaaaababaabaa$, de tamanho 13, obtida manualmente e denominada nesta seção como seqüência T, satisfaz as condições de suficiência propostas por Simão e Petrenko (2010), como será apresentado a seguir.

Para confirmar a completude da seqüência T por meio das condições de suficiência de Simão e Petrenko (2010), deve-se criar a árvore de teste. Nessa árvore, é representada a seqüência T, onde os nós representam prefixos de T e o estado atingido na MEF por meio da execução desse prefixo a partir do estado inicial. As transições da árvore indicam uma

entrada e a saída correspondente. Na Figura 2.10 é ilustrada a árvore de teste da MEF contida na Figura 2.6. Um exemplo de um nó existente na Figura 2.10 é o nó 11, que representa a seqüência *aaaaababaa*, prefixo da seqüência T, e está associado ao estado $S2$, pois a execução da seqüência *aaaaababaa* a partir do estado inicial leva a MEF ao estado $S2$.

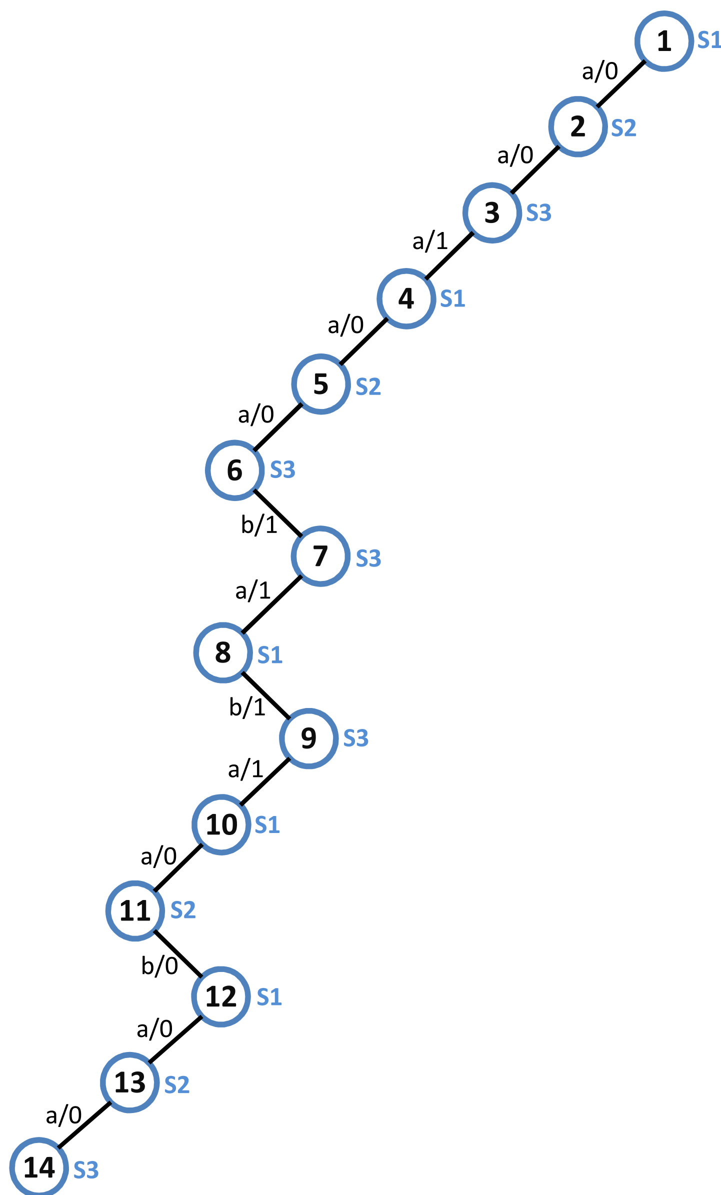


Figura 2.10: Árvore de teste.

O próximo passo após a construção da árvore é criar um grafo de distinção com a mesma quantidade de nós existentes na árvore de teste. A Figura 2.11 representa o grafo de distinção criado após a construção da árvore de teste do exemplo proposto. As arestas existentes no grafo são criadas a partir da distinção das saídas geradas pelos nós para

uma mesma entrada na árvore. Por exemplo, o nó 1 gera a saída 00 para a entrada *aa* e o nó 2 gera a saída 01 para essa mesma entrada. Dessa maneira, os nós 1 e 2 geram saídas diferentes para uma mesma entrada e, conseqüentemente, uma aresta é criada interligando esses dois nós. Como cada nó representa um estado, indicado na árvore de teste, pode-se afirmar que os nós que estão interligados no grafo por meio das arestas representam estados diferentes, visto que geram saídas diferentes para uma mesma entrada.

Após a construção do grafo de distinção, deve-se realizar a confirmação dos nós do grafo, processo que ocorre a partir da localização de um clique. Um *clique* no grafo é um conjunto de nós, onde cada nó desse conjunto está ligado aos demais nós do conjunto. Localizando um clique no grafo, é obtida uma seqüência confirmada, de acordo com o Lema 1, pois os estados representados pelos nós do clique são distintos par-a-par. No exemplo proposto, um clique no grafo da Figura 2.11 é o conjunto dos nós 1, 2 e 3, que possuem ligações entre si.

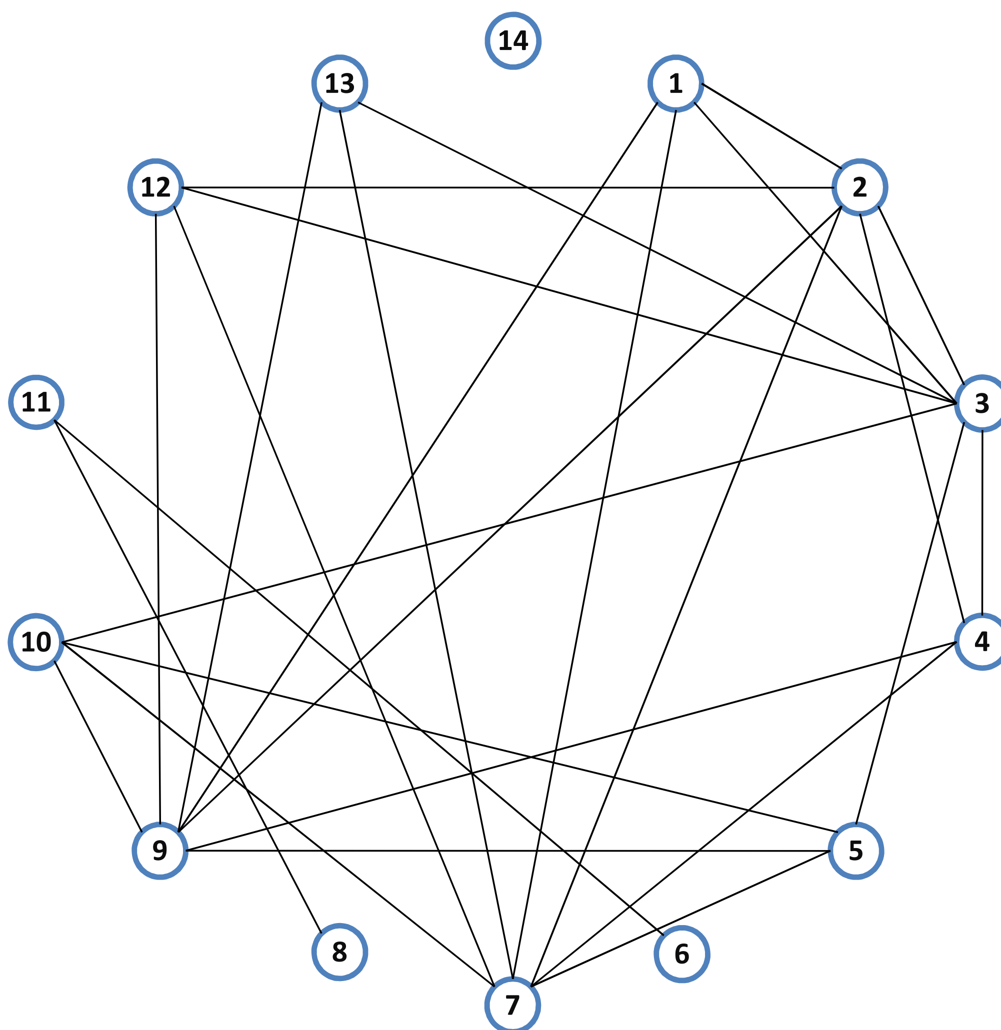


Figura 2.11: Grafo de distinção construído a partir da árvore de teste.

A partir da localização de um clique, os demais nós devem ser confirmados, conforme o Lema 2, baseado nos nós confirmados. Por exemplo, é possível confirmar o nó 7, uma vez que esse nó está ligado aos nós 1 e 2, que foram confirmados anteriormente. Os nós confirmados após a execução dessa fase foram: 1, 2, 3, 4, 7, 9 e 12. Assim, tem-se o conjunto $K = \{\epsilon, a, aa, aaa, aaaaab, aaaaabab, aaaaababaab\}$ de seqüências confirmadas.

Entretanto, ainda restam nós que não foram confirmados. Para a confirmação desses nós, é utilizado o Lema 3. A idéia contida nesse lema é a da existência de uma seqüência confirmada β que é prefixo de uma seqüência ainda não confirmada ω, porém, se a troca de β por outra seqüência confirmada α gerar uma seqüência confirmada, então a seqüência ω é considerada confirmada. Para confirmar os nós 5, 6, 8, 10, 11, 13 e 14, é apresentada a Tabela 2.2, onde β representa o prefixo da seqüência que se deseja confirmar e φ representa o sufixo dessa mesma seqüência.

Segundo o Teorema 1, a seqüência T é n-completo, visto que ϵ está no conjunto de seqüência confirmadas e existem seqüências α e $\alpha.x$ para todo o par (S, X) da MEF. Tal confirmação é representada na Tabela 2.3. Desse modo, comprova-se a completude do caso de teste $T = aaaaababaabaa$, de tamanho 13, significativamente menor que o caso de teste gerado pelo método de Ural et al. (1997), de tamanho 33, e menor que a seqüência gerada pelo método de Simão e Petrenko (2008) de tamanho 19.

Seqüências de verificação menores das demais MEFs apresentadas neste trabalho também foram obtidas manualmente para comprovar a sua completude com base nas condições de suficiência propostas por Simão e Petrenko (2010). Para a MEF ilustrada na Figura 2.2, foi obtida manualmente a seqüência $T = yyyyyyxyyyxyxxxyxyxyyy$, de tamanho 22. De maneira análoga ao exemplo apresentado nesta seção, a completude de tal seqüência foi confirmada pelas condições de suficiência propostas por Simão e Petrenko (2010), sendo que essa seqüência é menor que as geradas pelo método proposto por Gonenc (1970), que gerou a seqüência SV_{Gonenc}, de tamanho 45, e pelo método proposto por Simão e Petrenko (2008), que gerou a seqüência SV_{Simao_1}, de tamanho 23, ilustradas respectivamente nas Seções 2.5.2.1 e 2.5.2.6.

Outro estudo de caso realizado refere-se à seqüência $T = aabaaabbbababaababaaaa$, de tamanho 22, e a MEF ilustrada na Figura 2.7. Utilizando as condições de suficiência apresentadas por Simão e Petrenko (2010), confirmou-se que a seqüência é uma seqüência de verificação, sendo que essa seqüência é menor que as seqüências geradas pelos métodos propostos por Chen et al. (2005), que gerou a seqüência SV_{Chen}, de tamanho 44, Hierons e Ural (2006), cuja seqüência ($SV_{Hierons}$) gerada era de tamanho 64, e Simão e Petrenko (2008), que gerou a seqüência SV_{Simao_3}, de tamanho 32, ilustradas nas Seções 2.5.2.3, 2.5.2.4 e 2.5.2.6, respectivamente.

O último estudo de caso de confirmação de seqüências com base nas condições de suficiência de Simão e Petrenko (2010) está relacionado à MEF ilustrada na Figura 2.9 e à seqüência $T = aaaaabaabaaba$, de tamanho 13. As novas condições de suficiência confirmaram a completude da seqüência, que é menor que o caso de teste gerado pelo método proposto por Yalcin e Yenigun (2006), que gerou a seqüência SV_{Yalcin}, de tamanho 26, ilustrada na Seção 2.5.2.5.

2.6 Considerações Finais

Neste capítulo foram apresentados os principais conceitos relacionados ao teste de software e, devido ao escopo em que este trabalho está inserido, foram enfatizados os testes baseado em MEFs. Foram abordados também os principais aspectos relacionados à geração de seqüências de verificação, apresentando os conceitos mais importantes. Os métodos de geração de seqüências de verificação, tais como os métodos propostos em (Gonenc, 1970; Ural et al., 1997; Chen et al., 2005; Hierons e Ural, 2006; Yalcin e Yenigun, 2006), todos baseados na modelagem teórica de grafos, foram apresentados, exemplificados e, posteriormente, comparados ao método proposto em (Simão e Petrenko, 2008), que é baseado em busca local e que demonstrou ser mais eficiente do que os demais métodos.

Outro aspecto abordado refere-se ao uso de condições de suficiência para a confirmação de seqüências de verificação. Os estudos de caso apresentados utilizaram as condições de suficiência propostas por Simão e Petrenko (2010) para confirmar a completude de seqüências de entradas de MEFs ilustradas neste trabalho. As novas condições de suficiência forneceram um meio simples de confirmar que determinadas seqüências de entradas, que foram obtidas manualmente e que são menores que as seqüências geradas pelos métodos existentes, são seqüências de verificação. Dessa forma, torna-se necessária a investigação de estratégias para a utilização das novas condições de suficiências propostas para aprimorar a geração de seqüências de verificação.

Tabela 2.2: Confirmação de nós do grafo de distinção ilustrado na Figura 2.11.

Nó	ω	β e φ	α e φ	Conjunto K
5	aaaa	$\beta = aaa$ $\varphi = a$	$\alpha = \epsilon$ $\alpha \cdot \varphi = a$	$\{\epsilon, a, aa, aaa, aaaaab,$ $aaaaabab, aaaaababaab,$ $\mathbf{aaaa}\}$
6	aaaaa	$\beta = aaaa$ $\varphi = a$	$\alpha = a$ $\alpha \cdot \varphi = aa$	$\{\epsilon, a, aa, aaa, aaaaab,$ $aaaaabab, aaaaababaab,$ $aaaa, \mathbf{aaaaa}\}$
8	aaaaaba	$\beta = aaaaab$ $\varphi = a$	$\alpha = aa$ $\alpha \cdot \varphi = aaa$	$\{\epsilon, a, aa, aaa, aaaaab,$ $aaaaabab, aaaaababaab,$ $aaaa, aaaaa, \mathbf{aaaaaba}$
10	aaaaa baba	$\beta = aaaaabab$ $\varphi = a$	$\alpha = aa$ $\alpha \cdot \varphi = aaa$	$\{\epsilon, a, aa, aaa, aaaaab,$ $aaaaabab, aaaaababaab,$ $aaaa, aaaaa, aaaaaba,$ $\mathbf{aaaaababa}\}$
11	aaaaa babaa	$\beta = aaaaababa$ $\varphi = a$	$\alpha = \epsilon$ $\alpha \cdot \varphi = a$	$\{\epsilon, a, aa, aaa, aaaaab,$ $aaaaabab, aaaaababaab,$ $aaaa, aaaaa, aaaaaba,$ $aaaaababa, \mathbf{aaaaababaa}\}$
13	aaaaab abaaba	$\beta = aaaaababaab$ $\varphi = a$	$\alpha = \epsilon$ $\alpha \cdot \varphi = a$	$\{\epsilon, a, aa, aaa, aaaaab,$ $aaaaabab, aaaaababaab,$ $aaaa, aaaaa, aaaaaba,$ $aaaaababa, aaaaababaa,$ $\mathbf{aaaaababaaba}\}$
14	aaaaaba baabaa	$\beta = aaaaababaaba$ $\varphi = a$	$\alpha = a$ $\alpha \cdot \varphi = aa$	$\{\epsilon, a, aa, aaa, aaaaab,$ $aaaaabab, aaaaababaab,$ $aaaa, aaaaa, aaaaaba,$ $aaaaababa, aaaaababaa,$ $aaaaababaaba,$ $\mathbf{aaaaababaabaa}\}$

Tabela 2.3: Confirmação da cobertura das transições da MEF a partir das seqüências confirmadas.

Estado/entrada (S/x)	α	$\alpha \cdot x$
s_1 / a	ϵ	a
s_1 / b	aaaaaba	aaaaabab
s_2 / a	a	aa
s_2 / b	aaaaababaa	aaaaababaab
s_3 / a	aa	aaa
s_3 / b	aaaaa	aaaaab

Capítulo

3

Método de Geração de Seqüências de Verificação

3.1 Considerações Iniciais

Vários métodos de geração de seqüências de verificação são encontrados na literatura, porém é possível que sejam geradas seqüências menores que as seqüências obtidas por meio da utilização dos métodos existentes atualmente. Para explorar essa possibilidade, diferentes técnicas podem ser utilizadas com a finalidade gerar seqüências de verificação reduzidas.

Condições de suficiência podem ser úteis no desenvolvimento de métodos de geração de seqüências de verificação, pois a partir de tais condições é possível verificar e garantir a completude de seqüências de entradas. Desse modo, os métodos baseados em condições de suficiência utilizam-nas para verificar quais seqüências geradas são seqüências de verificação. Essa abordagem difere dos demais métodos, tais como (Hennie, 1964; Gonenc, 1970; Ural et al., 1997; Hierons e Ural, 2002; Chen et al., 2005; Ural e Zhang, 2006; Hierons e Ural, 2006), cujas seqüências são geradas de forma a garantir que serão sempre seqüências de verificação.

Neste capítulo é apresentado o algoritmo de geração de seqüências de verificação para MEFs proposto neste trabalho. O algoritmo é baseado fundamentalmente na técnica de algoritmos genéticos e nas condições de suficiência propostas por Simão e Petrenko

(2010). O algoritmo proposto consiste, basicamente, em criar novas seqüências a partir de mutações e utilizar as condições de suficiência para determinar quais seqüências são seqüências de verificação.

A divisão deste capítulo foi feita da seguinte forma. A Seção 3.2 apresenta o algoritmo de geração de seqüências de verificação proposto neste trabalho e na Seção 3.3 são descritos com mais detalhes os passos desse algoritmo. Na Seção 3.4 é ilustrado um exemplo de geração de uma seqüência de verificação a partir do algoritmo apresentado. Por fim, na Seção 3.5 são apresentados os aspectos de implementação relacionados aos aplicativos desenvolvidos neste trabalho de mestrado.

3.2 Algoritmo de Geração

O algoritmo apresentado neste trabalho foi desenvolvido baseado na técnica de algoritmos genéticos, nas condições de suficiência propostas por Simão e Petrenko (2010) e na abordagem proposta por Gonenc (1970), que testa cada transição da MEF para verificar a conformidade da especificação com a implementação, utilizando para isso seqüências de distinção.

O algoritmo, cujos parâmetros de entrada são uma MEF M fortemente conexa e um conjunto de distinção, é fundamentado em iterações que possuem o objetivo de gerar novas populações a partir da geração de cromossomos cada vez mais aptos. Os cromossomos são formados a partir de uma seqüência de genes, sendo que um gene, no contexto deste trabalho, é formado por uma seqüência de entradas da MEF M e o estado de origem que essa seqüência deve ser executada. Dessa maneira, um cromossomo cr representa uma seqüência de entradas válidas de M (seqüência χ), obtida a partir dos genes de cr.

Para cada iteração executada, o algoritmo classifica os cromossomos da população atual de acordo com a seqüência χ de cada cromossomo e gera novos cromossomos a partir dos cromossomos mais aptos (melhores classificados) da população atual. Esses novos cromossomos formam uma nova população e são classificados na iteração seguinte com a finalidade de gerar novos cromossomos, definindo assim o processo iterativo do algoritmo. Ao final de n_i iterações, a seqüência de verificação é obtida a partir do cromossomo melhor classificado da última população gerada, que representa a menor seqüência de verificação gerada pelo algoritmo. O algoritmo é dividido em seis passos, ilustrados na Figura 3.1 e descritos a seguir.

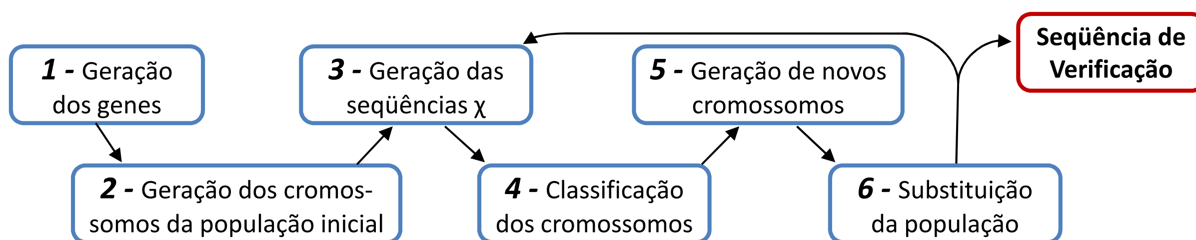


Figura 3.1: Passos do algoritmo genético.

3.3 Passos do Algoritmo de Geração

O Passo 1 do algoritmo consiste em gerar os genes do algoritmo genético, que compõem o conjunto GS (*Gene Set*). O conjunto GS é composto por dois tipos de genes, inspirados no método proposto por Gonenc (1970): os que representam as seqüências α (denominados de genes α) e os que representam as seqüências β (denominados de genes β).

As seqüências α têm como objetivo reconhecer os estados de M e são obtidas a partir de um conjunto de distinção, descritas anteriormente e que distinguem os estados de uma MEF. Assim, para cada estado s_i, tem-se o gene α_{s_i}, que é composto pela seqüência de distinção do estado s_i, contida no conjunto de distinção. As seqüências β são utilizadas para testar individualmente cada transição de M. As seqüências β são obtidas a partir da concatenação da entrada de uma transição com a seqüência α do estado de destino dessa transição. Dessa forma, para cada transição (s_i, x), tal que $s_j = \delta(s_i, x)$, tem-se o gene $\beta_{(s_i,x)} = (s_i, x\alpha_{s_j})$. No Passo 1, são gerados no máximo $|S|(|X| + 1)$ genes para o conjunto GS.

Com base na MEF da Figura 2.6, tem-se como exemplo a seqüência $\alpha = aa$ do estado s_1, gerada a partir do conjunto de distinção aa, aa, a, e a seqüência $\beta = baa$ da transição b do estado s_2, no qual a primeira entrada (b) representa a entrada da transição e as demais entradas (aa) representam a seqüência α do estado atingido pela transição, que nesse caso é o estado s_1.

No segundo passo do algoritmo são gerados N_P cromossomos, sendo N_P um valor estipulado previamente com a finalidade de compor a população inicial do algoritmo genético. Cada cromossomo de uma população é formado a partir de diferentes seqüências de genes de GS. Para gerar um cromossomo, genes do conjunto GS são selecionados aleatoriamente. Para um determinado cromossomo, essa seleção é feita até o momento em que o cromossomo possuir todos os genes β. Os genes α podem aparecer i-vezes em um cromossomo da população inicial, sendo $i \geq 0$, pois alguns genes α podem existir em genes β, como citado anteriormente e, desse modo, é possível gerar seqüências de verificação sem utilizar genes α. Os genes β aparecem ao menos uma vez em um cromossomo da

população inicial, visto que é necessário testar todas as transições de M. Essa estratégia é inspirada na abordagem adotada por Gonenc (1970).

No Passo 3, os cromossomos da população são convertidos em seqüências de entradas válidas da MEF M, denominadas como *seqüências* χ. Durante a conversão, três casos devem ser considerados, dependendo dos estados e entradas finais e iniciais de dois genes consecutivos. O primeiro caso é a sobreposição de entradas, caracterizada pela existência de uma mesma seqüência de entradas no início do gene g_{i+1} e no final do gene g_i, considerando para isso os estados de origem de cada entrada dessa seqüência. Essas entradas podem ser eliminadas (Ural e Zhang, 2006). Na Figura 3.2, esse caso é ilustrado entre o terceiro e quarto gene da seqüência, em que a entrada a do estado s_1 está presente no final do terceiro gene e no início do quarto gene e, dessa maneira, uma dessas entradas pode ser excluída da seqüência.

O segundo caso ocorre quando não há sobreposição e o estado final de um gene difere do estado inicial do gene seguinte: considerando-se os genes $g_1 = (s_i, \gamma)$ e $g_2 = (s_j, \varphi)$, com $\delta(s_i, \gamma) \neq s_j$, adiciona-se uma seqüência de transferência do estado $\delta(s_i, \gamma)$ ao estado s_j, entre os g_1 e g_2. Esse caso é ilustrado entre o primeiro e segundo gene da Figura 3.2, em que o estado final do primeiro gene é s_1 e o estado inicial do segundo gene é s_3 e, conseqüentemente, é adicionado à seqüência de transferência $TS = b$ que leva a MEF ilustrada pela Figura 2.6 do estado s_1 ao estado s_3.

Por fim, o terceiro caso ocorre quando o estado final de um gene coincide com o estado inicial do gene seguinte. Nesse caso, as seqüências dos genes são simplesmente concatenadas. Na Figura 3.2, o terceiro caso é ilustrado entre o segundo e terceiro gene e também entre o terceiro e quarto gene, uma vez que o estado inicial do quarto gene após a exclusão da entrada sobreposta é s_2.

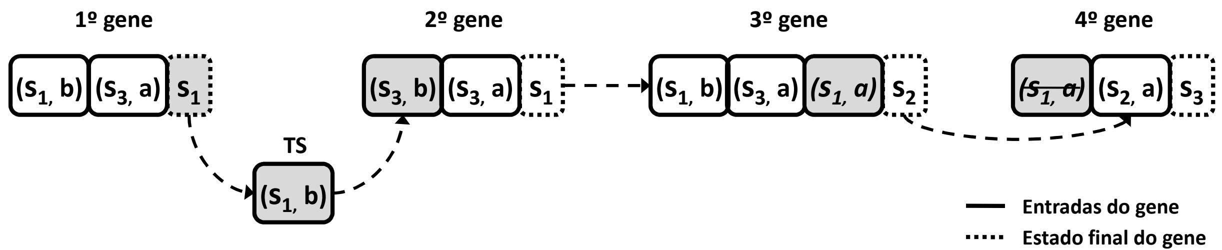


Figura 3.2: Exemplo de adição de seqüência de transferência e de remoção de entradas sobrepostas em uma *seqüência* χ.

No Passo 4, os cromossomos da população são classificados por meio de uma função *fitness*, definida com base na sua seqüência χ e nas condições de suficiência propostas por Simão e Petrenko (2010) , sendo que essa função é capaz de classificar distintamente cromossomos com seqüência χ n-completas de cromossomos com seqüência χ não n-completas. Assim, quanto menor o seu *fitness*, mais apto é um cromossomo. A popu-

lação é ordenada de modo crescente em função do valor do *fitness* de cada cromossomo e, por conseqüência, de modo decrescente de acordo com a aptidão de um cromossomo (do mais apto para o menos apto). O *fitness* de um cromossomo é definido da seguinte maneira: caso a seqüência χ de um cromossomo seja n-completa, o *fitness* desse cromossomo será o comprimento de χ, ou seja, a quantidade de entradas existentes na seqüência. Caso χ não seja n-completa, o *fitness* é definido como o comprimento da seqüência χ mais uma penalidade correspondente ao comprimento da maior seqüência χ existente da população. Dessa forma, garante-se que os cromossomos que correspondem a seqüências de verificação recebem um valor de *fitness* menor do que os cromossomos que não correspondem.

Após a classificação da população, no Passo 5 são gerados novos cromossomos a partir de mutações nos genes dos cromossomos da população atual. O objetivo dessa geração é obter seqüências χ n-completas menores que as seqüências χ n-completas existentes na população atual. Para gerar um novo cromossomo, é escolhido aleatoriamente um cromossomo da população atual e uma das mutações existentes, descritas a seguir. Entretanto, são selecionados apenas os N_M cromossomos mais aptos, ou seja, com o menor valor de *fitness*, sendo que N_M é definido como um parâmetro do algoritmo. Isso ocorre baseado no fato que esses cromossomos possuem as menores seqüências χ da população, podendo essas seqüências serem n-completas ou não, e conseqüentemente, tem maiores oportunidades de reproduzir indivíduos mais aptos.

Foram implementados seis operadores de mutação, descritos a seguir:

- **Remoção de um gene**: Um gene de uma posição aleatória é removido. Tanto o primeiro gene quanto o último, tem maior probabilidade de ser selecionado para remoção, pois identificou-se que a remoção desses genes acelera a obtenção de seqüências n-completa menores.

- **Alteração de posições de dois genes**: dois genes das posições aleatórias p_i e p_j (onde $p_i \neq p_j$) são alterados de posições no cromossomo.

- **Transferência dos k-primeiros genes para o final**: os k primeiros genes são removidos do início do cromossomo e transferidos para o final do cromossomo, sendo k um número obtido aleatoriamente.

- **Crossover entre duas seqüências**: os k últimos genes do cromossomo cr_i são removidos e os últimos genes do cromossomo cr_j, selecionado aleatoriamente, são inseridos em cr_i, sendo que os últimos genes de cr_j são obtidos a partir da remoção dos seus k primeiros genes e k é um número aleatório tal que $1 \leq k \leq |cr_i|$.

- **Inversão da ordem dos genes**: inverte a ordem dos genes do cromossomo e, conseqüentemente, a ordem das entradas da seqüência χ desse cromossomo. Esse

operador, apesar de não reduzir o tamanho da seqüência χ do cromossomo, pode permitir que uma seqüência χ não n-completa se torne uma seqüência n-completa.

- **Inserção de um gene** β: insere um gene β no cromossomo em uma posição definida aleatoriamente. Entretanto, o gene β, que também é selecionado de maneira aleatória, não deve estar presente no cromossomo. O objetivo desse operador é permitir que uma seqüência χ não n-completa, devido ao fato de não testar uma transição da MEF por conseqüência da inexistência de um gene β, se torne uma seqüência n-completa.

No Passo 6, uma nova população é formada a partir da composição dos N_M cromossomos mais aptos da população anterior e dos N_N novos cromossomos gerados no passo anterior, sendo $N_N = N_P - N_M$. Os cromossomos menos aptos da população anterior são descartados pelo algoritmo e substituídos pelos novos cromossomos gerados pelo Passo 5, que tendem a ser mais aptos que os cromossomos descartados e, dessa maneira, ocorre a evolução natural dos indivíduos e, conseqüentemente, da população do algoritmo. Essa iteração, de geração e classificação de novos cromossomos, é realizada enquanto houver evolução dos cromossomos mais aptos das populações formadas, sendo que essa evolução é determinada com base no *fitness* desses cromossomos. Após n_{se} iterações sem evolução dos cromossomos, o algoritmo é interrompido.

3.4 Exemplo

Para exemplificar o processo de utilização do algoritmo, considere a MEF da Figura 2.6 e o conjunto de distinção aa, aa, a para essa mesma MEF. A Tabela 3.1 contém os genes gerados pelo primeiro passo do algoritmo. Para facilitar a compreensão dos próximos passos, os genes possuem representações numéricas, sendo que para esse exemplo, os genes são representados do número 1 ao número 9.

A partir dos genes obtidos no primeiro passo e considerando $N_P = 10$, o Passo 2 irá gerar aleatoriamente os cromossomos da população inicial, como os descritos na Tabela 3.2, sendo que os genes dos cromossomos correspondem à representação numérica descrita na Tabela 3.1.

O terceiro passo é exemplificado pela Tabela 3.3, na qual são ilustrados os cromossomos da população atual do algoritmo com suas respectivas seqüências χ.

O Passo 4 do algoritmo é exemplificado pela Tabela 3.4, que exibe os cromossomos da população atual ordenados crescentemente a partir dos seus respectivos $fitness$, calculados nesse passo. Como a maior seqüência χ da população atual tem tamanho 40, tem-se *penalidade* $= 40$. Assim, os cromossomos da população atual que possuem seqüências

Tabela 3.1: Passo 1 - Genes gerados.

Gene	Representação numérica
$\alpha_{s_1} = (s_1, aa)$	1
$\alpha_{s_2} = (s_2, aa)$	2
$\alpha_{s_3} = (s_3, a)$	3
$\beta_{(s_1,a)} = (s_1, aaa)$	4
$\beta_{(s_1,b)} = (s_1, ba)$	5
$\beta_{(s_2,a)} = (s_2, aa)$	6
$\beta_{(s_2,b)} = (s_2, baa)$	7
$\beta_{(s_3,a)} = (s_3, aaa)$	8
$\beta_{(s_3,b)} = (s_3, ba)$	9

Tabela 3.2: Passo 2 - Cromossomos da população inicial.

Cromossomos
$cr_1 = 1, 7, 2, 2, 8, 9, 2, 5, 6, 3, 4$
$cr_2 = 7, 5, 8, 3, 2, 4, 2, 3, 1, 3, 3, 6, 9$
$cr_3 = 9, 5, 1, 7, 4, 1, 2, 8, 1, 2, 3, 6$
$cr_4 = 6, 1, 3, 7, 5, 9, 1, 8, 4, 1, 1, 1, 1, 3, 2$
$cr_5 = 5, 6, 2, 9, 2, 4, 1, 2, 8, 7, 3$
$cr_6 = 2, 8, 2, 6, 7, 9, 2, 5, 3, 1, 4$
$cr_7 = 6, 8, 2, 2, 2, 4, 7, 3, 1, 9, 2, 1, 3, 5$
$cr_8 = 5, 2, 9, 1, 4, 8, 1, 3, 1, 2, 3, 6, 7$
$cr_9 = 1, 3, 6, 5, 2, 2, 7, 4, 1, 9, 8$
$cr_{10} = 1, 7, 6, 1, 3, 4, 2, 5, 1, 9, 3, 8$

χ n-completas (verificadas por meio das condições de suficiência proposta por Simão e Petrenko (2010)) terão o valor de $fitness$ igual ao tamanho da sua seqüência χ. Para os cromossomos cuja seqüência χ não são n-completas, o seu $fitness$ será o tamanho da sua seqüência χ mais o valor da variável *penalidade*, sendo que o valor dessa variável é redefinido a cada iteração do algoritmo, ou seja, para toda vez que uma nova população é formada. Considerando a população atual, os cromossomos que possuem seqüência χ n-completa são: cr_1, cr_6, cr_9, cr_10, cr_7 e cr_4. Os demais cromossomos, cr_5, cr_3, cr_2 e cr_8, possuem seqüência χ não n-completa.

Para o exemplo apresentado, considerando $N_P = 10$ e $N_M = 3$, no quinto passo são gerados sete novos cromossomos a partir de mutações dos três melhores cromossomos da população atual obtida no passo anterior. A Tabela 3.5 apresenta os sete novos cromossomos gerados para o exemplo, no qual cada linha da tabela indica o cromossomo escolhido aleatoriamente a partir da população atual (que, no exemplo atual, deve estar entre os

Tabela 3.3: Passo 3 - Cromossomos da população atual com suas respectivas seqüências χ.

Cromossomo	Seqüência χ	$\lvert\chi\rvert$
cr_1 = 1, 7, 2, 2, 8, 9, 2, 5, 6, 3, 4	$aaaabaaaaaaaabaaaabaaaabaaaa$	28
cr_2 = 7, 5, 8, 3, 2, 4, 2, 3, 1, 3, 3, 6, 9	$abaaabaaaaaaaaaaaaabaaaabaaaabba$	32
cr_3 = 9, 5, 1, 7, 4, 1, 2, 8, 1, 2, 3, 6	$bbabaaaabaaaaaaaaaaaaaaabaaaa$	30
cr_4 = 6, 1, 3, 7, 5, 9, 1, 8, 4, 1, 1, 1, 1, 3, 2	$aaaaaaabaaababbaaaaaaaaaaaaaaaaaaaaaaaaa$	40
cr_5 = 5, 6, 2, 9, 2, 4, 1, 2, 8, 7, 3	$baaaaaaabbaaaaaaaaaaaaaabaaa$	28
cr_6 = 2, 8, 2, 6, 7, 9, 2, 5, 3, 1, 4	$aaaaaaaaaabaabaaaababaaaaaaa$	28
cr_7 = 6, 8, 2, 2, 2, 4, 7, 3, 1, 9, 2, 1, 3, 5	$aaaaaaaaaaaaaaaabaaaaabaaaaaaaaba$	32
cr_8 = 5, 2, 9, 1, 4, 8, 1, 3, 1, 2, 3, 6, 7	$baaaabbaaaaaaaaaaaaaaaabaaaaabaa$	32
cr_9 = 1, 3, 6, 5, 2, 2, 7, 4, 1, 9, 8	$aaaaaabaaaaaaaabaaaaaaaabaaa$	28
cr_{10} = 1, 7, 6, 1, 3, 4, 2, 5, 1, 9, 3, 8	$aaaabaaaaaaaaaaaaaaaabaaababaaa$	30

três melhores cromossomos da população), o operador de mutação também selecionado de forma aleatória e o cromossomo resultante da mutação.

Após a geração dos novos cromossomos no quinto passo, uma nova população é formada no Passo 6, excluindo da população os três indivíduos menos aptos e inserindo os três novos gerados no passo anterior. Nesse momento, não é possível afirmar se os novos cromossomos gerados (cr_{11} ao cr_{17}) são mais ou menos aptos que os cromossomos excluídos da população, algo que só é possível afirmar após gerar as seqüências χ dos novos cromossomos e classificá-los de acordo no contexto da nova população formada. Porém, mesmo que todos os novos cromossomos gerados sejam menos aptos, comparados àqueles que foram excluídos, esses indivíduos serão excluídos na iteração seguinte, sendo substituídos pelos novos cromossomos gerados (desde que estes sejam mais aptos). Desse modo, não há o risco de eliminar da população um elemento apto, visto que os indivíduos mais aptos da população atual são mantidos. A Tabela 3.6 ilustra a nova população, formada pelos três cromossomos mais aptos da população anterior (cr_1, cr_6 e cr_9) e pelos sete novos cromossomos criados no passo anterior (cr_{11} ao cr_{17}).

Tabela 3.4: Passo 4 - Cromossomos da população atual ordenados crescentemente a partir dos seus respectivos $fitness$.

Cromossomo	$\|\chi\|$	$fitness$	Classificação
$cr_1 = 1, 7, 2, 2, 8, 9, 2, 5, 6, 3, 4$	28	28	1
$cr_6 = 2, 8, 2, 6, 7, 9, 2, 5, 3, 1, 4$	28	28	2
$cr_9 = 1, 3, 6, 5, 2, 2, 7, 4, 1, 9, 8$	28	28	3
$cr_{10} = 6, 81, 7, 6, 1, 3, 4, 2, 5, 1, 9, 3, 8$	30	30	4
$cr_7 = 6, 8, 2, 2, 2, 4, 7, 3, 1, 9, 2, 1, 3, 5$	32	32	5
$cr_4 = 6, 1, 3, 7, 5, 9, 1, 8, 4, 1, 1, 1, 1, 3, 2$	40	40	6
$cr_5 = 5, 6, 2, 9, 2, 4, 1, 2, 8, 7, 3$	28	68	7
$cr_3 = 99, 5, 1, 7, 4, 1, 2, 8, 1, 2, 3, 6$	30	70	8
$cr_2 = 7, 5, 8, 3, 2, 4, 2, 3, 1, 3, 3, 6, 9$	32	72	9
$cr_8 = 5, 2, 9, 1, 4, 8, 1, 3, 1, 2, 3, 6, 7$	32	72	10

Após a formação da nova população, é necessário classificar os novos cromossomos. Para isso, o algoritmo retorna ao Passo 3, que tem como objetivo classificar os cromossomos da população, definindo o $fitness$ dos indivíduos. Entretanto, só é necessário classificar os novos cromossomos, uma vez que os cromossomos mais aptos da população anterior, e que foram mantidos na nova população, tiveram o seu $fitness$ definidos anteriormente. Ao voltar ao terceiro passo, o algoritmo irá classificar os novos cromossomos, definindo um novo valor para a variável *penalidade* e reordenando a nova população. Após isso, o algoritmo executa os passos seguintes até atingir novamente o sexto passo, onde uma nova população será formada e classificada ao retornar, novamente, ao terceiro passo. Dessa maneira, o algoritmo realiza n_i iterações, que se resumem na execução de n_i vezes dos Passos 3, 4, 5 e 6.

A quantidade de iterações é definida com base na evolução dos cromossomos mais aptos das populações formadas. Após n_{se} iterações do algoritmo sem que os N_M cromossomos evoluam, ou seja, possuam valores de *fitness* menores comparados aos *fitness* dos N_M cromossomos da população anterior, o algoritmo é interrompido. Para o exemplo atual, no qual $N_M = 3$, foi considerado $n_{se} = 3$. A Tabela 3.7 ilustra os *fitness* dos três cromossomos mais aptos em cada iteração do exemplo atual. A partir dessa tabela, é possível verificar que não houve evolução nos cromossomos mais aptos nas três últimas iterações e, assim, o algoritmo é interrompido.

Após a execução das n_i iterações, é possível obter a menor seqüência de verificação gerada a partir da seqüência χ do cromossomo mais apto da última população, pois esse indivíduo foi o mais apto de todas as populações criadas pelo algoritmo. Para o exemplo apresentado, após 13 iterações, obteve-se a seqüência de verificação *aaabaabaaba*, de tamanho 11, enquanto que o método proposto por Simão e Petrenko (2008) gerou

Tabela 3.5: Passo 5 - Novos cromossomos gerados.

Cromossomo Selecionado	Mutação	Novo Cromossomo
cr_1	Crossover entre cr_1 e cr_9	cr_{11} = 1, 7, 2, 2, 8, 9, 7, 4, 1, 9, 8
cr_1	Remoção dos genes da posição 2 e 6	cr_{12} = 1, 2, 2, 8, 2, 5, 6, 3, 4
cr_9	Remoção do primeiro gene	cr_{13} = 3, 6, 5, 2, 2, 7, 4, 1, 9, 8
cr_1	Troca dos genes da posição 7 e 10	cr_{14} = 1, 7, 2, 2, 8, 9, 3, 5, 6, 2, 4
cr_9	Transferência dos 9 primeiros genes para o final	cr_{15} = 9, 8, 1, 3, 6, 5, 2, 2, 7, 4, 1
cr_9	Inversão da ordem dos genes	cr_{16} = 8, 9, 1, 4, 7, 2, 2, 5, 6, 3, 1
cr_6	Remoção do terceiro gene	cr_{17} = 2, 8, 6, 7, 9, 2, 5, 3, 1, 4

Tabela 3.6: Passo 6 - Nova População.

Cromossomo
cr_1 = 1, 7, 2, 2, 8, 9, 2, 5, 6, 3, 4
cr_6 = 2, 8, 2, 6, 7, 9, 2, 5, 3, 1, 4
cr_9 = 1, 3, 6, 5, 2, 2, 7, 4, 1, 9, 8
cr_{11} = 1, 7, 2, 2, 8, 9, 7, 4, 1, 9, 8
cr_{12} = 1, 2, 2, 8, 2, 5, 6, 3, 4
cr_{13} = 3, 6, 5, 2, 2, 7, 4, 1, 9, 8
cr_{14} = 1, 7, 2, 2, 8, 9, 3, 5, 6, 2, 4
cr_{15} = 9, 8, 1, 3, 6, 5, 2, 2, 7, 4, 1
cr_{16} = 8, 9, 1, 4, 7, 2, 2, 5, 6, 3, 1
cr_{17} = 2, 8, 6, 7, 9, 2, 5, 3, 1, 4

a seqüência de verificação *aaaaababaabaa*, de tamanho 13, para o mesmo conjunto de distinção utilizado neste exemplo.

3.5 Aspectos da Implementação

Um dos resultados gerados por este trabalho de mestrado refere-se às implementações realizadas. No total, foram três aplicativos implementados, descritos resumidamente a seguir:

Tabela 3.7: Cromossomos mais aptos em cada iteração.

Iteração	*Fitness* do cromossomo 1	*Fitness* do cromossomo 2	*Fitness* do cromossomo 3
1	28	28	28
2	28	28	28
3	28	28	28
4	22	25	28
5	16	16	19
6	13	14	16
7	13	14	16
8	13	13	13
9	11	11	13
10	11	11	13
11	11	11	11
12	11	11	11
13	11	11	11

1. Gerador de Conjuntos de Distinção: aplicativo desenvolvido para gerar conjuntos de distinção para uma determinada MEF com base no método de geração de seqüências de distinção adaptativas proposto por Lee e Yannakakis (1994).

2. Verificador das Condições de Suficiências: aplicativo com o objetivo de verificar se um conjunto de casos de teste satisfaz as condições de suficiência propostas por Simão e Petrenko (2010).

3. Gerador de Seqüências de Verificação: implementação do algoritmo genético para gerar seqüências de verificação descrito na Seção 3.2. Há duas versões desse aplicativo: a primeira versão é executada utilizando apenas um processo, enquanto que a segunda faz uso de vários processos simultâneos, podendo ser executados em um *cluster* ou em um computador com mais de uma unidade de processamento, beneficiando assim o desempenho do aplicativo em relação ao tempo de execução.

Todos aplicativos foram implementados utilizando a linguagem de programação Java, o que permite a execução desses aplicativos em qualquer sistema operacional sem a necessidade de compilar novamente tais aplicativos. A seguir, são apresentados com mais detalhes os aplicativos citados.

3.5.1 Gerador de Conjuntos de Distinção

Esse aplicativo, denominado **GDS** (*Generator of Distinguishing Set*), foi desenvolvido com o propósito de gerar o conjunto de distinção, que é um conjunto com seqüências utilizadas para distinguir os estados de uma MEF.

O aplicativo implementa os dois algoritmos apresentados em (Lee e Yannakakis, 1994). Enquanto o primeiro algoritmo, que é polinomial, tem o objetivo de definir se uma determinada MEF possui ou não um conjunto de distinção, o segundo algoritmo, também polinomial, é utilizado para gerar o próprio conjunto de distinção.

O único parâmetro de entrada do aplicativo é uma MEF M, que deve ser definida de maneira textual em um arquivo obedecendo ao seguinte formato: `estadoOrigem -- entrada / saída -> estadoDestino`, conforme descrito na Tabela 3.8. Caso M tenha um conjunto de distinção, o aplicativo retorna como saída o valor 0 e um arquivo contendo o conjunto de distinção de M. Caso contrário, o aplicativo retorna o valor 255. O conjunto de distinção retornado pelo aplicativo é no formato: `estado entrada1 entrada2 entrada3`, também descrito na Tabela 3.8.

Tabela 3.8: Formato de entrada (MEF) e saída (Conjunto de Distinção) do Gerador de Conjuntos de Distinção.

MEF	Conjunto de Distinção
s1 -- a / 0 -> s2	s1 a a
s1 -- b / 1 -> s3	s2 a a
s2 -- a / 0 -> s3	s3 a
s2 -- b / 0 -> s1	
s3 -- a / 1 -> s1	
s3 -- b / 1 -> s3	

Apesar de seu desenvolvimento ser focado no Gerador de Seqüências de Verificação, o Gerador de Conjuntos de Distinção pode ser utilizado de maneira isolada, visto que para a sua execução, é necessário apenas que seja passada como parâmetro uma MEF no padrão descrito.

3.5.2 Verificador das Condições de Suficiência

Esse aplicativo, denominado como **CheSCon** (*Check Suficient Conditions*), desenvolvido em conjunto com outro aluno do grupo de pesquisa do ICMC/USP, tem o objetivo de verificar se um determinado conjunto de casos de teste de uma MEF satisfaz as condições de suficiência propostas em (Simão e Petrenko, 2010). Para realizar a sua execução, o soft-

ware deve receber como parâmetro de entrada uma MEF M e um conjunto de seqüências de teste, sendo que esse conjunto pode ser formado por apenas uma seqüência.

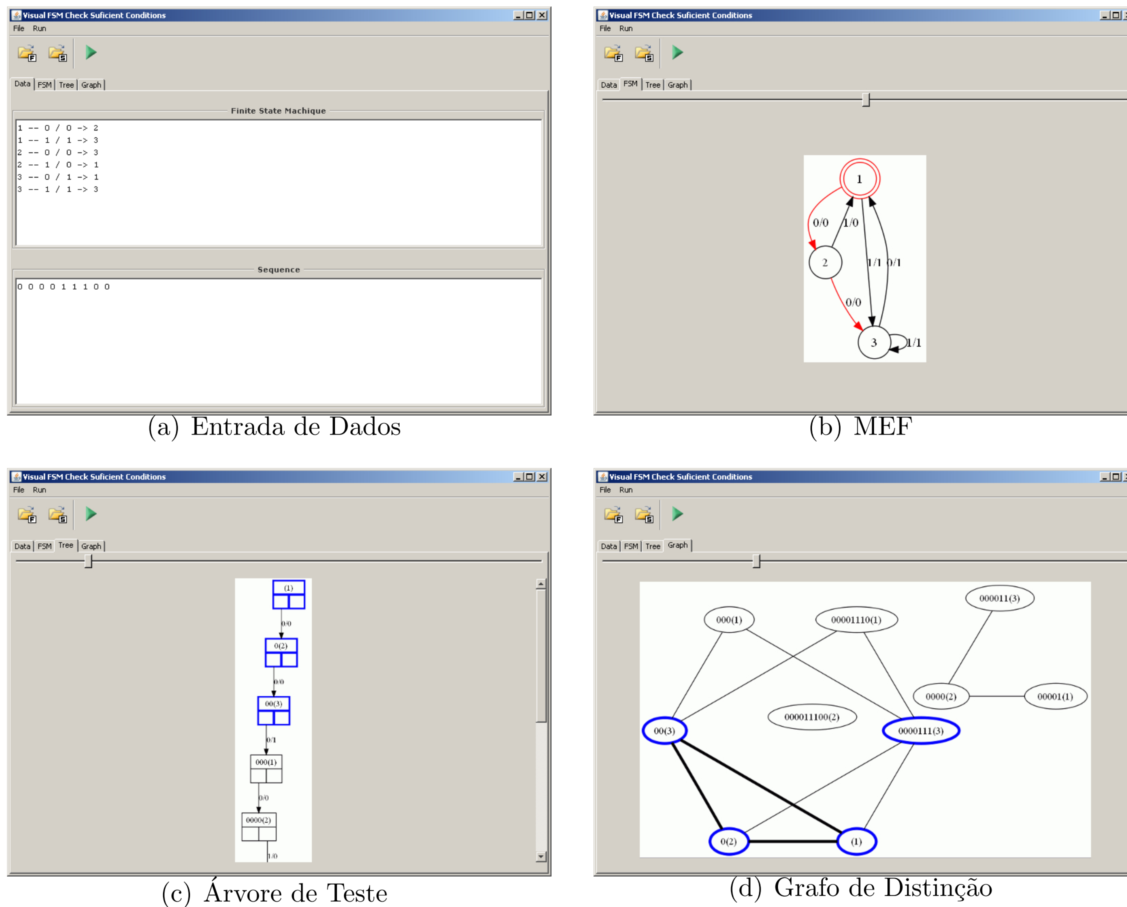
Assim como ocorre com o aplicativo **GDS**, os parâmetros são passados de maneira textual obedecendo a um determinado formato estabelecido pelo aplicativo. O primeiro parâmetro é uma MEF no mesmo formato do aplicativo **GDS**, igual descrito na Tabela 3.8. O segundo parâmetro é o conjunto de seqüências de teste a serem verificadas, que deve obedecer ao seguinte formato: `entrada1 entrada2 entrada3 ...`, nas quais as entradas são separadas por um espaço em branco e cada linha corresponde a uma seqüência do conjunto, sendo que cada linha é formada por uma ou mais entradas válidas da MEF. A Tabela 3.9 ilustra o formato do segundo parâmetro do aplicativo, que nesse exemplo é um conjunto de casos de teste formado por quatro seqüências. O formato do primeiro parâmetro está descrito na Tabela 3.8.

Tabela 3.9: Formato conjunto de casos de teste do Verificador das Condições de Suficiência.

Conjunto de casos de teste
a a a
b a
a b a
b b a a

Ao executar o aplicativo, devem ser passados como parâmetros dois arquivos distintos: o primeiro contendo a MEF e o segundo contendo o conjunto de seqüências de entradas a serem verificadas, sendo que o conteúdo dos dois arquivos devem obedecer o formato descrito anteriormente. Se a saída retornada pelo software for 0, o conjunto de seqüências satisfazem as condições de suficiência. Caso a saída seja 255, as condições de suficiência não foram satisfeitas.

Posteriormente, foi desenvolvida uma interface gráfica ao **CheSCon**, denominada **CheSCon Viz**. A implementação dessa versão do aplicativo com interface gráfica teve como objetivo o auxílio às pesquisas deste trabalho, identificando as características de cada abordagem levantada durante a implementação do método de geração de seqüências de verificação. Utilizando a interface gráfica desenvolvida, é possível visualizar, de maneira simples, o comportamento das estruturas utilizadas pelo algoritmo do **CheSCon** e, no caso do conjunto de seqüências de teste que não satisfizer as condições de suficiência, é possível visualizar quais transições da MEF que não foram percorridas e quais seqüências não foram confirmadas pelo algoritmo. A Figura 3.3 ilustra a interface desenvolvida para o **CheSCon**.



(a) Entrada de Dados (b) MEF

(c) Árvore de Teste (d) Grafo de Distinção

Figura 3.3: Interface desenvolvida para o **CheSCon**.

3.5.3 Gerador de Seqüências de Verificação

Esse aplicativo, denominado como **GGCS** (*Genetic Generator of Checking Sequence*), é a implementação do algoritmo descrito na Seção 3.2. Além de utilizar o aplicativo **GDS** para gerar o conjunto de distinção utilizado pelo algoritmo, o **GGCS** também faz uso de uma versão específica do aplicativo **CheSCon**, denominada como **CheSCon CS**, que é direcionada para verificar somente conjuntos de casos de teste unitários, ou seja, formado por apenas uma seqüência. Como o **GGCS** gera muitas seqüências durante a sua execução e é necessário verificar se essas seqüências geradas satisfazem as condições de suficiência, o **CheSCon** tem um papel importante no desempenho do **GGCS** em relação ao tempo de execução, visto que a maior parte do processamento do **GGCS** se concentra nessa verificação. O **CheSCon CS** tem o objetivo de minimizar o tempo de execução no momento de verificar se uma seqüência satisfaz as condições de suficiência. Como todas as seqüências geradas pelo **GGCS** são conjuntos unitários, não há a necessidade de utilizar a versão do **CheSCon** que verifica conjuntos não unitários. Com as alterações realizadas, as estruturas de dados do **CheSCon CS** foram simplificadas em comparação com a versão

original do aplicativo. Dessa maneira, conseguiu-se reduzir significativamente o tempo de execução do verificador das condições de suficiência e, conseqüentemente, reduzir o tempo de execução do **GGCS**.

Para executar o **GGCS**, é necessário passar como parâmetro dois arquivos no formato texto: o primeiro contendo a MEF e o segundo contendo o conjunto de distinção. Ambos arquivos devem estar de acordo com o formato ilustrado na Tabela 3.8. Um exemplo de comando para executar o **GGCS** é:

```
java -jar ggcs -fsm fsmFile.txt -ds dsFile.txt
```

Outro parâmetro que pode ser definido na execução do aplicativo é o tempo máximo de execução (parâmetro *timeLimit*). Com esse parâmetro, o **GGCS** será interrompido ao atingir o limite de tempo estipulado e retornará o cromossomo mais apto da população atual, que corresponde à menor seqüência de verificação gerada até aquele momento. Um exemplo de comando para executar o **GGCS** com limite de tempo de 10 segundos é:

```
java -jar ggcs -fsm fsmFile.txt -ds dsFile.txt -timeLimit 10
```

O tamanho da população e o tamanho do conjunto *Fittest* também podem ser definidos no momento da execução do aplicativo, utilizando para isso os parâmetros *populationSize* e $fittestSize$, respectivamente, conforme o exemplo a seguir:

```
java -jar ggcs -fsm fsmFile.txt -ds dsFile.txt

     -populationSize 200 -fittestSize 50
```

Caso não sejam informados os valores dos parâmetros *populationSize* e $fittestSize$ na execução do **GGCS**, o aplicativo atribui um valor padrão a estes: 100 e 30, respectivamente. Esses dois parâmetros influenciam diretamente na eficiência do algoritmo em relação ao tamanho das seqüências de verificação geradas e no seu desempenho em relação ao tempo de execução, conforme será demonstrado no Capítulo 4.

O **GGCS** pode ser executado em conjunto com o método de geração de seqüências de verificação proposto por Simão e Petrenko (2008). Ao ser executado, o referido método armazena em um arquivo todas as operações realizadas para a construção da seqüência de verificação. Por meio desse arquivo, que é passado por parâmetro ao **GGCS**, é possível gerar um cromossomo tal que a sua seqüência χ é idêntica à seqüência de verificação gerada pelo método de Simão e Petrenko (2008) e, dessa maneira, a população inicial do **GGCS** conterá a seqüência gerada pelo método referido. Assim, o **GGCS** pode ser utilizado como um aplicativo de redução de seqüências, pois a seqüência gerada pelo método de Simão e Petrenko (2008) e presente na população inicial do **GGCS** pode sofrer

mutações que a tornem uma seqüência de verificação menor. Para utilizar a seqüência de verificação gerada pelo método de Simão e Petrenko (2008), deve-se utilizar o parâmetro *sp08*, conforme o exemplo a seguir:

```
java -jar ggcs -fsm fsmFile.txt -ds dsFile.txt -sp08 outputSP08Method.txt
```

Caso o **GGCS** gere uma seqüência de verificação, o aplicativo retorna como saída o valor 0 e a menor seqüência de verificação gerada pelo algoritmo genético. A seqüência de verificação gerada pode ser impressa na tela ou pode ser armazenada em um arquivo, sendo que para realizar esse armazenamento é necessário informar o nome do arquivo na execução do aplicativo por meio do parâmetro *output*, conforme o exemplo a seguir:

```
java -jar ggcs -fsm fsmFile.txt -ds dsFile.txt -output outputGGCS.txt
```

Caso o **GGCS** não gere uma seqüência de verificação por um determinado motivo, como, por exemplo, parâmetros de entrada informados incorretamente, o aplicativo retorna como saída o valor 255.

Por se tratar de um aplicativo baseado no algoritmo genético, o desempenho em relação ao tempo de execução do **GGCS** é prejudicado a medida que a quantidade de transições das MEFs aumenta, fato que ocorre pelo acréscimo de estados ou de entradas válidas das MEFs. Para solucionar esse problema, foi desenvolvida uma versão específica do **GGCS**, denominada **GGCS Distributed**, que realiza o processamento dos dados utilizando vários processos para minimizar o seu tempo de execução. Essa versão pode ser utilizada tanto em *clusters* quanto em computadores com mais de um processador (ou com processadores com mais de um núcleo).

O **GGCS Distributed**, também implementado utilizando a linguagem de programação Java, possui o mesmo algoritmo de geração de seqüência de verificação do **GGCS**, porém com uma arquitetura específica para permitir a sua execução em vários processos ao mesmo tempo. O aplicativo utiliza a arquitetura **RMI** (*Remote Method Invocation*), que provê funcionalidades de uma plataforma de objetos distribuídos na arquitetura cliente-servidor. Por meio da utilização da arquitetura RMI, é possível que um objeto em uma máquina virtual Java (objeto cliente) possa interagir remotamente com objetos de outras máquinas virtuais Java (objetos servidores).

O **GGCS Distributed** é formado por um objeto cliente, que é considerado o processo principal do aplicativo, e por vários objetos servidores, que são considerados os processos auxiliares do aplicativo. O processo principal tem o objetivo de gerenciar a população e os processos auxiliares têm como finalidade duas tarefas: gerar as seqüências χ de um conjunto de cromossomos da população e verificar a completude dessas seqüências. Para cada processo auxiliar a ser utilizado, é necessário ter um objeto servidor do aplicativo

em execução. No caso do **GGCS Distributed**, tanto o objeto cliente quanto o objeto servidor são o próprio **GGCS Distributed**, sendo que no momento da execução de tais objetos, é necessário definir o tipo da execução. Como exemplo, tem-se, respectivamente, o comando para a execução de um objeto cliente, em que é necessário passar como parâmetro os arquivos que contêm uma MEF M e o conjunto de distinção de M, e o comando para a execução do objeto servidor, no qual devem ser informados os parâmetros $-cp$ e $Server$ para determinar a execução da classe $Server$ do aplicativo:

`java -jar ggcsDistributed.jar -fsm fsmFile.txt -ds dsFile.txt` e

`java -cp ggcsDistributed.jar Server`

O acesso do objeto cliente aos objetos servidores é realizado por meio do Protocolo da Internet (IP) do computador no qual o objeto servidor está sendo executado, sendo que durante toda a execução do objeto cliente, os objetos servidores utilizados pelo objeto cliente devem estar em execução. Os objetos servidores a serem utilizados pelo objeto cliente do **GGCS Distributed** devem ser definidos em um arquivo de propriedades do aplicativo. Como exemplo, considerando um *cluster* formado por 20 nós e o primeiro nó como o processo principal do aplicativo (objeto cliente), os 19 restantes serão os processos auxiliares (objetos servidores). Desse modo, deve-se executar o objeto servidor nos 19 nós citados e, posteriormente, deve-se executar o objeto cliente no primeiro nó do *cluster*, que irá acessar remotamente os objetos servidores por meio dos endereços IP dos 19 nós, sendo que esses endereços IP devem estar presentes no arquivo de propriedades do **GGCS Distributed**.

Considerando o algoritmo genético de geração de seqüências de verificação explicado anteriormente, quando uma nova população é formada, o processo principal do **GGCS Distributed** distribui os cromossomos entre os processos auxiliares para a geração das respectivas seqüências χ e para verificar a completude das próprias seqüências χ, que são os procedimentos responsáveis pela maior parte do processamento do algoritmo proposto. Considerando uma população com 100 cromossomos e que o **GGCS Distributed** será executado em um *cluster* com 20 nós, cada nó será responsável pelo processamento de apenas 5 cromossomos em cada iteração do algoritmo, gerando as suas seqüências χ e verificando a completude dessas 5 seqüências.

Após todos os processos calcularem as seqüências χ dos seus cromossomos, o processo principal reúne todos os cromossomos retornados pelos processos auxiliares e define o $fitness$ de cada cromossomo. Posteriormente, o processo principal irá gerar os novos cromossomos a partir de mutações. Como esse procedimento requer pouco processamento, não há a necessidade de fazer o uso dos processos auxiliares nesse ponto do algoritmo.

Após a geração dos novos cromossomos e, conseqüentemente, de uma nova população, o processo novamente divide os cromossomos da nova população entre os processos auxiliares para a geração das novas seqüências χ e para a verificação da completude dessas seqüências, definindo assim o processo iterativo do algoritmo.

Experimentos realizados no *cluster* disponível aos pesquisadores dos grupos de pesquisas do ICMC/USP demonstraram uma evolução significativa do **GGCS Distributed** comparado ao **GGCS** em relação ao tempo de execução, conforme será demonstrado no Capítulo 4. Outra vantagem do **GGCS Distributed** é a possibilidade de utilizar parâmetros otimizados para MEFs com uma quantidade maior de transições, permitindo gerar seqüências de verificação menores comparadas às seqüências geradas pelo **GGCS**. Como o tempo de execução do algoritmo genético é diretamente relacionado ao tamanho da população em cada iteração, a definição de uma população consideravelmente maior torna inviável a execução do **GGCS**. Entretanto, é possível definir uma população maior no **GGCS Distributed** e, ainda sim, obter a seqüência de verificação em um tempo aceitável, dependendo do computador ou do *cluster* utilizado. Com uma população maior, é possível gerar seqüências de verificação menores, conforme será demonstrado no Capítulo 4.

3.6 Considerações Finais

Neste capítulo foi apresentado o algoritmo de geração de seqüências de verificação desenvolvido neste trabalho de mestrado, assim como um exemplo de utilização desse algoritmo. Os aplicativos implementados no decorrer no trabalho também foram descritos, ilustrando os formatos dos parâmetros de entradas e as saídas retornadas por cada implementação.

O próximo capítulo visa a apresentar uma avaliação experimental do algoritmo de geração de seqüência de verificação, utilizando MEFs de diferentes características, identificando o comportamento do algoritmo desenvolvido.

Capítulo

4

Avaliação Experimental

4.1 Considerações Iniciais

Para avaliar o comportamento do algoritmo proposto foram realizados três estudos experimentais em diferentes contextos. No primeiro estudo experimental, apresentado na Seção 4.2, foram realizados quatro experimentos comparando o método proposto neste trabalho com o método proposto por Simão e Petrenko (2008) diante de MEFs de diferentes parâmetros. No primeiro experimento, os dois métodos são comparados com MEFs de diferentes quantidades de estados. No segundo experimento, a comparação é feita com MEFs de diferentes quantidades de entradas e no terceiro para diferentes quantidades de saídas. Por fim, o quarto experimento compara os métodos com MEFs de diferentes quantidades de transições. A Seção 4.3 apresenta o segundo estudo experimental, com dois experimentos com o método proposto. O primeiro experimento tem o objetivo de avaliar o comportamento do método proposto sob diferentes valores para os parâmetros de configuração do método (tamanho da população e tamanho do conjunto *Fittest*). O segundo experimento realiza uma comparação entre o **GGCS** utilizando as condições propostas por Simão e Petrenko (2010) e o **GGCS** utilizando as condições propostas por Ural et al. (1997). O terceiro estudo experimental é apresentado na Seção 4.4. Esse último estudo é composto por dois experimentos, ambos com a finalidade de comparar o **GGCS** com o **GGCS Distributed**, versão do **GGCS** que realiza o processamento dos dados utilizando vários processos, sendo que o primeiro experimento avalia o tempo de execução

entre as duas versões e o segundo avalia o tamanho das seqüências de verificação geradas entre as duas versões, sendo que nesse último experimento, os parâmetros de configuração do **GGCS Distributed** foram modificados.

4.2 Estudo Experimental 1

O primeiro estudo experimental tem o objetivo de avaliar o comportamento do aplicativo **GGCS** diante de MEFs de diferentes parâmetros, sendo que para cara experimento é variado apenas um parâmetro da MEF. No primeiro experimento realizado, o parâmetro variado é a quantidade de estados; no segundo, a quantidade de entradas das MEFs é o parâmetro utilizado para a variação; no terceiro experimento deste estudo experimental varia-se a quantidade de saídas das MEFs; por fim, no último experimento varia-se a quantidade de transições das MEFs.

Nos quatro experimentos as MEFs utilizadas são geradas aleatoriamente. Foram utilizadas MEFs aleatórias devido ao fato de permitirem determinar o comportamento médio do algoritmo proposto neste trabalho. Outros trabalhos fizeram uso de MEFs aleatórias, tais como (Dorofeeva et al., 2005a; Hierons e Ural, 2006; Simão et al., 2007). A geração de MEFs ocorre em três etapas. Primeiramente é selecionado um estado da MEF que será o estado inicial e é marcado como alcançável. Após a seleção do estado inicial, para cada estado s_i não marcado como alcançável é criada uma transição de s_j para s_i, sendo que o estado s_j deve estar marcado como alcançável e é escolhido aleatoriamente, assim como a entrada e a saída da transição. Dessa maneira, s_i é marcado como alcançável. Após todos os estados estarem marcados como alcançáveis, a segunda etapa adiciona novas transições aleatoriamente, caso haja a necessidade. Por fim, a última etapa verifica se a MEF é minimal e caso não seja, a MEF é rejeitada e é iniciado a geração de uma nova MEF (Simão e Petrenko, 2010).

Nesses experimentos, além do algoritmo **GGCS**, foi utilizado também o método proposto por Simão e Petrenko (2008) (denominado neste trabalho como **SP08**), para realizar uma comparação entre os dois métodos. O método proposto por Simão e Petrenko (2008) foi escolhido devido ao fato que esse método gera seqüências de verificação menores que os demais métodos de geração existentes na literatura (Simão e Petrenko, 2008). Os parâmetros utilizados pelo **GGCS** neste estudo experimental são $PopulationSize = 100$ e $FittestSize = 30$, que são os parâmetros que apresentaram os melhores resultados para o **GGCS**, conforme apresentado na seção 4.3.1.

4.2.1 Variação da Quantidade de Estados

O objetivo desse experimento é determinar o comportamento do algoritmo **GGCS** diante a MEFs com diferentes quantidades de estados. Para a realização desse experimento foram geradas aleatoriamente MEFs variando a quantidade de estados, no intervalo de 4 a 20 estados (variando de 2 em 2 estados). Desse modo, têm-se nove configurações de MEFs: 4, 6, 8, 10, 12, 14, 16, 18 e 20 estados. Cada uma das noves configurações são formadas por 50 MEFs, todas minimais, completas e com 4 entradas e 4 saídas.

A Figura 4.1 ilustra o gráfico do experimento, apresentando a relação da redução das seqüências de verificação obtidas pelo **GGCS** (diante do método **SP08**) com a quantidade de estados das MEFs. É possível observar que a quantidade de estados da MEF influencia diretamente a taxa de redução das seqüências de verificação geradas pelo **GGCS** em relação ao método **SP08**, sendo que a tendência é que quanto mais estados uma MEF possui, menor é a redução obtida pelo **GGCS** diante do método **SP08**.

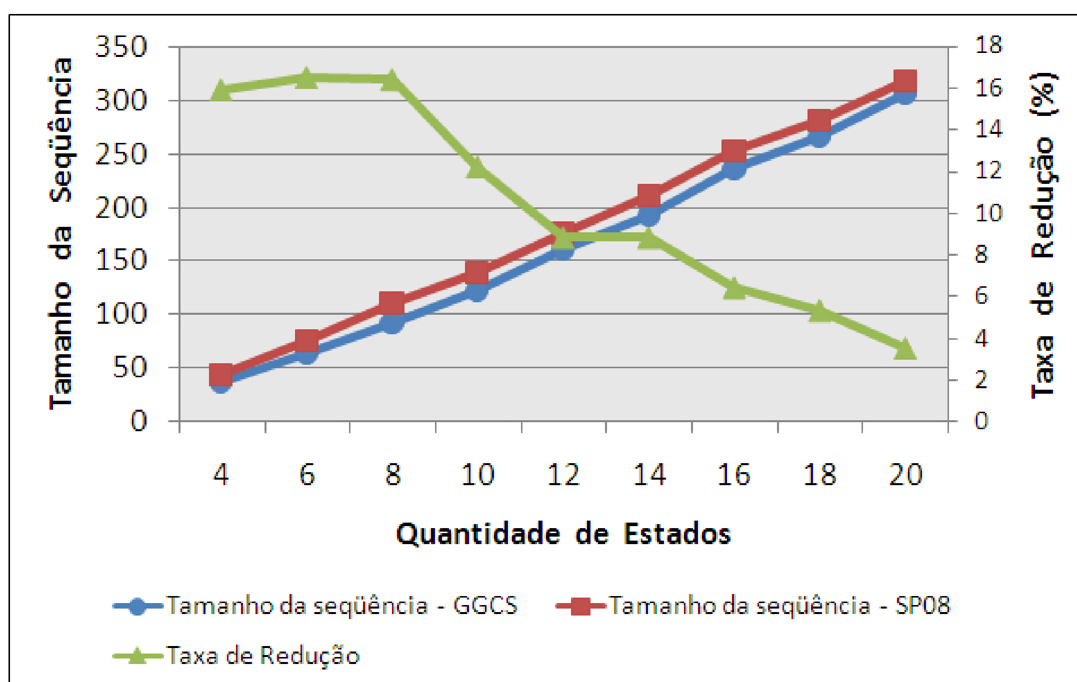


Figura 4.1: Comparação entre os métodos variando a quantidade de estados.

4.2.2 Variação da Quantidade de Entradas

De maneira análoga ao experimento anterior, esse experimento foi realizado para determinar o comportamento do **GGCS** com MEFs de diferentes quantidades de entradas. Foram utilizadas MEFs completas, com 10 estados e 4 saídas, variando-se a quantidade de entradas: 2, 3, 4 e 5 entradas. Desse modo, têm-se 4 configurações diferentes de MEFs, sendo que para cada configuração foram geradas aleatoriamente 50 MEFs.

Na Figura 4.2 é ilustrado graficamente os dados obtidos pelo experimento, apresentando a relação da redução das seqüências de verificação obtidas pelo **GGCS** (diante do método **SP08**) com a quantidade de entradas das MEFs. Nesse experimento, também é possível inferir que o tamanho da MEF influencia diretamente a taxa de redução do **GGCS** diante do método **SP08**, sendo que a tendência é que quanto maior a quantidade de entradas, menor é a redução obtida.

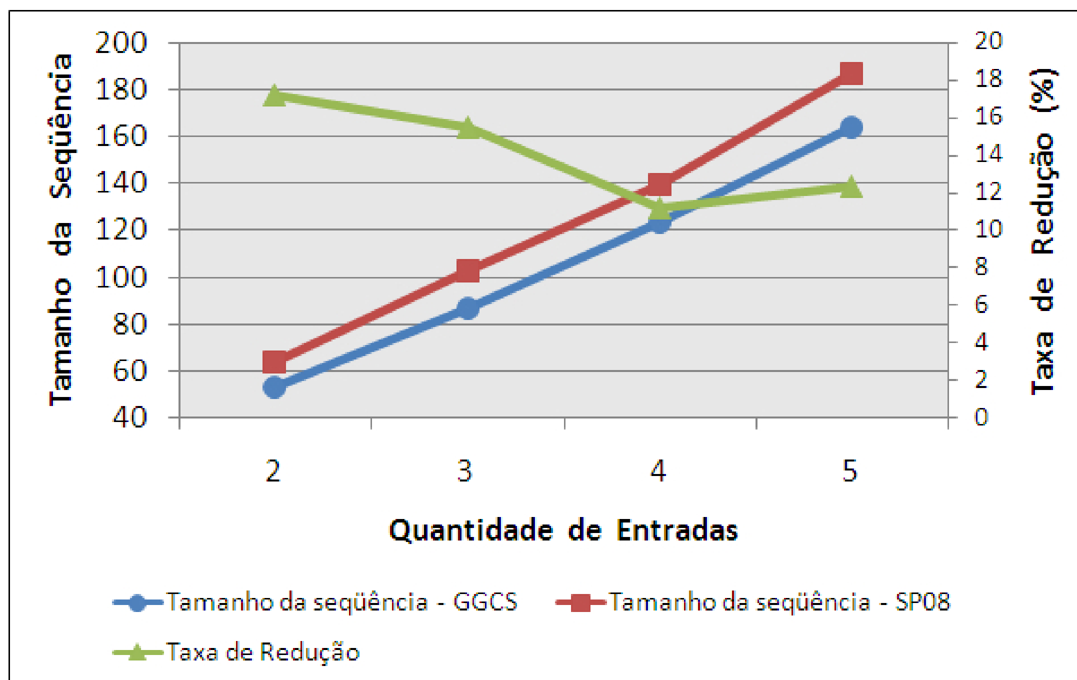


Figura 4.2: Comparação entre os métodos variando a quantidade de entradas.

4.2.3 Variação da Quantidade de Saídas

Para esse experimento foram geradas MEFs completas variando a quantidade de saídas e fixando os demais parâmetros: 10 estados e 4 entradas. Assim como nos experimentos descritos nas Seções 4.2.1 e 4.2.2, o objetivo é determinar o comportamento do **GGCS** diante a um conjunto de MEFs com configurações diferentes. A quantidade de saídas é variada de 2 a 5 saídas e, dessa maneira, têm-se 4 configurações de MEFs distintas. Para cada configuração, foram geradas aleatoriamente 50 MEFs.

A Figura 4.3 apresenta o gráfico do experimento, ilustrando a relação da redução das seqüências de verificação obtidas pelo **GGCS** (diante do método **SP08**) com a quantidade de saídas das MEFs. Apesar da quantidade de saídas não influenciar no tamanho de uma MEF, pode-se afirmar que esse parâmetro altera o comportamento do **GGCS** diante do método **SP08**. No experimento realizado, é possível observar que quanto maior a

quantidade de saídas de uma MEF, maior é a eficiência do **GGCS** em gerar seqüências de verificação menores que o método **SP08**.

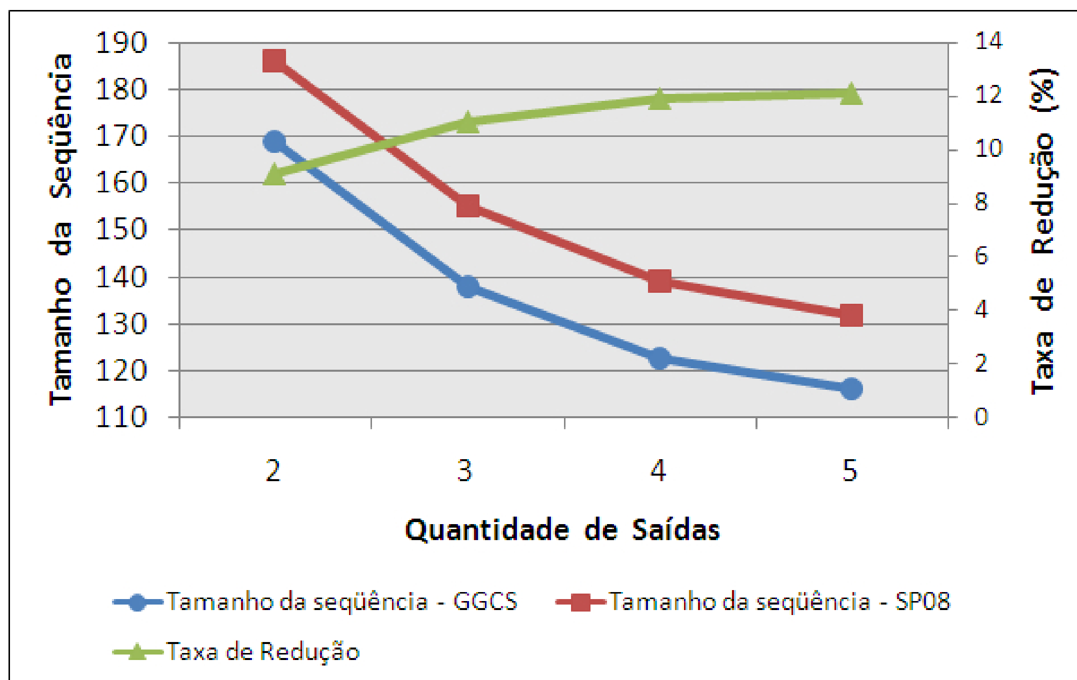


Figura 4.3: Comparação entre os métodos variando a quantidade de saídas.

4.2.4 Variação da Quantidade de Transições

Esse experimento tem a finalidade de avaliar o **GGCS** diante de MEFs com quantidades diferentes de transições. Dessa maneira, foram fixados os parâmetros das MEFs da seguinte maneira: 10 estados, 4 entradas e 4 saídas. Considerando que uma MEF completa com esses parâmetros possui 40 transições, a quantidade de transições foi variada de 20 transições a 40 transições, variando de 2 em 2 transições. Assim, têm-se 11 conjuntos de MEFs com configurações distintas em relação à quantidade de transições, sendo que cada conjunto é composto por 50 MEFs geradas de maneira aleatória.

A relação de redução das seqüências de verificação obtidas pelo **GGCS** (diante do método **SP08**) com a quantidade de transições das MEFs utilizadas no experimento é apresentada de maneira gráfica na Figura 4.4. Pode-se observar que a quantidade de transições é outro parâmetro que altera o comportamento do **GGCS**, alterando a taxa de redução do **GGCS** em relação ao método **SP08**. Quanto mais transições que uma MEF possui, maior é a taxa de redução obtida pelo **GGCS**.

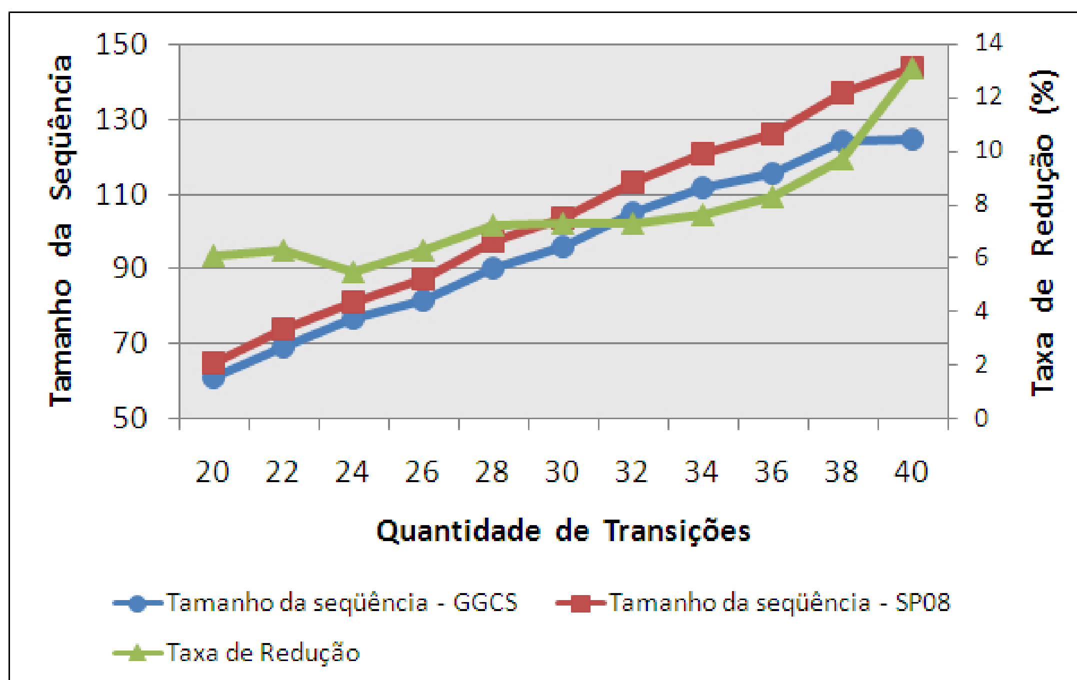


Figura 4.4: Comparação entre os métodos variando a quantidade de transições.

4.3 Estudo Experimental 2

O segundo estudo experimental tem como finalidade avaliar o comportamento do **GGCS** sob diferentes configurações. Desse modo, foram realizados dois experimentos. No primeiro experimento o **GGCS** é avaliado sob diferentes valores para os parâmetros de configuração do método (tamanho da população e tamanho do conjunto *Fittest*), determinando a influência de tais parâmetros na geração de seqüências de verificação pelo método proposto. O segundo experimento tem o objetivo realizar uma comparação entre as seqüências de verificação geradas pela versão do **GGCS** que utiliza as condições de suficiência propostas por Simão e Petrenko (2010), utilizadas nos experimentos anteriores, em relação às seqüências de verificação geradas pelpor uma versão do **GGCS** que utiliza as condições de suficiência propostas por Ural et al. (1997), avaliando assim a relação entre o tamanho das seqüências de verificação geradas de acordo com as condições de suficiência utilizadas. Nas próximas seções são apresentados os experimentos com mais detalhes.

4.3.1 Comparação entre Parâmetros de Configuração

Esse experimento tem como objetivo avaliar a relação dos dois parâmetros de configuração do método proposto neste trabalho (tamanho da população e tamanho do conjunto *Fittest*) com o tamanho das seqüências de verificação geradas pelo método e com o tempo de execução para essa geração. Nesse experimento, foi utilizado um conjunto de 50 MEFs

completas com os seguintes parâmetros: 10 estados, 4 entradas e 4 saídas. O experimento é dividido em duas etapas. A primeira avalia o **GGCS** fixando o tamanho da população ($PopulationSize = 100$) e variando o tamanho do conjunto *Fittest*. A segunda etapa, o parâmetro fixado é o conjunto *Fittest* ($FittestSize = 30$) e o parâmetro variado é o tamanho da população.

A Figura 4.5 apresenta os dados da primeira etapa, expondo para cada tamanho do conjunto *Fittest* o tamanho médio das seqüências de verificação geradas para o conjunto de MEFs citado anteriormente e a média do tempo de execução (em segundos) para a geração dessas seqüências. Os dados da segunda etapa são apresentados na Figura 4.6, expondo os mesmos tipos de dados da tabela referente à primeira etapa, porém nesse caso os dados são apresentados para cada tamanho da população.

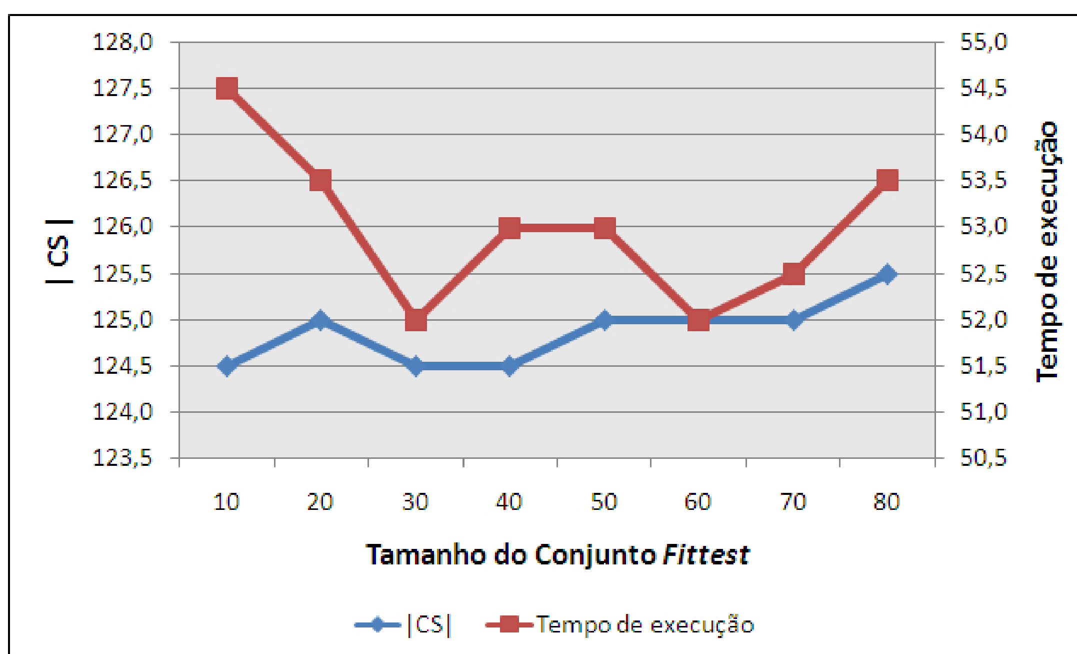


Figura 4.5: Tamanho das seqüências de verificação e o tempo de execução variando do tamanho do conjunto *Fittest*.

Considerando a primeira etapa do experimento, é possível observar que o melhor valor para o tamanho do conjunto *Fittest* é igual a 30, uma vez que a execução do **GGCS** para esse valor gerou as menores seqüências de verificação possíveis no menor tempo de execução quando comparado com os demais valores do experimento. Apesar de outros parâmetros terem gerado seqüências de verificação do mesmo tamanho, o tempo de execução nesses casos foi maior, o que faz o conjunto *Fittest* igual a 30 a melhor opção nesse contexto.

Para a segunda etapa, nota-se que o tamanho da população influencia no tamanho das seqüências de verificação geradas e que há uma relação inversamente proporcional entre o tamanho das seqüências de verificação geradas e o tempo de execução. Nessa etapa,

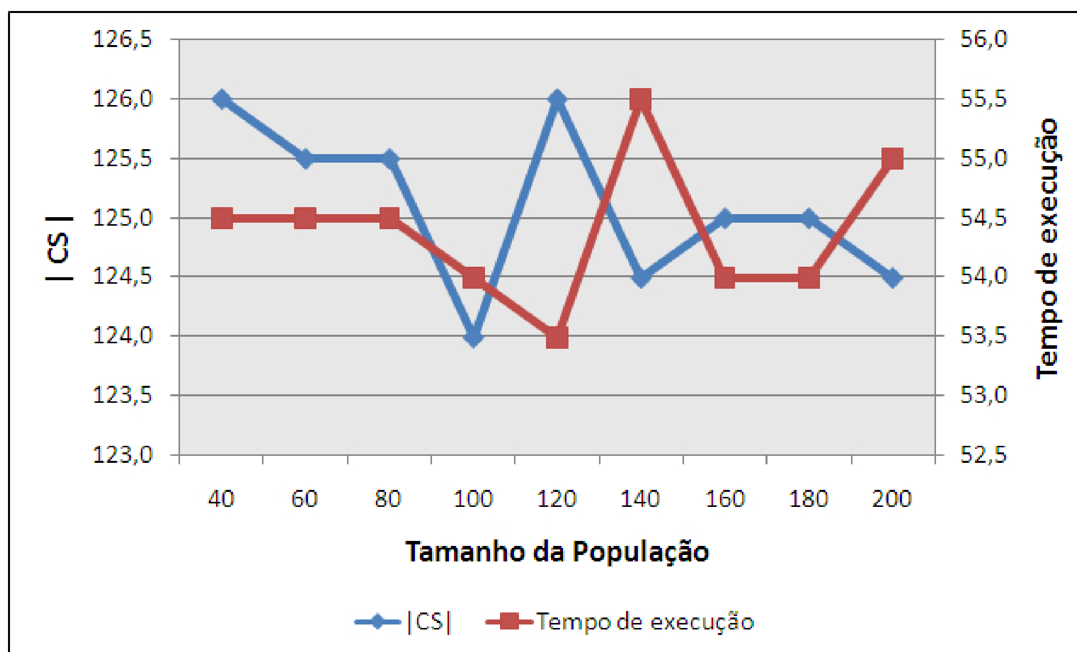


Figura 4.6: Tamanho das seqüências de verificação e o tempo de execução variando do tamanho da população.

é possível observar que o melhor valor para o tamanho da população é o valor igual a 100, uma vez que com esse valor, foi obtida a menor média do tamanho das seqüências de verificação geradas pelo **GGCS** junto com o menor tempo de execução.

Apesar dos valores do tamanho das seqüências de verificação e dos tempos de execução serem próximos para os diferentes valores dos dois parâmetros utilizados nesse experimento, tanto na primeira quanto na segunda etapa, é possível afirmar que os dois parâmetros do método proposto influenciam diretamente o tamanho das seqüências de verificação geradas e o tempo de execução para a geração das seqüências de verificação. Considerando os resultados obtidos, os melhores parâmetros para o **GGCS** são $PopulationSize = 100$ e $FittestSize = 30$, sendo que esses valores são utilizados pelo **GGCS** nos demais experimentos.

4.3.2 Comparação entre Condições de Suficiências

O método de geração de seqüências de verificação proposto neste trabalho utiliza as condições de suficiência apresentadas em (Simão e Petrenko, 2010). Entretanto, é possível utilizar outras condições de suficiência para verificar quais seqüências geradas pelo método são seqüências de verificação. Este estudo experimental foi realizado com a finalidade de comparar o **GGCS** com diferentes condições de suficiência. Para isso, além das condições de suficiência propostas por Simão e Petrenko (2010), também foram utilizadas as condições de suficiência propostas em (Ural et al., 1997). Apesar das condições de

Simão e Petrenko (2010) serem mais simples que as condições propostas por Ural et al. (1997), o objetivo do experimento é verificar a influência que as condições de suficiência exercem sobre o o método proposto. Para isso, foi realizado um experimento utilizando 50 MEFs completas, de 10 estados, 4 entradas e 4 saídas, todas geradas aleatoriamente conforme descrito na Seção 4.2. A Figura 4.7 apresenta o gráfico do experimento, ilustrando as reduções das seqüências de verificação obtidas pelo **GGCS** com as condições de suficiência propostas em (Simão e Petrenko, 2010) em relação às seqüências de verificação obtidas pelo **GGCS** com as condições propostas em (Ural et al., 1997) para cada MEF do experimento.

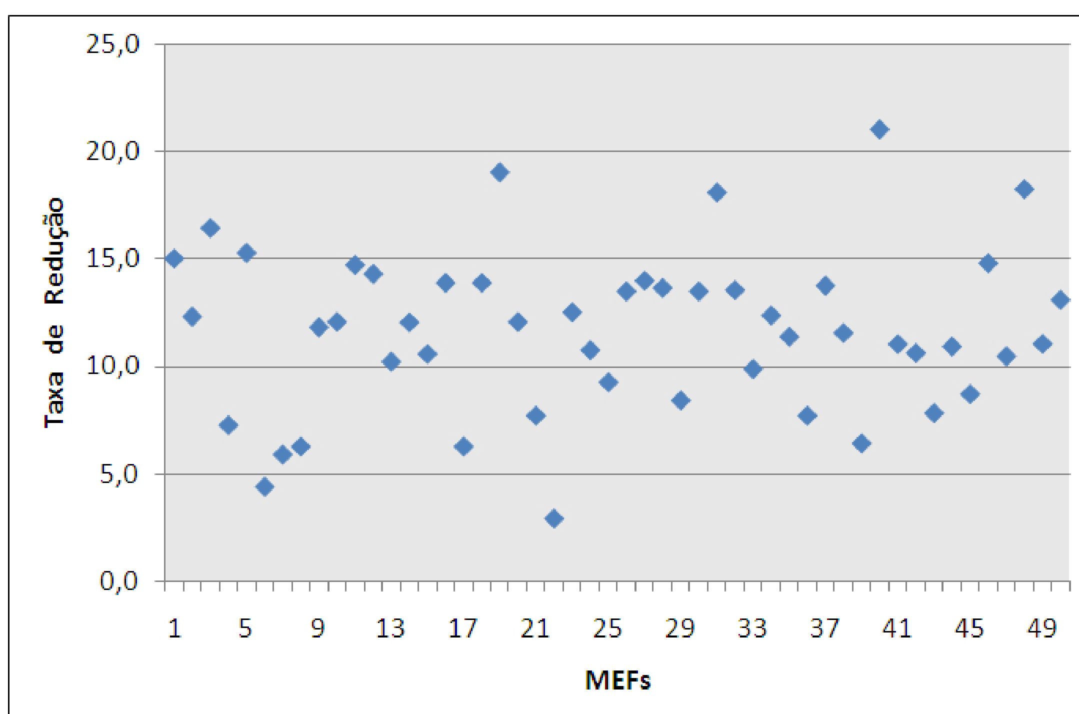


Figura 4.7: Comparação entre as condições de suficiência de Simão e Petrenko (2010) com as condições propostas por Ural et al. (1997).

A média de redução obtida pelo **GGCS** com as condições de suficiência propostas por Simão e Petrenko (2010) em relação ao **GGCS** com as condições propostas por Ural et al. (1997) foi de 11,6%, o que demonstra que as condições de suficiência utilizadas pelo **GGCS** influenciam diretamente o tamanho das seqüências de verificação geradas pelo aplicativo. Dessa maneira, é possível afirmar que a utilização de condições de suficiência mais simples do que as propostas por Simão e Petrenko (2010) poderia aumentar ainda mais a eficiência do **GGCS** em relação ao tamanho das seqüências de verificação geradas, produzindo seqüências menores do que as geradas atualmente pelo aplicativo.

4.4 Estudo Experimental 3

O último estudo experimental tem como finalidade avaliar o **GGCS** diante do **GGCS Distributed**, versão do **GGCS** descrita na Seção 3.5.3. Assim, dois experimentos foram realizados. O primeiro experimento avalia o tempo de execução do **GGCS** e do **GGCS Distributed** para a geração de uma seqüência de verificação para um determinado conjunto de MEFs. O segundo experimento avalia o comportamento do **GGCS** diante o **GGCS Distributed**, cujos parâmetros de configuração (tamanho da população e tamanho do conjunto *Fittest*) foram consideravelmente incrementados para avaliar se essa modificação pode resultar na geração de seqüências de verificação menores. Nas próximas seções são apresentados os experimentos com mais detalhes.

4.4.1 Comparação entre Tempos de Execução

Conforme descrito na Seção 3.5.3, foi desenvolvida uma nova versão do **GGCS**, denominada **GGCS Distributed**, para realizar o processamento dos dados utilizando vários processos para minimizar o tempo de execução do aplicativo **GGCS**. Para avaliar sua eficiência em relação ao tempo de execução, foi realizado um experimento com 50 MEFs geradas aleatoriamente com os seguintes parâmetros: completamente especificadas, 20 estados, 5 entradas e 5 saídas.

Com esse experimento, é possível determinar o ganho obtido com a nova versão do aplicativo. O experimento foi realizado no *cluster* disponível aos pesquisadores dos grupos de pesquisas do ICMC/USP. Esse *cluster* é composto por 14 nós, além do nó mestre, sendo que cada nó é composto por um computador equipado com: 2 processadores Intel Pentium 4 CPU 3.40 GHz, 3 GB de memória RAM, sistema operacional Linux e *Java Runtime Environment* (JRE) versão 1.6.

A Figura 4.8 apresenta o gráfico do experimento, ilustrando o tempo de execução do **GGCS** e do **GGCS Distributed** para cada MEF do experimento.

Enquanto o **GGCS** precisou de, em média, 40 minutos para gerar uma seqüência de verificação para uma MEF do experimento, o **GGCS Distributed** necessitou de 4 minutos, obtendo uma redução de 90% em relação ao tempo de execução do **GGCS**. Desse modo, o experimento demonstra que a utiliza da versão distribuída do **GGCS** reduz significativamente o tempo de execução para a geração de seqüências de verificação, sendo que esse tempo pode ser reduzido ainda mais com a utilização de cluster com uma quantidade maior de nós.

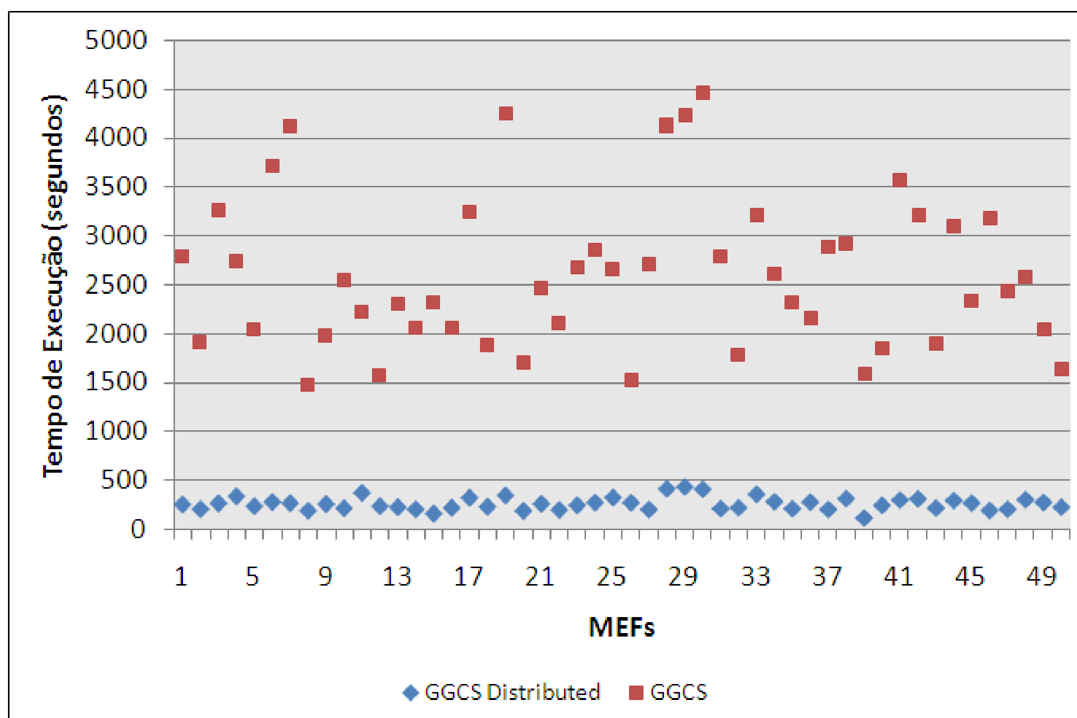


Figura 4.8: Tempo de execução do **GGCS** e do **GGCS Distributed**.

4.4.2 Comparação entre Parâmetros de Configuração

Como o *cluster* disponível aos pesquisadores dos grupos de pesquisas do ICMC/USP oferece um maior poder de processamento, é possível otimizar os parâmetros do **GGCS Distributed** para a geração de seqüências de verificação ainda menores. Dessa maneira, esse experimento tem o objetivo de avaliar o tamanho das seqüências de verificação geradas pelo **GGCS** com diferentes parâmetros de configuração:

- **GGCS**: $PopulationSize = 100$ e $FittestSize = 30$.
- **GGCS Distributed**: $PopulationSize = 1000$ e $FittestSize = 300$.

Esse experimento utilizou 50 MEFs completas com 20 estados, 5 entradas e 5 saídas, geradas aleatoriamente. O gráfico do experimento é ilustrado na Figura 4.9, apresentando as reduções obtidas pelo **GGCS Distributed** em relação ao **GGCS** para cada MEF do experimento.

O experimento demonstra que os parâmetros do aplicativo influenciam no tamanho das seqüências de verificação geradas. O **GGCS Distributed** gerou, em média, seqüências de verificação 5,8% menores que as seqüências geradas pelo **GGCS**. Apesar de utilizarem arquiteturas diferentes, as duas versões do aplicativo possuem o mesmo algoritmo de geração de seqüências de verificação. Assim, é possível afirmar que a diferença no tamanho das seqüências de verificação geradas entre as duas versões está relacionada à diferença de configuração de cada versão.

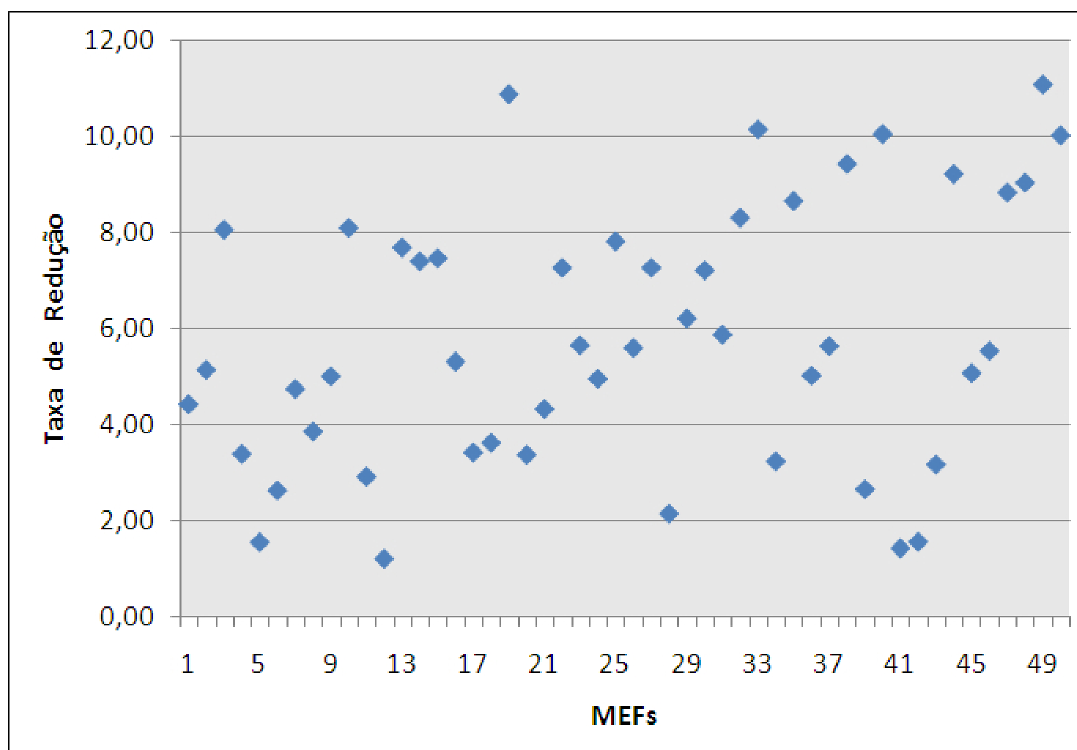


Figura 4.9: Taxa de redução obtida pelo **GGCS Distributed** em relação ao **GGCS**.

4.5 Considerações Finais

Este capítulo apresentou os estudos experimentais realizados com o objetivo de avaliar o método de geração de seqüências de verificação proposto neste trabalho. Os experimentos do primeiro estudo experimental se concentraram em determinar o comportamento do aplicativo **GGCS** com diferentes MEFs. Para isso, foram realizados quatro experimentos, sendo que em cada experimento um parâmetro das MEFs utilizadas era selecionado para sofrer variação de seus valores. No primeiro experimento, o parâmetro selecionado foi a quantidade de estados das MEFs. No segundo e no terceiro experimento, a variação foi realizada sobre a quantidade de entradas e de saídas, respectivamente. Por fim, no último experimento foi variada a quantidade de transições das MEFs.

No segundo estudo experimental, os experimentos foram direcionados para avaliar o comportamento do **GGCS** variando os seus próprios parâmetros, além de avaliar o **GGCS Distributed**. Nesse estudo experimental, o primeiro experimento comparou o comportamento do **GGCS** com diferentes condições de suficiência, utilizando para isso as condições de suficiência propostas por Simão e Petrenko (2010) e as condições propostas por Ural et al. (1997). No segundo experimento, foi avaliada a diferença do tempo de execução do **GGCS** e do **GGCS Distributed**. O terceiro experimento avaliou o tamanho das seqüências de verificação geradas pelo **GGCS** e pelo **GGCS Distributed**, porém essa segunda versão do aplicativo teve o tamanho da população e tamanho do conjunto

Fittest incrementados com a finalidade de gerar seqüências de verificação menores que o **GGCS**.

No próximo capítulo serão apresentadas as conclusões deste trabalho de mestrado, apresentando as dificuldades encontradas, as contribuições geradas, as limitações do método proposto e as direções para trabalhos futuros.

Capítulo

5

Conclusões

A geração de seqüências de verificação reduzidas pode agilizar a execução da atividade de testes, tornando-a menos complexa. Dessa forma, a pesquisa de métodos que gerem seqüências de verificação menores que os métodos atuais se torna relevante.

Neste trabalho foi abordado o problema da geração de seqüências de verificação reduzidas a partir de MEFs, propondo um método de geração baseado na estratégia de algoritmos genéticos e nas condições de suficiência propostas em Simão e Petrenko (2010). O método gerou seqüências de verificação menores que as geradas pelo método proposto por Simão e Petrenko (2008), o qual gerou seqüências menores que os métodos propostos por Chen et al. (2005) e Hierons e Ural (2006). Além disso, três estudos experimentais sobre os limites teóricos do método foram realizados: o primeiro para determinar o comportamento do método com diferentes MEFs, o segundo para avaliar o comportamento do método variando os seus próprios parâmetros e o terceiro para avaliar o **GGCS** diante do **GGCS Distributed**.

A seguir são apresentadas as contribuições geradas por esse trabalho de mestrado, as dificuldades encontradas, as limitações do método proposto e as direções para trabalhos futuros.

5.1 Contribuições

A principal contribuição deste trabalho de mestrado refere-se ao desenvolvimento de um novo método de geração de seqüências de verificação. Além das condições de suficiência propostas por Simão e Petrenko (2010), o método proposto é baseado na técnica de algoritmos genéticos, algo ainda pouco pesquisado nesse contexto. Baseado nesse método, foram implementados dois aplicativos para a geração de seqüências de verificação: o **GGCS**, voltado para computadores com apenas 1 processador, e o **GGCS Distributed**, para computadores com mais de 1 processador ou para *clusters*.

Foram apresentados estudos experimentais para avaliar o comportamento dos aplicativos desenvolvidos. No primeiro estudo experimental, foi identificado que o método proposto pode gerar seqüências de verificação consideravelmente menores que as seqüências geradas pelo método proposto por Simão e Petrenko (2008). No segundo estudo experimental foi constatada a influência das condições de suficiência utilizadas pelo método na geração das seqüências de verificação. Por fim, no terceiro estudo experimental foram demonstradas as vantagens da utilização do **GGCS Distributed** diante do **GGCS**. Além do **GGCS** e do **GGCS Distributed**, também foram implementadas as ferramentas **GDS**, **CheSCon** e **CheSCon Viz**.

5.2 Limitações

Conforme descrito nas Seções 4.2 e 4.3, o algoritmo proposto neste trabalho (implementado no aplicativo **GGCS**) não foi capaz de gerar seqüências de verificação significativamente menores que o método proposto por Simão e Petrenko (2008) para MEFs com uma quantidade elevada de estados. Para o caso de MEFs com 20 estados, 4 entradas e 4 saídas, as seqüências de verificação geradas pelo **GGCS** foi apenas 3,5% menores que as seqüências geradas pelo método de Simão e Petrenko (2008). Dessa maneira, o algoritmo proposto neste trabalho tem a sua eficiência em relação ao tamanho das seqüências de verificação geradas comprometida à medida que a quantidade de estados da MEF aumenta.

Outra limitação do algoritmo está associada ao tempo de execução do aplicativo. Considerando um computador equipado com um processador Intel Pentium 4 CPU 3.40 GHz, 3 GB de memória RAM, sistema operacional Linux e *Java Runtime Environment* (JRE) versão 1.6, o tempo de execução do **GGCS** para a geração de uma seqüência de verificação para uma MEF de 20 estados, 4 entradas e 4 saídas foi, em média, 24 minutos. Assim, torna-se inviável a utilização do método para MEFs com uma quantidade elevadas de estados ou entradas. Porém, essa limitação é contornada a partir da utilização do

GGCS Distributed, que pode ser utilizado em *clusters* com o objetivo de diminuir o tempo de execução do aplicativo.

5.3 Trabalhos Futuros

A indisponibilidade ou a limitação de algumas implementações inviabilizou realizar a comparação direta do método proposto com outros métodos de geração de seqüências de verificação, tais como os métodos propostos por Chen et al. (2005) e Hierons e Ural (2006), cujas implementações existentes não suportam o uso de conjuntos de distinção. Desse modo, os experimentos apresentados nas Seções 4.2 e 4.3 se concentraram a comparar o método proposto neste trabalho com o método proposto por Simão e Petrenko (2008). Porém, Simão e Petrenko (2008) demonstraram que o método proposto por eles geraram seqüências de verificação menores que os métodos propostos por Chen et al. (2005) e Hierons e Ural (2006). Porém, novos experimentos comparando o método proposto neste trabalho de mestrado com os outros métodos de geração de seqüências de verificação podem ser realizados em uma atividade futura, tendo que, para isso, obter as implementações de tais métodos.

Como trabalho futuro, pode-se investigar novos operadores de mutação, avaliando a eficácia de cada operador, a partir de outros trabalhos dessa área. Outra investigação a ser realizada está relacionada à geração da população inicial, com o objetivo de gerar cromossomos mais aptos desde o início do algoritmo. Essa otimização pode ser realizada a partir da implementação de heurísticas definidas por outros pesquisadores ou a partir da análise dos cromossomos mais aptos gerados pelo método proposto, descobrindo algum padrão entre eles e implementando esse padrão nos cromossomos da população inicial. A implementação **GGCS** e **GGCS Distributed** também podem ser aprimoradas com a finalidade de diminuir o tempo de execução. Essa diminuição do tempo de execução pode ser obtida com a otimização da população inicial, uma vez que o número de iterações do algoritmo seria reduzido, ou com a implementação de estruturas de dados mais eficientes.

A limitação referente às MEFs que possuem uma quantidade elevada de estados e entradas também deve ser considerada em uma atividade futura. Espera-se solucionar a limitação do **GGCS** para que as seqüências de verificação geradas para MEFs maiores possuam uma taxa de redução considerável quando comparado às seqüências de verificação geradas por outros métodos. Essa melhoria também pode ser obtida com a implementação de novas heurísticas e de novos operadores de mutação.

Referências

Aho, A. V.; Dahbura, A. T.; Lee, D.; Uyar, M. U. An optimization technique for protocol conformance test generation based on UIO sequences and rural chinese postman tours. *IEEE Transactions on Communications*, v. 39, n. 11, p. 1604–1615, 1991. 30

Boute, R. Distinguishing sets for optimal state identification in checking experiments. *IEEE Transactions on Software Engineering*, v. 23, n. 8, p. 874–877, 1974. 2, 3, 16, 22

Chen, J.; Hierons, R. M.; Ural, H.; Yenigun, H. Eliminating redundant tests in a checking sequence. In: *TestCom 2005*, n. 3502 in Lecture Notes on Computer Science, 2005, p. 146–158 (*Lecture Notes on Computer Science*,). v, vii, 3, 4, 20, 21, 23, 26, 27, 28, 32, 33, 36, 37, 39, 71, 73

Chow, T. S. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, v. 4, n. 3, p. 178–187, 1978. 2, 17, 18, 19

Davis, A. M. A comparison of techniques for the specification of external system behavior. *Communications of the ACM*, v. 31, n. 9, 1988. 13, 14

Dorofeeva, R.; El-Fakih, K.; Maag, S.; Cavalli, A. R.; Yevtushenko, N. Experimental evaluation of fsm-based testing methods. In: *Third IEEE International Conference on Software Engineering and Formal Methods*, 2005a, p. 23–32. vii, 24, 58

Dorofeeva, R.; El-Fakih, K.; Yevtushenko, N. An improved conformance testing method. In: *FORTE*, 2005b, p. 204–218. 2, 17, 20

Fujiwara, S.; Bochman, G. V.; Khendek, F.; Amalou, M.; Ghedamsi, A. Test selection based on finite state models. *IEEE Transactions on Software Engineering*, v. 17, n. 6, p. 591–603, 1991. 2, 17, 19

Gill, A. *Introduction to the theory of finite-state machines.* New York: McGraw-Hill, 1962. 13, 18

Gonenc, G. A method for the design of fault detection experiments. *IEEE Transactions on Computers*, v. 19, n. 6, p. 551–558, 1970. v, vii, 2, 3, 13, 23, 26, 27, 30, 32, 36, 37, 39, 40, 41, 42

Hennie, F. C. Fault-detecting experiments for sequential circuits. In: *Proceedings of Fifth Annual Symposium on Circuit Theory and Logical Design*, 1964, p. 95–110. 2, 3, 17, 22, 30, 39

Hierons, R. M.; Ural, H. Reduced length checking sequences. *IEEE Transactions on Computers*, v. 51, n. 9, p. 1111–1117, 2002. v, 3, 4, 26, 27, 28, 29, 31, 39

Hierons, R. M.; Ural, H. Optimizing the length of checking sequences. *IEEE Transactions on Computers*, v. 55, n. 5, p. 618–629, 2006. v, 3, 4, 17, 20, 21, 23, 26, 27, 28, 29, 30, 32, 33, 36, 37, 39, 58, 71, 73

Kohavi, I.; Kohavi, Z. Variable-length distinguishing sequences and their application to the design of fault-detection experiments. *IEEE Transactions on Computers*, v. 17, n. 8, p. 792–795, 1968. 22

Lee, D.; Yannakakis, M. Testing finite-state machines: State identification and verification. *IEEE Trans. Comput.*, v. 43, n. 3, p. 306–320, 1994. 16, 49, 50

Luo, G.; Petrenko, A.; v. Bochmann, G. *Selecting test sequences for partially-specified nondeterministic finite state machines.* Relatório Técnico, Department d'IRO, Université de Montreál, 1994. 17

Maldonado, J. C. *Critérios potenciais usos: Uma contribuição ao teste estrutural de software.* Tese de doutoramento, Campinas, SP, 1991. 12

Naito, S.; Tsunoyama, M. Fault detection for sequential machines by transition tours. In: *Proceedings of the 11th IEEE Fault Tolerant Computing Conference (FTCS 1981)*, IEEE Computer Society Press, 1981, p. 238–243. 15

Neto, L. F. M.; Simão, A. S. Minimização de conjuntos de casos de teste por meio de condições de suficiência. In: *Primeiro Workshop Brasileiro de Teste Sistemático e Automatizado*, João Pessoa, PB, 2007, p. 1–8. 20

Petrenko, A.; Yevtushenko, N. Testing from partial deterministic fsm specifications. *IEEE Transactions on Computers*, v. 54, n. 9, p. 1154–1165, 2005. 2, 14, 19

Petrenko, A.; Yevtushenko, N.; Lebedev, A.; Das, A. Nondeterministic state machines in protocol conformance testing. In: *Protocol Test Systems*, 1993, p. 363–378. 2, 17, 19

Pressman, R. S. *Software engineering — a practitioner's approach.* 6 ed. McGraw-Hill, 2005. 8, 11, 12

Sabnani, K. K.; Dahbura, A. A protocol test generation procedure. *Computer NetWorks and ISDN Systems*, v. 15, n. 4, p. 285–297, 1988. 2, 30

Sidhu, D. P.; Leung, T. K. Formal methods for protocol testing: A detailed study. *IEEE Transactions on Software Engineering*, v. 15, n. 4, p. 413–426, 1989. 18

Simão, A. S. Teste baseado em modelos. In: *Introdução ao Teste de Software*, cáp. 3, Rio de Janeiro, Brasil: Campus, p. 27–45, 2007. 15

Simão, A. S.; Petrenko, A. Generating checking sequences for partial reduced finite state machine. In: *20th IFIP Int. Conference on Testing of Communicating Systems (TESTCOM 2008)*, Tokyo, Japan, 2008. v, 3, 4, 22, 23, 31, 32, 33, 36, 37, 47, 53, 54, 57, 58, 71, 72, 73

Simão, A. S.; Petrenko, A. Checking sequence generation using state distinguishing subsequences. In: *ICSTW '09: Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation Workshops*, Washington, DC, USA: IEEE Computer Society, 2009, p. 48–56. 16

Simão, A. S.; Petrenko, A. Checking completeness of tests for finite state machines. *IEEE Transactions on Computers*, v. 99, n. PrePrints, 2010. vii, 3, 4, 5, 20, 21, 33, 36, 37, 39, 40, 42, 45, 49, 50, 57, 58, 62, 64, 65, 68, 71, 72

Simão, A. S.; Petrenko, A.; Maldonado, J. C. Experimental evaluation of coverage criteria for fsm-based testing. In: *XXI Simpósio Brasileiro de Engenharia de Software*, João Pessoa, PB, 2007, p. 1–16. 58

Ural, H.; Wu, X.; Zhang, F. On minimizing the lengths of checking sequences. *IEEE Transactions on Computers*, v. 46, n. 1, p. 93–99, 1997. v, vii, 3, 4, 17, 19, 20, 21, 23, 26, 27, 30, 32, 33, 36, 37, 39, 57, 62, 64, 65, 68

Ural, H.; Zhang, F. Reducing the lengths of checking sequences by overlapping. *Lecture Notes on Computer Science*, , n. 3964, p. 274–288, 2006. 3, 4, 26, 31, 32, 39, 42

Vuong, S. T.; Chan, W. W. L.; Ito, M. R. The uiov-method for protocol test sequence generation. In: *Proc. of the IFIP TC6 2nd IWPTS*, North-Holland, 1989, p. 161–175. 2

Yalcin, M. C.; Yenigun, H. Using distinguishing and uio sequences together in a checking sequence. In: *TestCom 2006*, n. 3964 in Lecture Notes on Computer Science, 2006, p. 274–288 (*Lecture Notes on Computer Science*,). v, vii, 3, 20, 21, 23, 26, 30, 31, 32, 33, 37