UNIVERSIDADE DE SÃO PAULO
Instituto de Ciências Matemáticas e de Computação

Otimização bioinspirada para apoio à geração de dados de teste para software concorrente

Ricardo Ferreira Vilela

Tese de Doutorado do Programa de Pós-Graduação em Ciências de Computação e Matemática Computacional (PPG-CCMC)

ICMC USP SÃO CARLOS

Ricardo Ferreira Vilela

Otimização bioinspirada para apoio à geração de dados de teste para software concorrente

Tese apresentada ao Instituto de Ciências Matemáticas e de Computação – ICMC-USP, como parte dos requisitos para obtenção do título de Doutor em Ciências – Ciências de Computação e Matemática Computacional. *VERSÃO REVISADA*

Área de Concentração: Ciências de Computação e Matemática Computacional

Orientadora: Profa. Dra. Simone do Rocio Senger de Souza

USP – São Carlos
Agosto de 2021

Ficha catalográfica elaborada pela Biblioteca Prof. Achille Bassi
e Seção Técnica de Informática, ICMC/USP,
com os dados inseridos pelo(a) autor(a)

Vilela, Ricardo Ferreira
V699o Otimização bioinspirada para apoio à geração de dados de teste para software concorrente / Ricardo Ferreira Vilela; orientadora Simone do Rocio Senger Souza. -- São Carlos, 2021.
129 p.

Tese (Doutorado - Programa de Pós-Graduação em Ciências de Computação e Matemática Computacional) -- Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, 2021.

1. Teste de Software. 2. Programas Concorrentes. 3. Geração de dados de teste. 4. Otimização bioinspirada. 5. Algoritmo Genético . I. Souza, Simone do Rocio Senger, orient. II. Título.

Bibliotecários responsáveis pela estrutura de catalogação da publicação de acordo com a AACR2:
Gláucia Maria Saia Cristianini - CRB - 8/4938
Juliana de Souza Moraes - CRB - 8/6176

Ricardo Ferreira Vilela

Bio-inspired optimization to support the test data generation for concurrent software

Doctoral dissertation submitted to the Institute of Mathematics and Computer Sciences – ICMC-USP, in partial fulfillment of the requirements for the degree of the Doctorate Program in Computer Science and Computational Mathematics. *FINAL VERSION*

Concentration Area: Computer Science and Computational Mathematics

Advisor: Profa. Dra. Simone do Rocio Senger de Souza

USP – São Carlos
August 2021

AGRADECIMENTOS

À Deus, primeiramente, pela sua presença constante na minha vida, pelo apoio nas minhas escolhas, por estar sempre comigo nos momentos mais difíceis e por fazer com que mais esse sonho se realize. Agradeço ainda por ter colocado em meu caminho pessoas tão especiais, que me ajudaram durante a realização deste Doutorado.

À Universidade de São Paulo (USP) e ao Instituto de Ciências Matemáticas e de Computação (ICMC), pela oportunidade concedida para realização do doutorado.

Ao Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) pela concessão da bolsa de estudos. À Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) pelo apoio financeiro para condução da pesquisa.

À minha orientadora Prof. Simone do Rocio Senger de Souza, por toda paciência, dedicação e pela postura exemplar como pessoa e pesquisadora. Agradeço também pela preocupação comigo e com meus familiares em todas as situações, pela amizade que construímos ao longo desses sete anos e por ser essa pessoa incrível que eu tenho orgulho de chamar de orientadora.

Ao Prof. Paulo Sérgio Lopes de Souza, por estar sempre disponível para ensinar, ajudar, conversar e comemorar as minhas vitórias. Agradeço também pela amizade durante todos esses anos e pelos ensinamentos que contribuíram para o meu crescimento pessoal e profissional.

Aos meus Pais Cairo Viana Vilela e Maria Aparecida Ferreira Vilela por serem a minha referência de honestidade, persistência, carinho e amor. Agradeço por me ajudarem a realizar cada um dos meus sonhos e por se alegrarem comigo em cada conquista. Se hoje estou podendo realizar este grande sonho, vocês são os responsáveis por essa conquista.

Um agradecimento especial aos meus irmãos Marcos Vinícius e Laíze Vilela por acreditarem na minha capacidade, pelo apoio constante, pelos conselhos e pelo exemplo como pesquisadores. Tenho muito orgulho de seguir os passos de vocês e ser o terceiro a receber um título de Doutor.

Aos meus amigos do LabES agradeço pela força, apoio, por estarem sempre presentes, sem vocês o caminho teria sido muito mais difícil.

Aos professores do Instituto de Ciências Matemáticas e de Computação, meu agradecimento por todas as oportunidades de crescimento e aprendizagem e pelos ensinamentos sempre valiosos.

Ao Roger Levy Vigula Boy pelos longos anos de amizade e companheirismo. Agradeço também por toda a ajuda que concedeu durante a execução dos experimentos, permitindo que eu utilizasse recursos que foram essenciais para condução deste trabalho.

Aos membros da banca pelas valiosas contribuições com este trabalho, e pela disponibilidade em aceitar participar da minha banca de defesa.

A todos aqueles que, de uma forma ou de outra, contribuíram para a realização deste trabalho.

A todos, muito obrigado!

"O sábio não se senta para lamentar-se,
mas se põe alegremente em sua
tarefa de consertar o dano feito."
(William Shakespeare)

RESUMO

VILELA, R. F. **Otimização bioinspirada para apoio à geração de dados de teste para software concorrente**. 2021. 129 p. Tese (Doutorado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos – SP, 2021.

A programação concorrente está cada vez mais presente nas aplicações modernas. Embora esse modelo de programação forneça maior desempenho e melhor aproveitamento dos recursos disponíveis, os mecanismos de interação entre processos/threads resultam em maior desafio para atividade de teste. O não determinismo presente nessas aplicações é um dos principais desafios na atividade de teste, uma vez que ainda com uma mesma entrada de teste o programa concorrente pode executar caminhos distintos, os quais podem ou não apresentar defeitos. A geração automática de dados de teste pode contribuir para essa atividade garantindo maior rapidez e confiabilidade no teste de software. Neste trabalho, a geração automática de dados de teste é explorada para o domínio de programas concorrentes por meio de uma técnica bioinspiradas de otimização, o Algoritmo Genético. Este estudo propõe uma abordagem de geração de dados para programas concorrentes denominada BioConcST. Além disso, propõe-se um novo operador de seleção de indivíduos de teste utilizando lógica fuzzy, denominado FuzzyST. Essas contribuições são avaliadas em um estudo experimental utilizado para validar as abordagens propostas. Os resultados obtidos do experimento demonstraram que a BioConcST é mais promissora que as demais abordagens utilizadas em todos os níveis analisados. Além disso, o operador FuzzyST também obteve os melhores resultados juntamente com os operadores Elitismo e Torneio. Contudo, o operador FuzzyST mostrou-se mais indicado para programas concorrentes de maior complexidade.

Palavras-chave: Programas concorrentes, Teste de software, Geração de dados de teste, Otimização bioinspirada, Algoritmo Genético.

ABSTRACT

VILELA, R. F. **Bio-inspired optimization to support the test data generation for concurrent software**. 2021. 129 p. Tese (Doutorado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos – SP, 2021.

Concurrent programming is increasingly present in modern applications. Although this programming model provides greater performance and better use of available resources, the mechanisms of interaction between processes/threads result in a greater challenge for software testing activity. The non-determinism present in these applications is one of the main challenges in the test activity since even with the same test input, the concurrent program can execute different paths, which may or may not present defects. The automatic generation of test data can contribute to this activity, ensuring greater speed and reliability in software testing. In this work, the automatic test data generation is explored for the domain of concurrent programs through a bioinspired optimization technique, the Genetic Algorithm. We propose a test data generation approach for concurrent programs called BioConcST. Also, we propose a new operator for selecting test subjects using fuzzy logic, called FuzzyST. We evaluated these approaches in an experimental study to validate. The results obtained from the experiment showed that BioConcST is more promising than the other approaches used at all levels analyzed. The operator FuzzyST also obtained the best results, together with the Elitism and Tournament operators. Nevertheless, the FuzzyST operator proved to be more suitable for concurrent programs of greater complexity.

Keywords: Concurrent programs, Software testing, Test data generation, Bio-inspired optimization, Genetic Algorithm.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

SUMÁRIO

CAPÍTULO

1

INTRODUÇÃO

1.1 Contextualização e Motivação

Embora a maior parte das pessoas não saiba, todos os usuários de computadores e *smartphones* atuais utilizam diariamente software concorrente. Os sistemas operacionais (SOs), inclusos nesses dispositivos, utilizam mecanismos da programação concorrente para gerenciar o escalonamento e a execução dos processos, seja em ambientes monoprocessados, com o pseudoparalelismo, ou multiprocessados, onde há de fato a execução paralela entre processos.

Até onde se sabe, a primeira contribuição da programação concorrente para Sistemas Operacionais foi proposta por Dijkstra (1965), que tinha como objetivo identificar e solucionar a exclusão mútua para impedir o acesso simultâneo a um recurso compartilhado, também denominado por região crítica. Esse e demais mecanismos da programação concorrente permitiram aos SOs otimizar a utilização dos recursos computacionais e, consequentemente, melhorar o seu desempenho.

Nos dias atuais, a programação concorrente não mais se restringe aos SOs, uma vez que as aplicações modernas também demandam melhor aproveitamento dos recursos e maior desempenho na execução de rotinas e tarefas. Como exemplo, a programação paralela e programação distribuída, ramificações da programação concorrente, proporcionaram o surgimento de novas tecnologias amplamente utilizadas como *Web services* e *Cloud computing* que empregam conceitos da programação concorrente para prover mecanismos de comunicação e otimização de serviços (MARINESCU, 2018; YUAN; YANG, 2020).

Existe uma variedade de elementos necessários para o desenvolvimento de programas concorrentes, todavia, podem-se citar três primitivas básicas que permeiam a construção de todas as aplicações desse domínio: 1) a primeira etapa do processo é a definição

de quais tarefas serão executadas de forma concorrente (ou paralela). Essa etapa deve analisar minunciosamente as dependências entre as tarefas para que defeitos não sejam inseridos em função de atribuições inapropriadas; 2) Após definidas na etapa anterior, essas tarefas serão abstraídas em processos (ou *threads*), para os quais serão definidas estratégias de inicialização e finalização de processos concorrentes; e 3) por último, estabelece-se a coordenação, em relação à ordem de execução, entre os processos concorrentes enquanto estes estiverem executando (ALMASI; GOTTLIEB, 1994; WONG; LEI; MA, 2005).

Em adição as primitivas, mecanismos de interação permitem a comunicação e a sincronização entre os processos/*threads* que são fundamentais para condução das etapas citadas. Essa interação entre processos/*threads* ocorre por meio de dois paradigmas distintos, os quais se distinguem, principalmente, em função da organização de memória disponível (GRAMA *et al.*, 2003).

O mecanismo de memória compartilhada, como o próprio nome sugere, considera um mesmo espaço de endereçamento para comunicação, no qual processos concorrentes podem compartilhar variáveis de memória. Nesse cenário, a sincronização é usualmente realizada por meio da construção de semáforos ou monitores (TANENBAUM, 2007).

Por sua vez, o mecanismo de comunicação por passagem de mensagem considera a presença de memória distribuída, onde processos concorrentes possuem acesso apenas ao seu próprio espaço de endereçamento, isto é, não há o compartilhamento de variáveis. Nesse caso, quando há necessidade de interação entre processos, essa só poderá ser realizada por meio da troca de mensagens. Existem diferentes primitivas que abstraem a troca mensagem entre processos. Neste estudo considera-se a interação por meio da primitiva ***Send*/*Receive***, que sugere o envio e o recebimento de mensagens, respectivamente.

Vale ressaltar, que as aplicações concorrentes também podem utilizar modelos híbridos de interação entre processos, empregando ambos os tipos de paradigmas. Esse cenário pode ocorrer com *threads* iniciadas em um mesmo processador, quando há a presença de duas ou mais unidades de processamento (CPUs), ou com o uso de plataformas heterogêneas que incluam máquinas com memória compartilhada e memória distribuída.

Diante do contexto apresentado, percebe-se que as etapas que envolvem o processo de desenvolvimento de programas concorrentes diferem-se das aplicações sequenciais. Como exemplo, a atividade de teste, no contexto de software concorrente, apresenta novos desafios em função das características e dos tipos de defeitos dessas aplicações. O esforço em identificar um defeito nesse tipo de programa está associado, principalmente, a execução não-determinística que em outras palavras significa que um programa pode percorrer caminhos diferentes em múltiplas execuções ainda que o mesmo conjunto de entradas seja utilizado em todas as execuções.

O não-determinismo, por si só, não constitui-se propriamente em um defeito, con-

tudo, essa característica acrescenta maior complexidade e custo na atividade de teste. Essa particularidade ocorre devido à imprevisibilidade na ordem e tempo de execução entre *threads* ou processos, pois assim como os demais programas em execução, um processo concorrente disputa por recursos e não há como prever quando e por quanto tempo esses recursos estarão disponíveis.

A diferença nesse cenário é que os programas sequencias podem sofrer preempções e continuar posteriormente sua execução sem que haja conflitos, já os programas concorrentes utilizam memórias compartilhadas que podem ser declaradas erroneamente durante uma preempção ou dependem da comunicação com outros processos que pode não ocorrer ou ser estabelecida de modo incorreto. Cabe a atividade de teste, nesse cenário, identificar se todas as interações possíveis foram executadas e se as saídas obtidas estão corretas (SOUZA *et al.*, 2015).

Em razão dos desafios impostos por essas aplicações, a condução manual da atividade de teste pode ser impraticável considerando o grande número de requisitos de teste gerados durante a verificação e o alto custo necessário para testa-los. Nessa perspectiva, a geração automática de dados de teste é vista como grande aliada durante a verificação, diminuindo significativamente o esforço necessário e otimizando o processo de seleção de entradas de teste (MAIRHOFER; FELDT; TORKAR, 2011; SCALABRINO *et al.*, 2016).

Embora a automatização do processo de seleção de entradas (ou geração automática de dados de teste) seja útil para testar aplicações de domínio concorrente, essa tarefa é classificada como um problema indecidível, mesmo no domínio de aplicações sequenciais, uma vez que não existe um algoritmo de propósito geral que satisfaça um determinado critério de teste e nem mesmo é possível saber se existe um conjunto de teste capaz de satisfazer esse critério (ALEB; KECHID, 2013; DELAMARO; MALDONADO; JINO, 2016). Ainda assim, há uma grande concentração de estudos na literatura que têm em vista a otimização da seleção automática de dados de teste, principalmente para programas sequenciais.

Uma das técnicas mais simples e baratas para geração automática de dados de teste é a geração aleatória. Essa técnica seleciona aleatoriamente locais específicos do domínio de entrada até que um dado critério de teste seja satisfeito. Apesar disso, não há garantias que requisitos importantes de testes serão executados com o uso dessa técnica (MCMINN, 2004; NIKRAVAN; PARSA, 2019). Por sua vez, a técnica de geração automática de dados de teste com execução simbólica, utilizada juntamente com um critério de teste baseado em caminhos, deriva expressões simbólicas a partir de variáveis de entrada do programa abstraindo os possíveis caminhos de execução em expressões algébricas. Desse modo, o problema da geração de dados de teste é transformado em um problema de resolução de expressões algébricas. Embora a natureza do problema seja de ordem mais simples, a presença de laços indefinidos, ponteiros e uma grande quantidade de desvios funcionais

pode inviabilizar o uso dessa técnica em função do custo e limitação para definição de expressões algébricas (BOYER; ELSPAS; LEVITT, 1975; GUO; RUBIO-GONZáLEZ, 2020).

A técnica de geração baseada em busca (*search-based*) tem sido uma grande aliada para a geração de dados de testes. Essa técnica transforma o processo de seleção de dados de teste em um problema de busca, onde meta-heurísticas são empregadas para identificar melhores soluções no domínio de entrada das aplicações sob teste (McMinn, 2011). Nesse contexto, existe uma diversidade de meta-heurísticas disponíveis para tratar o problema de otimização, ainda assim, pode-se destacar as meta-heurísticas bioinspiradas que têm sido frequentemente usadas em estudos que abordam a geração de dados de teste, como é o caso do Algoritmo Genético (AG) (KHARI; KUMAR, 2017).

A investigação da geração automática no contexto de software concorrente ainda é recente, os trabalhos existentes (NISTOR *et al.*, 2012; TIAN; GONG, 2013; TIAN; GONG, 2014; TIAN; GONG, 2016; MIRHOSSEINI; HAGHIGHI, 2020) empenham-se em mapear as técnicas consolidadas de aplicações sequenciais para o domínio de aplicações concorrentes, contudo, ainda existem lacunas que precisam ser exploradas para potencializar a geração automática de dados de teste nesse domínio.

Com o intuito de investigar as lacunas e contribuições das técnicas consolidadas para programas sequenciais, um estudo prévio experimental foi realizado em busca de evidências sobre a efetividade, eficácia e o custo dessas técnicas quando aplicadas em programas concorrentes (VILELA, 2016). De modo geral, as técnicas apresentaram um alto custo computacional para alcançar os objetivos de teste das aplicações. Apesar disso, a maior deficiência das técnicas foi em relação à eficácia em revelar defeitos e a adequação dos critérios concorrentes. Em todos os cenários observados a técnica aleatória (*baseline*) foi superior às demais técnicas investigadas. Em vista disso, observou-se que os mecanismos de interação concorrente e os desafios intrínsecos dessas aplicações demandavam maior esforço na tarefa de geração de dados de teste.

No estudo proposto por Khanna, Purandare e Sharma (2020) são apresentadas novas teorias sobre a otimização desse problema que demonstram a importância da representação do espaço de busca considerando não apenas o universo de entradas do programa, como ocorre nos programas sequenciais, mas também as possíveis interações entre processos/*threads* e critérios de teste específicos de comunicação. Ainda assim, no que diz respeito ao processo de geração de dados de teste, o estudo limitou-se ao universo de programas com memória compartilhada para tratar problemas de *deadlocks*[1].

Em um estudo preliminar desta tese, propôs-se uma abordagem bioinspirada para

[1] Um *deadlock* ocorre quando um conjunto de processos estiver esperando por um evento que somente um outro processo do mesmo conjunto pode desempenha, portanto, todos ficam bloqueados indefinidamente

geração de dados de teste baseada em critérios de teste para programas concorrentes com memória compartilhada e passagem de mensagem (VILELA *et al.*, 2019). A premissa dessa abordagem considera a permanência de indivíduos (dados de teste) ruins, em relação ao valor de *fitness*, como estratégia para manter a população (conjunto de teste) diversificada e alcançar melhores resultados. Os resultados foram satisfatórios demonstrando avanços significativos em relação à cobertura de critérios de teste concorrente.

No entanto, observou-se três limitações principais neste estudo preliminar que foram a base de investigação para as principais contribuições desta tese de doutorado. A primeira limitação refere-se ao artefato de saída da abordagem, esse artefato é constituído apenas pela entrada de teste do programa concorrente, desse modo, não é possível garantir que uma reexecução do programa, sob o mesmo dado de entrada, irá alcançar os mesmos objetivos de teste.

A segunda limitação ocorre na atribuição do valor de *fitness* para as entradas de teste, na qual os indivíduos são submetidos à uma avaliação binária de cobertura, ou seja, se um indivíduo alcançou um determinado requisito de teste, o mesmo é considerado apto para futuras gerações, caso contrário, todos os indivíduos que não alcançaram um objetivo de teste são vistos igualmente inaptos para passar informações genéticas para futuras gerações. Essa característica faz com que informações relevantes dos testes que não alcançaram um determinado objetivo sejam perdidas, além disso, pode contribuir para uma população menos diversificada.

Por último, a terceira limitação envolve o processo de seleção de indivíduos que determina o processo de transferência de informações genéticas para gerações posteriores. Embora a manutenção do pior indivíduo da população contribua para uma diversidade na população, essa estratégia não garante que o pior indivíduo possui realmente informações relevantes e singulares que possam contribuir para futuras gerações.

Considerando os resultados obtidos e os demais relatos na literatura, nota-se que a geração de dados de teste no domínio de software concorrente ainda apresenta lacunas no processo de otimização. Até onde se sabe, este é o primeiro estudo que investiga essa tarefa para ambos os paradigmas de comunicação de programas concorrentes por meio de uma estratégia bioinspirada, fornecendo meios para reexecução do dado de teste nas mesmas condições sob o qual foi gerado, estabelecendo uma estratégia de diferenciação de indivíduos que não alcançaram um mesmo objetivo de teste e determinando atributos de lógica multivalorada como estratégia de seleção de indivíduos.

1.2 Objetivos e Hipótese

Com base no contexto apresentado, destaca-se a importância em definir novas estratégias para a geração de dados de teste no domínio de software concorrente, uma vez

que ainda há um número limitado de trabalhos sob essa perspectiva e ainda existem grandes lacunas que podem permitir contribuições inéditas para o tema de estudo. Ainda vale ressaltar, que esse tipo de aplicação tem se tornado cada vez mais presente nos sistemas modernos e quaisquer contribuições no teste dessas aplicações devem afetar diretamente na qualidade dos produtos de software. Dessa forma, a hipótese principal deste estudo é caracterizada da seguinte forma:

É possível melhorar a geração de dados de teste para programas concorrentes, a partir da diferenciação entre dados de testes que falharam na cobertura de um requisito de teste. Além disso, é possível melhorar o processo genético de seleção considerando não apenas o valor de fitness, mas também a originalidade e o valor alcançado de cobertura global de um dado de teste.

Tendo em vista a motivação e a hipótese descritas, o objetivo geral deste trabalho é apresentar uma abordagem bioinspirada denominada BioConcST, que visa a geração de dados de testes, compostos pelo seu respectivo caminho (rastro) de execução, para apoiar o teste de software concorrente. O objetivo ainda pode ser subdividido nos seguintes objetivos específicos:

1. Definir uma função de *fitness*, baseada na distância entre o objetivo de teste e o caminho percorrido pelo indivíduo, para caracterizar indivíduos que atingiram um requisito de teste e diferenciar indivíduos que falharam;
2. Elaborar e desenvolver um novo operador genético de seleção, denominado FuzzyST, que emprega a Lógica *Fuzzy* para selecionar indivíduos a partir dos valores de *fitness*, originalidade e cobertura global;
3. Automatizar o processo de seleção de dados de testes para programas concorrentes por meio da implementação de um protótipo que reúne as contribuições deste estudo a fim de avaliar o estudo proposto; e
4. Condução de um estudo experimental para avaliação controlada das contribuições propostas por meio do protótipo desenvolvido.

1.3 Organização

Esta tese está disposta em seis capítulos. Neste Capítulo 1 foram apresentadas a Contextualização e Motivação deste estudo. Além disso, são descritos a hipótese, os objetivos geral e específicos deste trabalho. No capítulo 2 são descritos os conceitos teóricos que fundamentam a pesquisa investigada. No Capítulo 3 descreve-se os trabalhos relacionados a este estudo. Em seguida, no Capítulo 4, detalha-se as contribuições propostas

desta tese. No Capítulo 5 descreve-se o *design* experimental da avaliação da abordagem proposta, juntamente com a análise e discussão dos resultados. Por último, no Capítulo 6 são apresentadas as conclusões deste estudo e trabalhos futuros.

CAPÍTULO

2

REFERENCIAL TEÓRICO

2.1 Considerações Iniciais

O desenvolvimento de software concorrente apresenta desafios inerentes à complexidade e a qualidade dos produtos desenvolvidos. Nesse contexto, atividades de verificação e validação (V&V) são relevantes e primordiais para aumentar a garantia de qualidade dos produtos de software desenvolvidos. Essas atividades devem ser empregadas ao longo de todo o processo de desenvolvimento, verificando se o software se comporta conforme esperado (MYERS; SANDLER; BADGETT, 2011; SOUZA *et al.*, 2015).

Este capítulo tem como objetivo apresentar os conceitos primordiais de programas concorrentes. Em especial, são discutidos os elementos que impactam sobre a complexidade na atividade de teste de software. Além disso, também são discutidos os principais conceitos de teste de software, em especial da geração automática de dados de teste, e como estratégias baseadas em busca têm contribuído para automatização da seleção de entradas para programas concorrentes.

O capítulo está disposto da seguinte forma. Na Seção 2.2 são apresentados os principais conceitos sobre as aplicações concorrentes, incluindo os tipos de interações e os mecanismos de proteção de recursos compartilhados. Por sua vez, a Seção 2.3 apresenta conceitos básicos do teste de software seguidos pelas diretrizes do teste de software concorrente. Por último, são apresentados os conceitos da geração automática de dados de teste, incluindo mecanismos de Inteligência Artificial que apoiam a realização dessa tarefa.

2.2 Programas Concorrentes

Tradicionalmente, a programação paralela/concorrente é empregada, principalmente, para a realização de tarefas que demandam computação de alto desempenho, tais

como simulações numéricas e processamento de imagens. Contudo, com o surgimento de processadores de múltiplos núcleos em computadores pessoais, dispositivos móveis e *clusters* de computadores, o crescimento do grau de paralelismo disponível proporcionou novas perspectivas no desenvolvimento de software concorrente (BERKA; HAGENAUER; VAJTERSIC, 2011).

De modo geral, um programa é definido pelo conjunto de instruções para realização de uma ou mais tarefas que podem ser executadas em um computador. Quando ocorre de fato a execução desse programa em um computador, essa execução desse programa é abstraída em um Processo. O processo mantém todas as informações necessárias para execução do programa, como dados de registradores, especificações de limites, características dos recursos que serão utilizados e uma região de memória onde instruções e dados são armazenados (MACHADO; MAIA, 2013).

Processos são ditos concorrentes quando dois ou mais processos iniciaram suas execuções, mas ainda não finalizaram. Esses processos disputam por recursos do sistema, tais como processadores, memórias e rotinas de Entrada/Saída. Quando dois ou mais processos estão sendo executados em processadores diferentes, ao mesmo tempo, esses são ditos paralelos. Por outro lado, quando dois ou mais processos são executados em apenas um processador, são considerados concorrentes. De fato, a definição de processos concorrentes é mais abrangente do que o de paralelismo e, portanto, todos os programas paralelos são concorrentes, mas nem todos os programas concorrentes são paralelos. Em outras palavras, o paralelismo é um tipo de concorrência (NICHOLS; BUTTLAR; FARRELL, 1996).

Uma das principais diferenças entre um programa sequencial e um programa concorrente é que o último envolve processos concorrentes que interagem entre si para realização de tarefas, enquanto o primeiro executa um conjunto de instruções sequencialmente. A interação nesses programas pode ocorrer de forma síncrona ou assíncrona. A execução é dita síncrona quando, em um determinado momento, um processo aguarda pelo outro para continuar sua execução. Na execução assíncrona, após a inicialização, os processos serão executados de maneira independente, retornando o resultado da computação ao término da execução. Ainda vale ressaltar, que esses processos também podem ou não concorrer pelos mesmos recursos computacionais.

Os processos concorrentes são usados em diferentes modelos de computação, tais como computação paralela, sistemas distribuídos e sistemas operacionais. Quando os processos concorrentes são usados apenas no domínio de sistemas distribuídos, visando, por exemplo, o compartilhamento de dados e recursos, tais processos são ditos distribuídos.

O berço dos processos concorrentes, como conhecidos atualmente, é o surgimento dos sistemas operacionais multiprogramados (HANSEN, 2002). Posteriormente as mesmas técnicas, antes usadas apenas para a construção dos programas concorrentes em sistemas operacionais, foram utilizadas na computação paralela e nos sistemas distribuídos. As

tarefas de uma aplicação concorrente são codificadas em uma linguagem específica para programação concorrente ou em ambientes instanciados por linguagens tradicionais, essas tarefas são atribuídas a processos ou *threads*.[1] As tarefas de uma aplicação podem ser independentes, mas também podem depender uma da outra, resultando em dependências de dados ou de controle de execução, ou seja, se uma tarefa necessita de dados ou da execução de uma outra tarefa, a execução desta tarefa só pode iniciar após a conclusão da segunda tarefa ou até que essa envie os dados necessários para execução (RüNGER; RAUBER, 2013).

Para que os processos ou *threads* possam transmitir dados ou organizar a ordem de execução eles dependem de mecanismos de comunicação e sincronização, respectivamente. A comunicação permite que a execução de um processo influencie a execução de outro. Dessa forma, a comunicação pode ocorrer com o uso de variáveis compartilhadas (variáveis que podem ser referenciadas por mais de um processo) ou por passagem de mensagem. A sincronização pode ser vista como um conjunto de restrições na ordenação dos eventos. Dois modelos de comunicação e sincronização mais populares para programas concorrentes são os modelos de passagem de mensagem e memória compartilhada. Cada modelo apresenta características que refletem suposições sobre a máquina subjacente. O modelo de memória compartilhada utiliza a memória para comunicação entre os processos, e normalmente é associada a multiprocessadores de memória compartilhada (Figura 1A). O modelo de passagem de mensagem utiliza mensagens para comunicação entre processos e é tipicamente associado a multicomputadores com memória distribuída (Figura 1B) (LEBLANC; MARKATOS, 1992).

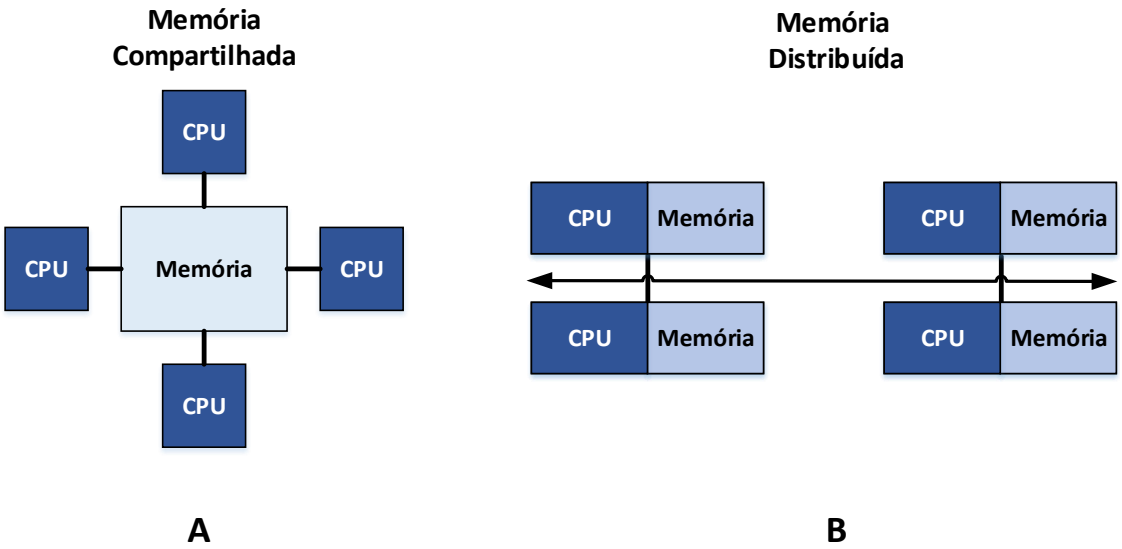
O modelo de memória compartilhada é uma abstração do processador genérico centralizado. O hardware subjacente é assumido como sendo um conjunto de processadores, cada um com acesso à mesma memória compartilhada (Figura 2). Esses processadores possuem acesso aos mesmos locais de memória e, portanto, podem interagir e sincronizar uns com os outros pelo uso de variáveis compartilhadas.

Quando dois ou mais processos estão lendo ou escrevendo algum dado compartilhado, cujo o resultado depende de quem executa precisamente e quando, é dito que existe uma condição de corrida (*race condition*). Nessas situações, a proteção da região crítica é fundamental para impedir que dois processos utilizem, ao mesmo tempo, um só recurso. Garantir a exclusividade ao acesso evita que dados sejam perdidos devido à sobreposição da escrita e/ou utilização de dados antigos. Para garantir a eficácia da sincronização foram desenvolvidos mecanismos como monitores e semáforos.

Um semáforo é um mecanismo de sincronização proposto por Dijkstra (1965). Esse

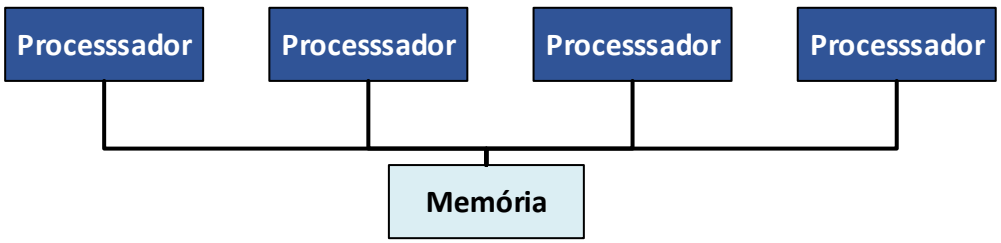
[1] Thread é o fluxo de controle de um processo que permite que múltiplas execuções ocorram no mesmo ambiente do processo com um grau de independência uma da outra (TANENBAUM, 2007).

Figura 1 – Modelo de memória em arquiteturas paralelas



Fonte: Adaptada de Barney (2010).

Figura 2 – Modelo de memória compartilhada de computação paralela

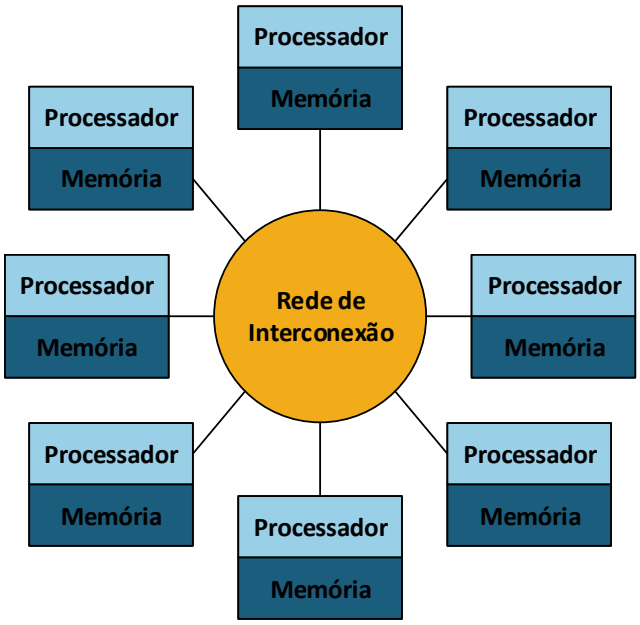


Fonte: Adaptada de Quinn (2003).

mecanismo é composto por uma variável inteira, normalmente iniciada em 1, que é associada a uma lista de processos inicialmente vazia. Dijkstra (1965) propôs a existência de duas operações, *down* e *up*. A operação *down* sobre um semáforo verifica se seu valor é maior que 0, neste caso, o valor é decrementado e o processo continua sua execução. Contudo, se o valor for 0, o processo será alocado como inativo (ou dormindo) antes mesmo de realizar a operação de *down*. A operação *up* incrementa o valor de um semáforo, se um ou mais processos estiverem inativos neste semáforo, um deles é escolhido pelo Sistema Operacional, de acordo com uma ordem de prioridade, para continuar sua execução e realizar a operação de *down* (TANENBAUM, 2007).

O mecanismo de monitores consiste em uma coleção de rotinas, variáveis e estruturas de dados agrupados em um tipo especial de módulo ou pacote. Neste mecanismo, proposto por (HANSEN, 1973) e (HOARE, 1974), a exclusão mútua (ou proteção da região crítica) é dada pela propriedade que diz que apenas um processo pode estar ativo, em um instante de tempo, no monitor. Em outras palavras, se um processo chamar um procedimento do monitor e outro processo estiver ativo dentro do monitor, o primeiro

Figura 3 – Modelo de passagem de mensagens onde cada processador possui seu próprio espaço de endereçamento e uma rede de interconexão de apoio entre os processadores



Fonte: Adaptada de Quinn (2003).

processo será suspenso até que o processo que detém a região crítica deixe o monitor. As primitivas de monitores estão em alto nível, enquanto as primitivas de semáforos são de mais baixo nível. Vale ressaltar, que a ordem das operações fica sob responsabilidade do programador, tornando ainda mais propensa a ocorrência de enganos.

O modelo de memória compartilhada também suporta a paralelização incremental, um processo de transformação de um programa sequencial em um programa paralelo. A capacidade do modelo de memória compartilhada para apoiar paralelização incremental é uma de suas maiores vantagens sobre o modelo de passagem de mensagens. Esse processo permite, a partir da execução do programa sequencial, classificar blocos do programa de acordo com a demanda por recursos, paralelizar cada bloco passível de execução paralela e finalizar quando não houver mais indícios de melhorias no desempenho (QUINN, 2003).

O modelo ou paradigma de passagem de mensagem é composto por duas principais características que definem esse tipo de comunicação. A primeira delas é o espaço de endereçamento particionado, no qual cada processo ou *thread* recebe uma partição lógica da memória (Figura 3). Em segundo, esse paradigma fornece suporte apenas à paralelização explícita, em outras palavras, o paralelismo fica a cargo do programador responsável pelo desenvolvimento dessa aplicação. Essa última característica, provê menor abstração ao programador, entretanto, quando programas são escritos adequadamente, esse aspecto pode tornar-se um grande potencial da aplicação, pois pode resultar em alto nível de desempenho e escalabilidade para um número grande de processos (GRAMA *et al.*, 2003).

Para realizar a comunicação e a sincronização de processos por meio de passagem de mensagem, os programas necessitam implementar primitivas que viabilizem a ordem de execução e a forma como as mensagens serão trocadas. Existem diferentes definições na literatura para os conceitos de síncrono/assíncrono e bloqueante/não-bloqueante. Neste trabalho utiliza-se a notação apresentada por Souza, Vergilio e Souza (2007), a qual é apresentada a seguir.

A troca de mensagens é realizada por meio de primitivas de *send/receive* ou por meio de chamadas de procedimentos remotos. A sintaxe das primitivas é: $send(msg, destino)$ e $receive(msg/origem)$. Em vista disso, a comunicação entre processos pode ser considerada síncrona quando o *send* (ou *receive*) aguarda que o *receive* (ou *send*) seja executado. Para garantir que as mensagens sejam enviadas, esse modelo utiliza confirmações (*ack*) a cada troca de mensagens. Dessa forma, considera-se que toda comunicação síncrona é bloqueante. Por outro lado, na comunicação assíncrona o comando *send* (ou *receive*) não necessita aguardar o comando *receive* (ou *send*) para continuar a sua execução. Para implementar a comunicação assíncrona, algumas bibliotecas oferecem a utilização de *buffers*. Neste contexto, as mensagens são copiadas/lidas do *buffer*, que fornece aos processos uma falsa impressão de sincronismo.

Em relação aos conceitos de bloqueante/não-bloqueante, os mesmos são considerados similares aos de síncrono/assíncrono, se e somente se, não houver *buffer* no transmissor e no receptor da mensagem. Ainda assim, um *send* pode ser considerado bloqueante se garantir que o *buffer* com a mensagem possa ser reutilizado logo após a liberação do *send* sem que a mensagem seja perdida. Para que um *receive* seja bloqueante ele deve esperar até que a mensagem esteja disponível no *buffer*, e somente após o recebimento continuar sua execução. Nestes dois casos, não é possível garantir a sincronização, mas sim assegurar o envio e recebimento de mensagens.

Considerando o contexto apresentado, percebe-se que os programas concorrentes apresentam características adicionais em relação aos programas sequenciais. Embora essas aplicações forneçam muitos benefícios, existe um custo adicional na atividade de teste, pois os defeitos originados das tarefas concorrentes demandam maior esforço para identificação. Na seção seguinte descreve-se os conceitos da atividade de teste, incluindo o teste de software concorrente.

2.3 Teste de Software

2.3.1 Conceitos Básicos

A atividade de teste de software tem como objetivo verificar se as funcionalidades de um sistema estão de acordo com suas especificações, essa atividade atua na detecção

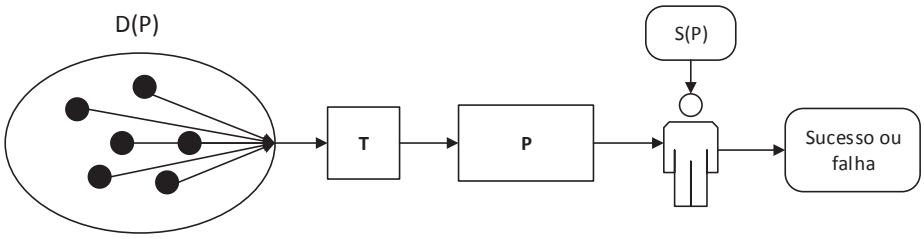
de defeitos, anomalias e informações sobre atributos não funcionais do software. O teste de software consiste em uma atividade dinâmica, pois requer a execução do software ou modelo de forma controlada a fim de observar o seu comportamento. Sendo assim, a principal meta do teste de software é revelar defeitos no programa em análise, em outras palavras, o teste bem sucedido é dado por entradas que resultem na falha do programa sob teste (SOMMERVILLE, 2010).

Para compreensão da atividade de teste alguns conceitos básicos devem ser considerados, como exemplo, o **domínio** de um programa consiste no conjunto de todos os valores que um programa pode assumir como entrada. Por sua vez, cada elemento desse conjunto é chamado de **dado de teste**. Finalmente, a tupla composta pelo dado de teste e sua respectiva saída esperada é denominada por **caso de teste**. Um aspecto importante que deve ser analisado no teste de software são os tipos de problemas, erros ou defeitos que os software apresentam, ou seja, para cada termo existe uma definição diferente, as quais são descritas a seguir:

- Engano (*mistake*): ação incorreta tomada pelo programador;
- Defeito (*fault*): passo, processo ou definição de dados incorreta, incompleta, ausente ou extra que, ao ser executada, pode produzir um erro no programa;
- Erro (*error*): ocorre quando o valor obtido não coincide com o valor esperado;
- Falha (*failure*): evento manifestado em que o sistema viola suas especificações.

Em adição aos termos descritos, a atividade de teste compreende algumas etapas já definidas para sua execução, sendo elas: planejamento, projeto de casos de teste, execução e análise. Este cenário pode ser observado na Figura 4.

Figura 4 – Cenário típico da atividade de teste



Fonte: Adaptada de Delamaro, Maldonado e Jino (2007).

Dado um programa **P**, tal que o seu domínio de entrada seja denotado por **D(P)**, são definidos elementos específicos desse domínio para execução de P. Em um cenário ideal todos os elementos desse domínio seriam selecionados, entretanto, esse tipo de teste,

conhecido como teste exaustivo, na maioria das vezes, é impraticável devido a cardinalidade de D(P). Sendo assim, no planejamento do teste, o conjunto de casos de teste **T** normalmente é composto apenas por elementos com maior probabilidade de revelar defeitos.

Em seguida, o programa é executado com **T**, ao passo que seu resultado obtido é verificado. Quando o resultado produzido pela execução de **P** coincide com o resultado esperado, significa que nenhum erro foi identificado. Por outro lado, se para algum caso de teste, o resultado obtido for diferente do resultado esperado, então o defeito foi revelado. Em geral, essa atividade é realizada por um testador, como sugere a Figura 4, o qual baseado em uma especificação do programa **S(P)** verifica se o resultado obtido é o mesmo resultado esperado. Nesse caso, o testador desempenha o papel de um oráculo de teste, ou seja, aquele instrumento capaz de verificar se a saída obtida de uma determinada execução coincide com a saída esperada.

De acordo com Ammann e Offutt (2008), os testes podem ser derivados a partir de requisitos e especificações do software, artefatos de projeto ou até mesmo do código do programa. Diferentes tipos de teste podem ser aplicados, os quais testam o software em ocasiões diferentes durante o processo de desenvolvimento. Em geral, os seguintes tipos de teste são encontrados:

- **Teste de unidade**: tem como objetivo testar unidades menores de um programa, como funções, procedimentos, métodos e classes;
- **Teste de integração**: é realizado após o teste de unidade e tem como foco o teste na construção de estruturas do sistema. Esse tipo de teste é fundamental para verificar a consistência da interação entre as unidades do sistema;
- **Teste de sistemas**: ocorre após a conclusão do sistema, o objetivo é verificar se as funcionalidades do sistema foram implementadas corretamente e se estão de acordo com a especificação do programa;
- **Teste de regressão**: é aplicado após o desenvolvimento do software na fase de manutenção. Esse tipo de teste é utilizado devido aos erros que podem ser inseridos no programa a cada nova versão implementada.

Independentemente da etapa de teste, diferentes técnicas podem ser empregadas para apoiar a seleção de casos de teste. A diferença entre essas técnicas está na informação utilizada por cada uma para estabelecer os requisitos de teste. Um requisito de teste é uma parte do software ou especificação que um caso de teste deve satisfazer ou cobrir (AMMANN; OFFUTT, 2008). As técnicas de teste mais consolidadas são as técnicas funcional, baseada em erros e estrutural.

A técnica funcional normalmente é utilizada para projetar casos de teste quando o código do programa é desconhecido (caixa preta) e, para testa-los é necessário fornecer entradas e avaliar as saídas geradas para estabelecer se estas estão em conformidade com a especificação do programa.

Na técnica baseada em erros são utilizados enganos típicos do processo de desenvolvimento de software para derivar requisitos de teste. Por sua vez, a técnica estrutural utiliza como informação aspectos internos do código para derivar os requisitos de teste (DELAMARO; MALDONADO; JINO, 2007). A Seção 2.3.2 apresenta os principais critérios utilizados na técnica estrutural, dado a sua importância para o projeto desenvolvido nesta tese.

2.3.2 Teste Estrutural

A técnica estrutural é uma filosofia de projeto de casos de teste que utiliza a estrutura do código de programas para derivar casos de teste (PRESSMAN, 2010). Essa técnica estabelece os requisitos de teste com base em uma implementação específica, requisitando a execução de trechos ou componentes elementares do software.

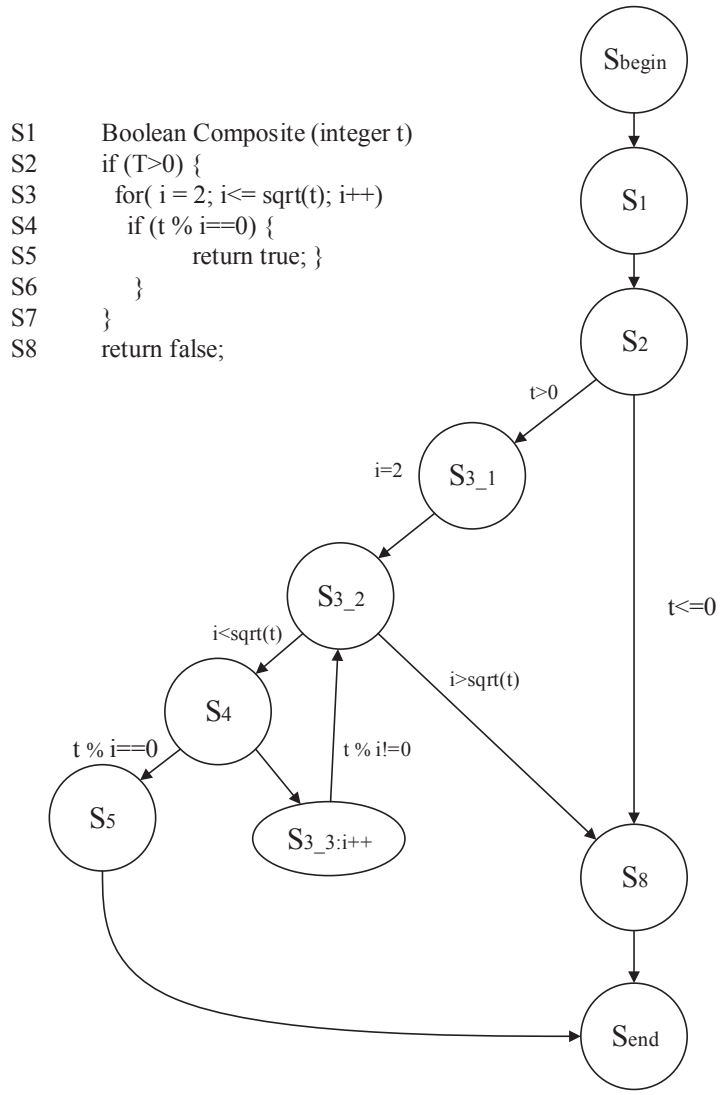
Em geral, os critérios pertencentes a essa técnica utilizam uma representação do programa conhecido como grafo de fluxo de controle (GFC). Um GFC=(N,E,s) é um grafo dirigido com apenas um único nó de entrada s e um único nó de saída, onde $\boldsymbol{N}$ é o conjunto de nós e cada nó $n \in N$ representa um bloco de execução. Um bloco de execução consiste em uma sequência máxima de comandos indivisíveis. Por sua vez, $\boldsymbol{E}$ representa o conjunto de arestas no grafo, as quais representam o controle de movimentação de um nó para o outro (GAO *et al.*, 2003; BARBOSA *et al.*, 2007).

Na Figura 5 é apresentado o GFC extraído de um pseudocódigo para determinar se um valor inteiro é um número composto. O grafo possui alguns nós com mais de uma aresta (arco) de saída, essas arestas representam fluxos de execuções que o programa pode assumir, comandos como if possuem mais de uma saída e, portanto, devem ser expressos no grafo.

O conceito de caminho no GFC é dado quando para uma sequência finita de nós $(n_1, n_2, n..., n_k)$, $k \geq 2$, existe uma aresta de n_i para $n_i + 1$ para $i = 1, 2, ...k - 1$. Em adição, existem diferentes tipos de caminhos que podem estar presentes em um GFC, são eles:

- Caminho simples: é um caminho em que todos os nós, exceto possivelmente o primeiro e o último, são distintos;
- Caminho livre de laço: é definido como um caminho em que todos os nós são distintos;

Figura 5 – Exemplo de um grafo de fluxo de controle



Fonte: Adaptada de Gao *et al.* (2003).

- Caminho completo: é um caminho em que o primeiro nó coincide com o primeiro nó do grafo e seu último nó é também o último nó do grafo.

Os critérios pertencentes a técnica estrutural são classificados com base no fluxo de controle, no fluxo de dados e complexidade dos programas sob teste (BARBOSA *et al.*, 2007).

Os critérios baseados em fluxo de controle utilizam características de controle da execução, como comandos ou desvios de um programa para derivar requisitos de teste. Nesse caso, um cenário ideal para atividade de teste seria executar todos os possíveis caminho do GFC de um programa, entretanto, embora desejável, existem desafios que impedem essa prática. Como exemplo, existem caminhos que não pode ser executados, ou seja, não existe nenhuma dado de teste no domínio de entrada que permita a execução de

um determinado caminho. Além disso, a quantidade de caminhos lógicos de um programa pode ser muito grande, ocasionando em um longo período de tempo para identificar, projetar casos de teste e executá-los. Portanto, o teste de caminhos exaustivo pode ser impraticável, se não impossível (MYERS; SANDLER; BADGETT, 2011).

Ainda assim, a técnica estrutural fornece um conjunto de critérios que permitem verificar o fluxo de controle dos programas, a seguir são apresentados os principais critérios de fluxo de controle.

- Todos-Nós: esse critério requer que a execução do programa passe, ao menos uma vez, em cada vértice do GFC. Dessa forma, todos os comandos devem ser executados para satisfazer este critério;
- Todas-Arestas (Arcos): exige que cada aresta do grafo (ou fluxo de controle), seja exercitada pelo menos uma vez;
- Todos-Caminhos: esse critério requer que todos os possíveis caminhos do programa sejam exercitados.

É importante ressaltar, que a presença de laços em um programa pode implicar em um número de caminhos muito grande ou até mesmo infinito, o que pode inviabilizar a utilização de alguns critérios. Desse modo, critérios de fluxo de controle testam o programa considerando um número n de repetições (DELAMARO; MALDONADO; JINO, 2007).

Os critérios de fluxo de dados baseiam-se no pressuposto de que para testar um programa de forma adequada é necessário avaliar o fluxo de valores de dados. Dessa forma, é importante assegurar que os valores assumidos em um determinado ponto do código são criados e usados adequadamente, o que implica em um destaque para as definições e usos de variáveis.

A definição (def) consiste em um local do programa onde um valor é atribuído a uma variável (Exemplo: $var\_a = 1$), já o uso de uma variável é dado por um local do código onde o valor da variável é acessado. Existem dois tipos de acesso a uma variável, o uso predicativo (p-uso) e uso computacional (c-uso). O p-uso ocorre quando o acesso ao valor de uma variável ocorre em um comando predicativo que determina um fluxo de execução, como um comando de laço ou de decisão (Exemplo: $if(var\_a > 2)$). Por sua vez, o c-uso é dado pelo uso do valor da variável em uma computação, ou seja, quando o valor de uma variável é utilizado para atribuição de outra (Exemplo: $var\_a = var\_b - 1;$, existe um uso computacional de var_b) (AMMANN; OFFUTT, 2008).

Existem diferentes critérios na literatura propostos para verificar o fluxo de dados. Os primeiros e mais conhecidos critérios de fluxo de dados foram propostos por Rapps e Weyuker (1985) em uma família de critérios para o fluxo de dados, a seguir são apresentados os principais critérios.

- Todas-Definições: Exige que cada definição de variável seja exercitada por pelo menos uma vez, seja ela c-uso ou p-uso;

- Todos-Usos: Requer que todas as associações entre uma definição de uma variável e seus usos (c-uso ou p-uso) sejam exercitadas pelos casos de teste, por meio de pelo menos um caminho livre de definição;

- Todos-Du-Caminhos: este critério requer que toda associação entre uma definição de variável e seus subsequentes usos, seja exercitada por todos os caminhos livres de definição e de laços que contemplem esta associação.

Por último, os critérios estruturais baseados na complexidade utilizam dados sobre a complexidade do software para gerar requisitos de teste. Uma medida de complexidade muito utilizada é a complexidade ciclomática (MCCABE, 1976), que pode ser definida da seguinte forma: se $\boldsymbol{n}$ é o número de estruturas de seleção e repetição no programa, a complexidade ciclomática do programa é $\boldsymbol{n}$+1 (WAZLAWICK, 2013). A partir do valor de complexidade determina-se o número de caminhos independentes do programa, ou seja, caminhos que introduzem no mínimo uma nova aresta. Em seguida, são projetados casos de teste que executem todos os caminhos independentes.

Conforme descrito anteriormente, os programas concorrentes adicionam novas características que exigem mudanças na atividade de teste. Essas características, em grande parte, estão relacionadas com a estrutura do código dessas aplicações. Por esse motivo, a técnica estrutural é uma das técnicas de teste que foi estendida para o teste de software concorrente. Na seção seguinte serão apresentados conceitos que abrangem especificamente o teste de software concorrente, incluindo o teste estrutural para esse domínio.

2.3.3 Teste de Software Concorrente

De fato, os programas concorrentes apresentam características adicionais em relação aos programas sequenciais. Essas características envolvem, principalmente, mecanismos de interação essenciais para realização das tarefas concorrentes. Embora essa interação seja beneficiada por esses mecanismos, existe um custo adicional para atividade de teste. Diante disso, um dos principais objetivos no teste de aplicações concorrentes é identificar defeitos relacionados à comunicação, paralelismo e à sincronização (SOUZA; SOUZA; ZALUSKA, 2014).

O principal desafio no teste de programas concorrentes é o não-determinismo, o qual ocorre quando em duas ou mais execuções com a mesma entrada, saídas diferentes são obtidas. Isso pode ocorrer devido à ordem de sincronização que pode ocorrer entre os processos concorrentes. A ordem de sincronização, por si só, não caracteriza uma situação

de erro, porém é necessário garantir que o programa funciona corretamente para todas as possíveis sincronizações (TIAN; GONG, 2014; SOUZA; VERGILIO; SOUZA, 2007).

Nesse contexto, as rotinas de programas concorrentes podem apresentar diferentes tipos de defeitos, os quais ocasionam diferentes falhas. Desses, dois são mais comuns entre as aplicações, os erros de observabilidade e travamento.

O erro de observabilidade está associado ao ambiente de teste e pode ocorrer quando o testador ou ferramenta de teste não possui o controle do comportamento do programa executado em paralelo. Como exemplo, esse tipo de erro pode ocorrer quando apenas uma ordem específica de acessos a um recurso compartilhado pode resultar em uma falha. Por sua vez, o erro de travamento pode ocorrer quando um processo não se comunica com o processo correto e, consequentemente, um processo ficará à espera de uma sincronização infinitamente ou até que um tempo limite seja excedido, pois a mensagem esperada não será recebida.

A partir desses erros, existem tipos de defeitos concorrentes resultantes. Alguns desses defeitos são mapeados em taxonomias de defeitos concorrentes. Wu e Kaiser (2011) descrevem um modelo geral de defeitos para programas concorrentes (Quadro 1).

Quadro 1 – Defeitos típicos de programas concorrentes.

Defeito	Descrição
Data Race	Ocorre quando múltiplas *threads* realizam operações de escrita e leitura em um mesmo arquivo, e o resultado da execução depende exclusivamente da ordem em que o arquivo é acessado.
Memory Inconsistency	Ocorre quando diferentes *threads* possuem visões inconsistentes de uma mesma variável, em outras palavras, dada uma variável "*var*"e duas *threads* "*t1*"e "*t2*", o valor armazenado em *var* é considerado diferente entre *t1* e *t2*.
Atomicity Violation	Originada da execução concorrente de múltiplas *threads* que transgridem a atomicidade de uma determinada região do código, ou seja, dado um trecho do código que deve ser executado sem interrupção, por algum motivo o mesmo é interrompido, assim, alguma computação pode ser perdida durante essa interferência.
DeadLock	Ocorre quando múltiplas *threads* são bloqueadas para sempre, isto é, dadas três *threads* "*t1*","*t2*"e "*t3*", em um dado momento *t1* pode ficar bloqueada na espera de uma sincronização com *t2*, posteriormente *t2* pode ser bloqueada antes de sincronizar com *t1*, na espera de uma sincronização com *t3*, nesse momento se *t3* for bloqueada, esperando por exemplo por uma sincronização com *t1*, ambas as *threads* ficarão bloqueadas para sempre.
Livelock	A ação de uma *thread* pode ser realizada em resposta à ação de outra *thread*, assim, se a ação de uma *thread t2* ocorrer em resposta de *t1*, ao passo que um ciclo de ações e reações seja iniciado, ambas as *threads* ficarão executando infinitamente ocasionando um *Livelock*, apesar disso, diferentemente do deadlock, as *threads* não serão bloqueadas.
Starvation	Ocorre quando uma *thread* está bloqueada, impedida de acessar recursos compartilhados, e nunca é escalonada para acessar regiões compartilhadas, consequentemente não consegue progredir em suas atividades.
Suspension	Ocorre quando uma *thread* suspende suas atividades ou espera indefinidamente.

Em vista de revelar os defeitos concorrentes, a atividade de teste nesse domínio estende alguns conceitos do teste de programas sequenciais com ajustes para atender mecanismos e aspectos dessas aplicações. Como exemplo, o teste estrutural de programas concorrentes define um modelo para abstrair aspectos estruturais do programa. Existem diferentes modelos baseados no fluxo de controle e no fluxo de dados dessas aplicações, o modelo utilizado nesta tese foi proposto por Souza *et al.* (2008).

O modelo considera um número n fixo e conhecido de processos paralelos criados durante a inicialização da aplicação concorrente. Nesse modelo, a comunicação entre processos ocorre de duas formas: ponto a ponto, onde um processo pode enviar uma mensagem para outro usando primitivas como *send* e *receive*, e comunicação coletiva, na qual um processo pode enviar uma mensagem para todos os processos da aplicação (ou para um grupo de processos) (SOUZA *et al.*, 2008).

Dessa forma, para um determinado programa paralelo *ProgPar* composto pelos processos $ProgPar = P^0, P^1, ...P^{n-1}$, cada processo P_i possui seu GFC, construído da mesma forma que em programas sequenciais. Sendo assim, um nó n pode ou não estar associado a uma função de comunicação do tipo *send* ou *receive*. Com base nisso, um Grafo de Fluxo de Controle Paralelo (GFCP) é construído para *ProgPar*, sendo composto pelos GFC de todos os processos presentes em *ProgPar* e pelas arestas de comunicação dos processos paralelos. Vale ressaltar que, neste modelo o conjunto de nós e arestas são representados por N e E, respectivamente. Um nó i em um processo p é representado pela notação n_i^p. Além disso, dois subconjuntos de nós são definidos: N_s, o qual é formado por nós que contém funções de envio de mensagens e N_r que é formado por nós que contém primitivas de recebimento de mensagens.

Para cada $n_i^p \in N_s$, um conjunto R_i^p é associado, o qual apresenta os possíveis nós que recebem a mensagem enviada pelo nó n_i^p. Já as arestas do GFCP podem ocorrer de duas formas:

- Arestas intraprocessos (E_i): são arestas que estão internas a um processo p;
- Arestas interprocessos (E_s): são arestas que representam a comunicação entre processos distintos.

No contexto de programas concorrentes, a definição de critérios considera, principalmente, os aspectos relacionados a interação entre processos (ou *threads*). Em vista disso, uma família de de critérios de testes estruturais para programas concorrentes baseados no fluxo de controle foram definida (SOUZA *et al.*, 2008; SOUZA *et al.*, 2013). Esses critérios verificam as interações entre processos e são definidos a partir do GFCP do programa sob teste, a saber:

- **Todos-Nós-s (*all-nodes-s*)**: Este critério requer que todos os nós do conjunto N_s sejam exercitados pelo menos uma vez pelo conjunto de casos de teste.
- **Todos-Nós-r (*all-nodes-r*)**: Requer que todos os nós do conjunto N_r sejam exercitados pelo menos uma vez pelo conjunto de teste.
- **Todos-Nós (*all-nodes*)**: Este critério requer que todos os nós do conjunto N sejam exercitados pelo menos uma vez pelo conjunto de teste.
- **Todas-Arestas-Sincronização (*all-sync-edges*)**: Requer que todas as arestas do conjunto E_s sejam exercitadas pelo menos uma vez pelo conjunto de teste.
- **Todas-Arestas (*all-edges*)**: Requer que todas as arestas do conjunto E sejam exercitadas pelo menos uma vez pelo conjunto de teste.

Assim como revelar o defeito de um programa concorrente, selecionar a entrada de teste e seu respectivo caminho de execução possui forte relevância para o processo de qualidade, uma vez que, um defeito só poderá ser reportado se houver meios de reproduzi-lo (HUANG; ZHANG, 2016). As propriedades de um programa concorrente tendem a dificultar essa tarefa, pois um mesmo dado de teste pode percorrer caminhos diferentes e, consequentemente, pode não revelar a possível presença de um defeito.

Nesse sentido, a execução determinística permite a execução da mesma sequência de eventos de sincronização para uma mesma entrada de teste. Para esse fim, informações sobre rastros e sincronizações são necessárias para possibilitar a execução determinística e definir mecanismos que permitam a reexecução de uma entrada de teste sob as mesmas condições anteriores (Carver; Tai, 1991).

As informações de rastro armazenam dados sobre a ordem em que os eventos de sincronização ocorreram e os pares de sincronização (Emissor/Receptor) que foram utilizados durante a execução do programa. Essas informações são essenciais para reexecução do programa e, além disso, permitem avaliar a cobertura das primitivas de sincronização (SOUZA; SOUZA; ZALUSKA, 2014). Em alguns trabalhos da literatura, as informações de rastro também são utilizadas para derivar novas possíveis combinações de pares de sincronização, também conhecida por *reachability testing*. Essa prática, embora dispendiosa, contribui para aumentar a cobertura dos requisitos de teste (Lei; Carver, 2006).

As atividades de identificar requisitos de teste, executar o programa, gerar rastros da execução e verificar o comportamento de saída são dispendiosas. Dessa forma, a automatização dessa função pode diminuir os esforços gastos na atividade de teste e ainda agilizar esse processo. Com este propósito, a ferramenta ValiPar tem como objetivo apoiar a aplicação de modelos e critérios de teste em programas concorrentes. Uma breve descrição da ferramenta ValiPar é apresentada a seguir.

2.3.3.1 Ferramenta ValiPar

A ValiPar utiliza a técnica estrutural para testar programas concorrentes que utilizam mecanismos para comunicação baseados nos paradigmas de passagem de mensagem e memória compartilhada. A ValiPar possui diferentes versões desenvolvidas para apoiar o teste de programas concorrentes, as quais distinguem-se, principalmente, pela linguagem de programação utilizada. Neste projeto, a versão utilizada considera o teste de aplicações desenvolvidas na linguagem Java. Diferentemente das versões anteriores, essa versão considera a composição dos modelos de passagem de mensagem e memória compartilhada (SOUZA *et al.*, 2013; PRADO *et al.*, 2015).

A ferramenta ValiPar é composta por cinco módulos, cada modo recebe artefatos de entrada específicos. A arquitetura da ValiPar é apresentada na Figura 6, onde os módulos são representados pelos retângulos enquanto as entradas são representadas pelas elipses.

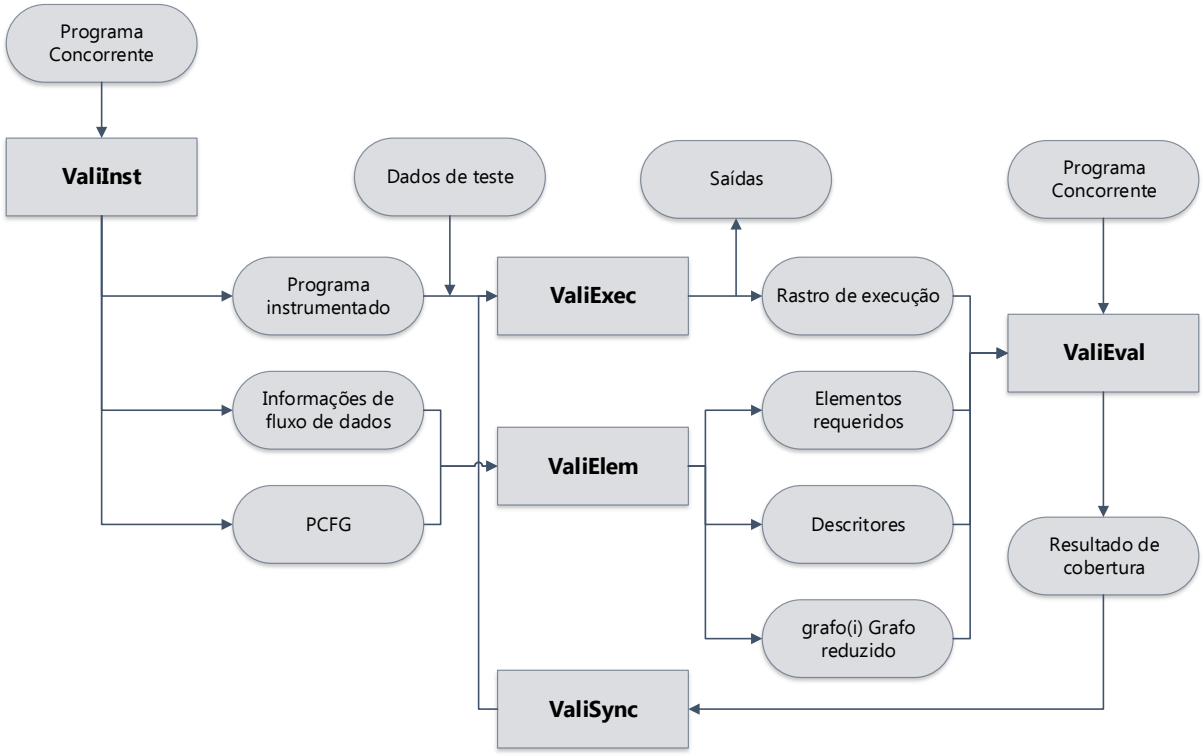


Figura 6 – Arquitetura ferramenta ValiPar.

O módulo **ValiInst** recebe como entrada o programa concorrente a ser testado. Este módulo realiza a análise estática do programa e extrai as informações de fluxo de controle, fluxo de dados e fluxo de sincronização, os quais serão utilizados pelos demais módulos durante a atividade de teste. Além disso, o módulo ValiInst realiza a instrumentação do código fonte, tarefa necessária para permitir que posteriormente um rastro da execução seja gerado a partir da execução de um dado de teste.

Por sua vez, o módulo **ValiElem** recebe como entrada as informações de fluxo e o PCFG gerado pelo módulo ValiInst. Este módulo tem como objetivo gerar os elementos requeridos para os critérios de teste. Em adição, é gerado um descritor responsável por estabelecer possíveis caminhos que podem cobrir cada elemento requerido.

O módulo **ValiExec** recebe o programa instrumentado e o executa com os dados de teste fornecidos pelo usuário. Para cada execução do programa, este módulo armazena as informações sobre entradas e saídas, a relação de caminhos percorridos por cada processo e a sequência de sincronizações sucedida. Este módulo também permite a execução determinística e não determinística dos programas sob teste, isto é, com a execução não determinística (ou livre) é possível que qualquer aresta de sincronização seja exercitada durante a execução, já com a execução determinística o usuário pode repetir uma sequência de sincronizações executada previamente, considerando os mesmos dados de teste.

O módulo **ValiEval** utiliza os dados de rastro e os elementos requeridos gerados pelo módulo ValiElem para calcular a cobertura obtida e, então apresentar quais elementos foram cobertos pelo conjunto de teste utilizado. Além disso, este módulo fornece a alcançabilidade dos critérios obtida pelos dados fornecidos.

Finalmente, o módulo **ValiSync** tem como objetivo realizar a geração automática de variantes, isto é, impor a execução de diferentes pares de sincronização com um mesmo dado de teste, aumentando, se possível, a cobertura das arestas de sincronização (BATISTA, 2015).

Em vista dos desafios elencados, a seleção automática de dados de teste guiada por heurísticas bioinspiradas tem sido uma grande aliada no teste de programas concorrentes. Nessa técnica, a seleção é transformada em um processo de busca, onde cada entrada é vista como um indivíduo de uma população onde indivíduos mais aptos ao ambiente possuem maior chance de sobrevivência e consequentemente maior probabilidade de passar informações genéticas para futuras gerações. A seção seguinte descreve detalhadamente esse processo.

2.3.4 Geração automática de dados de teste

A geração automática de dados de teste tem sido amplamente empregada para apoiar a atividade de teste na seleção de entradas mais significativas (BERTOLINO, 2007; MOHI-ALDEEN; DERIS; MOHAMAD, 2014). A seleção geralmente é guiada por critérios de testes que determinam requisitos básicos do programa sob teste, para os quais são definidos dados de teste que cobrem esses elementos.

Embora a automatização dessa atividade seja muito desejada, não existe um algoritmo de propósito geral e eficiente que seja capaz de gerar automaticamente e a baixo custo um conjunto de teste efetivo que seja adequado a um determinado critério de teste.

Nem mesmo é possível determinar automaticamente se esse conjunto existe (VERGILIO; MALDONADO; JINO, 2007). Além disso, existem restrições inerentes à atividade de teste que dificultam essa automatização, destacando:

- **Correção coincidente**: Ocorre quando o programa em teste apresenta um resultado correto para um dado de teste particular, não revelando a presença de um erro presente, ou seja, o erro existe mas devido ao valor do dado de teste coincidentemente apresentou um valor correto;
- **Caminho ausente**: corresponde a uma determinada funcionalidade requerida para o programa, mas que por algum motivo não foi implementada, ou seja, o caminho desejado não existe no programa;
- **Caminhos não executáveis**: ocorre quando não existe dado de teste que seja capaz de exercitar esse caminho. Isso ocorre com frequência devido principalmente a informações sobre o fluxo de dados de um programa;
- **Equivalência de programas**: não existe algoritmo de propósito geral capaz de determinar se dois programas apresentam, para qualquer dado possível do domínio de entrada, o mesmo comportamento.

Apesar das limitações, várias técnicas de geração de dados de teste são encontradas na literatura (AHMED; HERMADI, 2008; BAGNARA *et al.*, 2013; GHIDUK, 2014; SILVA; SOUZA; SOUZA, 2014). Em geral, essas técnicas utilizam informações sobre a cobertura em relação a critérios de teste para avaliar o grau de importância de cada entrada para o programa sob teste.

Para melhor exemplificar os desafios da geração automática de dados de teste no contexto de programas concorrentes, considere a classe Java *RollerCoaster* apresentada no Código-fonte 1. Essa classe tem como objetivo simular, de forma simplificada, o funcionamento de uma montanha-russa sendo composta por dois métodos: *startRollerCoaster()* e *startEngine()*, os quais tem como objetivo iniciar o funcionamento da montanha-russa e ligar a quantidade de motores necessários, respectivamente.

O desafio da geração automática de dados de teste nesse contexto é identificar não apenas entradas que executem todos os cenários factíveis, mas também cenários que não estão previstos e podem ocorrer durante uma execução. Como exemplo, o método *startRollerCoaster()* possui como domínio de entrada valores inteiros entre 1 e 4, apesar disso, de acordo com as técnicas de teste, valores fora desse intervalo e suas extremidades também devem ser verificadas a fim de uma melhor garantia de qualidade, consequentemente durante a seleção todos esses valores devem ser considerados.

Código-fonte 1 – Classe RollerCoaster

```
1:  public class RollerCoaster {
2:
3:  public void startRollerCoaster(int numberCars){
4:
5:    if(numberCars <=2 && numberCars >0){
6:      startEngine(1);
7:    }else if(numberCars>2 && numberCars<=4){
8:      startEngine(2);
9:    }else
10:   {
11:     System.out.println(``The value is wrong!! Try Again.'');
12:   }
13: }
14: public void startEngine(int numEngines){
15:   if(numEngines==1){
16:     Thread t1;
17:     t1.run();
18:     t1.Brakes.increment();
19:   }else if (numEngines==2){
20:     Thread t1;
21:     Thread t2;
22:     t1.run();
23:     t1.Brakes.increment();
24:     t2.run();
25:     t2.Brakes.increment();
26:   }
27: }
28: }
```

Durante a otimização da seleção de entradas de testes uma abordagem deve avaliar não apenas a contribuição individual de cada dado de teste, mas também o peso que esse dado possui em relação ao conjunto de teste que o mesmo pertence. Por exemplo, se as entradas 1 e 2, pertencentes ao mesmo conjunto, fossem utilizadas como teste para função *startRollerCoaster*() o programa seria iniciado utilizando apenas um motor em ambas as execuções, vide linhas [15-18] do Código 1, o trecho de código responsável pela utilização de dois motores não seria exercitado, linhas [19-25].

Apesar de ambos os trechos, [15-18] e [19-25], realizarem chamadas para o método *startEngine*(), uma execução concorrente entre *threads*, presente no trecho [19-25], pode apresentar falhas no programa se não tratada corretamente. Esse cenário será apresentado posteriormente na classe *Breaks*, Código-fonte 2. Desse modo, o desafio é selecionar um

conjunto de teste com entradas que executem características distintas garantindo maior cobertura do programa.

Na literatura é possível encontrar trabalhos que buscam realizar o teste minimizando os efeitos do não-determinismo. Essa tarefa não é trivial, pois para minimizar esses efeitos é necessário o uso de técnicas/ferramentas de alto custo computacional que podem inviabilizar a atividade de teste dependendo das características do software sob teste e do projeto. Os programas que utilizam mecanismos de comunicação apresentam grande possibilidade de caminhos, ou seja, sequências de execuções diferentes também conhecidas como *Interleavings*.

Para melhor exemplificar os *Interleavings* considere a classe java *Breaks* apresentada no Código-fonte 2. A classe ***Breaks*** é uma classe simples projetada para que em cada chamada do método ***increment***() seja acrescido 1 na variável ***c*** e decrescido 1 quando chamado o método ***decrement***(). O método ***value***() retorna o valor atual de ***c***.

Código-fonte 2 – Classe Breaks

```
1:  class Breaks {
2:    private static int c = 0;
3:    public static void increment() {
4:        c++;
5:    }
6:    public static void decrement() {
7:        c--;
8:    }
9:    public static int value() {
10:       return c;
11:   }
12: }
```

Essa classe não apresenta grande complexidade se executada com apenas uma única instância (*thread*), entretanto, o mesmo não pode ser garantido quando existem duas ou mais instâncias da mesma classe executando concorrentemente. Para entender melhor esse cenário, considere duas *Threads* T1 e T2 que possuem instâncias da classe ***Breaks***, no qual T1 realiza uma chamada do método ***increment***() enquanto T2 chama o método ***decrement***(). Para obter o valor de ***c*** cada *thread* chama o método ***value***() após incrementar ou decrementar a variável, conforme apresentado a seguir:

T1 *increment()*
T1 *value()*
T2 *decrement()*
T2 *value()*

Diferentes sequências de execução podem ocorrer neste cenário resultando em maior complexidade para atividade de teste, pois do ponto de vista de teste podem existir defeitos em cada uma das sequências e consequentemente todas devem ser verificadas de modo a melhorar a garantia de qualidade do software. Uma representação das possíveis sequências de execução para este cenário é apresentada no Quadro 2. Os métodos Increment e Decrement são representados no quadro como incr() e decr(), respectivamente. O símbolo # corresponde a coluna com a ordem de execução das *threads*.

Quadro 2 – Sequências de execução para classe Breaks com duas threads.

#	SEQUÊNCIAS DE EXECUÇÃO											
	SEQ-1		SEQ-2		SEQ-3		SEQ-4		SEQ-5		SEQ-6	
	T1	T2	T1	T2	T1	T2	T1	T2	T1	T2	T1	T2
1	*incr()*		*incr()*		*incr()*			*decr()*		*decr()*		*decr()*
2	*value()*			*decr()*		*decr()*		*value()*	*incr()*		*incr()*	
3		*decr()*	*value()*			*value()*	*incr()*			*value()*	*value()*	
4		*value()*		*value()*	*value()*		*value()*		*value()*			*value()*

O número de sequências obtidas para um programa depende da quantidade de métodos, *threads* e rotinas utilizadas. Usualmente, os programas apresentam mais de uma classe e também vários métodos para realizar seus objetivos, assim percebe-se como a complexidade do teste pode aumentar considerando esses fatores. Não obstante, outro desafio para o teste está relacionado a violação de atomicidade, ou seja, um bloco de execução que realiza alterações em uma variável compartilhada poderá sofrer preempções no meio de sua execução e o valor da variável pode não ser consistente nesse processo.

No cenário anterior, devido ao nível de abstração da linguagem de programação, não é possível identificar se cada rotina é executada de forma atômica. Para isso deve-se observar as instruções de baixo nível, as quais não são passíveis de preempção após o início de sua execução. Considerando o exemplo da classe *Breaks*, o seguinte cenário pode ocasionar uma falha no programa:

$$
\begin{array}{ll}
\text{T1: } \textbf{GET } C & c = 0 \\
\text{T1: } \textbf{ADD } 1 & \\
\text{T2: } \textbf{GET } C & c = 0 \\
\text{T2: } \textbf{SUB } C & \\
\text{T2: } \textbf{WRITE RESULT } C & c = -1 \\
\text{T1: } \textbf{WRITE RESULT } C & c = 1
\end{array}
$$

Neste exemplo a *thread* T1 realiza uma chamada da função *increment*() para incrementar o valor de c, entretanto, após ler o valor da variável e realizar a soma, T1 sofre uma preempção antes de gravar o valor na variável e T2 inicia sua execução. T2 faz uma chamada ao método *decrement*() que realiza todas as instruções de baixo nível para concluir sua execução. Assim, o valor que T1 possui da variável C já não corresponde mais ao valor atual da mesma, consequentemente, atribui um valor errado para variável C que poderá ocasionar uma falha no programa.

O cenário apresentado é conhecido como condição de corrida, sendo uma das possíveis categorias de defeitos dos programas concorrentes. Uma forma de diminuir o risco desse e de outros defeitos, é a verificação de todas as possíveis sequências de execução de um programa, entretanto, essa tarefa exige alto custo computacional e tempo. Em razão disso, as abordagens de geração buscam identificar trechos específicos do código que possuem maior probabilidade de haver defeitos.

A otimização da etapa de seleção de dados de teste para programas concorrente ainda é uma área recente de pesquisa. O processo de seleção utiliza técnicas de busca e otimização matemática para resolução de problemas complexos provenientes das atividades de teste. Normalmente, as técnicas de busca são empregadas para otimização de problemas em que o espaço de busca é muito grande, envolvendo a seleção de soluções ótimas ou quase ótimas para um determinado problema. Dessa forma, um novo campo de pesquisa denominado *Search-Based Software Testing* (SBST) vêm sendo amplamente investigado por diversos trabalhos na literatura (COLANZI *et al.*, 2013).

Em geral, os trabalhos de SBST utilizam técnicas meta-heurísticas para otimização de buscas. Uma meta-heurística é composta por formas genéricas de heurísticas que podem ser utilizadas para resolução de diferentes problemas. Sendo assim, as técnicas pertencentes a essa classe utilizam ideias de diversos domínios como inspiração para a busca de soluções para problemas de otimização.

No que tange a geração de dados de teste, o objetivo do teste é transformado em um problema de otimização, onde o espaço de busca é dado pelo domínio de entrada dos programas em teste. Os algoritmos de busca exploram o espaço de busca para encontrar dados de teste que satisfaçam o objetivo do teste. Neste trabalho, a geração de dados de teste foi explorada pelo Algoritmo Genético, uma das meta-heurísticas bioinspiradas, essa técnica é detalhada a seguir.

2.3.5 Algoritmo Genético

Um Algoritmo Genético (AG) utiliza uma coleção (população) de soluções, que por meio de estratégias de reprodução e recombinação seletivas, melhores soluções podem ser produzidas.

A geração de dados de teste com algoritmos genéticos utiliza conceitos da computação evolutiva, que baseia-se na evolução natural das espécies. Com essa técnica pode-se ter várias soluções que se encaixam em problemas com dificuldade na obtenção de resultados ótimos. Quando se implementa um algoritmo genético é preciso representar o espaço de busca do problema e suas possíveis soluções de forma computacional e encontrar a função de aptidão que é usada para avaliar quão boa é uma dada solução. O algoritmo genético parte da premissa de que dada duas soluções com resultados aproximados para

um determinado problema, essas duas soluções combinadas podem conduzir uma melhor solução.

Operadores genéticos como Cruzamento *crossover* e mutação são utilizados para a construção de novas soluções a partir de soluções antigas, de tal maneira que, para muitos problemas a população melhora constantemente. O operador *crossover* pode ser entendido da seguinte forma: dadas duas cadeias **a** e **b**, compostas por 6 variáveis (genes),

$$(a1, a2, a3, a4, a5, a6) \text{ e } (b1, b2, b3, b4, b5, b6),$$

as mesmas representam duas soluções para um determinado problema. Um ponto de *crossover* é escolhido aleatoriamente para gerar novas soluções. Neste exemplo considera-se que o ponto escolhido seja a posição 2. A nova geração será composta pelos seguintes indivíduos:

$$(a1, a2, \mathbf{b3}, \mathbf{b4}, \mathbf{b5}, \mathbf{b6}) \text{ e } (b1, b2, \mathbf{a3}, \mathbf{a4}, \mathbf{a5}, \mathbf{a6}),$$

A maioria das aplicações utilizam cadeias compostas por apenas 0s e 1s, por facilitar a tarefa e por vantagens impostas por esta abordagem. Outro operador utilizado para geração de novos indivíduos é a mutação. O operador de mutação oferece a oportunidade de alcançar espaços de busca não alcançados pelo *crossover*. Cada gene da cadeia é examinado e possui uma pequena probabilidade de ocorrer uma alteração arbitrária em sua estrutura. Para ilustrar a situação, considere um individuo composto por 10 genes:

$$1001010100,$$

após aplicação desse operador, suponha que o terceiro e oitavo gene sofram uma mutação. A sequência obtida após a mutação resulta no novo individuo:

$$10\mathbf{1}1010\mathbf{0}00$$

Em adição aos operadores citados, os indivíduos também são escolhidos com base em uma função de fitness. A função de aptidão avalia cada indivíduo com base no problema a ser otimizado. Dessa forma, a função de aptidão é considerada uma das partes mais importantes do AG. Após esta fase, existem abordagens utilizadas para seleção dos indivíduos, dentre elas pode-se destacar a Roleta, Torneio, Elitismo e a Amostragem universal estocástica.

A seleção por Roleta (*Roulette Wheel*) foi um dos primeiros operadores definidos para o Algoritmo Genético, sendo amplamente utilizado em diversos problemas de otimização. O objetivo desse operador consiste em simular uma roleta física dividida por porções que representam cada indivíduo da população. No entanto, diferentemente de uma roleta

tradicional em que existe uma proporção igual para todos os elementos, esse operador divide a proporção em função do valor de *fitness* que cada indivíduo possui, dessa forma, há maior probabilidade de selecionar um indivíduo com *fitness* mais significativo. Em notação matemática, a probabilidade P, que um indivíduo X de *fitness* F_x possui para ser escolhido é dada por: $P = \frac{F_x}{\sum_{i=1}^{n} i}$ (GOLDBERG, 1989; PRATIHAR, 2013).

O operador de seleção por Torneio (*Tournament*) é um mecanismo de seleção amplamente utilizado em AGs. A ideia central desse operador é promover torneios entre grupos de n ($n \geq 2$) indivíduos, selecionados aleatoriamente, da população. Em seguida, o indivíduo com melhor valor de *fitness*, ou seja, o melhor indivíduo desse conjunto é selecionado para compor a próxima geração. Esse processo ocorre iterativamente até que o número de indivíduos selecionados nos torneios seja igual ao tamanho da população (MILLER; GOLDBERG, 1995).

A seleção pelo operador de Elitismo (*Elitism*) considera a permanência de bons indivíduos fundamental para o processo de busca do AG. Essa estratégia considera que ao aplicar operadores de cruzamento e mutação bons indivíduos possam ser perdidos. Dessa forma, o mecanismo de seleção define uma parcela da população atual, contendo os melhores indivíduos, que será levada sem alterações para a próxima geração. Embora seja promissor, essa estratégia de seleção também pode ocasionar em um convergência precoce resultando em mínimos locais (THIERENS, 1998).

Por sua vez, a seleção por Amostragem Universal Estocástica (*Stochastic Universal Sampling*-SUS) é uma variação do operador Roleta que baseia-se na seleção proporcional de *fitness*. Diferentemente do operador Roleta, onde há a abstração de apenas uma única agulha de seleção, esse método determina um número n de agulhas (n = tamanho da população) igualmente espaçadas. Contudo, o espaço de cada indivíduo na roleta é proporcional ao valor de *fitness*, consequentemente, os indivíduos que possuem maior área terão maior probabilidade de serem selecionados. Outra diferença em relação a roleta tradicional é que para a seleção dos indivíduos o valor aleatório, que corresponde ao giro da roleta, será gerado apenas uma única vez. Essa estratégia visa garantir que indivíduos com valor de *fitness* ruins, possuam maior chance de ser selecionados e, consequentemente, contribui para diversificação da população (BAKER, 1987).

Uma técnica que tem sido empregada de forma colaborativa aos Algoritmos Genéticos é lógica difusa (ou lógica *fuzzy*). Essa técnica é utilizada para lidar com conceitos de verdade parcial, ou seja, quando não há total certeza se um valor é totalmente falso ou totalmente verdadeiro (KHMELEVA *et al.*, 2017; PLEROU; VLAMOU; PAPADOPOULOS, 2017). Uma visão geral sobre os conceitos dessa abordagem são apresentados a seguir.

2.3.6 Lógica Fuzzy

A lógica fuzzy foi proposta por Zadeh (1965), desde então vêm sendo utilizada em diversas áreas do conhecimento. Essa lógica pode ser entendida como uma generalização da lógica clássica que admite infinitos valores lógicos intermediários entre o falso e verdadeiro. Além disso, esse método também é visto com um tipo de sistema especialista, no qual utiliza-se o conhecimento adquirido para tomada de decisões.

Diferentemente da lógica booleana, que define apenas os valores lógicos 0 (falso) e 1 (verdadeiro), a lógica *fuzzy* determina que os valores verdade podem assumir qualquer número real entre 0 (totalmente falso) e 1 (totalmente verdadeiro). Esses resultados não são expressos de forma bem definida, mas linguisticamente como: “baixo”, “muito baixo”, “alto” e “muito alto”. Tais valores estão contidos em um conjunto *fuzzy*.

Cada conjunto *fuzzy* A é definido em termos de relevância a um conjunto universal U, por uma função de pertinência, associando a cada elemento x um número, $\mu_A(x)$, no intervalo fechado [0,1] que caracteriza o grau de pertinência de x em A. O fator de pertinência pode assumir qualquer valor entre 0 e 1, representando completa exclusão e completa pertinência, respectivamente. A função que determina a pertinência é dada por $\mu_A(x) : X \rightarrow [0,1]$.

Os intervalos linguísticos são representados por meio de variáveis linguísticas. Em uma variável linguística os valores são expressos em formato de texto que representam os nomes dos conjuntos *fuzzy*. Os conjuntos *fuzzy* são representados por funções de pertinência.

As funções de pertinência podem assumir diferentes formatos, dependendo do conceito que se deseja representar e do contexto em que são utilizadas, as mais comuns são: trapezoidal, triangular e gaussiana.

Os sistemas *fuzzy* empregam um conjunto de regras do tipo “Se-Então” baseadas nas variáveis linguísticas. Inicialmente, as variáveis de entrada passam por um processo denominado *fuzzificação*, onde é realizado um mapeamento do conjunto de números reais para um conjunto *fuzzy*. Em seguida, efetua-se a inferência sobre o conjunto de regras *fuzzy* obtendo os valores dos termos das variáveis de saída.

O mecanismo de inferência define a base para tomada de decisões. Por fim, as variáveis de saída passam por um processo denominado *defuzzificação* que consiste em transformar dados *fuzzy* para valores numéricos reais. Para isto, são utilizadas várias técnicas, tais como valor máximo, média dos máximos, média local dos máximos, centro de gravidade, ponto central da área e o centro da média (DRIANKOV; HELLENDOORN; REINFRANK, 1993).

2.4 Considerações Finais

A programação concorrente foi fundamental para evolução dos Sistemas Operacionais, entretanto, atualmente esse tipo de programação é amplamente utilizada em diversas aplicações, contribuindo para um melhor aproveitamento de recursos e desempenho. Ainda assim, o teste dessas aplicações é uma tarefa complexa e dispendiosa.

A geração automática de dados de teste, no contexto concorrente, visa facilitar o processo de seleção de entradas e otimizar a verificação dos produtos de software. A otimização bioinspirada tem sido uma técnica significativa para alcançar esses objetivos, nesse sentido, existem diferentes tipos de técnicas que podem apoiar essa otimização.

Este capítulo apresentou as principais técnicas de teste de software, juntamente com os respectivos critérios, e também discute conceitos de geração automática de dados de teste baseada em busca. Além disso, também foram explorados conceitos de metaheurísticas, mais precisamente o Algoritmo Genético, e sobre a Lógica *fuzzy*. O capítulo seguinte apresenta os principais trabalhos identificados na literatura que investigam o problema da seleção automática de dados de teste para programas concorrentes.

CAPÍTULO

3

TRABALHOS RELACIONADOS

3.1 Considerações iniciais

A geração automática de dados de teste não é um tema recente na literatura, entretanto, existem ramificações dessa área que não possuem ainda um número expressivo de relatos científicos, como é o caso da geração automática para software concorrente. Por pelo menos uma década a geração automática de dados de teste para programas concorrentes vem sendo explorada por meio de técnicas já consolidadas para programas sequenciais, apesar disso, as contribuições não são tão significativas se comparadas as de programas sequenciais.

Este capítulo apresenta estudos que sugerem ou exploram abordagens de geração para o contexto de programas concorrentes. Diferentemente da proposta deste trabalho, na literatura encontram-se trabalhos que investigam esse problema isolando um ou outro paradigma de comunicação para abordar o problema de seleção de dados de teste. Por esse motivo, os trabalhos serão divididos em dois grupos que representam os paradigmas de comunicação, geração de dados para programas de passagem de mensagem e memória compartilhada.

3.2 Geração automática de dados de testes para programas com memória compartilhada

O trabalho de Eytani (2006) foi um dos pioneiros na seleção automática de dados de teste para programas concorrentes que usam comunicação por memória compartilhada. O problema do não determinismo, presente nas aplicações concorrentes, já era conhecido nesse período e, consequentemente, sabia-se que o número de possíveis *interleavings* crescia em função do número de variáveis compartilhadas e do número *threads* que acessavam

e manipulavam essas variáveis. Além disso, também era de conhecimento compartilhado que apenas um pequeno número de *interleavings* seria capaz de revelar defeitos como *deadlocks* e *race conditions*, assim, somente a re-execução múltipla dos testes não garantiria que os defeitos pudessem ser revelados.

A geração aleatória já havia sido empregada em estudos anteriores, devido principalmente ao baixo custo computacional e simplicidade, entretanto, o uso dessa técnica não fornecia garantias sobre a qualidade do teste. Por esse motivo, Eytani (2006) propôs uma reformulação da geração de teste aleatório para programas concorrentes em Java como um problema de busca.

A solução proposta foi implementada em um protótipo denominado GeneticFinder, uma ferramenta de *noise-maker* que empregava um algoritmo genético como método de busca. A abordagem visava maximizar duas funções objetivo predefinidas: i) alcançar uma alta probabilidade de que o defeito se manifeste em todas as execuções do programa; e ii) exportar uma grande quantidade de informações de depuração para o usuário, a fim de permitir a reprodução do defeito.

Na representação do AG no GeneticFinder cada cromossomo modela uma configuração usando uma tabela *hash* de três níveis. O primeiro nível na tabela contém os nomes das heurísticas, usados como uma chave para acessar o segundo nível que contém, para cada heurística, as variáveis e os locais do programa emitidos por essa heurística.

Os parâmetros de ruído de cada heurística (para cada local ou variável) são encontrados no terceiro nível da tabela hash e são acessados de maneira semelhante. A recombinação (ou *crossover*) é feita escolhendo aleatoriamente um par de configurações "pais". Cada pai contribui com um subconjunto aleatório de suas heurísticas (e depois um subconjunto de variáveis e localizações do programa) para a nova configuração de descendência. Quando os dois pais adicionam a mesma heurística e localização (ou variável) ao seu descendente, seus parâmetros de ruído são escolhidos aleatoriamente e delimitados pelos valores dos pais.

A mutação é precedida escolhendo-se um subconjunto de variáveis ou locais de uma heurística e removendo-os da configuração descendente. Esse procedimento é realizado para criar uma configuração menor que ainda manifesta os erros concorrentes. O operador de mutação é alcançado criando-se aleatoriamente uma série de novas configurações para cada geração.

A validação do estudo foi realizada por meio de estudos experimentais empregando-se programas concorrentes com defeitos já conhecidos e documentados. Os defeitos foram semeados em diferentes variáveis e locais do programa, sendo necessária a aplicação de ruídos nesses locais para que os defeitos fossem devidamente identificados. No experimento, os locais já eram conhecidos, ou seja, o protótipo foi configurado previamente para gerar

ruídos nesses locais.

Apesar de ser uma abordagem promissora, considerando principalmente a época que este estudo foi desenvolvido, existem alguns desafios que podem tornar o uso dessa abordagem inviável. Quando os locais não são conhecidos, cenário padrão na a atividade do teste, faz-se necessário executar o programa sob teste diversas vezes para que os diferentes locais sejam identificados, entretanto, não há garantias que o local defeituoso será mapeado durante essas execução. Apesar disso, a abordagem permite que, uma vez o defeito identificado, sempre será possível reproduzir esse defeito.

Em conformidade aos problemas citados acima, Steenbuck e Fraser (2013) descrevem alguns desafios inerentes à geração de dados de teste para defeitos concorrentes em programas de memória compartilhada, são eles: i) ausência de controle sobre o escalonador de processos, como ocorre na linguagem Java; ii) compreensão limitada do programador sobre as consequências de várias threads acessando uma memória compartilhada; e iii) escalonamento específico de um defeito de natureza do concorrente pode não ser de conhecimento do programador.

Como proposta, frente aos desafios citados, o trabalho proposto por Steenbuck e Fraser (2013) apresenta uma técnica que viabiliza a geração de dados de teste e o escalonamento de execuções no teste de unidade em programas concorrentes em Java. A técnica concebe os dados de testes para que combinações de acessos à memória compartilhada (variáveis compartilhadas) sejam percorridas por diferentes threads e, em seguida, explora diferentes interleavings para essas combinações.

Todo o processo de busca é guiado por um critério de concorrência definido pelos autores, cuja definição é dada por: **Cobertura de concorrência** - Todas as combinações executáveis de η pontos de sincronização para cada grupo de ϑ variáveis, sendo acessadas por ε threads, devem ser exercitadas por pelo menos um dado de teste.

A cobertura de concorrência, usualmente, produz muitos objetivos de cobertura que precisam ser exercitados pelos testes. Em decorrência disso, o objetivo da abordagem de geração de testes proposta é produzir automaticamente conjuntos de testes que alcancem uma alta cobertura de concorrência. O algoritmo 1 descreve a abordagem em alto nível de abstração.

A saída do algoritmo é um conjunto de dados de teste, em que cada teste é formado por uma tupla que consiste em uma sequência de prefixos p que determina um objeto compartilhado, um conjunto de m sequências de métodos que executam operações no objeto compartilhado e um planejamento s que determina como as m threads são intercaladas. Esse conjunto de testes busca abranger o maior número possível de todas as n combinações de pontos de sincronização de v variáveis com m threads.

Na avaliação do estudo, um protótipo foi implementado com a abordagem proposta,

Algoritmo 1 – Visão conceitual de geração de suíte de teste
Fonte: Adaptada de Steenbuck e Fraser (2013).

Requer: Classe C
Requer: Número de pontos de sincronização n
Requer: Número de threads m
Requer: Número de variáveis v

```
1:  procedimento GenerateSuite(C,n,m,v)
2:      p ← GeneratePrefixSequence(C)
3:      N ← GetSynchronizationPoints(C)
4:      S ← GenerateSchedules(n,m,v,N)
5:      P ← GeneratePartialGoals(S,m)
6:      S ← remove statically infeasible schedules from S
7:      A ← generate method sequences for P
8:      T ← {}
9:      para s ∈ S faça
10:         para sequences ∈ GetMatchings(s,A) faça
11:             se seq.execution of seq reaches s então
12:                 se par.execution of seq reaches s então
13:                     T ← T ∪ {(p,sequences,s)}
14:                 fim se
15:             fim se
16:         fim para
17:     fim para
18:     retorna T
19: fim procedimento
```

denominado CONSUITE, uma extensão da já consolidada ferramenta de geração de testes de unidade EVOSUITE. A CONSUITE consiste em uma ferramenta baseada em busca, que emprega algoritmo genético para esse processo, visando a geração de dados de teste de programas concorrentes desenvolvidos em Java.

Guo, Kusano e Wang (2016) apresentam um método de execução simbólica incremental para software concorrente com o objetivo de gerar testes automaticamente. Esse método tem como foco apenas as execuções afetadas por alterações no código entre duas versões do programa. Uma análise inter-*threads* e inter-procedimental de impacto de mudanças foi proposta para verificar se uma declaração é afetada pelas mudanças, e em seguida, alavancar a informação para escolher as execuções que precisam ser reexploradas.

A abordagem incremental é apresentada no Algoritmo 2, existem duas diferenças significativas do procedimento estabelecido no estudo de *baseline*. Primeiramente, a entrada do algoritmo sofreu alterações, enquanto a *baseline* considera apenas um programa como entrada a nova abordagem considera tanto a versão atual quanto a versão antiga do programa (P *e* P*'*).

Antes da execução do programa P*'* é computado o conjunto a frente impactado (*ISfwd*) e também o conjunto anterior impactado (*ISbwd*). Em adição, a tabela PS com

Algoritmo 2 – Execução simbólica incremental. (GUO; KUSANO; WANG, 2016)

Requer: $IS_{fwd} \leftarrow$ ComputeForwardImpactedSet(P, P')
Requer: $IS_{bwd} \leftarrow$ ComputeBackwardImpactedSet(P, P')
Requer: $PS[s] \leftarrow$ the summary at s computed in previous program P

```
1:  procedimento NextState(s,t)
2:      s ←(pcon, M, enabled, branch, done)
3:      se t is halt então
4:          s' ← normal_end_state
5:      senão se t is abort então
6:          s' ← falty_end_state
7:      senão se t is assignment υ := exp então
8:          se t.inst ∈ IS_bwd and pcon ⟹ PS[s] então
9:              s' ← early_termination_state
10:         senão
11:             s' ← (pcon, M[υ ⟼ exp])
12:         fim se
13:     senão se t is ASSUME(c) and M[pcon ∧ c] is satisfiable então
14:         se t.inst ∈ IS_fwd and another branch has been explored então
15:             s' ← early_termination_state
16:         senão
17:             s' ← (pcon ∧ c, M)
18:         fim se
19:     senão
20:         s' ← infeasible_state
21:     fim se
22:     retorna s'
23: fim procedimento
```

os resumos de execução computados em P é transferida para o novo programa P'. Para cada estado s, o conjunto de execuções exploradas iniciado a partir de s é denotado $PS[s]$

Por conseguinte, as linhas [8-10] e [14-16], contidas no procedimento *NextState*, são responsáveis por levantar *ISfwd, ISbwd* e $PS[s]$ para decidir, para cada etapa da execução simbólica ($s^t s$), se todas as execuções iniciadas em *NextState s'* são redundantes. Especificamente, se $t.inst \in ISfwd$, a declaração do ramo corrente não sofre impacto pelo conjunto.

Uma vez que o ramo para executar em s é infactível, se um dos ramos já foi explorado, pode-se forçar uma rescisão antecipada da execução atual. Similarmente, se $t.inst \notin ISbwd$, o cálculo de precondição mais fraco, sobre o qual o resumo da execução é computado, não seria afetado pelas mudanças de código. Portanto, é possível enviar o resumo $PS[s]$ de P para P'. Se a condição atual de caminho *pcon*, no programa modificado, for submetida por $PS[s]$, então a continuação da execução de s não causaria novos erros. Nesse caso pode-se forçar uma rescisão antecipada da execução atual diminuindo o tempo gasto durante a geração dos testes.

Para o método de geração por execução simbólica incremental foi desenvolvida uma ferramenta denominada Conc-iSE, a qual foi avaliada por meio de um conjunto de programas C *multi-threads*. Durante a avaliação, foram produzidos resultados que demonstram uma redução do tempo de execução quando comparado a outras ferramentas de execução simbólica.

O trabalho proposto por Schimmel *et al.* (2015) apresenta uma abordagem para geração de casos de testes paralelos. Essa abordagem utiliza os testes de unidade convencionais como entrada e gera automaticamente casos de teste paralelos. A principal diferença entre os testes de unidade convencionais e os testes paralelos é o número de métodos sob teste (MUT - *Methods under test*). Enquanto os testes convencionais verificam apenas um MUT sob a perspectiva de um *Assertion*, os testes paralelos consistem em pelo menos dois métodos, dependendo do número de *threads* envolvidas no cenário em observação.

A definição dos testes paralelos segue o padrão convencional de execução para testes convencionais, esse padrão é composto por: **i)** ***Arrange***: que define e inicializa as variáveis, realiza a chamada aos métodos requeridos e configura as dependências; **ii)** ***Act***: responsável por executar o método sob teste; e **iii)** ***Assert***: que define e valida as restrições esperadas causadas pela execução do MUT.

Embora os testes de unidade paralelos também empreguem esse padrão de execução, o padrão foi estendido para a abordagem proposta. Além disso, também foi adicionado um novo estágio para segurança de múltiplas *threads*. O novo padrão foi estabelecido da seguinte forma: **i)** ***Arrange*** define e inicializa as variáveis de entrada e as dependência para todos os MUTs e executa, de modo sequencial, o código de inicialização; **ii)** ***Act*** instancia e executa uma *thread* para cada MUT; **iii)** ***Wait*** determina a espera de todas as *threads* executando MUTs; e **iv)** ***Assert*** valida os *assertations* definidos previamente (opcional).

Antes de executar um MUT de modo paralelo, os testes de unidade necessitam que todas as inicializações necessárias estejam disponíveis. Após essa execução, cada teste deve esperar que todos os demais MUTs terminem suas execuções com sucesso. A abordagem de geração foi estabelecida seguindo duas etapas principais, sendo a identificação dos MUTs e extração/combinação dos testes.

A primeira etapa do processo de geração é a identificação dos MUTs. A entrada para abordagem proposta são pares de métodos que são executados em paralelo em tempo de execução que podem apresentar condições de corrida. A ferramenta AutoRT, proposta em (Schimmel *et al.*, 2013), foi empregada para identificar automaticamente essas seções de código. Para identificar um teste de unidade paralelo, para um dado par de métodos, é necessário encontrar todos os testes de unidade convencionais que testam qualquer método. Os MUTs são identificados por meio de chamadas de métodos imediatas antes da primeira declaração de *assert*.

A extração e combinação dos testes estabelece a segunda etapa do processo de geração. Todas as declarações antes da chamada do MUT são consideradas para inicialização do teste. Essas declarações são copiadas para o caso de teste paralelos. Qualquer declaração após a chamada do MUT é requerida para o *assertation*, então devem ser ignoradas ou adicionadas para os casos de teste paralelos, enriquecendo os casos de teste. Para melhor exemplificar a abordagem, considere os métodos apresentados na Figura 7.

```
1:  @TestMethod
2:  public void AddLine_ValidLine_Success(){
3:      StubTextFormatService service = new
        StubTextFormatService();
4:      String line = TestData.
        SAMPLE_LINE_1_UNFORMATTED;
5:      srvice.FormatLineString = (x) =>
6:      {return TestData.
        SAMPLE_LINE_1_UNFORMATTED;};
7:      TextManager tm = new TextManager(
        service);
8:  /**** Estrutura arrange-act-assert *****/
9:      tm.AddLine(line);
10: /**************************************/
11:     Assert.IsTrue(tm.Lines.Count == 1);
12: }
```

```
1:  @TestMethod
2:  public void AddLine_ValidLine_Success(){
3:      StubTextFormatService service = new
        StubTextFormatService();
4:      String chapter = TestData.
        SAMPLE_CHAPTER_1_UNFORMATTED;
5:      srvice.FormatLineString = (x) =>
6:      {return TestData.
        SAMPLE_CHAPTER_1_UNFORMATTED;};
7:      TextManager tm = new TextManager(
        service);
8:  /**** Estrutura arrange-act-assert *****/
9:      tm.AddChapter(chapter);
10: /**************************************/
11:     Assert.IsTrue(tm.Chapters.Count == 1);
12: }
```

Figura 7 – Dois testes de unidade convencionais. As linhas entre comentários indicam a estrutura *arrange-act-assert*

Fonte: Adaptada de Schimmel *et al.* (2015).

No exemplo apresentado na Figura 7 dois casos de teste são selecionados para um par de métodos que pode executar em paralelo, um teste de unidade paralelo é então definido (Código 3) a partir de ambos os testes. Os dois testes são divididos em três partes: Inicialização, chamada do método sob teste e *assertion*.

Os testes convencionais seguem as boas práticas e utilizam apenas um único objeto *mock* (tm). No entanto, o caso de teste paralelo resultante inclui ambos os códigos dos objetos *mock*. É assumido um objeto *mock* para cada MUT a ser validado, desde que os objetos não sejam usados por vários métodos ao mesmo tempo. As declarações de *assert* em ambos os testes convencionais referem-se a diferentes campos da classe que contém o MUT. Se as declarações de assert testassem o mesmo valor do campo, pelo menos um dos asserts falharia no caso do teste paralelo, pois esse valor agora seria influenciado por dois métodos.

Para essa situação, não é possível reutilizar os *asserts* em uma execução bem sucedida do caso de teste paralelo. Isso explica por que os casos de teste paralelo genéricos sem o enriquecimento ainda são necessários. Ainda assim, pode-se destacar essa situação, pois ocorre quando uma variável de asserção é escrita por ambos os MUTs. Sempre que é detectado essa situação, um caso de teste paralelo é executado com objetivo de revelar a falha.

Código-fonte 3 – Caso de teste paralelo gerado a partir dos casos de teste da Figura 7

Fonte: Adaptada de Schimmel *et al.* (2015).

```
1: @TestMethod
2: public void AddChapter_AddLine_ParallelTest(){
3:     StubTextFormatService service = new StubTextFormatService();
4:     String line = TestData.SAMPLE_LINE_1_UNFORMATTED;
5:     service.FormatLineString = (x) =>
6:     /*************************************************/
7:     {return TestData.SAMPLE_CHAPTER_1_FORMATTED;}
8:     /*************************************************/
9:     TextManager tm = new TextManager(service);
10:    Task.WaitAll(
11:    Task.Factory.StartNew(() => {tm.AddChapter(chapter);})
12:    Task.Factory.StartNew(() => {tm.AddLine(line);})
13:    );
14:    Assert.IsTrue(tm.Chapters.Count == 1);
15:    Assert.IsTrue(tm.Lines.Count == 1);
16: }
```

3.3 Geração automática de dados de teste para programas com passagem de mensagem

Embora o número de requisitos de teste produzidos por programas com memória compartilhada seja grande, o problema de geração de dados de teste não se restringe a esses programas. A comunicação por passagem de mensagem também produz diferentes sequencias de sincronização que são afetadas pelo não-determinismo. Nesse sentido, a geração automática de dados de teste visa contribuir na execução de diferentes pares de sincronização e na detecção de defeitos provenientes desse tipo de comunicação.

Um dos primeiros trabalhos sobre a geração de dados de teste para software concorrente com comunicação por passagem de mensagem foi proposto por Korel, Wedde e Ferguson (1991). O trabalho foi estabelecido por meio da execução dinâmica do programa, onde um escalonador de tarefas de tempo real, métodos de minimização de função e uma análise dinâmica do fluxo de dados eram utilizados no processo de seleção automática dos dados de teste. O processo de geração da abordagem proposta foi definido com base em critérios de caminhos (*path-oriented*).

Na primeira etapa do processo de geração, um caminho do programa concorrente, também visto como uma ramificação de programas concorrentes, é derivado para que

então uma entrada do programa que resulte na execução desse caminho seja produzida. O seletor de caminhos gera, automaticamente, um caminho do programa concorrente seguindo as orientações de um determinado critério de teste. Se a entrada do programa não é encontrada para o caminho selecionado, então um outro caminho é escolhido.

Esse processo é repetido até que um caminho do programa concorrente seja selecionado para o qual uma entrada do programa seja encontrada, ou até que o limite do tempo de busca seja atingido. Na abordagem proposta por Korel, Wedde e Ferguson (1991) o objetivo de encontrar uma entrada para o caminho do programa **P** é alcançado após resolver uma sequência de sub-objetivos, para os quais os métodos de minimização de função são empregados. Para melhor exemplificar esse cenário considere x^0 como entrada inicial do programa, selecionada aleatoriamente, para qual o programa será executado. Se **P** é percorrido, x^0 é a solução para o problema de geração de dados de teste, caso contrário o programa concorrente é executado desde que a execução atual do programa não viole o caminho selecionado **P**, isto é, a execução continua até ocorrer uma violação em algum ramo em uma das tarefas, caso isso ocorra a execução do programa é suspensa.

Os métodos de minimização de função são usados quando durante a execução do programa for observado um fluxo de execução indesejável, nesse caso os métodos localizam automaticamente os valores das variáveis de entrada para as quais o caminho selecionado seja percorrido, esse processo é realizado em tempo de execução. Além disso, uma análise dinâmica do fluxo de dados é usada para determinar as variáveis de entrada responsáveis pelo comportamento indesejável do programa, o que pode levar a uma especificação significativa do processo de busca.

Gong, Tian e Yao (2012) apresentam um método para minimizar os desafios na geração de dados de teste guiada por cobertura de caminhos. Conforme apresentado pelo autor, o critério de cobertura de caminhos oferece uma grande quantidade de requisitos de teste, a depender do tamanho do programa sob teste, que dificulta o processo de geração. Isto é, para cada um dos objetivos de teste será necessário encontrar uma entrada de teste capaz de executá-lo.

A solução apresentada por Gong, Tian e Yao (2012) consiste no agrupamento de caminhos associado à geração de dados de teste por meio de um Algoritmo Genético. O número de caminhos objetivo (*target paths*) em cada grupo é determinado de acordo com um cálculo de similaridade entre os caminhos visando reduzir o número de caminhos diferentes, os quais serão mapeados como objetivos de busca posteriormente.

A similaridade entre os caminhos é dada pela seguinte forma. Considere dois caminhos objetivo P_i e P_j, e seus respectivos nós $P_{i1}, P_{i2}, ..., P_{i|p_i|}$ e $P_{j1}, P_{j2}, ..., P_{j|p_j|}$, onde $|p_i|$ e $|p_j|$ representam o comprimento dos caminhos P_i e P_j, respectivamente. Compara-se P_j e P_i de P_{j1} para verificar se ambos possuem os mesmos nós, e determinar o número de nós sucessivos equivalentes, como $|P_i \cap P_i|$. A similaridade entre $|p_i|$ e $|p_j|$ é definida

como a razão do número de nós equivalentes sucessivamente para o maior comprimento do caminho e denotada como $s(P_i, P_j)$, cuja expressão é a seguinte:

$$s(p_i, p_j) = \frac{|P_i \cap P_j|}{max\{|p_i|, |p_j|\}} \tag{3.1}$$

A ideia do método proposto para agrupar os caminhos objetivo pode ser descrita da seguinte forma. O número dos caminhos objetivo em cada grupo é determinado de acordo com os recursos de cálculo disponíveis para reduzir a diferença do número de caminhos objetivo em diferentes grupos. Os caminhos objetivo pertencentes a um grupo são determinados de acordo com as semelhanças dos caminhos para melhorar a semelhança entre o caminho base e os demais no grupo.

Como exemplo, suponha que exista recursos de cálculo de N_r disponíveis e eles são homogêneos. Assim, nós Nr com a mesma configuração podem ser usados para gerar simultaneamente dados de teste que cobrem os caminhos objetivo. Para reduzir o consumo de tempo gasto na geração de dados de teste, a carga desses recursos de cálculo deve ser balanceada. Em outras palavras, espera-se gerar dados de teste usando diferentes recursos..

Após todos os caminhos objetivos serem divididos em N_r grupos, um problema de otimização de múltiplos objetivos é formulado com base nos caminhos objetivo de cada grupo. O problema original de otimização m-objetivo, pode ser convertido em N_r problemas de sub-otimização, e o i-ésimo tem $|g_i|$ objetivos que correspondem à geração de dados de teste que cobrem os caminhos de destino do i-ésimo grupo.

Tian e Gong (2016) apresentam um método de geração de dados de testes para programas paralelos com passagem de mensagem. Nesse método um algoritmo genético co-evolucionário é empregado para geração dos dados de teste seguindo as regras estabelecidas por critérios de cobertura de caminho.

Algoritmos Genéticos Co-evolucionários foram propostos aplicando-se a ideia da co-evolução, ou seja, influências mútuas entre a evolução das plantas e insetos comedores de plantas. Existem dois tipos básicos de classes de algoritmos genéticos co-evolucionários: Co-evolução competitiva, em que o *fitness* de cada indivíduo é determinado por uma série de competições com indivíduos de outras populações, e Co-evolução cooperativa, na qual o *fitness* de cada indivíduo é determinado por uma série de colaborações com indivíduos de outras populações (WIEGAND; LILES; JONG, 2001).

O AG co-evolucionário para geração de dados de teste é constituído por dois tipos de população: diversas sub-populações e população cooperativa. Essas populações envolvem diferentes caminhos, assim as etapas de evolução de cada tipo de população são realizadas separadamente, conforme apresentado na Figura 8.

Apenas as variáveis de decisão relacionadas com p^i (caminhos percorridos pela

Figura 8 – Etapas do método co-evolucionário. **A** Etapas evolucionárias da sub-população M^i. **B** Etapas evolucionárias da população *GM*

Sub-população M^i

Passo 1: Inicializar a população
Passo 2: Julgue se o número máximo de gerações foi atingido ou não. Se sim, encerre o algoritmo. Caso contrário, decodifique todos os indivíduos como entrada de S e calcule o fitness dos indivíduos de acordo com a Equação 3.2.
Passo 3: Julgue se o período evolutivo definido foi atingido ou não. Se sim, selecione um número de indivíduos representativos como EM^i definido, envie EM^i ao *GM* e receba os indivíduos dominantes de *GM*. Caso contrário, vá para o passo 5.
Passo 4: Julgue se os indivíduos recebidos são NULOS ou não. Se sim, encerre o algoritmo.
Passo 5: Executar operadores genéticos, seleção, cruzamento e mutação. Ir para a Passo 2.

A

População cooperativa *GM*

Passo 1: Receber EM^0, EM^1, ..., EM^{m-1} e constituir *GM*.
Passo 2: Julgar se o período evolutivo definido foi alcançado. Se sim, vá para o Passo 5. Caso contrário, decodifique os indivíduos como entrada de *S*, execute *S* para obter o caminho percorrido e calcule o fitness dos indivíduos de acordo com a fórmula 3.3.
Passo 3: Julgar se o fitness de um indivíduo é igual a um ou não. Se sim, produza o indivíduo, envie *NULL* para M^0, M^1, ..., M^{m-1} e encerre o algoritmo.
Passo 4: Execute os operadores genéticos, seleção, cruzamento e mutação. Ir para a Passo 2.
Passo 5: Decompor os indivíduos dominantes e enviar as variáveis de decisão correspondentes para M^0, M^1, ..., M^{m-1}

B

Fonte: Adaptada de Tian e Gong (2016).

entrada de teste) são otimizadas por M^i. No método, o caminho percorrido é obtido por meio da execução de S^i (i-ésimo processo do programa paralelo *S*) com o indivíduo decodificado sendo a entrada de S^i. O *fitness* é então calculado para cada indivíduo de acordo com a Fórmula 3.2 (**A**-Passo 2). Quando M^i evoluiu por um determinado período, uma série de melhores indivíduos são selecionados como representantes. Esses Indivíduos são enviados para *GM* (população cooperativa), e então os indivíduos dominantes de *GM* são recebidos (Step 3).

$$F^i(X^i) = \frac{\left|p^{*i} \cap p^i\right|}{max\{|p^{*i}|, |p^i|\}} \tag{3.2}$$

Se um valor nulo é recebido, então a evolução de M^i é finalizada, indicando que um dado de teste desejado foi encontrado por GM (**A**-Passo 4). Por outro lado, as operações genéticas são realizadas para gerar indivíduos descendentes por meio informações genéticas da população atual (**A**-Passo 5).

Quando os indivíduos representativos a partir dessas sub-populações são recebidos,

os indivíduos iniciais de ***GM*** são constituídos pelo método de indivíduos cooperativos (**B**-Passo 3), e evoluiu para gerar dados de teste cobrindo o caminho alvo. Em cada geração de um dado período evolutivo, o caminho percorrido é obtido juntamente com a execução do programa paralelo com cada indivíduo decodificado, e o *fitness* de cada indivíduo é calculado de acordo com a Equação 3.3 (**B**-Passo 2).

$$F(X) = \frac{1}{m} \sum_{i=0}^{m-1} s(p^{*i}, p^{i}) \tag{3.3}$$

Se houver um indivíduo cuja aptidão seja igual a um, ele é emitido como o dado de teste desejado e o valor de *NULL* é enviado para todas as subpopulações para finalizar suas evoluções, ao mesmo tempo, a evolução da ***GM*** também é encerrada (**B**-Passo 3). Caso contrário, as mesmas operações genéticas, como cada sub-população, são também realizadas para gerar seus descendentes (**B**-Passo 4). Uma vez que ***GM*** evoluiu para um dado período evolutivo, os indivíduos dominantes são decompostos e enviados para as sub-populações correspondentes de acordo com as variáveis de decisão otimizadas.

Apesar das contribuições do modelo co-evolucionário , ainda há um grande desafio em relação ao custo do processo de otimização. O teste dinâmico de aplicações concorrentes possui um custo expressivo para atividade de teste, o qual é intensificado pela verificação de suas possíveis sincronizações.

Nessa linha, Khanna, Purandare e Sharma (2020) elencam alguns desafios no teste dinâmico de aplicações concorrentes, a saber: i) mecanismos de verificação são prejudicados pelo problema de escalabilidade devido ao tamanho do espaço de estados alcançáveis que cresce exponencialmente a medida que o número de entidades paralelas aumenta; ii) dependência da qualidade dos dados de teste para eficiência na verificação e na detecção de defeitos concorrentes; e iii) ausência de um *benchmark* significativo de defeitos para validar as abordagens e ferramentas propostas nessa direção.

Diante disso, Khanna, Purandare e Sharma (2020) apresentam contribuições para verificação dinâmica de software concorrente com foco nas seguintes questões de pesquisa:

- **Q1:** A combinação de *Constraint-solving* e verificação dinâmica pode contribuir para um teste escalonável e, assim, garantir a ausência de *deadlocks* em programas concorrentes?
- **Q2:** Como sintetizar estrategicamente casos de teste inteligentes que possam conduzir mecanismos de verificação dinâmica para explorar o espaço de escalonamento de estados corretamente?
- **Q3:** Como sintetizar defeitos reais e profundos em programas concorrentes para avaliar a qualidade de ferramentas de verificação dinâmica em um grande número

de sujeitos?

O primeiro problema (Q1) é explorado no contexto de programas concorrentes com comunicação por passagem de mensagem, em especial programas desenvolvidos em MPI (*Message Passing Interface*). A técnica desenvolvida para esse problema combina aspectos das técnicas de verificação dinâmica e verificação de rastros de execução, que fornecem escalabilidade e cobertura de múltiplos caminhos sobre essas técnicas, respectivamente.

O método híbrido, estabelecido para identificar *deadlocks* em programas de múltiplos caminhos em MPI, explora exaustivamente as execuções do programa sob uma entrada fixada seguinte forma: i) primeiramente uma execução concreta ρ do programa é obtida por meio da análise dinâmica, para isso utiliza-se um escalonador que orquestra a execução; ii) então o conjunto de execuções viáveis (*feasible runs*) é codificado simbolicamente a partir do mesmo conjunto de eventos observados em ρ, de modo que cada processo desencadeia as mesmas decisões do fluxo de controle e executa a mesma sequência de chamadas de comunicação, tal como ocorre em ρ; iii) em seguida, verifica-se a presença de violações de qualquer propriedade (nesse caso, *deadlocks* de comunicação); e iv) por último, se nenhuma propriedade for violada, a codificação simbólica é alterada para explorar a viabilidade de tomar um caminho de fluxo de controle alternativo que seja diferente de ρ. Nesse caso, inicia-se a execução concreta desse fluxo alternativo.

A segunda questão (Q2) é investigada no contexto de aplicações Java *multi-thread*. Uma abordagem é proposta para sintetizar casos de teste para revelar problemas de *deadlocks* nesses programas, a qual foi implementada em um protótipo denominado REVELIO. A abordagem possui duas etapas principais: i) primeiramente encontra-se rastros distintos de duas execuções *single-thread*, posteriormente determina-se se a combinação desses rastros pode revelar um *deadlock* presente no programa sob teste; e ii) se o defeito puder de fato ser revelado, um registro dos rastros e dos parâmetros de entrada são armazenados para compor um caso de teste *multi-thread*.

Para a última questão (Q3), os autores conduzem um estudo de revisão para identificar características e padrões de defeitos concorrentes na literatura. As descobertas revelam que muitos *benchmarks* identificados na literatura já são identificados por ferramentas existentes, contudo, não há direcionamento para defeitos que ainda não possuem soluções propostas. Os autores também ressaltam a importância de mecanismos que atuem não somente no espaço de busca de entradas, mas também no espaço de *interleavings*. Com base nessas descobertas, Khanna, Purandare e Sharma (2020) apresentam aspectos de qualidade que julgam necessários para proposição de uma técnica de sintetização de defeitos concorrentes, a saber:

1. O defeito deve ser manifestado com uma entrada real do programa, ou seja, uma

entrada que não necessita de configurações ou mudanças específicas para ser executada;

2. O defeito deve ser raro, somente deve ser manifestado com *interleavings* específicos de uma entrada em particular;

3. Os defeitos devem estar distribuídos randomicamente no espaço de busca, dessa forma não favorece alguma técnica de detecção em particular.

As contribuições citadas avançam significamente no teste de programas concorrentes, em especial para as etapas que compõem a geração automática de dados de teste. Os defeitos de aplicações concorrentes não se limitam aos *deadlocks* de comunicação, há diferentes tipos de defeitos que podem não ser endereçados com o uso das contribuições apresentadas.

Em vista disso, em um estudo preliminar desta tese Vilela *et al.* (2019), foi proposta uma abordagem de otimização bio-inspirada que empregou um algoritmo genético na geração de dados de testes para programas com comunicação por passagem de mensagem e memória compartilhada. A abordagem sintetiza critérios estruturais de programas concorrentes em duas funções fitness para ambos os paradigmas de comunicação.

A principal característica dessa abordagem é a preservação de indivíduos com baixo valor de fitness para novas gerações. Nessa etapa do trabalho observou-se que alguns indivíduos ruins, em relação ao valor de *fitness*, possuíam características singulares em relação aos requisitos de teste. Em outras palavras, alguns requisitos de teste contemplados por indivíduos de baixo valor de *fitness*, em uma função de maximização, não eram alcançados pelos melhores indivíduos de uma população. Desse modo, a proposta foi elaborada com a premissa de que novos indivíduos melhores poderiam ser gerados se uma parcela de indivíduos antes considerados ruins pudessem ser selecionados para futuras gerações.

Um protótipo foi desenvolvido para validar a aplicação da abordagem, denominado BioConcST (Versão 1.0). A validação considerou aplicações provenientes de um conjunto de *benchmarks* comparando a abordagem proposta com uma abordagem elitista, na qual somente os indivíduos de melhor fitness são considerados em futuras gerações. Uma visão geral da abordagem é apresentada no Algoritmo 3.

Em sua primeira versão, a abordagem BioConcST requer como entrada um software concorrente sob teste (*SUT*) e uma *flag* que determina o paradigma de comunicação do programa sob teste, podendo variar entre passagem de mensagem e memória compartilhada. Como saída, o algoritmo fornece um conjunto de teste otimizado que consiste nos dados de testes obtidos pelo processo de otimização.

Algoritmo 3 – Visão Geral BioConcST (Versão 1.0).
Fonte: Adaptada de Vilela *et al.* (2019).

```
1:  procedimento OptimizeTestSuite(SUT, CP)
2:      TS ← InitializeRandomPopulation
3:      PopEvaluation(TS)
4:      enquanto StopCondition is not TRUE faça
5:          Parents ← selectParents(TS, TS_size)
6:          para P1, P2, ∈ Parents faça
7:              Ch1, Ch2 ← Crossover(P1, P2, C_method, C_rate)
8:              Children.Push(Ch1, Ch2)
9:              Children.Push(Mutation(Ch1, M_method, M_rate))
10:             Children.Push(Mutation(Ch2, M_method, M_rate))
11:         fim para
12:         PopEvaluation(Children)
13:         TempPOP.Push(GetBestSolution(Children))
14:         TempPOP.Push(GetWorstSolution(Children))
15:         TempPOP.Push(Tournament(Children))
16:         TS ← Replace(TS, TempPOP)
17:     fim enquantoretorna TS
18: fim procedimento
```

No primeiro estágio (linha 2), uma população aleatória de dados de teste é gerada, que será o ponto de partida para otimizar o problema. A população consiste em um indivíduo (ou cromossomos) que correspondem aos dados de teste (ou entradas de teste) do programa sob teste. Os dados de teste consistem em um ou mais parâmetros de entrada, cada parâmetro sendo considerado um gene nessa abordagem.

No próximo estágio (linha 3), uma avaliação de *fitness* da população é realizada para que os indivíduos possam ser categorizados em termos de sua capacidade de atingir uma meta de teste (por exemplo, uma meta de teste deve cumprir um critério de teste específico). O critério de parada dessa abordagem inclui uma população capaz de atingir 100% de cobertura para todos os critérios de teste. Se o critério de parada não for atendido, o processo evolutivo continua até que um número máximo de gerações (previamente definido) seja alcançado.

Para ampliar a busca por indivíduos, dois tipos de operadores genéticos são usados para recombinação (C_{method}). Quando dois indivíduos são selecionados para recombinação, seguindo um valor de taxa (C_{rate}), um método aleatório define qual será usado para recombinação. O objetivo dessa estratégia é tornar a busca mais dinâmica ao longo das gerações. O primeiro método de recombinação é o cruzamento de um único ponto, por meio do qual, após a recombinação, dois novos indivíduos filhos são gerados. No segundo método, um valor aleatório é escolhido para cada pai (intervalo de 0 a 1) e, em seguida, um gene do cromossomo é selecionado aleatoriamente e modificado. Este método pode permitir uma maior diversidade na busca, uma vez que aceita um pequeno grau de aleatoriedade no

processo de recombinação, ao contrário do primeiro método.

A mutação também ocorre por meio de diferentes operadores (M_{method}), seguindo uma taxa previamente definida (M_{rate}). No primeiro caso, um gene é selecionado e um valor aleatório é redefinido para esse gene. No segundo, dois genes são selecionados no mesmo cromossomo e posteriormente a posição entre os mesmos é invertida.

Uma população *Children* é gerada a partir dos operadores genéticos e uma avaliação de aptidão é necessária para medir a aptidão dos novos indivíduos. Em seguida, os melhores e os piores indivíduos dessa população são selecionados para formar a nova geração, e os demais indivíduos são selecionados pelor operador de seleção Torneio. Como resultado, o processo de otimização é iterado até que o critério de parada seja alcançado. Um conjunto de teste otimizado é obtido após os ciclos de gerações.

A abordagem foi avaliada em um estudo experimental considerando um conjunto de *benchmarks* desenvolvidos em Java. Para fins de comparação, uma abordagem elitista foi utilizada sob os mesmas configurações da abordagem BioConcST. Os resultados indicaram que a seleção de uma parcela de indivíduos ruins contribui para o processo de otimização, pois em todos os critérios avaliados a abordagem BioConcST teve resultados superiores e significativos.

3.4 Considerações Finais

Diante dos trabalhos apresentados percebe-se uma preocupação no teste de aplicações concorrentes devido ao alto nível de dificuldade imposto para atividade de teste, esse interesse surge, principalmente, devido a presença cada vez maior de código concorrente nas aplicações e também na dificuldade de identificar os defeitos dos programas.

Os trabalhos descritos nas seções anteriores investigam a geração de teste de forma distinta e também em momentos diferentes do processo de desenvolvimento de software. Alguns trabalhos vêm explorando a especificação do software para derivar seus testes (SUN *et al.*, 2015; MAHALI *et al.*, 2016; ARORA; BHATIA; SINGH, 2017), entretanto, existem características dos programas que estão intrinsecamente ligadas a nível de abstração da linguagem de programação que podem ocasionar defeitos, assim acredita-se que esses métodos são importantes, pois verificam o software no início do desenvolvimento, mas uma verificação do software após o seu desenvolvimento também é considerada fundamental.

Trabalhos que empregam execução simbólica para a geração dos testes também são utilizados para o teste de programas concorrentes (ALBERT *et al.*, 2014; GUO *et al.*, 2015; GUO; KUSANO; WANG, 2016), essa técnica contribuiu para que diversos trabalhos voltados a programas sequencias, e alguns concorrentes, obtivessem bons resultados para geração, entretanto, as abordagens que empregam esse método não consideram alguns

aspectos concorrentes importantes para a verificação dos programas, tais como sequencias de execução e sincronização. Apesar disso, acredita-se que esse método pode contribuir, em conjunto com outras técnicas de geração, para uma verificação mais precisa.

Outro tipo de abordagem utilizada para a geração dos testes são os métodos de otimização por meio de meta-heurísticas. Essa área é amplamente explorada para programas sequencias e recentemente ganhou atenção para programas concorrentes (GERONIMO *et al.*, 2012; TIAN; GONG, 2013; TIAN; GONG, 2016). Os trabalhos propõem otimizações para o problema da seleção de entradas de teste, entretanto, um desafio para este problema é avaliação dos indivíduos que podem obter níveis de aptidão diferentes em execuções distintas devido ao não-determinismo. Em razão desse obstáculo muitas abordagens deixam de considerar os efeitos causados ou utilizam estratégias de minimização do problema, como por exemplo a repetição de execuções com a mesma entrada de teste. Ambos os casos podem não garantir uma correta seleção pois são sustentados pela casualidade, ou seja, um defeito poderá ou não ser revelado durante as execuções. Em vista dos aspectos observados, este estudo visa avançar na atividade de geração automática de dados de teste empregando uma abordagem de otimização bioinspirada. O próximo capítulo descreve em detalhes os conceitos e as etapas dessa abordagem.

CAPÍTULO

4

BIOCONCST - OTIMIZAÇÃO BIOINSPIRADA PARA O TESTE DE SOFTWARE CONCORRENTE

4.1 Considerações Iniciais

A geração automática de dados de teste é amplamente investigada na literatura sendo caracterizada como um problema crítico e indecidível, independentemente da técnica de teste aplicada. Associado a esse problema, o teste de programas concorrentes, o qual também se trata de um problema indecidível, apresenta novos desafios para automação da etapa de seleção de dados de teste.

Considerando a inexistência de uma solução ótima para o problema da seleção de dados de teste, os estudos relacionados empregam técnicas de otimização, como meta-heurísticas, para obtenção de uma solução aceitável. Uma questão chave a ser explorada nesse contexto é a forma como ocorre a avaliação dos testes gerados automaticamente. Usualmente, critérios de testes são empregados nessa avaliação com objetivo de guiar a otimização e/ou como critério de parada.

Apesar da grande contribuição dos critérios de testes, o processo de otimização deve considerar outros aspectos que permitam uma otimização contínua. Técnicas de otimização bio-inspiradas, como algoritmos genéticos e enxame de partículas, necessitam de características distintas para uma evolução satisfatória, caso contrário a otimização pode ser prejudicada (HARMAN *et al.*, 2007; Gay, 2017). Nesse sentido, o fornecimento de um feedback significativo para essas técnicas é um dos desafios deste projeto.

A otimização bioinspirada consiste em um termo guarda-chuva que abrange uma variedade de abordagens que são baseadas nos princípios dos sistemas biológicos, tais

como otimização por colônia de formigas (*colony opti-mization - ACO*), otimização por colônia de abelhas (*bee colony optimization - BCO*), algoritmos evolucionários, como o Algoritmo Genético (RAI; TYAGI, 2013). Esse tipo de otimização tem sido utilizada com sucesso para apoiar diferentes áreas do desenvolvimento de software, tais como: i) priorização de requisitos, design, reengenharia e engenharia reversa, inspeção, medição de software e seleção de dados de teste (HARMAN; JONES, 2001; ALI *et al.*, 2020). No contexto deste trabalho, uma abordagem é proposta visando apoiar a automatização do teste de programas concorrentes, com ênfase na seleção automática de dados de teste.

O capítulo está organizado da seguinte forma, na Seção 4.2 apresenta-se a proposta da abordagem bioinspirada no teste de programas concorrentes. Na Seção 4.3 é descrito o operador genético FuzzyST proposto para seleção de indivíduos. Por último são apresentadas as considerações finais deste capítulo na Seção 4.4.

4.2 Descrição da abordagem BioConcST

Tendo em vista a otimização do processo de busca para o problema da seleção de entradas de programas concorrentes, este estudo propõe uma abordagem bioinspirada, denominada BioConcST (*Bio-inspired Optimization for Concurrent Sofware Testing*). A BioConcST utiliza um Algoritmo Genético para o processo de otimização e acrescenta estratégias que visam apoiar o teste de aplicações concorrentes.

Antes de apresentar as definições e etapas desta proposta, é importante definir alguns conceitos importantes. A entrada de teste para a otimização proposta neste trabalho é composta pelo dado de teste juntamente com um caminho de execução percorrido pelo mesmo. Assim. uma entrada de teste (I) para otimização é dada por $I = \{X, \Pi\}$, tal que X seja o dado de teste composto por todos os argumentos (x) de entrada do programa sob teste $X = \{x_1, x_2, x_3, ..., x_n\}$ e Π corresponde ao caminho de execução percorrido por X. Um caminho Π é composto por uma sequência lógica de execução $\Pi = \{\pi_1, \pi_2, \pi_3, ..., \pi_n, \}$, onde π é dado por uma tupla $\pi = \{p, t, \delta\}$ composta pelo processo (p) e *thread* (t) executados e a ordem de execução (δ) no espaço lógico de tempo.

A seleção da entrada de teste é realizada por meio de um processo de busca metaheurística, onde I é dado como um indivíduo de uma população. O processo de busca permite evoluir desde uma população inicial, formada por dados de teste aleatórios, até uma população ótima (ou sub-ótima) constituída por um conjunto de I_s (população) otimizados. Um dado de teste I é codificado em um Algoritmo Genético que representa o indivíduo em uma hierarquia de dados, a qual descreve as diferentes unidades de dados de I. Na Figura 9, apresenta-se um modelo conceitual que ilustra essa hierarquia.

Assim como na Genética, o Gene na hierarquia de uma entrada de teste para a otimização constitui a unidade fundamental de um indivíduo, entretanto, diferentemente

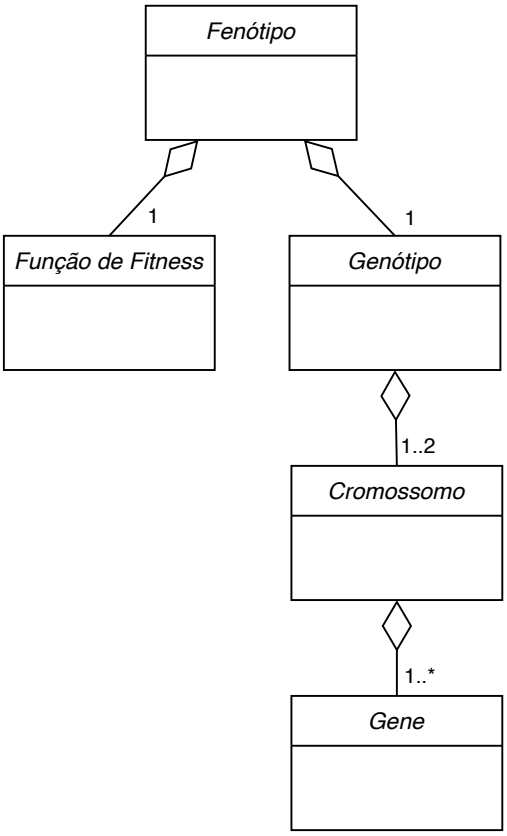


Figura 9 – Hierarquia de unidades de dados da entrada de otimização *I*

da genética tradicional, o Gene de *I* somente pode ser representado por dois tipos de alelos[1]. O primeiro refere-se aos argumentos de uma entrada do programa sob teste, neste caso o alelo poderá assumir qualquer valor do domínio de entrada da aplicação, como exemplo podem-se citar os conjuntos numéricos e o conjunto de caracteres alfanuméricos. Por último, o alelo também poderá assumir as unidades de caminho de execução. Neste trabalho essas unidades são representadas por nós e arestas de um Grafo de Fluxo de Controle Paralelo (GFCP).

As unidades de Gene constituem o Cromossomo de uma solução. Cada Cromossomo é formado por um ou mais genes que podem caracterizar tanto a entrada do programa quanto o caminho de execução, sendo que esses dois tipos de cromossomos representam a solução. O tamanho de cada tipo de Cromossomo é variável para atender diferentes configurações e domínios de aplicação, sendo assim é possível aplicar a seleção para diferentes tipos de programas sob teste. Além disso, o Cromossomo é a unidade essencial para aplicação dos operadores de mutação e recombinação (*crossover*), por meio dessa coleção de Genes, novos indivíduos podem ser gerados com características hereditárias.

A combinação entre os cromossomos de entrada de teste e caminho de execução constituem o Genótipo da solução. Na Figura 10, apresenta-se a estrutura do Genótipo

[1] Alelos são formas alternativas de um determinado gene.

contendo as informações genéticas do dado de teste para o programa concorrente. Um Genótipo consiste de um ou dois cromossomos e o número total de genes é obtido pela soma dos genes de todos os cromossomos. Ainda é importante ressaltar que um Cromossomo pode apresentar tamanhos distintos em um mesmo Genótipo, pois cada cromossomo em um genótipo representa informações genéticas diferentes.

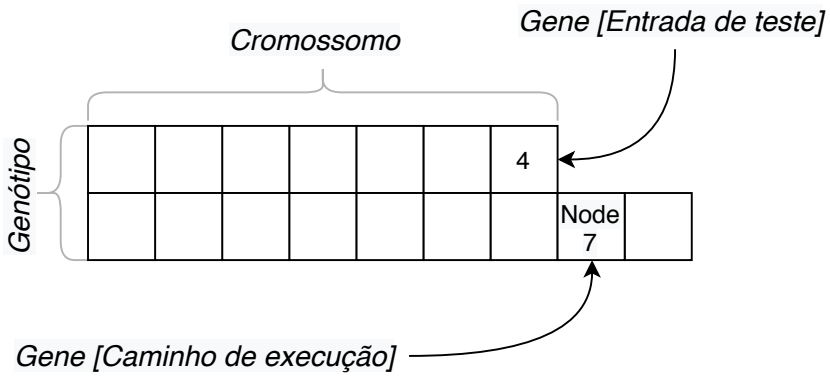


Figura 10 – Estrutura do Genótipo do dado de teste concorrente

A variação no número de cromossomos (1 ou 2) em um Genótipo ocorre em decorrência da aquisição do Cromossomo de caminho da execução, pois este somente é alcançado após a avaliação de fitness do indivíduo, em outras palavras, apenas é possível obter o caminho, e consequentemente a formação de seu cromossomo, após ao menos uma execução dinâmica do dado de entrada no programa sob teste.

A formação final de um indivíduo I é representada pela unidade Fenótipo. O Fenótipo, assim como na genética, resulta da expressão dos Genes no ambiente, um mesmo Genótipo pode expressar características diferentes devido a condições e propriedades do ambiente, o que neste contexto trata-se do software sob teste e das condições imprevisíveis durante a sua execução. Essa característica possui forte relação com o não-determinismo presente nas aplicações concorrentes, uma vez que, um mesmo dado de teste pode executar caminhos distintos e livre de defeitos. A composição do Fenótipo possui todas as informações genéticas do indivíduo bem como a sua aptidão ao ambiente, essa aptidão, também conhecida como *fitness*, é mensurada por uma função de avaliação que fornece um valor de aptidão para cada indivíduo avaliado. Na seção seguinte, descreve-se em detalhes o processo proposto para avaliação dos indivíduos.

4.2.1 Função de fitness para avaliação de dados de teste para programas concorrentes

Conforme destacado anteriormente, em um estudo prévio, Vilela *et al.* (2019) investigou a aplicabilidade de critérios concorrentes como parâmetros de avaliação para uma função de *fitness*. Os resultados foram satisfatórios quando comparados com uma aborda-

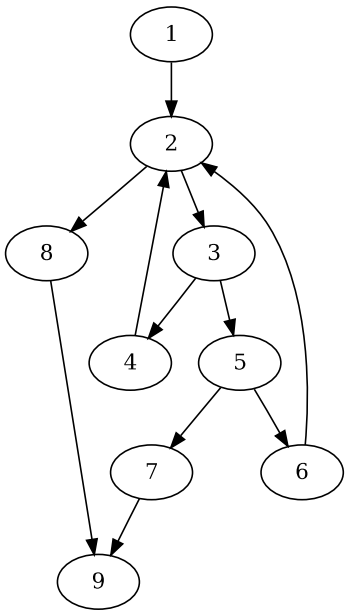
gem elitista, entretanto, observou-se ainda, que existem desafios para o cumprimento dos objetivos de teste. Um dos desafios identificados foi a estratégia de avaliação de *fitness*. Essa avaliação emprega lógica binária (requisito coberto ou não coberto) para diferenciar os indivíduos, deixando de capturar informações relevantes sobre os mesmos. Embora a seleção de uma parcela de indivíduos ruins para próximas gerações contribua para melhorar o processo evolucionário, essa estratégia, por si só, não garante que esses indivíduos possuem realmente características importantes.

Considerando essas limitações, esta abordagem visa mensurar e analisar a distância entre o caminho percorrido pela entrada e o requisito de teste como estratégia de avaliação de *fitness*. Em outras palavras, a abordagem visa minimizar a distância entre esses elementos, assumindo que dessa forma seja possível diferenciar dois ou mais dados de teste que não alcançaram um requisito de teste, e assim otimizar a atribuição do valor de *fitness* para os dados de teste.

Para melhor ilustrar esse cenário, na Figura 11, apresenta-se um trecho de código do método busca binária e seu respectivo grafo de fluxo de controle. A busca binária trata-se de um algoritmo com objetivo de encontrar um elemento específico em uma lista ordenada de elementos. O algoritmo recebe como entrada um vetor ordenado de elementos (**int** *v*[ ]), o número de elementos contidos no vetor (**int** *n*) e o valor objetivo (**int** *x*), para o qual a busca é realizada. O processo de busca divide a lista ao meio repetidamente, em cada divisão é determinada uma sublista onde o elemento se encontra, o processo termina quando a sublista possui apenas um elemento, ou seja, o elemento procurado.

Figura 11 – Grafo de fluxo de controle do método Busca Binária

```
1   int binsearch(int x, int v[],
      int n) {
2  /**1**/ int low, high, mid;
3  /**1**/ low = 0;
4  /**1**/ high = n - 1;
5  /**2**/ while (low <= high) {
6  /**3**/   mid = (low + high) / 2;
7  /**3**/   if (x < v[mid])
8  /**4**/     high = mid - 1;
9  /**5**/   else if (x > v[mid])
10 /**6**/     low = mid + 1;
11 /**7**/   else
12 /**7**/     return mid;
13       }
14 /**8**/ return -1;
15   } /**9**/
```



Fonte: Adaptada de Bowring (2006).

Considere, para o cenário ilustrado, que existe um nó associado a cada requisito de teste. Assuma que um dado requisito de teste no Nó 4 (Linha 8) ainda não foi alcançado por nenhum dado de teste. Sendo assim, o desafio será encontrar um dado de teste onde o valor procurado seja menor que valor o armazenado na posição $mid(v)$, conforme a condição **if** $(x < v[mid])$ ilustrada na Figura 11.

Para representar a distância entre o elemento requerido (Nó 4) e o caminho percorrido, assume-se que a distância é dada pelo número de nós existentes, e alcançáveis, entre o elemento requerido e o nó percorrido mais próximo desse elemento, incluindo o nó objetivo. Como exemplo, considere como dado de teste, para o algoritmo de busca binária, o vetor ordenado $v[\,] = \{3,5,6,7,8,10,14,19\}$ de tamanho $n = 8$, tal que 10 seja o elemento procurado. Após sua execução, o caminho percorrido é dado por C_{Ei} $\{1,2,3,5,6,2,3,5,7,9\}$, onde C_{Ei} representa o conjunto de nós, em sequência, do caminho de execução do dado de teste I. Neste exemplo, o nó 3 apresenta a menor distância alcançável entre todos os nós percorridos pelo dado de teste, apenas um nó separa o caminho percorrido do objetivo de teste, essa distância é o valor usado para definir quão longe um dado de teste está de seu objetivo.

Em outro cenário, ainda assumindo que o nó 4 não foi alcançado, considere como dado de teste o vetor vazio $v[\,] = \{\}$ de tamanho $n = 0$, tal que 10 seja o elemento objetivo. Neste exemplo, o caminho percorrido é dado por C_{Ei} $\{1,2,8,9\}$, sendo o nó 2 o mais próximo do elemento requerido, dois nós separam este nó do objetivo de teste. No contexto de geração de dados bioinspirada, percebe-se que, apesar de ambos dados de teste não atingirem seus objetivos, o primeiro possui informações genéticas mais adaptativas ao ambiente (requisito de teste), sendo assim, os descendentes desse indivíduo possuem maior expectativa para solucionar o problema de busca.

A distância de um indivíduo em relação ao elemento requerido é medida por meio de um algoritmo que, empregando a busca em largura, percorre o GFC na procura do nó executado mais próximo ao elemento requerido de teste. A busca em largura realiza a travessia no GFC a partir de um nó raiz, que consiste no nó associado ao elemento requerido, visitando todos os nós vizinhos da raiz, depois todos os vizinhos dos vizinhos, e assim por diante. Cada nó visitado é verificado quanto à sua presença ou ausência em C_{Ei}. Se presente, há a finalização da busca e o armazenamento do valor de distância. Se ausente, a busca continua até atender essa condição, ou seja, um nó presente em C_{Ei}. Um pseudo-código da função *Distance*, que desempenha essas etapas, é apresentado no Algoritmo 4.

O GFC do programa sob teste é armazenado em uma matriz de adjacência, essa matriz dispõe os nós e arestas conforme estruturado no grafo do programa sob teste, com algumas exceções. O nó de origem é dado pelo nó associado ao requisito de teste, e não mais pelo nó de origem do programa. Além disso, os nós vizinhos são formados pelos

Algoritmo 4 – Cálculo da distância do elemento requerido de teste

```
1:  procedimento DISTANCE(CFG, Path, Target)                ▷ Calculates distance
2:      distance ← 1
3:      Q ← Target                                          ▷ Q: Queue
4:      enquanto Q is not empty faça
5:          u ← Remove(Q)
6:          para cada node ∈ u → adj faça                   ▷ adj: Neighboring nodes
7:              se node is not visited então
8:                  se node ∈ Path então
9:                      node.SETDISTANCE(distance)
10:                     retorna node                        ▷ Returns the node and its distance
11:                 senão
12:                     node.ISVISITED(true)
13:                     ENQUEUE(Q, node)
14:                 fim se
15:             fim se
16:         fim para
17:         INCREMENT(distance)
18:     fim enquanto
19:     retorna -1                                          ▷ Wrong entry
20: fim procedimento
```

nós que possuem arestas dirigidas ao nó raiz, depois os nós com arestas dirigidas aos nós vizinhos, e assim por diante.

Conforme descrito anteriormente, o cerne da avaliação do indivíduo está na comunicação entre processos/*threads*, portanto, o critério de teste *all-sync-edges* é responsável por estabelecer quais serão os elementos requeridos de um programa concorrente sob teste. A aplicação do critério incorpora uma representação do programa concorrente em um GFCP. Nessa representação, existem dois nós associados a um elemento requerido, ou seja, nó remetente (*send*) e nó destinatário (*receive*). Contudo, existem dois GFCs associados na troca de uma mensagem, pois o GFCP representa cada *thread* existente, como um GFC do programa sob teste.

Em decorrência disso, para todo e qualquer elemento requerido, do critério de teste *all-sync-edges*, será necessário calcular a distância do objetivo de teste em cada ponta dessa comunicação. Portanto, a distância de *I* em relação a um elemento requerido será dada pela média entre as distâncias dos nós *send* e *receive*. Além disso, o valor final de distância de um indivíduo será composto pelo resultado da média aritmética entre as distâncias de todos os elementos requeridos do programa sob teste. Uma ilustração desse cenário é apresentada na Figura 12.

O valor de *fitness* dado pelo cálculo de distância estabelece quão boa é uma entrada de teste em relação aos requisitos de teste. Essa função de fitness permite explorar diferentes caminhos do programa concorrente, ainda que com uma mesma entrada do pro-

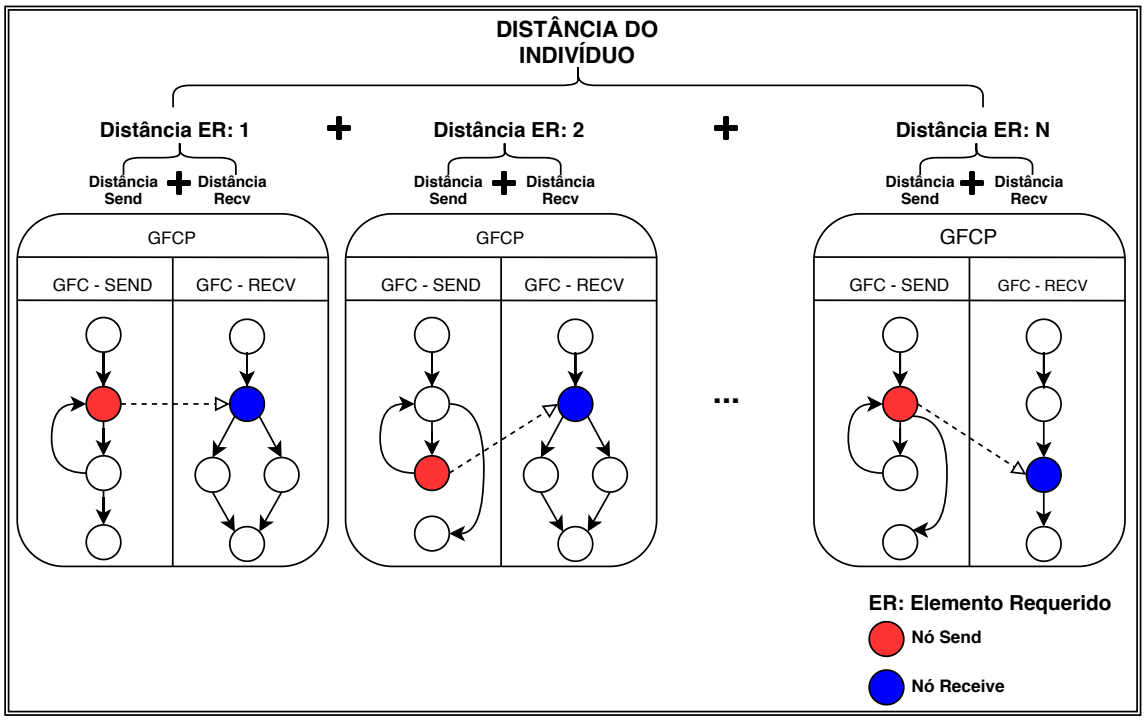


Figura 12 – Composição do valor de fitness de um dado de teste baseado na distância para programas concorrentes

grama. As características mais revelantes dos indivíduos são priorizadas em um processo de otimização evolucionária. Nessa otimização, os melhores indivíduos terão maior chance de passar suas características, ou até mesmo permanecerem, nas próximas gerações. Na seção seguinte, descrevem-se as etapas necessárias do processo evolutivo para obtenção de um conjunto de teste otimizado

4.2.2 Algoritmo Genético multithreading para geração de dados de teste para programas concorrentes

A abordagem BioConcST emprega o Algoritmo Genético para criar uma população de soluções candidatas (i.e., um conjunto de dados de teste concorrente), guiada pela função de *fitness* descrita anteriormente. Em adição ao cálculo de *fitness*, a BioConcST realiza uma busca dinâmica empregando operadores genéticos durante o processo evolutivo. Uma visão geral das etapas da abordagem BioConcST é ilustrada na Figura 13.

A primeira etapa da abordagem consiste na inicialização de uma população aleatória de indivíduos. Essa população é gerada a partir de um intervalo previamente definido que diz respeito ao espaço de busca do problema investigado, i.e., as possíveis entradas do programa sob teste. Em seguida, os indivíduos dessa população são submetidos à uma avaliação de *fitness*, a qual irá classificar a aptidão de cada indivíduo frente ao problema de busca.

A etapa de avaliação separa os indivíduos em *threads* exclusivas que executam

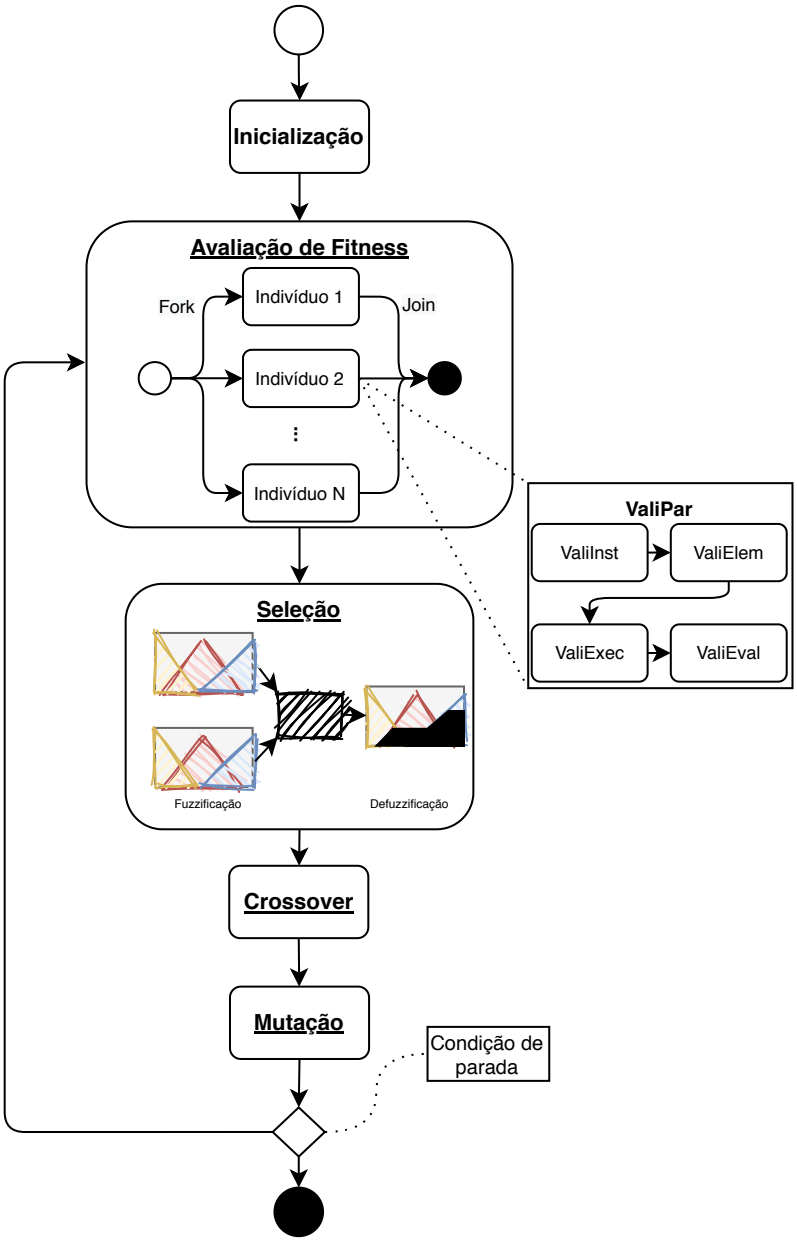


Figura 13 – Visão geral da abordagem BioConcST

concorrentemente para definir o valor *fitness* do indivíduo. Conforme descrito na Seção 4.2.1, a aptidão do indivíduo é obtida por uma distância global entre o caminho percorrido e os elementos requeridos do programa sob teste. As informações necessárias para essa computação, como elementos requeridos, caminho percorrido, GFCs e GFCPs, são obtidas por uma ferramenta de apoio ao teste de programas concorrentes, denominada ValiPar.

A ValiPar é uma ferramenta de teste estrutural para programas concorrentes. Ela realiza a execução dinâmica do programa sob teste por meio de módulos distintos que sistematizam o teste de software concorrente. Em cada *thread* de indivíduos, existe uma instância da ValiPar que desempenha as fases de teste sob uma entrada fornecida. Após a execução do teste na ValiPar, uma função faz a leitura dos arquivos de *report* que servirão

como entrada para computação do valor de *fitness*. A etapa de avaliação somente será encerrada quando todas as *threads* forem finalizadas e os valores de *fitness* atribuídos aos respectivos indivíduos.

Nesse momento, os indivíduos serão submetidos aos operadores genéticos responsáveis por definir a nova população. A seleção é o primeiro operador empregado nesse processo, esse operador seleciona um número X de indivíduos que irão compor a nova população com base nas características dos indivíduos. Neste trabalho, propõe-se um operador de seleção baseado em Lógica *Fuzzy* que sugere a permanência de indivíduos mais aptos em relação ao teste de software concorrente e características inéditas de um indivíduo. Uma descrição dos conceitos e etapas desse operador será apresentada em detalhes na Seção 4.3.

Na etapa seguinte, o operador *Crossover* (ou Recombinação), em um processo análogo à reprodução genética, combina informações de dois indivíduos (*parents*) para gerar um par de novos indivíduos (*offspring*). O operador *Single-point crossover* é definido como padrão na abordagem, entretanto, também é possível estender outros tipos de operadores de recombinação. Além disso, uma taxa também é estabelecida para definir a probabilidade de ocorrer uma recombinação entre dois indivíduos.

Por último, o operador Mutação é empregado para manter a diversidade de uma população ao longo das gerações. Para esse cenário optou-se pelo *Swap Mutator*, que consiste na troca da posição entre os genes que constituem o cromossomo, entretanto, outros tipos de mutações também podem ser empregadas. Assim como na recombinação, uma taxa de probabilidade também é definida para este operador.

Em seguida, o processo é submetido à uma condição de parada, a qual estabelece critérios em que a busca deve ser encerrada. A BioConcST estabelece três tipos diferentes de critérios de parada, os quais podem ser utilizados de forma exclusiva ou em conjunto. O primeiro critério estabelece um valor máximo de gerações para o processo de busca, ou seja, a solução final (i.e. o conjunto de entradas de testes de otimização I) será composta pela melhor população de indivíduos ao longo de um número de gerações predefinidas. O segundo critério, denominado *Steady Fitness*, estabelece um número máximo de gerações em que não há melhoria no valor atual de *fitness*.

Por último, o critério *Coverage* determina que o processo somente será interrompido quando todos os elementos requeridos do SUT forem alcançados pela solução atual. Enquanto um critério de parada não é alcançado, o processo de busca desempenha as atividades de forma iterativa, buscando sempre encontrar melhores indivíduos para solucionar o problema de geração de dados de teste concorrente.

4.3 *Fuzzy Selector* - Um operador genético de Seleção baseado em lógica *fuzzy*

Os operadores de seleção normalmente distinguem os indivíduos pelo valor de *fitness* associado a cada um deles. No processo de seleção, estratégias de priorização dos indivíduos com melhor *fitness* são aplicadas para aumentar as chances de que as informações genéticas desses indivíduos sejam passadas para futuras gerações. Neste trabalho, o *fitness* do indivíduo é uma importante característica para classificação do indivíduo, entretanto, existem outras informações que também são consideradas durante a etapa de seleção.

Conforme descrito anteriormente, este trabalho também propõe um operador genético de seleção utilizando lógica *fuzzy*, denominado FuzzyST. Informações relevantes dos dados de teste concorrente, como originalidade, *fitness* e cobertura de elementos requeridos, são definidas como variáveis linguísticas em um Sistema de Inferência *Fuzzy* (SIF).

A originalidade do indivíduo diz respeito à capacidade que este possui para cobrir elementos requeridos ainda não alcançados, ou pouco alcançados, pelos demais indivíduos da população. A computação de originalidade de um indivíduo X em relação a um indivíduo Y é dada pela distância de *Hamming*, para a qual duas *strings* binárias de tamanhos equivalentes são fornecidas como entrada. Cada posição de uma *string* representa um requisito de teste do programa sob teste, o status de cobertura dos elementos são dados como 1 para verdadeiro e 0 para falso. A Equação 4.1, apresenta o método empregado para calcular a distância de *Hamming*.

$$H_D = \frac{\sum_{k=1}^{NE} |X_k - Y_k|}{NE} \tag{4.1}$$

Seja X e Y indivíduos distintos, NE o tamanho da *string* (i.e. o número total de elementos requeridos) e k o k-ésimo elemento dessa *string*. A distância de *Hamming* (H_D) entre dois indivíduos é dada pelo soma de valores diferentes entre as *strings*, dividida pelo número total de elementos (NE).

O valor final de originalidade (*Originality*) considera a distância entre todos os demais indivíduos da população. Assim, uma média aritmética da distância entre um elemento X e todos os demais elementos da população é calculada por meio da Equação 4.2.

$$Originalidade = \frac{\sum_{i=1}^{n} H_D(X,i)}{n} \tag{4.2}$$

Tal que, X seja o indivíduo para o qual pretende-se encontrar o valor de origina-

lidade, n o número de indivíduos na população e i o i-ésimo indivíduo da população. A distância é calculada entre todos os indivíduos, o valor de originalidade é dado pela média entre todos os valores de distância computados.

A segunda variável linguística diz respeito à cobertura (*Coverage*) dos elementos requeridos, essa variável mede a capacidade de um indivíduo de exercitar o conjunto de elementos requeridos de teste. A cobertura é dada pelo percentual de elementos cobertos pelo indivíduo em relação ao conjunto total de elementos. A terceira variável trata-se do valor de *fitness* do indivíduo, os valores de *fitness* são padronizados em um intervalo que permitem estabelecer valores estáticos para as funções de pertinência, caso contrário, seria necessário estabelecer uma nova função de pertinência para cada programa sob teste.

Em adição, a variável *Survivor* estabelece a saída do sistema *fuzzy*, permitindo classificar quais serão os indivíduos mais aptos para próxima geração. As quatro variáveis linguísticas são caracterizadas por funções de pertinência, conforme ilustrado na Figura 14.

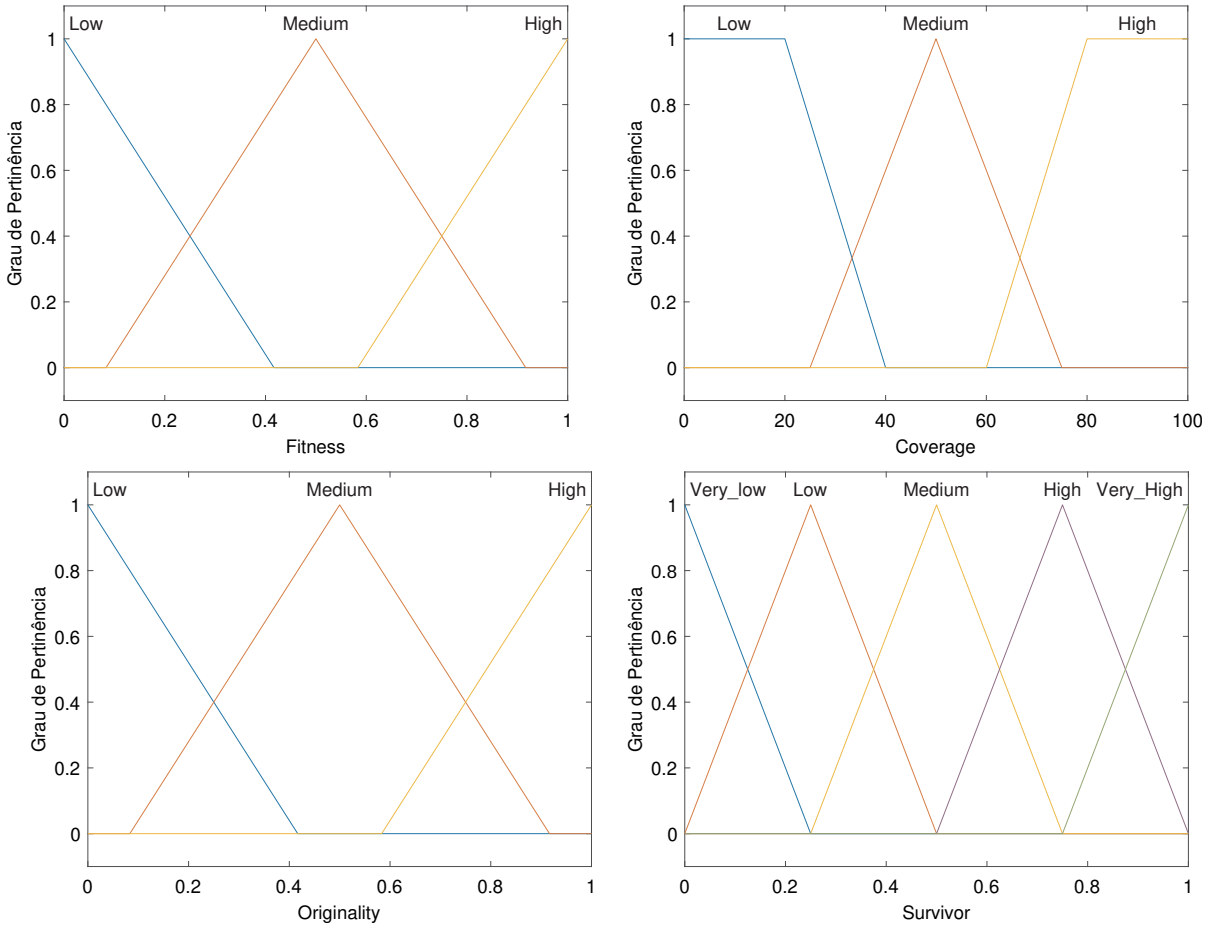


Figura 14 – Funções de pertinência das variáveis linguísticas

As três variáveis de entrada, *Fitness*, *Coverage* e *Originality*, possuem os mesmos valores linguísticos (*Low, Medium e High*), entretanto, há diferenças em suas funções de pertinência. As funções foram inicialmente definidas considerando seus respectivos

conceitos, mais tarde um processo de calibração foi realizado para ajusta-las. A função *Survivor* apresenta cinco valores linguísticos (*Very low, low, Medium, High e Very High*), os quais irão classificar, por meio de inferência *fuzzy*, os indivíduos de uma população. Em adição as variáveis linguísticas, um conjunto de regras é utilizado para o processo de inferência, este mecanismo define a base para tomada de decisões.

Na Tabela 1, apresenta-se o conjunto de regras definido. As entradas das regras de inferência do SIF são definidas a partir da combinação dos valores linguísticos incorporados nas variáveis de entrada. Essa combinação constitui um conjunto de 27 regras, as quais são resultado da permutação entre três valores linguísticos de cada variável de entrada. A saída de uma regra é definida por uma condição (Se-Então) que estabelece um valor linguístico na variável de saída. As condições definidas tem como objetivo manter os indivíduos com maior cobertura e menor *fitness*, mas também prioriza indivíduos heterogêneos.

Como exemplo, a Regra 13 ilustra um cenário onde a cobertura de um indivíduo é média, a distância dos elementos requeridos representada pelo valor de *fitness* também é média, mas a originalidade desse indivíduo é alta. Nesse caso, a probabilidade de permanência (*Survivor*) desse indivíduo na próxima geração será alta, pois ele possui características raras e importantes que devem permanecer ao longo das gerações. Uma visão geral de como o processo de seleção de indivíduos é realizado em conjunto com um sistema *fuzzy* é ilustrada na Figura 15.

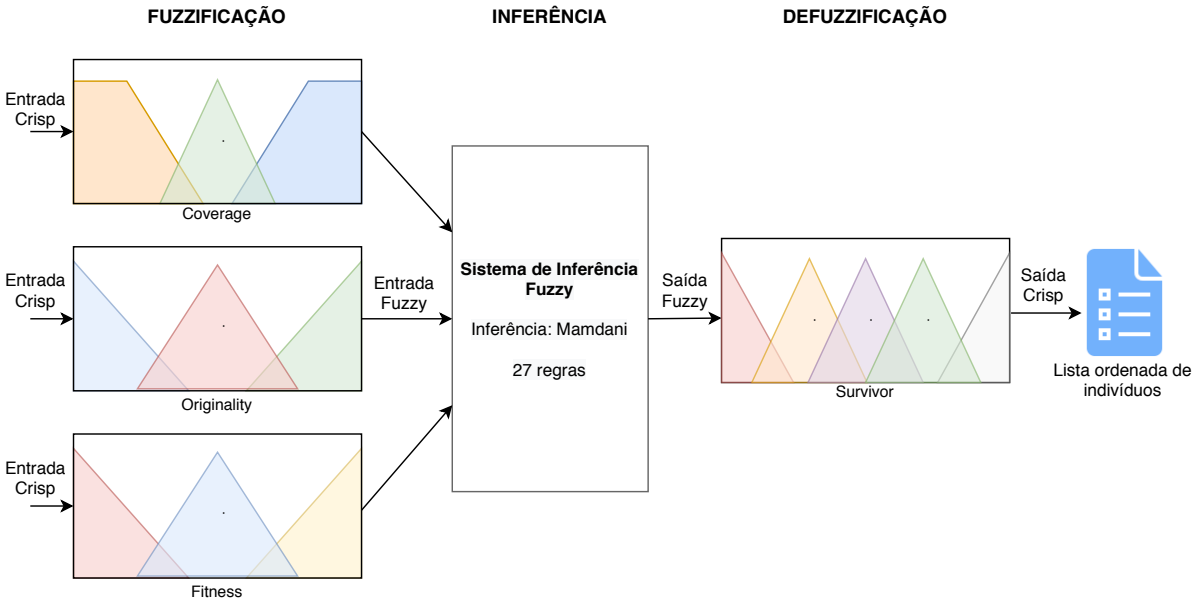


Figura 15 – Visão geral do operador genético de seleção baseado em Lógica *Fuzzy*

Após a concluir a etapa de avaliação de *fitness*, é possível extrair os valores de cada indivíduo para representação das variáveis linguísticas do FIS, entretanto, esse valores ainda necessitam de uma transformação para o procedimento de inferência. Para que os

Tabela 1 – Conjunto de regras *fuzzy* para seleção de dados de teste concorrente

	Entrada			**Saída**
Regra	⇑ ***Coverage***	⇓ ***Fitness***	⇑ ***Originality***	⇑ ***Survivor***
1	*High*	*Low*	*High*	*Very High*
2	*High*	*Low*	*Medium*	*Very High*
3	*High*	*Low*	*Low*	*High*
4	*High*	*Medium*	*High*	*Very High*
5	*High*	*Medium*	*Medium*	*High*
6	*High*	*Medium*	*Low*	*Medium*
7	*High*	*High*	*High*	*High*
8	*High*	*High*	*Medium*	*Medium*
9	*High*	*High*	*Low*	*Medium*
10	*Medium*	*Low*	*High*	*High*
11	*Medium*	*Low*	*Medium*	*Medium*
12	*Medium*	*Low*	*Low*	*Medium*
13	*Medium*	*Medium*	*High*	*High*
14	*Medium*	*Medium*	*Medium*	*Medium*
15	*Medium*	*Medium*	*Low*	*Low*
16	*Medium*	*High*	*High*	*High*
17	*Medium*	*High*	*Medium*	*Medium*
18	*Medium*	*High*	*Low*	*Low*
19	*Low*	*Low*	*High*	*High*
20	*Low*	*Low*	*Medium*	*Medium*
21	*Low*	*Low*	*Low*	*Low*
22	*Low*	*Medium*	*High*	*Medium*
23	*Low*	*Medium*	*Medium*	*Medium*
24	*Low*	*Medium*	*Low*	*Low*
25	*Low*	*High*	*High*	*Medium*
26	*Low*	*High*	*Medium*	*Very Low*
27	*Low*	*High*	*Low*	*Very Low*

valores absolutos (*crisp*) possam ser processados em um sistema *fuzzy*, eles precisam ser transformados em valores nebulosos, ocorrendo então o processo de fuzzificação.

A fuzzificação ocorre por meio das funções de pertinência (*Coverage, Originality e Fitness*), onde cada função entrega um valor fuzzificado para o sistema de inferência. A inferência é realizada empregando-se um conjunto de regras, nas quais os antecedentes (variáveis de entrada) e o consequente (variável de saída) são conjuntos nebulosos. A inferência sobre os valores nebulosos é realizada pelo método Mamdani, o qual baseia-se em uma estrutura simples de operações *min-max*, envolvendo regras de inferência como *Se-E-Então*.

Após a inferência dos valores nebulosos, ocorre a defuzzificação. Nesse processo o valor nebuloso é novamente transformado em um absoluto, que corresponde ao grau de importância de um indivíduo para uma nova geração. Neste trabalho, utiliza-se o método

do centro geométrico para o processo de defuzzificação.

A seleção efetiva do AG só ocorre quando todos os indivíduos da população possuem um valor associado de sobrevivência (*Survivor*). Uma lista ordenada pelo valor de sobrevivência contém todos os indivíduos da população, assim, somente os N primeiros indivíduos dessa lista serão selecionados, tal que N representa o tamanho da população estabelecido no AG.

4.4 Considerações Finais

Neste capítulo foi apresentada a abordagem BioConcST para geração de dados de teste para programas concorrentes. A abordagem emprega uma técnica meta-heurística baseada na teoria da evolução das espécies e seleção natural de indivíduos. A técnica é guiada por uma função de *fitness* que utiliza informações de distância de alcançabilidade dos objetivos de teste do critério *all-sync-edges*. Além disso, também foi descrita a proposta de um operador genético de seleção de dados de teste, denominado FuzzyST. Esse operador considera informações adicionais, como originalidade e cobertura de critérios de teste, para selecionar um indivíduo, e não apenas o *fitness* como os demais operadores de seleção. No capítulo seguinte apresenta-se um estudo experimental que visa avaliar a abordagem proposta considerando outros métodos de geração e outros operadores de seleção.

CAPÍTULO

5

AVALIAÇÃO EXPERIMENTAL

5.1 Considerações Iniciais

A avaliação experimental fornece a base essencial para o avanço no entendimento e conhecimento sobre o tema investigado (SHULL *et al.*, 2004). O processo experimental na engenharia de software é empregado para apoiar a avaliação, predição, entendimento e controle no desenvolvimento de um artefato científico. Neste trabalho, a avaliação da abordagem BioConcST será explorada por meio da condução de experimentos, os quais avaliarão diferentes abordagens de geração de dados de teste para o contexto de programas concorrentes. A definição deste estudo considera as diretrizes do *framework* de experimentação proposto por Wohlin *et al.* (2000). A caracterização do presente estudo é formalmente sumarizada da seguinte forma:

Analisar *a abordagem BioConcST para seleção automática de dados de teste com a* ***finalidade de*** *comparar a qualidade dos conjuntos de testes gerados,* ***no que diz respeito a*** *cobertura de critérios* ***do ponto de vista do*** *teste para software concorrente* ***no contexto*** *da academia.*

Este capítulo está organizado da seguinte forma, na Seção 5.2 é apresentado o planejamento do estudo experimental, o qual descreve as variáveis empregadas e as atividades que serão realizadas durante o estudo experimental.

5.2 Planejamento

A finalidade deste estudo experimental é analisar como a estratégia de avaliação e seleção de indivíduos da abordagem BioConcST contribui na seleção automática de dados de teste para programas concorrentes. Primeiramente, almeja-se analisar como a função de *fitness*, baseada na distância entre objetivo de teste e caminho percorrido, con-

tribui para a seleção do conjunto de dados de teste concorrente otimizado. Posteriormente, pretende-se verificar se o operador de seleção proposto neste trabalho contribui para o processo evolutivo do Algoritmo Genético. Com esse intuito, um conjunto diversificado de programas concorrentes é empregado em uma avaliação comparativa entre os artefatos propostos neste trabalho e os principais métodos reportados na literatura.

Considerando o contexto apresentado, a definição deste estudo aborda duas questões de pesquisa principais:

- **QP1** *Qual a influência da atribuição de distância usada durante a avaliação de fitness para seleção automática de dados de teste concorrente*?
  - **QP1.1** Existe diferença, em relação ao processo de busca, entre indivíduos que falharam na cobertura de um objetivo de teste?
  - **QP1.2** A classificação de indivíduos que não alcançaram um objetivo de teste pode contribuir para seleção de conjuntos de teste melhores?
  - **QP1.3** A avaliação baseada na distância apresenta contribuições ao processo de busca?

Para responder estas perguntas, uma comparação entre a abordagem proposta BioConcST e duas outras abordagens estabelecidas na literatura, aleatória e baseada em cobertura é realizada. Pretende-se analisar quais os fatores que influenciam na cobertura do critério de teste *all-sync-edges* e como a atribuição de distância é propagada para o custo no número de gerações. As seguintes hipóteses foram definidas para esta questão de pesquisa:

- **Hipótese Nula ($H1_0$)** Não há diferença, em razão da cobertura para o critério *all-sync-edges*, entre os conjuntos de teste fornecidos pela abordagem BioConcST ($\mu CT_{BioConcST}$), aleatória ($\mu CT_{Aleatória}$) e baseada em cobertura ($\mu CT_{Cobertura}$).

$$H1_0 : \mu CT_{BioConcST} = \mu CT_{Aleatória} = \mu CT_{Cobertura}$$

- **Hipótese Alternativa ($H1_1$)** Existe diferença, em razão da cobertura para o critério *all-sync-edges*, entre os conjuntos de teste fornecidos pela abordagem BioConcST ($\mu CT_{BioConcST}$), aleatória ($\mu CT_{Aleatória}$) e baseada em cobertura ($\mu CT_{Covertura}$).

$$H1_1 : \mu CT_{BioConcST} \neq \mu CT_{Aleatória} \neq \mu CT_{Cobertura}$$

- **QP2** Qual a influência da combinação entre os fatores cobertura, *fitness* e originalidade durante a seleção de indivíduos no processo de busca?

- **QP2.1** A inferência *fuzzy* a partir das informações cobertura, *fitness* e originalidade contribui para formação de novos indivíduos melhores?
- **QP2.2** A convergência precoce pode ser evitada com o uso da combinação entre originalidade, fitness e cobertura?
- **QP2.3** Os operadores de seleção baseados exclusivamente no *fitness* são as melhores alternativas para otimização do teste de software concorrente?

Para responder a esta questão, um estudo comparativo será realizado entre o operador proposto e os principais operadores reportados na literatura. Pretende-se descobrir como a seleção de indivíduos influencia na cobertura de critérios de teste. Além disso, espera-se esclarecer o comportamento do processo de busca quando submetido a seleção baseada em um sistema *fuzzy*. A seguir, apresenta-se as hipóteses estabelecidas para esta questão de pesquisa.

- **Hipótese Nula ($H1_0$)** Não há diferença, em razão da cobertura para o critério *all-sync-edges*, do valor de *fitness* e da taxa de convergência, entre o método de seleção proposto ($\mu FuzzyST$) e os métodos Torneio ($\mu Tournament$), Roleta ($\mu RoulleteWheel$), Elitismo ($\mu Elite$) e Amostragem universal estocástica (μSUS).

$$H1_0 : \mu FuzzyST = \mu Tournament = \mu RoulleteWheel = \mu Elite = \mu SUS$$

- **Hipótese Alternativa ($H1_1$)** Existe diferença, em razão da cobertura para o critério *all-sync-edges*, do valor de *fitness* e da taxa de convergência, entre o método de seleção proposto ($\mu FuzzyST$) e os métodos Torneio ($\mu Tournament$), Roleta ($\mu RoulleteWheel$), Elitismo ($\mu Elite$) e Amostragem universal estocástica (μSUS).

$$H1_0 : \mu FuzzyST \neq \mu Tournament \neq \mu RoulleteWheel \neq \mu Elite \neq \mu SUS$$

5.2.1 Variáveis e Métricas

5.2.1.1 Variáveis Independentes

Cinco variáveis independentes serão manipuladas, o método de seleção de dados de teste concorrente (VI-1), a estratégia de avaliação de indivíduos (VI-2), os operadores de seleção de indivíduos (VI-3) e os programas concorrentes sob teste (VI-4). Na Tabela 2 são apresentadas as variações (manipulação) de cada variável independente.

O **método de seleção** (VI-1) diz respeito a estratégia empregada para definir um conjunto de dados de teste. Neste trabalho três métodos diferentes são avaliados (Tabela

Tabela 2 – Níveis de manipulação por variável independente

Variáveis	**Níveis de Manipulação**				
VI-1	Aleatória (Random)	Cobertura (CoverageBased)	Distância (BioConcST)		
VI-2	Avaliação por Cobertura	Avaliação por distância			
VI-3	*FuzzyST*	Elitismo	SUS	Roleta	Torneio
VI-4	Memória compartilhada	Passagem de Mensagem	Híbrido		

2), um método de seleção aleatória, definido como *baseline* da avaliação, uma busca meta-heurística guiada pela cobertura de critérios e a abordagem BioConcST guiada por uma combinação entre distância, cobertura e originalidade do indivíduo.

A **avaliação do indivíduo** (VI-2) concentra-se no método utilizado para definir a aptidão ao dado de teste concorrente diante do problema busca. A manipulação dessa variável é dada em dois níveis, o primeiro nível avalia os requisitos de teste de forma binária, isto é, para atribuição do valor de *fitness* somente é julgado se o objetivo de teste foi ou não alcançado. O segundo nível, que consiste na proposta deste trabalho, considera a proximidade entre o caminho percorrido e o objetivo de teste para realização dessa tarefa.

Os **operadores de seleção** (VI-3) distinguem-se pela estratégia utilizada na composição de novas populações no processo evolutivo. Serão utilizados cinco operadores neste estudo, a saber, *FuzzyST*, Elitismo, Amostragem universal estocástica (*Stochastic universal sampling* - SUS), Roleta e Torneio.

O paradigma do **programa concorrente** (VI-4), o qual difere-se pelas estratégias de comunicação, seja essa realizada entre processos ou *threads*. A manipulação dessa variável considera três tipos de comunicação, a saber, passagem de mensagem, memória compartilhada e comunicação híbrida, onde há a presença de ambos os paradigmas de comunicação.

5.2.1.2 Variáveis Dependentes

Na condução do estudo experimental serão computadas três variáveis, divididas em relação à cobertura do critério de teste *all-sync-edges* (VD-1), taxa de convergência do algoritmo de busca (VD-2) e no valor de *fitness* dos melhores indivíduos (VD-3) no decorrer das gerações.

A cobertura de um critério compreende a razão entre o número de requisitos de teste atingidos e a quantidade total de elementos de um programa sob teste, conforme apresentado na Equação 5.1.

$$Cobertura = \frac{Requisitos\ atingidos}{Total\ de\ requisitos} \tag{5.1}$$

A **Taxa de convergência** verifica quão rápido, em função do número de gerações, as populações podem vir a incluir os indivíduos que juntos apresentam melhor cobertura dos requisitos de teste. A formalização da taxa convergência é apresentada na Equação 5.2.

$$T_C = \frac{N_{GC}}{N_{TG}} \tag{5.2}$$

O valor de T_C caracteriza a taxa de convergência do método investigado, tal que N_{GC} representa o número de gerações necessárias para alcançar melhor cobertura dos conjuntos de teste, enquanto N_{TG} corresponde ao número total de gerações previamente definido como critério de parada.

O **valor de *fitness*** equivale a adequação de um indivíduo ao problema de otimização. A função de avaliação determina quão apto é um indivíduo para solucionar o problema de busca. Essa métrica caracteriza os métodos de seleção de acordo com os melhores indivíduos alcançados no decorrer das gerações.

As variáveis dependentes, também conhecidas como variáveis de resposta, estão diretamente ligadas aos efeitos provenientes da variação entre as variáveis independentes. Neste estudo, uma mesma variável dependente será utilizada para compreender os efeitos de um ou mais fatores. Na Tabela 3 é apresentada a relação entre cada variável e seus respectivos fatores.

Tabela 3 – Relação entre variáveis dependentes e independentes

	Variáveis independentes			
Variáveis dependentes	VI-1	VI-2	VI-3	VI-4
Cobertura (VD-1)	✓	✓	✓	✓
Taxa de convergência (VD-2)		✓	✓	✓
Fitness (VD-3)			✓	✓

5.2.2 Seleção dos Sujeitos de Teste

Assim como descrito anteriormente, o objetivo deste estudo é avaliar os conjuntos de teste fornecidos pela abordagem BioConcST. Nesse sentido, a seleção dos sujeitos foi realizada para contemplar sujeitos heterogêneos para maior garantia de generalização dos resultados. Assim, programas concorrentes, extraídos do *benchmark* TestPar definido por Dourado (2015), foram empregados a fim de avaliar os diferentes artefatos sob investigação.

Os programas foram implementados na linguagem Java com o objetivo de apoiar a validação e avaliação de teste estrutural de programas concorrentes com passagem de

Tabela 4 – Programas concorrentes sob teste

ID	Programa concorrente sob teste	LOC	Padrão de Comunicação	Paradigma	Processos	*Threads*	Sync-edge
1	*Greatest Common Divisor*	180	Mestre-Escravo	PM	4	4	33
2	*Greatest Common Divisor- Two Slaves*	201	Mestre-Escravo	PM	3	3	16
3	*Greatest Common Divisor/LeastCommon Multiple*	259	Mestre-Escravo	PM	4	4	57
4	*Token Ring Iterations*	175	Produtor-Consumidor	PM	4	4	12
5	*Token Ring Both Directions Same Primitives*	221	Produtor-Consumidor	PM	4	4	12
6	*Token Ring Both Directions Diferent Primitives*	224	Produtor-Consumidor	PM	4	4	51
7	*Parallel Sieve of Eratosthenes*	274	Árvore	PM	4	4	15
8	*Roller Coaster*	374	Produtor-Consumidor	PM	6	4	206
9	*Producer Consumer SemaphoreLock Conditions Iterations*	241	Todos-para-todos	MC	1	6	20
10	*Cigarette Smokers*	253	Todos-para-todos	MC	1	5	42
11	*Matrix*	228	Todos-para-todos	MC	1	5	192
12	*Jacobi*	411	Todos-para-todos	MC	1	13	532
13	*Token Ring Broadcast*	294	Vizinhos	PM/MC	3	7	46
14	*Greatest Common Divisor / Least Common Multiple Both*	462	Mestre-Escravo	PM/MC	4	10	147

mensagem e memória compartilhada. A amostra considerada neste estudo, para a análise de cobertura, é composta por 14 programas (Tabela 4), os quais apresentam diferentes características relacionadas à concorrência, como: padrão de comunicação, número de processos e *threads* e número de requisitos de teste. Oito desses programas comunicam-se por meio de passagem de mensagem, quatro por memória compartilhada e dois programas de comunicação híbrida.

5.2.3 Instrumentação

Na instrumentação do estudo admitiu-se a implementação de um protótipo para prova de conceitos, incorporando as diretrizes apresentadas na abordagem BioConcST. O protótipo foi implementado na linguagem Java considerando o framework *Jenetics* (WILHELMSTÖTTER, 2020) na implementação do Algoritmo Genético. Além disso, a ferramenta ValiPar (PRADO *et al.*, 2015) também foi empregada para execução dos testes das aplicações concorrentes.

Na implementação da BioConcST funcionalidades adicionais foram desenvolvidas para mediar a avaliação de *fitness*, em parte realizada pela ValiPar, e a execução do AG. Como exemplo, a ferramenta ValiPar ainda não possui atributos necessários para execução *multithread*, consequentemente o desempenho do protótipo era prejudicado por essa limitação.

Para garantir o desempenho da BioConcST, a ValiPar passou a ser instanciada em cada avaliação de *fitness*, isolando cada indivíduo em sua própria instância. Contudo, para algumas funcionalidades, a ValiPar utiliza recursos exclusivos do SO, e somente a criação de uma instância exclusiva não poderia garantir uma execução *thread-safe* da BioConcST. Por essa razão, essas funcionalidades foram utilizadas na BioConcST em funções síncronas, com o intuito de garantir uma forma segura para que os indivíduos fossem avaliados de forma concorrente.

Além dos recursos desenvolvidos para execução *multithread*, a BioConcST necessita de informações sobre o teste concorrente relevantes para o processo evolutivo. As

interfaces do framework Jenetics não permitem a adição de informações extras nas classes que controlam o processo evolutivo, somente classes relacionadas à representação do indivíduo e operadores genéticos são passíveis de mudança. Em virtude disso, novas funcionalidades foram incorporadas no Jenetics e uma versão exclusiva do framework é utilizada na BioConcST.

Na Figura 16 apresenta-se um modelo conceitual da relação entre a BioConcST e as outras ferramentas utilizadas. Os conceitos ilustrados na cor azul compõem o conjunto de classes desenvolvidas neste estudo, esses conceitos são divididos entre a função de *fitness* e o operador de seleção. A comunicação entre a função de avaliação e a ferramenta ValiPar ocorre através de um *middleware* que é responsável por atender as requisições, gerenciar as tarefas concorrentes e criar diferentes instâncias da ValiPar.

Além disso, a avaliação de *fitness* também utiliza métodos do conceito *Distance* para definir a distância do indivíduo para um dado objetivo de teste. As funções desse conceito permitem adquirir a distância para o envio (*send*) da mensagem e também para o recebimento (*recv*). Para realização dessas tarefas é necessário um objeto contendo todos os elementos requeridos do programa sob teste (*Required Elements*) e também uma busca em largura sobre um GFCP para mensurar a distância entre o caminho percorrido e o objetivo de teste.

A construção do operador de seleção implementa a interface *Selector* do framework Jenetics, a qual possui o conjunto métodos essenciais para comunicação com os demais módulos do framework. Além disso, o operador estabelecido neste trabalho utiliza a biblioteca *jFuzzyLogic* (CINGOLANI; ALCALá-FDEZ, 2013) para definição das variáveis linguísticas, funções de pertinência, regras, e para execução do sistema de inferência *fuzzy*.

Os conceitos ilustrados na cor vermelha representam o conjunto de classes que sofreram alterações para adequar as necessidades da BioConcST. Na representação do indivíduo foram necessárias alterações nos conceitos *Phenotype*, *Genotype* e *AnyGene*. As alterações dizem respeito a adição de um valor de cobertura do critério *all-sync-edges* para um indivíduo no conceito *Phenotype*, inserção do caminho de execução no conceito *Genotype* e definição de uma estrutura de gene que corresponde a entrada do programa sob teste.

No processo evolutivo foram necessárias alterações no controlador principal (*Engine*) do algoritmo genético, as mudanças compreendem a inserção de um mecanismo para armazenar e recuperar a cobertura de teste da população de indivíduos e manter uma lista da melhor população, em relação à cobertura de critérios, ao longo das gerações.

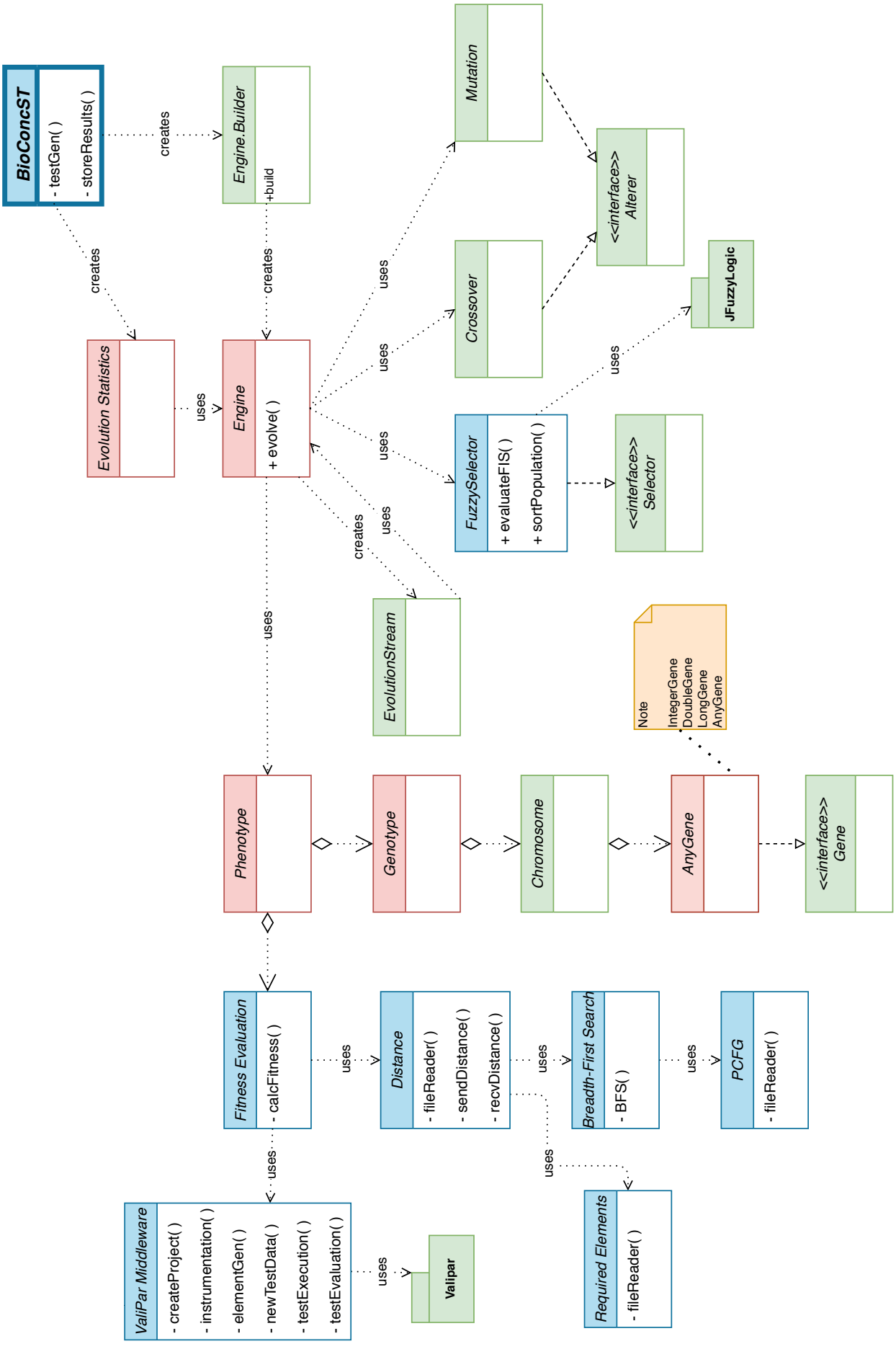


Figura 16 – Modelo conceitual do protótipo BioConcST

Por último, informações da cobertura de critérios foram adicionadas nas estatísticas do processo evolutivo (*Evolution Statistics*), os resultados que antes apenas eram exibidos no console, agora são armazenados de forma estruturada em planilhas na extensão XLSX.

Ainda com relação ao modelo ilustrado na Figura 16, os conceitos apresentados na cor verde representam os artefatos utilizados integralmente, em outras palavras, esses conceitos possuem os atributos necessários para construção do protótipo e não foram necessárias adaptações nas bibliotecas e interfaces desse conjunto.

No que diz respeito a instrumentação para a execução das atividades experimentais, o estudo foi conduzido a partir de um terminal com acesso remoto ao servidor de execução. O servidor está situado na costa leste dos Estados Unidos, as configurações do servidor são apresentadas na Tabela 5.

Tabela 5 – Configuração de hardware e software

Fator	**Configuração**
Processador	AMD Ryzen 7 3800x
Nº de núcleos de CPU	8
Nº de threads	16
Clock básico	3.9 GHz
Memória	64GB RAM
GPU	Radeon 5700xt 8GB
Armazenamento	Inland Performance 1TB SSD M.2 2280
Sistema Operacional	Ubuntu 20.04 LTS
Versão Java	1.8 build 25.265-b01 & 11 build 11.0.8+10

Vale ressaltar que a BioConcST utiliza duas versões Java para otimização dos conjuntos de teste, a versão 8 é utilizada para execução da ferramenta ValiPar e a versão 11 é utilizada no framework Jenetics e também para as demais funcionalidades implementadas neste projeto.

5.2.4 Ameaças à Validade

Um ponto importante sobre um estudo experimental diz respeito a validade dos resultados obtidos, considerando a possibilidade de generalização dos resultados. Por essa razão, é importante considerar a questão da validade dos resultados ainda na etapa de planejamento do estudo, pois dessa forma é possível elaborar um estudo adequado frente as possíveis ameaças.

Validade de conclusão

A validade de conclusão diz respeito às características que afetam a habilidade de inferir conclusões corretas sobre a relação entre os tratamentos e saídas do experimento. Neste estudo, as conclusões obtidas como resultado podem ser influenciadas pelo teste estatístico escolhido. Para mitigar essa ameaça, adotou-se as diretrizes da estatística experimental para o planejamento e condução do estudo. Os testes estatísticos foram escolhidos com base na normalidade dos dados, no número de amostras e no número de fatores investigados. No que se refere ao risco *Fishing*, isto é, o favorecimento de uma determinada hipótese durante a análise dos dados, o mesmo não é considerado uma ameaça neste estudo, pois não há interesse em concluir em nenhuma das hipóteses, mas sim demonstrar as descobertas do problema investigado.

Validade interna

Ameaças à validade interna são influências que podem afetar as variáveis independentes no que diz respeito à casualidade, as quais o pesquisador não possui conhecimento ou controle. Nessa linha, acredita-se que esse grupo de ameaças não é propagado para o presente estudo, pois este experimento trata-se de um estudo controlado onde todas as variáveis são de conhecimento do pesquisador e a manipulação das mesmas é realizada uniformemente para todos os artefatos avaliados. Além disso, o pesquisador possui acesso ao código fonte de todos os artefatos externos utilizados neste trabalho.

Validade de construção

A validade de construção refere-se à relação entre a teoria e a observação de um experimento. Em outras palavras, a validade de construção está relacionada ao fato do experimento refletir de maneira realística a influência dos tratamentos nos resultados observados. Embora o presente estudo apresente uma nova abordagem para o teste de programas concorrentes, resultados insatisfatórios também são bem vistos para este estudo, uma vez que trata-se de um problema indecidível que necessita de um estudo contínuo para avanços na área de investigação. Dessa forma, não existe intenções para o favorecimento de uma abordagem em detrimento de outra.

Validade externa

A validade externa refere-se à generalização dos resultados do experimento para outros contextos, como por exemplo programas de software industriais. Nesse estudo, a escolha dos programas concorrentes pode ser considerada uma ameaça à validade dos resultados, contudo, para minimizar os efeitos dessa ameaça optou-se pela utilização de programas com características distintas no que diz respeito ao objeto de estudo deste trabalho. Conforme mencionado anteriormente, os programas são provenientes de um

benchmark previamente avaliado e com artefatos já utilizados em diferentes estudos na literatura.

5.3 Operação

A operação de um estudo experimental consiste na preparação dos artefatos e ferramentas para execução das atividades definidas no planejamento do estudo. As seções seguintes descrevem em detalhes as atividades desenvolvidas em cada etapa desse processo.

5.3.1 Preparação

A etapa de preparação do experimento determina o ambiente no qual o mesmo será desenvolvido. Neste estudo, faz-se necessária a configuração do ambiente para execução da BioConcST, assim como os artefatos que representam as demais abordagens investigadas neste estudo.

A configuração da BioConcST requer um ambiente com distribuição do Linux e com variável de ambiente previamente estabelecida para a ferramenta ValiPar. A configuração da variável de ambiente pode ser realizada conforme apresentado no Código-fonte 4.

Código-fonte 4 – Script para configuração de variável ambiente ValiPar

```
1: sudo nano ~/.bashrc #Abrir o script
2: export PATH=$PATH:caminho/valipar/bin #adicionar ao script
3: source ~/.bashrc #implementar e ler mudanças
```

Conforme mencionado anteriormente, a execução da BioConcST requer duas versões da Máquina virtual Java para sua execução, a instalação das versões pode ser realizada conforme apresentado no Código-fonte 5.

Código-fonte 5 – Script para instalação das versões JVM

```
1: sudo apt update #Atualizar lista de pacotes
2: apt install openjdk-11-jre-headless #Java Virtual Machine 11
3: apt install openjdk-8-jre-headless  #Java Virtual Machine 8
```

5.3.2 Condução

A avaliação foi dividida em dois experimentos principais, o primeiro experimento tem como objetivo avaliar o operador de seleção FuzzyST proposto neste trabalho em relação aos demais operadores já conceituados na literatura. Os resultados desse experimento deverão contribuir para tomada de decisão na definição do operador de seleção mais

apropriado para BioConcST. Uma visão geral do experimento definido para avaliação dos operados de seleção é ilustrada na Figura 17.

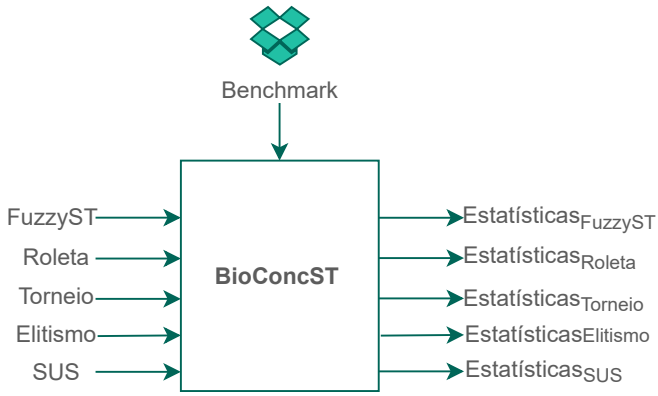


Figura 17 – Visão geral do experimento de operadores de seleção

Os operadores de seleção serão utilizados no protótipo da BioConcST, todos serão submetidos aos programas sob teste para os quais serão geradas estatísticas que deverão refletir a capacidade de cada operador para o processo evolutivo e para a cobertura dos objetivos de teste.

No segundo experimento pretende-se investigar a abordagem BioConcST em relação aos demais métodos na literatura para geração automática de dados de testes para programas concorrentes. Conforme mencionado anteriormente, esta análise tem como objetivo verificar se avaliação de *fitness* baseada na distância apresenta melhorias em relação à avaliação de cobertura binária e geração aleatória. Uma visão geral dessa investigação é apresentada na Figura 18.

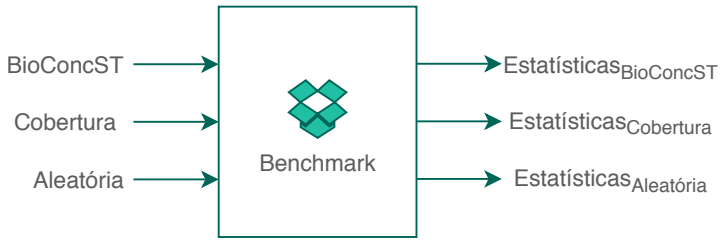


Figura 18 – Visão geral do experimento de estratégias de geração de dados de teste

Assim como no primeiro experimento, os métodos de geração serão submetidos ao conjunto de programas sob teste. Após a execução dos métodos para todos os programas considerados, estatísticas serão obtidas de acordo com as variáveis dependentes definidas para esta investigação, conforme apresentado anteriormente na Tabela 3.

Em adição aos aspectos levantados, também foram definidas configurações para o delineamento do experimento, as quais dizem respeito aos atributos do algoritmo genético e o número de repetições dos estudos. Em ambos os experimentos as configurações

foram mantidas para garantir uma comparação justa entre os artefatos, na Tabela 6 são apresentadas as configurações dos estudos experimentais.

Tabela 6 – Configuração dos parâmetros do experimento

Fator	**Configuração**
Taxa de Mutação	0.1
Taxa de *Crossover*	0.6
Offspring	90%
Survivor	10%
População	10 indivíduos
Nº Gerações	100
Nº Repetições	20

A definição dos parâmetros foi inicialmente realizada seguindo as diretrizes apresentadas em um estudo prévio (VILELA *et al.*, 2016), em seguida, experimentos piloto foram conduzidos para ajustar uma configuração ideal dos parâmetros. Os parâmetros *Offspring* e *Survivor* referem-se ao percentual de indivíduos que serão selecionados para a próxima geração, o primeiro remete-se as proles obtidas por meio dos operadores genéticos enquanto o segundo trata-se dos indivíduos antecessores.

5.4 Análise dos Resultados

As análises delineadas para este estudo tem como propósito avaliar, experimentalmente, as principais contribuições deste trabalho em comparação aos métodos consolidados na literatura e, assim, evidenciar como esta tese de doutoramento avança cientificamente contribuindo para o estado da arte na geração automática de dados de teste para programas concorrentes. As análises foram subdivididas em dois grupos, os quais procuram responder as duas principais questões de pesquisa caracterizadas anteriormente. Ainda vale ressaltar que em todos os ensaios realizados observou-se o número de amostras e a distribuição dos dados para a escolha do teste estatístico adequado.

O primeiro grupo tem como objetivo avaliar a contribuição da estratégia de seleção de dados de teste estabelecida na BioConcST, para isso foram realizadas comparações com outros métodos de seleção e estratégias de busca. O segundo grupo aborda os resultados do operador genético de seleção *FuzzyST*, para o qual foram realizadas análises sob diferentes óticas visando responder a segunda questão de pesquisa e suas respectivas subquestões. Em todas as análises realizadas, operadores de seleção consolidados em diversos cenários de otimização foram empregados para fins de comparação com o operador proposto. As subseções seguintes apresentam de forma crítica e circunstanciada os resultados das análises desempenhadas para cada questão de pesquisa estabelecida.

5.4.1 QP1 Qual a influência da atribuição de distância usada durante a avaliação de fitness para seleção automática de dados de teste para programas concorrentes?

Em vista de esclarecer a QP1, foram analisados os resultados de cobertura do critério *all-sync-edges* entre as estratégias de seleção. As hipóteses foram analisadas sob três cenários, o primeiro cenário (QP1.1) verifica, sob uma perspectiva meta-heurística, como diferentes funções de avaliação contribuem para seleção de indivíduos com melhor aderência aos requisitos de teste e qual o impacto do custo, em termos de gerações, nos resultados de cobertura. O segundo cenário (QP1.2) também compreende uma análise de cobertura, contudo, a avaliação não é limitada ao domínio de meta-heurísticas. Por último, o terceiro cenário (QP1.3) investiga a relação entre os fatores cobertura e *fitness*. As subquestões seguintes descrevem em detalhes os resultados obtidos para os cenários supracitados.

QP1.1 Existe diferença, em relação ao processo de busca, entre indivíduos que falharam na cobertura de um objetivo de teste?

Essa questão de pesquisa endereça uma das principais contribuições deste projeto. A função de *fitness* estabelecida na BioConcST caracteriza os indivíduos em razão da proximidade do caminho percorrido até o objetivo de teste. Por essa razão, indivíduos que não cobriram um determinado requisito de teste, podem apresentar diferentes valores de aptidão, pois cada um possui seu próprio caminho de execução e, consequentemente, uma distância do requisito de teste. Na literatura encontram-se trabalhos que avaliam um indivíduo em função da cobertura de elementos requeridos alcançados, empregando uma espécie de avaliação binária, ou seja, os indivíduos que cobriram um determinado requisito de teste são promissores para gerar novos indivíduos ainda melhores. Contudo, os indivíduos que falharam na cobertura desse mesmo critério são vistos como equivalentes e portanto são considerados igualmente inaptos para composição de futuras gerações.

Nessa perspectiva, para a análise da QP1.1 foi realizada uma comparação entre as abordagens BioConcST e *CoverageBased*, as quais representam as diferentes estratégias de avaliação de indivíduos citadas anteriormente. Essa investigação se deu por meio de quatro níveis no número de gerações, sendo eles 25, 50, 75 e 100 gerações. Na Figura 19 são apresentados os resultados obtidos por níveis de gerações.

Os resultados reforçam a premissa de que indivíduos possuem diferenças em relação ao processo de busca, ainda que os mesmos sejam semelhantes do ponto de vista de cobertura dos requisitos de teste. As análises realizadas revelam que há diferenças significativas entre as abordagens BioConcST e *CoverageBased* e, portanto, demonstram que a avaliação de indivíduos pode ser melhor empregada utilizando aspectos de distância do requisito de teste para o contexto de programas concorrentes. Além disso, nota-se que

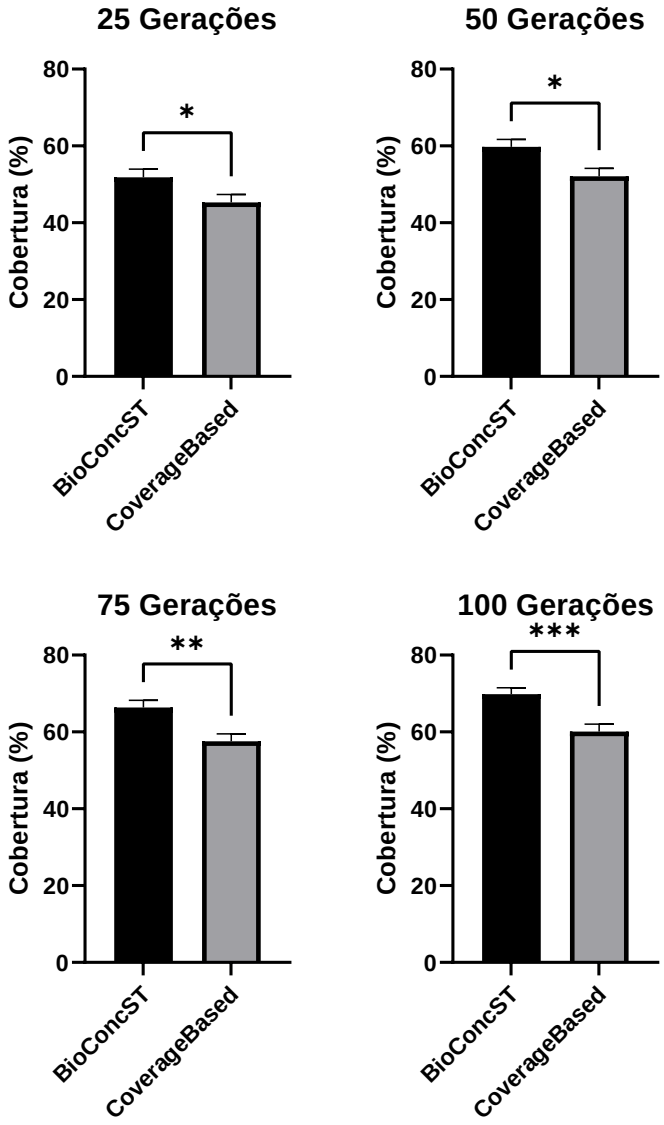


Figura 19 – Análise de cobertura do critério *all-sync-edges* entre funções de avaliação das abordagens BioConcST e *CoverageBased* sob os níveis 25, 50, 75 e 100 de gerações. Os dados foram analisados pelo teste não paramétrico *Mann-Whitney*. N = 280. Dados expressos como média ± erro padrão da média. (* $p \leq 0.05$, ** $p \leq 0.01$, *** $p \leq 0.001$)

essa hipótese é confirmada em todos os níveis de gerações, com os últimos níveis (75 e 100) apresentando maior nível de confiabilidade com p-valor ≤ 0.001. Sendo assim, rejeita-se a hipótese nula (H_0) que supõe que não há diferença significativa entre as amostras analisadas.

QP1.2 A classificação de indivíduos que não alcançaram um objetivo de teste pode contribuir para seleção de conjuntos de teste melhores?

Essa questão é complementar a questão anterior, pois uma vez comprovado que há diferenças entre indivíduos que falharam na cobertura de um determinado requisito de teste, o desafio é estabelecer um mecanismo para ordenar a importância desses indivíduos

em função de um requisito de teste ou de um conjunto de requisitos. Uma das atividades da BioConcST é a classificação desses indivíduos, estabelecida inicialmente pelo *fitness* do indivíduo e posteriormente pelo operador de seleção FuzzyST.

Dessa forma, esta análise emprega diferentes estratégias de geração de dados de teste para investigar as contribuições e desafios da atividade de classificação de indivíduos em razão da distância do requisito de teste. Sendo assim, as abordagens de geração CoverageBased, *Random* e BioConcST foram investigadas por meio da geração de conjuntos de testes para o conjunto de *benchmarks* descrito anteriormente. Na Figura 20 são apresentados os resultados obtidos por esta análise.

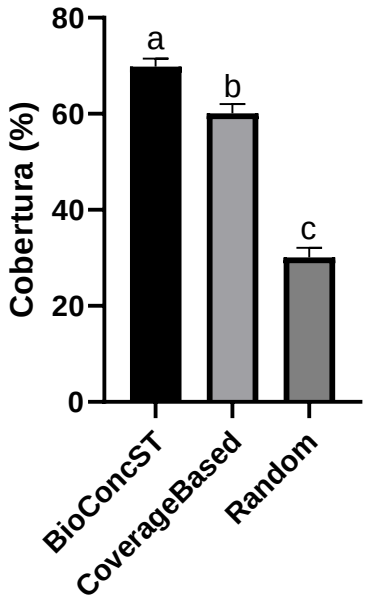


Figura 20 – Análise de cobertura entre as abordagens de geração de dados de teste BioConcST, *CoverageBased* e Random. Os dados foram analisados pelo teste não paramétrico Kruskal-Wallis. N=280. Os resultados foram submetidos ao teste de comparações múltiplas de Dunn. Dados expressos como média ± erro padrão da média. Valores diferentes são marcados com letras diferentes sobrescritas ($p < 0.05$)

Os resultados obtidos demonstram que ambas abordagens meta-heurísticas apresentam contribuições ao teste de software concorrente. A abordagem *Random*, ou aleatória, possui um baixo custo de geração e é amplamente utilizada como *baseline* para estudos experimentais no contexto de geração automática de dados de teste. Neste estudo, é possível observar que há diferenças entre todos as abordagens avaliadas e que as abordagens meta-heurísticas apresentam melhores resultados. Ainda assim, nota-se uma diferença significativa que pode demonstrar maior contribuição na abordagem BioConcST.

Portanto, a hipótese nula (H_0) que supõe que não existe diferença entre as abordagens analisadas é rejeitada com um nível de confiança de 95%. Em outras palavras, percebe-se que avaliação de *fitness* considerando apenas a cobertura como critério de ava-

liação fornece bons resultados ao processo de seleção de dados de teste, entretanto, quando realizada a distinção e classificação de indivíduos em função da distância do requisito de teste, os resultados podem ser mais promissores.

QP1.3 A avaliação baseada na distância apresenta contribuições ao processo de busca?

A análise realizada para o esclarecimento desta questão de pesquisa busca investigar se as variáveis cobertura e *fitness* possuem uma relação a partir dos dados extraídos. Essa questão endereça uma possível ameaça que pode ocorrer quando o valor de *fitness* não corresponde ao critério de parada do algoritmo de busca, neste caso a cobertura do critério *all-sync-edges*.

Em vista disso, uma análise de correlação entre as variáveis *Fitness* e Cobertura foi conduzida para medir o grau de dependência entre as variáveis. A Figura 21 apresenta os resultados dessa análise.

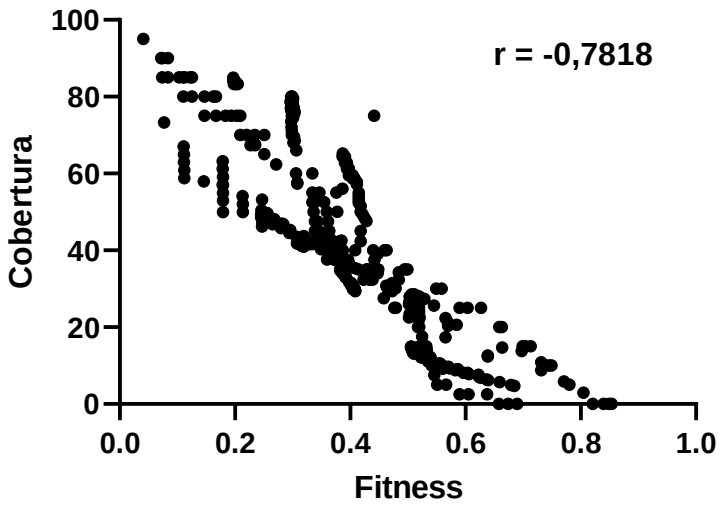


Figura 21 – Coeficiente de correlação entre as variáveis Cobertura e *Fitness*. Os dados foram submetidos ao teste não paramétrico: Correlação de Spearman. N=1400, r = -0,7818 e p<0.0001

Os resultados exibidos na Figura 21 indicam que há uma correlação negativa forte entre as duas variáveis, uma vez que r= -0,7818, para o qual rejeita-se a hipótese nula (H_0) com um nível de extrema significância (p<0.0001). Em outras palavras, este resultado demonstra que há uma relação inversamente proporcional entre as variáveis *fitness* e cobertura.

Desse modo, a minimização da função de *fitness* na BioConcST apresenta resultados favoráveis em direção a cobertura de sincronizações/comunicações dos programas concorrentes, reforçando que a análise de distância entre o caminho percorrido pelo indivíduo e o objetivo de teste pode ser uma estratégia promissora na avaliação dos dados de testes (indivíduos).

Os resultados descritos acima ilustram cenários onde a atribuição de distância na função de *fitness* pode contribuir para o teste de programas concorrentes. Embora os programas utilizados na avaliação não sejam provenientes da indústria, o conjunto de *benchmarks* utilizados neste estudo possui características essenciais e diversificadas de programas concorrentes. Em virtude disso, acredita-se que esse resultado ainda poderia ser generalizado com o uso de programas reais na avaliação.

Os resultados caracterizados na QP1.1 indicam que a estratégia de avaliação proposta pode contribuir não apenas para melhorar a cobertura do critério *all-sync-edges* em relação à abordagem *CoverageBased*, mas também demonstra ser mais indicada em relação ao custo, pois existem evidências significativas que demonstram melhores resultados em todos os níveis de gerações investigados.

Essa descoberta é de extrema importância para o contexto de programas concorrentes, pois há um alto custo computacional envolvido no teste dessas aplicações, principalmente quando os requisitos de testes envolvidos são direcionados a comunicação e sincronização desses programas. Em adição, o uso de meta-heurísticas, embora muito útil, insere custo adicional ao processo de teste, pois em sua maioria requer a execução do programa para avaliação dos indivíduos.

A análise conduzida na QP1.2, ilustrada na Figura 20, reforça a importância do uso de meta-heurísticas, neste caso o Algoritmo Genético, para o teste de programas concorrentes. Vale ressaltar que, embora seja uma técnica simples e de baixo custo, a geração aleatória é promissora em muitos cenários relacionados à geração de dados de teste, ou seja, os resultados obtidos demonstram um avanço significativo para o teste das aplicações concorrentes com o uso de meta-heurísticas, em especial a abordagem BioConcST que apresenta os melhores resultados dessa avaliação.

A QP1.3, conforme mencionado anteriormente, endereça um desafio importante na condução deste trabalho, uma vez que o sucesso de uma meta-heurística está intimamente ligado a representação correta do problema em sua estrutura genérica de otimização. A projeção incorreta dessa representação pode implicar no uso de uma técnica de alto custo com resultados similar ao de uma técnica aleatória.

Nesse sentido, observa-se que a relação entre as variáveis *fitness* e cobertura, ilustrada na Figura 21, indica uma relação positiva na avaliação dos indivíduos e na representação do problema em função da cobertura do critério *all-sync-edges*. Esse resultado reforça as indagações anteriores sobre a contribuição da abordagem BioConcST na geração de dados de testes para o cenário de programas concorrentes.

De modo geral, observa-se que a atribuição de distância durante a avaliação de *fitness* para seleção dos dados de teste pode ser promissora para investigar os aspectos de comunicação e sincronização desse tipo de aplicação. Em virtude disso, acredita-se que a

proposta deste estudo pode ser uma boa alternativa para identificar os principais tipos de erros das aplicações concorrentes, a observabilidade e o travamento. Em ambos os tipos de erros observa-se a comunicação e a sincronização envolvidas nos cenários de falha, sendo assim, a cobertura de um critério que explora essas características associada ao dado de teste e seu respectivo caminho determinístico de execução pode ser a composição chave para determinar a presença de defeitos em aplicações concorrentes.

5.4.2 QP2 Qual a influência da combinação entre os fatores cobertura, fitness e originalidade durante a seleção de indivíduos no processo de busca?

A questão QP2 endereça a segunda principal contribuição desta tese de doutorado, a qual tem por objetivo a avaliar o operador genético de seleção denominado FuzzyST. A motivação para criação de um novo operador de seleção surge em um estudo prévio, no qual encontrou-se evidências que a utilização de indivíduos inicialmente ”inaptos”poderia contribuir ao processo evolutivo da BioConcST (Versão 1.0) (VILELA *et al.*, 2019).

Em seguida, observou-se que esses indivíduos poderiam apresentar características originais em relação aos melhores indivíduos da população. Contudo, a originalidade desses indivíduos não era a única informação relevante para o processo de busca. Por essa razão, o operador foi elaborado a partir de três informações pertinentes, apresentadas no Capítulo 4. Essas informações são utilizadas como entrada para um sistema de inferência Fuzzy que fornece uma variável de saída, a partir de um conjunto de regras, utilizada para classificação e seleção dos indivíduos que irão compor as próximas gerações.

Nesta análise foram investigados os principais operadores genéticos de seleção disponíveis na literatura, os quais são amplamente utilizados e consolidados em diferentes áreas de conhecimento para otimização, objetivando a comparação e avaliação do operador FuzzyST. As subquestões seguintes visam detalhar esta questão de pesquisa sob diferentes pontos de investigação.

QP2.1 A inferência *fuzzy* a partir das informações cobertura, *fitness* e originalidade contribui para formação de novos indivíduos melhores?

Esta questão de pesquisa envolve a premissa principal de um AG, a qual enfatiza que o processo de busca contribui para seleção de indivíduos melhores no decorrer das gerações. Esse princípio é alcançado, principalmente, pelos operadores genéticos que circundam o processo de busca. Para tal, a avaliação desta questão considerou a comparação do valor de *fitness* dos melhores indivíduos por níveis de geração (25, 50, 75 e 100) de cada operador de seleção investigado. Na Figura 22 são apresentados os resultados dessa comparação.

Em sua grande maioria os operadores de seleção empregam suas estratégias uti-

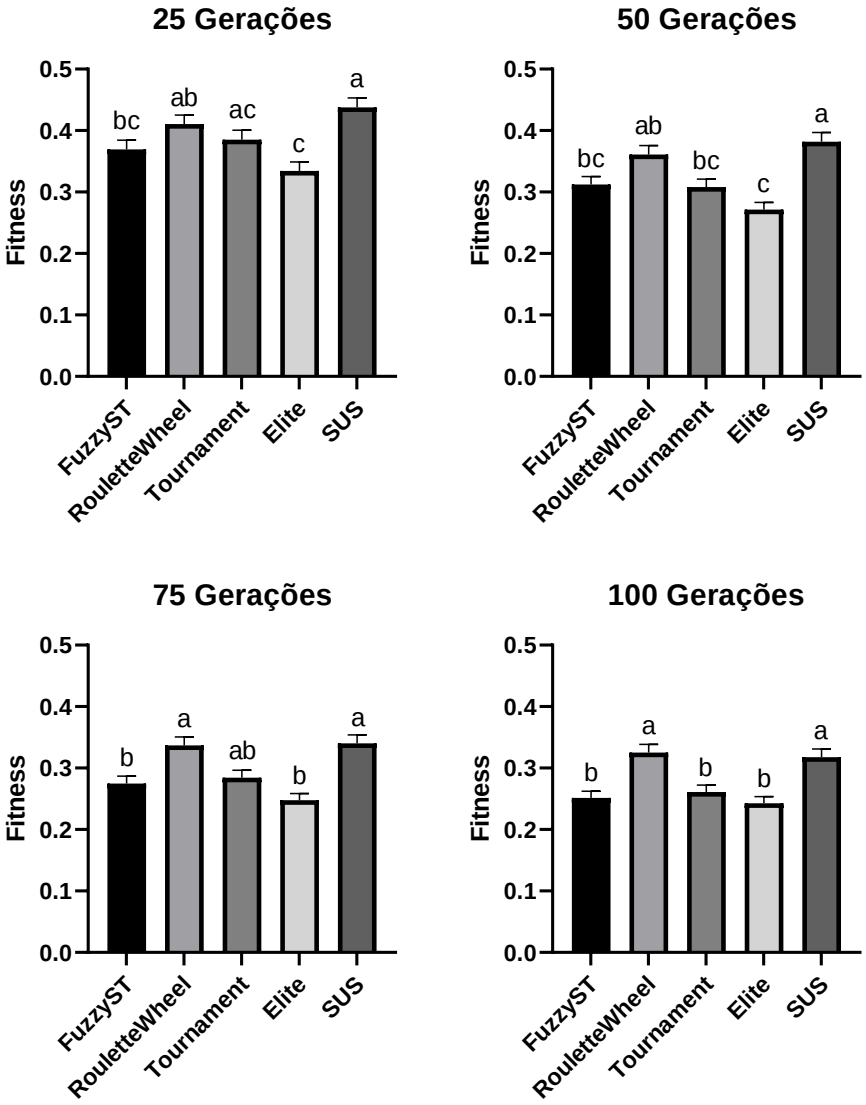


Figura 22 – Análise de *fitness* dos melhores indivíduos ao longo de diferentes níveis de gerações entre os operadores genéticos de seleção: FuzzyST, Roleta (RouletteWheel), Torneio (Tournament), Elitismo (Elite) e *Stochastic Universal Sampling* (SUS). Os dados foram analisados pelo teste não paramétrico Kruskal-Wallis. N=280. Os resultados foram submetidos ao teste de comparações múltiplas de Dunn. Dados expressos como média ± erro padrão da média. Valores diferentes são marcados com letras diferentes sobrescritas ($p < 0.05$)

lizando como referência o valor de *fitness* na seleção dos indivíduos. Nesse sentido, a obtenção de bons indivíduos também está associada a qualidade do operador de seleção pra direcionar o processo evolutivo ao longo das gerações. Os resultados obtidos, em razão do valor de *fitness*, sugerem diferentes cenários para os operadores investigados, contudo, percebe-se que em todos os níveis de gerações os critérios *FuzzyST*, Torneio e Elitismo apresentam os melhores resultados dessa avaliação.

Embora os resultados não demonstrem diferença significativa entre o operador *FuzzyST* e os operadores Torneio e Elitismo, a hipótese nula (H_0) é rejeitada com um

nível de significância de 95%, pois há diferença entre o operador proposto e os operadores Roleta e *Stochastic Universal Sampling* para todos os níveis de gerações investigados.

QP2.2 A convergência precoce pode ser evitada com o uso da combinação entre originalidade, fitness e cobertura?

A questão QP2.2 investiga um problema recorrente na proposição de abordagens que empregam o Algoritmo Genético como estratégia de otimização. A convergência precoce pode induzir o processo de busca a ótimos locais, os quais não refletem os melhores indivíduos da população. Nesse sentido, o operador de seleção pode ser um dos principais fatores que contribuem para convergência precoce durante o processo de busca.

Considerando o contexto apresentado, esta análise investigou os métodos de seleção observando o número de gerações que um processo de busca chega ao seu estado estacionário, isto é, o processo não consegue encontrar melhores indivíduos. Nesse sentido, para esta análise definiu-se que após 10 gerações sem alteração no melhor valor de *fitness* o processo chegou ao seu estado estacionário. Esse número representa 10% de todo o processo de busca do experimento, sendo considerado um alto custo computacional para o processo de teste. Além disso, também observou-se a cobertura alcançada até que o processo chegasse ao seu estado estacionário. A Figura 23 ilustra os resultados dessa análise.

Os resultados desta análise indicam que não há diferença entre os operadores de seleção em relação à análise de convergência, ou seja, todos os operadores apresentaram comportamentos semelhantes em relação ao estado estacionário. Esta descoberta é vista como um resultado significativo para o operador *FuzzyST*, pois todos os demais operadores analisados já são amplamente utilizados em outros estudos com resultados positivos. Contudo, observa-se diferenças em relação à cobertura do critério *all-sync-edges* entre os operadores investigados.

Nesta comparação observa-se que os resultados indicam um melhor cenário para os operadores *FuzzyST* e Elitismo, uma vez que não há diferença significativa entre os mesmos e ambos apresentam diferença entre quase todos os demais operadores. Por outro lado, os operadores *Stochastic Universal Sampling* e Roleta apresentam resultados inferiores aos demais operadores. O operador torneio apresenta-se de forma intermediária entre os demais operadores, pois os resultados não diferentes aos piores operadores dessa avaliação e também entre um dos melhores.

Diante dos resultados apresentados, observa-se como o operador de seleção pode influenciar na qualidade dos conjuntos de testes otimizados. Embora os operadores sejam equivalentes na busca por ótimos globais, a cobertura dos requisitos de teste foi comprometida com o uso de alguns operadores. Vale ressaltar que os baixos valores de cobertura e do número de gerações possuem explicações plausíveis para os resultados encontrados.

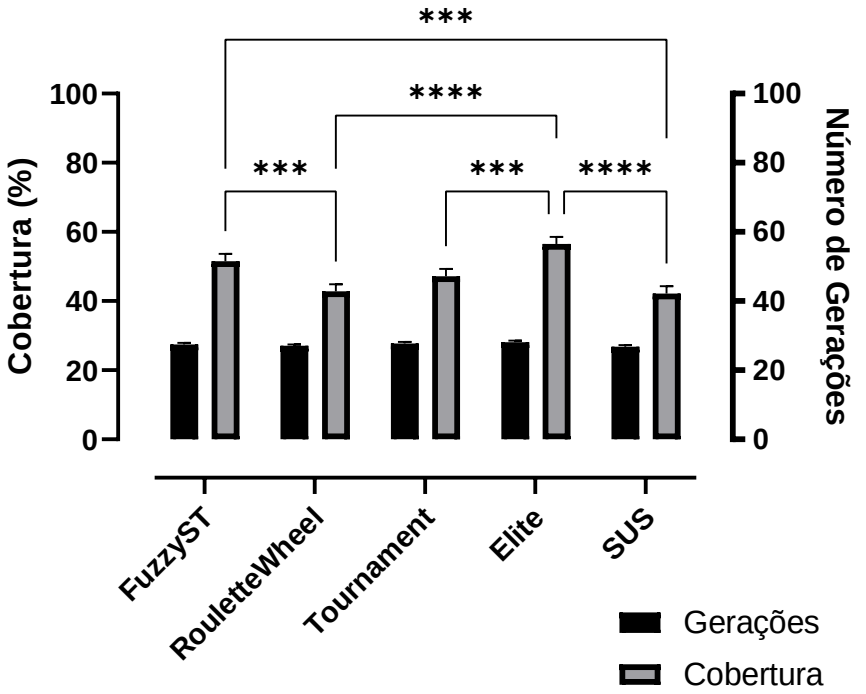


Figura 23 – Análise de Convergência e Cobertura entre os operadores genéticos de seleção: FuzzyST, Roleta (RouletteWheel), Torneio (Tournament), Elitismo (Elite) e *Stochastic Universal Sampling* (SUS). Os dados estão apresentados como média ± erro padrão. Os dados foram analisados por ANOVA Two-way; Dados submetidos ao teste não paramétrico de múltiplas comparações de Tukey; N=280 (*** $p \leq 0.001$, **** $p \leq 0.0001$)

Em relação ao valor de cobertura, existe um problema recorrente na atividade de teste que contribui para que o processo de otimização não alcance o valor máximo de cobertura. O problema é conhecido como requisitos de teste não executáveis, os quais não possuem entradas de teste capazes de exercitá-los.

No que se diz respeito ao baixo número de gerações para que o processo alcance o estado estacionário, acredita-se que o uso de programas considerados simples para o processo de busca contribuem para que a média seja baixa. Alguns dos programas utilizados possuem características essenciais para os programas concorrentes, contudo, na maioria dos casos, o alcance dessas características não estão em função da seleção da entrada de teste, tonando o problema simples para o processo de busca.

No que diz respeito ao ponto central desta questão de pesquisa, embora exista a presença de programas simples na avaliação, acredita-se que a convergência precoce pode ser evitada no operador FuzzyST. As razões para essa indagação consideram a equivalência entre os operadores investigados e a cobertura alcançada pelo operador proposto. Nessa linha, acredita-se que os resultados são significativos existindo a possibilidade de uma generalização para outros tipos de cenários.

QP2.3 Os operadores de seleção baseados exclusivamente no *fitness* são as melhores alternativas para otimização do teste de software concorrente?

Em complemento as questões anteriores, esta questão de pesquisa busca esclarecer se os operadores de seleção apresentam diferenças em relação à cobertura do critério *all-sync-edges*. Contudo, esta avaliação não considera como critério de parada um valor de estado estacionário, mas sim números fixos de gerações que representam o custo do processo de otimização sob diferentes níveis de gerações. Os resultados desta análise são ilustrados na Figura 24.

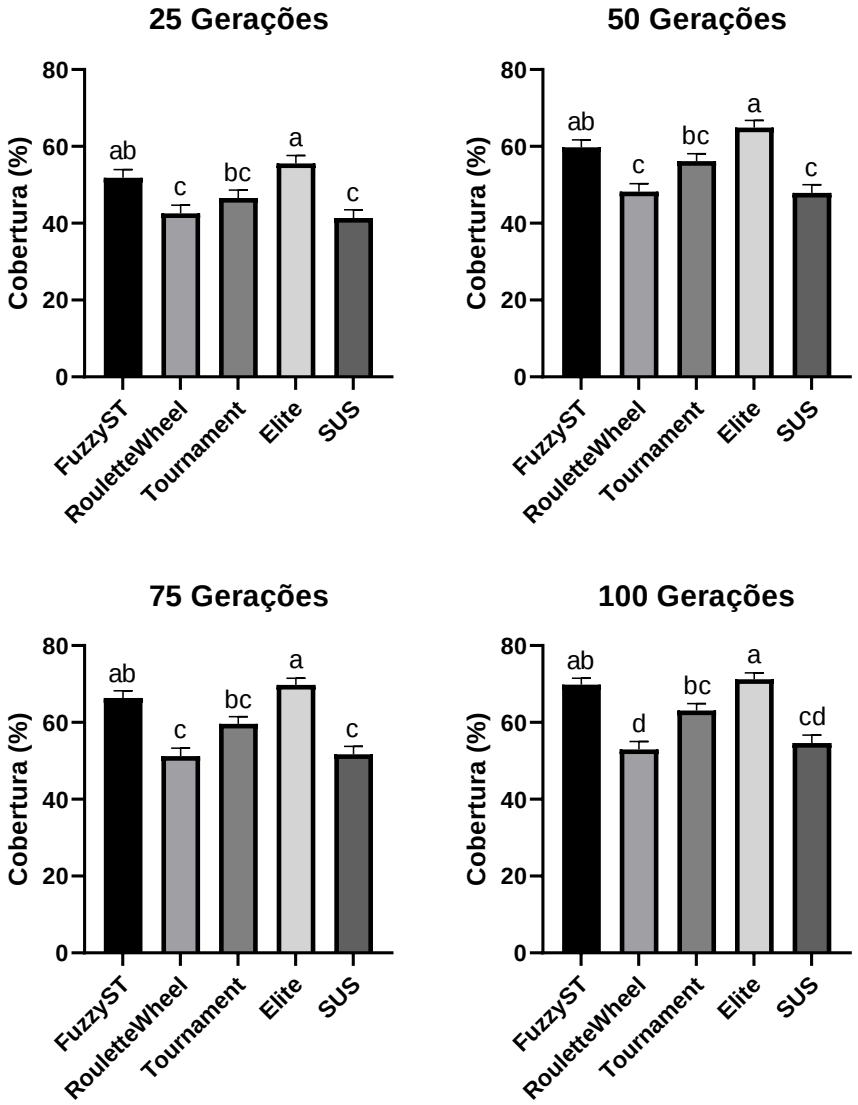


Figura 24 – Análise de cobertura ao longo de diferentes níveis de gerações entre os operadores genéticos de seleção: FuzzyST, Roleta (RouletteWheel), Torneio (Tournament), Elitismo (Elite) e *Stochastic Universal Sampling* (SUS). Os dados foram analisados pelo teste não paramétrico Kruskal-Wallis. N=280. Os resultados foram submetidos ao teste de comparações múltiplas de Dunn. Dados expressos como média ± erro padrão da média. Valores diferentes são marcados com letras diferentes sobrescritas ($p < 0.05$)

Os resultados reforçam novamente a superioridade dos operadores *FuzzyST* e Elitismo em todos os níveis de gerações, os quais permitem rejeitar a hipótese nula (H_0). O operador Torneio ainda apresenta-se de forma intermediária nos níveis 25, 50 e 75, entretanto, no nível de 100 gerações percebe-se um declínio expressivo do operador Roleta em relação aos demais operadores, sendo possível notar não apenas a diferença com os operadores *FuzzyST* e Elitismo, mas também com o operador Torneio.

Esses resultados revelam uma característica importante sobre operadores elitistas para o contexto analisado. Embora o operador Torneio seja conhecido por manter uma diversidade na população pela determinação aleatória de uma subpopulação para seleção dos indivíduos, a estratégia de seleção é similar ao método elitista, onde o indivíduo selecionado sempre será aquele de maior *fitness* entre os demais indivíduos dessa subpopulação. Os operadores elitistas tendem a diminuir a diversidade entre os indivíduos de uma população, enquanto os operadores como Roleta e SUS tendem a manter maior diversidade da população.

Há uma descoberta nesse sentido, pois, embora o elitismo esteja entre os melhores operadores desta investigação, o operador FuzzyST possui mecanismos para garantia de diversidade em seu processo de seleção e ainda assim apresenta resultados significativos. Diante disso, observou-se que uma das possíveis razões para esse resultado seria a complexidade dos programas investigados, considerando que o elitismo seja um operador promissor para problemas de baixa complexidade.

Considerando os aspectos apresentados, optou-se por investigar individualmente programas que apresentassem maior complexidade ao processo de busca utilizando-se como critérios o número de requisitos de teste o número de argumentos que constituem a entrada do programa e os paradigmas de comunicação envolvidos. Conforme os dados apresentados na Tabela

O programa *Jacobi* possui o maior número de requisitos de teste entre todos os demais *benchmarks* analisados, totalizando 532 pares de sincronização (*Send/Receive*). Acredita-se que uma parte desses requisitos é considerada não executável, considerando a baixa cobertura (28%) em todos os operadores investigados. Contudo, ainda trata-se de um problema complexo para encontrar soluções aproximadas de sistemas lineares.

Os resultados apresentados na Figura 25 indicam que o operador FuzzyST é mais promissor no processo de busca para o cenário analisado. Essa superioridade pode ser observada a partir de 50 gerações, em que o operador proposto apresenta diferença significativa entre todos os demais operadores analisados. Ainda assim, percebe-se que há diferença significativa entre os operadores FuzzyST e SUS já no nível de 25 gerações.

Acredita-se que esses resultados são, em parte, contraditórios aos resultados anteriores em razão da complexidade do software sob teste, pois nota-se que apesar de todos os

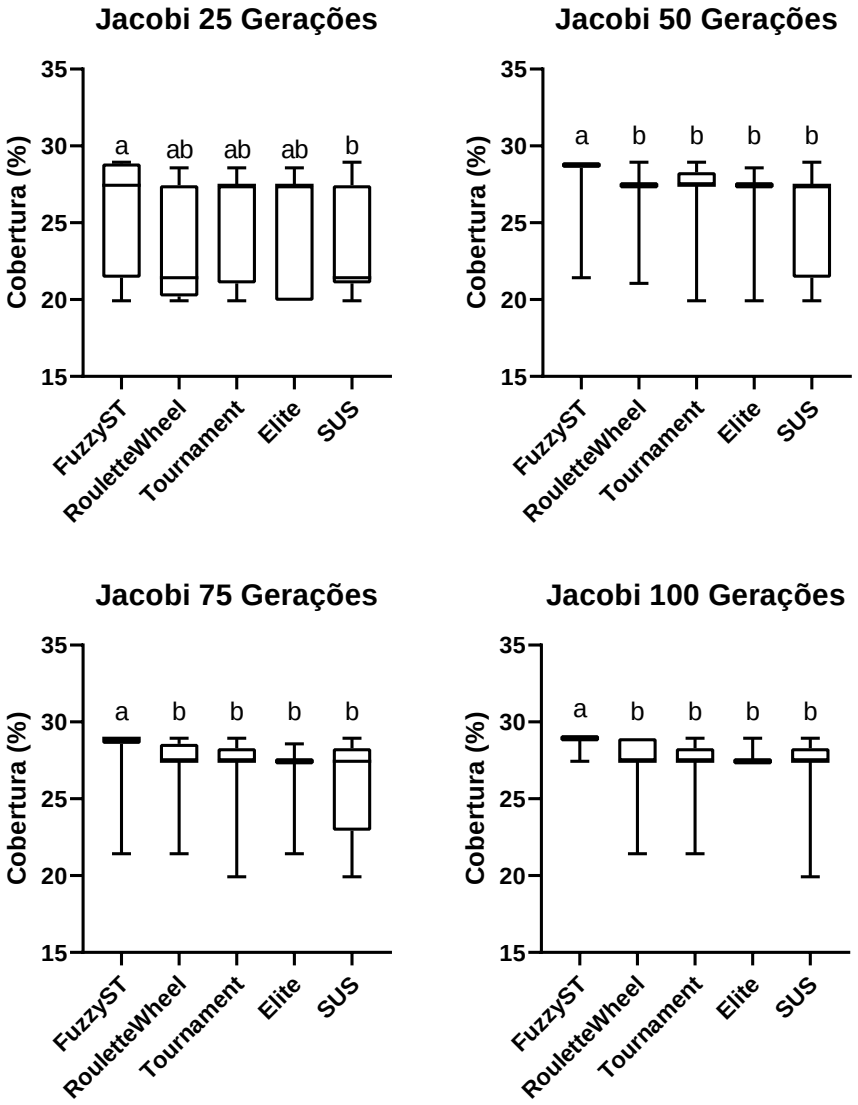


Figura 25 – Análise de cobertura do *benchmark Jacobi* ao longo de diferentes níveis de gerações entre os operadores genéticos de seleção: FuzzyST, Roleta (RouletteWheel), Torneio (Tournament), Elitismo (Elite) e *Stochastic Universal Sampling* (SUS). Os dados foram analisados pelo teste não paramétrico Kruskal-Wallis. N=20. Os resultados foram submetidos ao teste de comparações múltiplas de Dunn. Dados expressos como média ± erro padrão da média. Valores diferentes são marcados com letras diferentes sobrescritas ($p < 0.05$)

operadores apresentarem melhorias ao longo do número de gerações, o operador FuzzyST apresenta menor variação nos resultados obtidos, como pode ser observado nos *boxplots* ilustrados.

O *benchmark Token Ring Broadcast* apresenta características singulares se comparado ao restante dos demais programas, visto que possui ambos os paradigmas de comunicação (passagem de mensagem e memória compartilhada) e utiliza-se de um padrão de comunicação entre vizinhos. Além disso, possui um espaço de busca com maior complexidade em função das possíveis combinações de entradas que resultam em diferentes

tipos de funcionalidades.

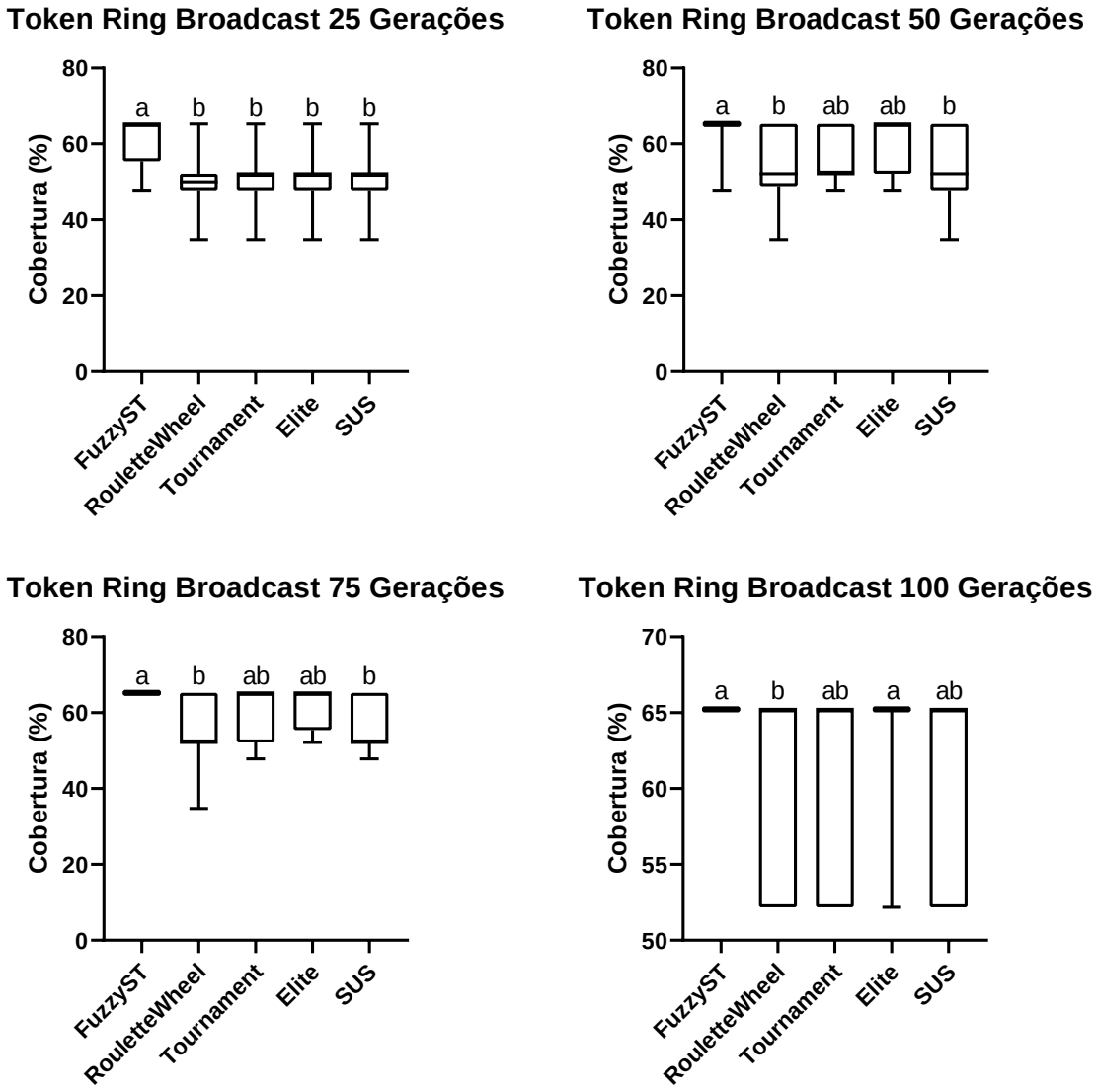


Figura 26 – Análise de cobertura do *benchmark Token Ring Broadcast* ao longo de diferentes níveis de gerações entre os operadores genéticos de seleção: FuzzyST, Roleta (RouletteWheel), Torneio (Tournament), Elitismo (Elite) e *Stochastic Universal Sampling* (SUS). Os dados foram analisados pelo teste não paramétrico Kruskal-Wallis. N=20. Os resultados foram submetidos ao teste de comparações múltiplas de Dunn. Dados expressos como média ± erro padrão da média. Valores diferentes são marcados com letras diferentes sobrescritas ($p < 0.05$)

Os resultados obtidos ilustram um cenário inverso ao citado anteriormente, pois a principal diferença observada ocorre no nível de 25 gerações. A cobertura alcançada é relativamente alta já nas primeira interações, especialmente no operador FuzzyST. Os resultados indicam que o operador proposto alcança um possível ótimo global já no nível de 25 gerações, uma vez que não há grande diferença entre os demais níveis e o restante dos requisitos de teste podem ser não executáveis.

Esses resultados apontam duas contribuições principais, a primeira contribuição

é em relação à cobertura obtida, pois em todos os níveis o operador FuzzyST encontra-se entre os melhores operadores. Em seguida, observa-se que o custo também é uma contribuição importante diante dos resultados obtidos, considerando a superioridade no primeiro nível de gerações e a descoberta de um possível ótimo global com apenas 25 iterações no processo de busca.

Na literatura existem estudos que visam identificar e remover os requisitos não executáveis de programas concorrentes (VERGILIO; MALDONADO; JINO, 2006). Nessa situação, percebe-se que o cenário descoberto pode ser ainda mais promissor considerando a ausência ou diminuição de falsos positivos que dificultam o processo de busca.

Considerando o fator central desta questão de pesquisa, não é possível afirmar que os operadores baseados apenas no *fitness* são piores ou melhores na geração automática de dados de teste para programas concorrentes, uma vez que os resultados ilustram cenários onde há equivalência entre os operadores FuzzyST, Elitismo e Torneio. Entretanto, conforme descrito anteriormente, há uma suposição que a complexidade no processo de busca pode ser melhor tratada com o uso do operador proposto. No entanto, só é possível afirmar essa indagação sob o ponto de vista dos cenários investigados, uma futura experimentação sob essa ótica pode afirmar ou não essa hipótese.

5.5 Considerações Finais

Neste capítulo foram apresentados o design experimental do estudo e os resultados obtidos por meio da condução dos experimentos. Primeiramente foram definidas as variáveis e os fatores que compuseram essa análise, em seguida apresentam-se discussões frente aos resultados identificados.

Considerando os resultados apresentados, foi possível observar contribuições significativas da abordagem BioConcST, a qual apresenta resultados satisfatórios em todas as análises realizadas na questão QP1. Por sua vez, o operador FuzzyST, investigado na questão QP2, apresenta resultados significativos na seleção de indivíduos. Apesar disso, não é possível afirmar que o operador proposto é melhor que o operador de Elitismo. Vários cenários avaliados indicaram que esse operador é promissor, o que motiva estudos futuros.

CAPÍTULO

6

CONCLUSÕES

6.1 Caracterização da Contribuição

As aplicações concorrentes estão cada vez mais presentes no contexto atual. Essas aplicações têm contribuído significativamente para melhorar o desempenho da execução de tarefas simples até as mais complexas. A qualidade desses produtos impacta diretamente na percepção dos usuários finais sobre o desempenho e assertividade na execução de tarefas. Diante disso, a geração automática de dados de teste no contexto de programas concorrentes visa otimizar o processo de seleção de entradas de teste significativas contribuindo para o teste dessas aplicações.

O estudo desenvolvido nesta tese almejou melhorar o processo de geração automática de dados de teste para programas concorrentes sob a seguinte hipótese: ***É possível melhorar a geração de dados de teste para programas concorrentes, a partir da diferenciação entre dados de testes que falharam na cobertura de um requisito de teste. Além disso, é possível melhorar o processo genético de seleção considerando não apenas o valor de fitness, mas também a originalidade e o valor alcançado de cobertura global de um dado de teste.***

A pesquisa desenvolvida resultou na proposição de uma abordagem bioinspirada, denominada BioConcST, que apresenta novas contribuições em relação: i) ao artefato de saída do processo de geração; ii) ao processo de avaliação de *fitness* dos indivíduos de teste; e iii) ao processo evolucionário de seleção de indivíduos. Nessa linha, foram definidas questões de pesquisa que abordam essas contribuições sob uma perspectiva de análise experimental.

A partir dos resultados, foi possível esclarecer as questões de pesquisa que relatam avanços importantes em direção ao teste de software concorrente. Em relação à questão QP1, que investiga a estratégia de avaliação de *fitness*, observou-se que a abordagem Bi-

oConcST supera as demais abordagens avaliadas em todos os cenários considerados. Esse resultado reforça a premissa de que mesmo os indivíduos de teste que não alcançaram um determinado requisito, podem possuir informações importantes que conduzam o processo de busca em direção ao requisito de teste.

Em relação à questão de pesquisa QP2, que analisa a estratégia de seleção do operador genético FuzzyST, constatou-se que o uso de informações complementares no processo de seleção pode ser visto como promissor se comparado aos operadores Roleta e SUS. Apesar disso, considerando a avaliação geral do conjunto de *benchmarks*, não foi possível afirmar que esse operador apresenta diferença significativa em relação aos operadores Elite e Torneio, embora todos esses estejam com os melhores resultados da avaliação.

Ainda assim, foi possível constatar que os cenários que possuem maior complexidade para atividade de teste foram melhor explorados quando submetidos ao operador genético de seleção FuzzyST. Entretanto, para uma melhor constatação dessa descoberta, faz-se necessário uma investigação sob um número maior de *benchmarks* com alto nível de complexidade.

6.2 Contribuições Principais

O principal objetivo desta tese de doutorado foi a definição da abordagem BioConcST, a qual visa contribuir com teste de software concorrente otimizando a etapa de seleção de entradas de teste. Essa abordagem pode ajudar na detecção de defeitos concorrentes que estejam ligados, principalmente, aos mecanismos de interação entre processos ou *threads*, os quais compreendem os principais e mais complexos tipos de defeitos. As principais contribuições deste trabalho são descritas a seguir:

- **Composição do dado de teste com o caminho de execução;**
- **Mecanismo de reexecução do dado de teste sob as condições no qual foi gerado;**
- **Avaliação de distância entre requisito de teste e caminho percorrido para diferenciação entre indivíduos de teste; e**
- **Proposição de um novo operador genético de seleção, denominado FuzzyST, sob as variáveis *fitness*, originalidade e cobertura.**

6.3 Trabalhos Futuros

Ao longo da produção científica desta tese de doutorado, foi possível identificar novas lacunas de pesquisa que, se solucionadas, podem contribuir diretamente para melhorias na abordagem BioConcST e, consequentemente, no teste de aplicações concorrentes. A seguir destaca-se possíveis trabalhos futuros que devem dar continuidade as contribuições desta tese.

- Replicação do estudo experimental considerando a análise de eficácia em revelar defeitos concorrentes, parâmetros diferentes de configuração genética e *benchmarks* de maior complexidade;
- Definição de novos operadores genéticos de *Crossover* e Mutação com implicações diretas ao teste de software concorrente, tais como: alteração nas sequências dos pares de sincronização e cruzamento entre caminhos de execução;
- Definição de mecanismos de instrumentação de código concorrente de forma dinâmica, visando abranger um número maior de aplicações; e
- Evolução do processo de otimização com técnicas de priorização de requisitos, investigando a possibilidade de utilização de uma técnica multi-objetiva.

REFERÊNCIAS

AHMED, M. A.; HERMADI, I. Ga-based multiple paths test data generator. **Computers and Operations Research**, Elsevier Science, v. 35, n. 10, p. 3107–3124, out. 2008. ISSN 0305-0548. Citado na página 46.

ALBERT, E.; ARENAS, P.; GÓMEZ-ZAMALLOA, M.; ROJAS, J. M. Test case generation by symbolic execution: Basic concepts, a clp-based instance, and actor-based concurrency. In: _____. **Formal Methods for Executable Software Models: 14th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2014, Bertinoro, Italy, June 16-20, 2014, Advanced Lectures**. Cham: Springer International Publishing, 2014. p. 263–309. ISBN 978-3-319-07317-0. Disponível em: <https://doi.org/10.1007/978-3-319-07317-0_7>. Citado na página 70.

ALEB, N.; KECHID, S. Automatic test data generation using a genetic algorithm. In: MURGANTE, B.; MISRA, S.; CARLINI, M.; TORRE, C. M.; NGUYEN, H.-Q.; TANIAR, D.; APDUHAN, B. O.; GERVASI, O. (Ed.). **Computational Science and Its Applications – ICCSA 2013**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. p. 574–586. ISBN 978-3-642-39643-4. Citado na página 23.

ALI, S.; ARCAINI, P.; PRADHAN, D.; SAFDAR, S. A.; YUE, T. Quality indicators in search-based software engineering: An empirical evaluation. **ACM Trans. Softw. Eng. Methodol.**, Association for Computing Machinery, New York, NY, USA, v. 29, n. 2, mar. 2020. ISSN 1049-331X. Disponível em: <https://doi.org/10.1145/3375636>. Citado na página 74.

ALMASI, G. S.; GOTTLIEB, A. **Highly Parallel Computing**. 2nd. ed. Redwood City, CA, USA: Benjamin-Cummings Publishing Co., Inc., 1994. ISBN 0805304436 9780805304435. Citado na página 22.

AMMANN, P.; OFFUTT, J. **Introduction to Software Testing**. 1. ed. New York, NY, USA: Cambridge University Press, 2008. ISBN 0521880386, 9780521880381. Citado nas páginas 36 e 39.

ARORA, V.; BHATIA, R.; SINGH, M. Synthesizing test scenarios in uml activity diagram using a bio-inspired approach. **Computer Languages, Systems & Structures**, v. 50, p. 1 – 19, 2017. ISSN 1477-8424. Disponível em: <http://www.sciencedirect.com/science/article/pii/S1477842417300209>. Citado na página 70.

BAGNARA, R.; CARLIER, M.; GORI, R.; GOTLIEB, A. Symbolic path-oriented test data generation for floating-point programs. In: **Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on**. [S.l.: s.n.], 2013. p. 1–10. Citado na página 46.

BAKER, J. E. Reducing bias and inefficiency in the selection algorithm. In: **Proceedings of the Second International Conference on Genetic Algorithms on Genetic**

Algorithms and Their Application. USA: L. Erlbaum Associates Inc., 1987. p. 14–21. ISBN 0805801588. Citado na página 52.

BARBOSA, E. F.; CHAIM, M. L.; VINCENZI, A. M. R.; DELAMARO, M. E.; JINO, M.; MALDONADO, J. C. Introdução ao teste de software. In: _____. [S.l.]: Elsevier, 2007. cap. Teste Estrutural, p. 46–76. Citado nas páginas 37 e 38.

BARNEY, B. **Introduction to Parallel Computing**. 2010. Disponível em: <https://computing.llnl.gov/tutorials/parallel_comp/>. Citado na página 32.

BATISTA, R. N. **Otimizando o teste estrutural de programas concorrentes : uma abordagem determinística e paralela.** Dissertação (Mestrado) — Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, 2015. Citado na página 45.

BERKA, T.; HAGENAUER, H.; VAJTERSIC, M. A middleware for concurrent programming in mpi applications. In: **40th International Conference on Parallel Processing Workshops (ICPPW)**. [S.l.: s.n.], 2011. p. 269–278. ISSN 1530-2016. Citado na página 30.

BERTOLINO, A. Software testing research: Achievements, challenges, dreams. In: **Future of Software Engineering**. [S.l.: s.n.], 2007. p. 85–103. Citado na página 45.

BOWRING, J. F. **Modeling and Predicting Software Behaviors**. Tese (Doutorado) — Georgia Institute of Technology, Atlanta, GA, USA, 2006. Disponível em: <http://hdl.handle.net/1853/19754>. Citado na página 77.

BOYER, R. S.; ELSPAS, B.; LEVITT, K. N. A formal system for testing and debugging programs by symbolic execution. **SIGPLAN Not.**, ACM, New York, NY, USA, v. 10, n. 6, p. 234–245, abr. 1975. ISSN 0362-1340. Citado na página 24.

Carver, R. H.; Tai, K. . Replay and testing for concurrent programs. **IEEE Software**, v. 8, n. 2, p. 66–74, 1991. Citado na página 43.

CINGOLANI, P.; ALCALá-FDEZ, J. jfuzzylogic: a java library to design fuzzy logic controllers according to the standard for fuzzy control programming. **International Journal of Computational Intelligence Systems**, v. 6, p. 61–75, 2013. ISSN 1875-6883. Citado na página 95.

COLANZI, T. E.; VERGILIO, S. R.; ASSUNçãO, W. K. G.; POZO, A. Search based software engineering: Review and analysis of the field in brazil. **Journal of Systems and Software**, v. 86, n. 4, p. 970–984, 2013. Citado na página 50.

DELAMARO, M.; MALDONADO, J.; JINO, M. **Introdução ao teste de software**. [S.l.]: Elsevier, 2007. ISBN 9788535226348. Citado nas páginas 35, 37 e 39.

_____. **Introdução ao teste de software**. 2 edição. ed. Rio de Janeiro: Elsevier, 2016. Citado na página 23.

DIJKSTRA, E. W. **Cooperating Sequential Processes, Technical Report EWD-123**. [S.l.], 1965. Citado nas páginas 21, 31 e 32.

DOURADO, G. G. M. **Contribuindo para a avaliação do teste de programas concorrentes: uma abordagem usando benchmarks**. Dissertação (Mestrado) — Universidade de São Paulo, 2015. Citado na página 93.

DRIANKOV, D.; HELLENDOORN, H.; REINFRANK, M. **An Introduction to Fuzzy Control**. Springer Berlin Heidelberg, 1993. Disponível em: <https://doi.org/10.1007/978-3-662-11131-4>. Citado na página 53.

EYTANI, Y. Concurrent Java Test Generation as a Search Problem. **Electronic Notes in Theoretical Computer Science**, v. 144, n. 4 SPEC. ISS., p. 57–72, 2006. ISSN 15710661. Citado nas páginas 55 e 56.

GAO, J. Z.; TSAO, J.; WU, Y.; JACOB, T. H.-S. **Testing and Quality Assurance for Component-Based Software**. Norwood, MA, USA: Artech House, Inc., 2003. ISBN 1580534805. Citado nas páginas 37 e 38.

Gay, G. The fitness function for the job: Search-based generation of test suites that detect real faults. In: **2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)**. [S.l.: s.n.], 2017. p. 345–355. Citado na página 73.

GERONIMO, L. D.; FERRUCCI, F.; MUROLO, A.; SARRO, F. A parallel genetic algorithm based on hadoop mapreduce for the automatic generation of junit test suites. In: **2012 IEEE Fifth International Conference on Software Testing, Verification and Validation**. [S.l.: s.n.], 2012. p. 785–793. ISSN 2159-4848. Citado na página 71.

GHIDUK, A. S. Automatic generation of basis test paths using variable length genetic algorithm. **Information Processing Letters**, v. 114, n. 6, p. 304 – 316, 2014. ISSN 0020-0190. Citado na página 46.

GOLDBERG, D. E. **Genetic Algorithms in Search, Optimization and Machine Learning**. 1st. ed. USA: Addison-Wesley Longman Publishing Co., Inc., 1989. ISBN 0201157675. Citado na página 52.

GONG, D.; TIAN, T.; YAO, X. Grouping target paths for evolutionary generation of test data in parallel. **Journal of Systems and Software**, Elsevier Inc., v. 85, n. 11, p. 2531–2540, 2012. ISSN 01641212. Disponível em: <http://dx.doi.org/10.1016/j.jss.2012.05.071>. Citado na página 63.

GRAMA, A.; KARYPIS, G.; KUMAR, V.; GUPTA, A. **Introduction to Parallel Computing**. 2. ed. [S.l.]: Addison-Wesley Longman Publishing Co., Inc., 2003. ISBN 9780201648652. Citado nas páginas 22 e 33.

GUO, H.; RUBIO-GONZáLEZ, C. Efficient generation of error-inducing floating-point inputs via symbolic execution. In: **Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2020. (ICSE '20), p. 1261–1272. ISBN 9781450371216. Disponível em: <https://doi.org/10.1145/3377811.3380359>. Citado na página 24.

GUO, S.; KUSANO, M.; WANG, C. Conc-ise: Incremental symbolic execution of concurrent software. In: **Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering**. New York, NY, USA: ACM, 2016. (ASE

2016), p. 531–542. ISBN 978-1-4503-3845-5. Disponível em: <http://doi.acm.org/10.1145/2970276.2970332>. Citado nas páginas 58, 59 e 70.

GUO, S.; KUSANO, M.; WANG, C.; YANG, Z.; GUPTA, A. Assertion guided symbolic execution of multithreaded programs. In: **Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering**. New York, NY, USA: ACM, 2015. (ESEC/FSE 2015), p. 854–865. ISBN 978-1-4503-3675-8. Disponível em: <http://doi.acm.org/10.1145/2786805.2786841>. Citado na página 70.

HANSEN, P. B. **Operating System Principles**. USA: Prentice-Hall, Inc., 1973. ISBN 0136378439. Citado na página 32.

_____. **The Origin of Concurrent Programming**. New York, NY, USA: Springer-Verlag New York, Inc., 2002. 3-61 p. ISBN 0-387-95401-5. Citado na página 30.

HARMAN, M.; HASSOUN, Y.; LAKHOTIA, K.; MCMINN, P.; WEGENER, J. The impact of input domain reduction on search-based test data generation. In: . New York, NY, USA: Association for Computing Machinery, 2007. (ESEC-FSE '07), p. 155–164. ISBN 9781595938114. Disponível em: <https://doi.org/10.1145/1287624.1287647>. Citado na página 73.

HARMAN, M.; JONES, B. F. Search-based software engineering. **Information and Software Technology**, v. 43, n. 14, p. 833 – 839, 2001. ISSN 0950-5849. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0950584901001896>. Citado na página 74.

HOARE, C. A. R. Monitors: An operating system structuring concept. **Commun. ACM**, Association for Computing Machinery, New York, NY, USA, v. 17, n. 10, p. 549–557, out. 1974. ISSN 0001-0782. Disponível em: <https://doi.org/10.1145/355620.361161>. Citado na página 32.

HUANG, J.; ZHANG, C. Debugging concurrent software: Advances and challenges. **Journal of Computer Science and Technology**, Springer Science and Business Media LLC, v. 31, n. 5, p. 861–868, set. 2016. Disponível em: <https://doi.org/10.1007/s11390-016-1669-8>. Citado na página 43.

Khanna, D.; Purandare, R.; Sharma, S. Verifying and testing concurrent programs using constraint solver based approaches. In: **2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)**. [S.l.: s.n.], 2020. p. 834–838. Citado nas páginas 24, 66 e 67.

KHARI, M.; KUMAR, P. An extensive evaluation of search-based software testing: a review. **Soft Computing**, Springer Science and Business Media LLC, v. 23, n. 6, p. 1933–1946, nov. 2017. Disponível em: <https://doi.org/10.1007/s00500-017-2906-y>. Citado na página 24.

KHMELEVA, E.; HOPGOOD, A. A.; TIPI, L.; SHAHIDAN, M. Fuzzy-logic controlled genetic algorithm for the rail-freight crew-scheduling problem. **KI - Künstliche Intelligenz**, Springer Science and Business Media LLC, v. 32, n. 1, p. 61–75, out. 2017. Disponível em: <https://doi.org/10.1007/s13218-017-0516-6>. Citado na página 52.

Korel, B.; Wedde, H.; Ferguson, R. Automated test data generation for distributed software. In: **[1991] Proceedings The Fifteenth Annual International Computer Software Applications Conference**. [S.l.: s.n.], 1991. p. 680–685. Citado nas páginas 62 e 63.

LEBLANC, T.; MARKATOS, E. Shared memory vs. message passing in shared-memory multiprocessors. In: **Proceedings of the Fourth IEEE Symposium on Parallel and Distributed Processing**. [S.l.: s.n.], 1992. p. 254–263. Citado na página 31.

Lei, Y.; Carver, R. H. Reachability testing of concurrent programs. **IEEE Transactions on Software Engineering**, v. 32, n. 6, p. 382–403, 2006. Citado na página 43.

MACHADO, F. B.; MAIA, L. P. **Arquitetura de Sistemas Operacionais**. 5. ed. [S.l.]: LTC, 2013. Citado na página 30.

MAHALI, P.; ARABINDA, S.; ACHARYA, A. A.; MOHAPATRA, D. P. Test case generation for concurrent systems using uml activity diagram. In: **2016 IEEE Region 10 Conference (TENCON)**. [S.l.: s.n.], 2016. p. 428–435. Citado na página 70.

MAIRHOFER, S.; FELDT, R.; TORKAR, R. Search-based software testing and test data generation for a dynamic programming language. In: **Proc. of the 13th Annual Conference on Genetic and Evolutionary Computation**. [S.l.: s.n.], 2011. (GECCO '11). Citado na página 23.

MARINESCU, D. C. Chapter 3 - concurrency in the cloud. In: MARINESCU, D. C. (Ed.). **Cloud Computing (Second Edition)**. Second edition. Morgan Kaufmann, 2018. p. 53–111. ISBN 978-0-12-812810-7. Disponível em: <https://www.sciencedirect.com/science/article/pii/B9780128128107000042>. Citado na página 21.

MCCABE, T. A complexity measure. **IEEE Transactions on Software Engineering**, SE-2, n. 4, p. 308–320, Dec 1976. ISSN 0098-5589. Citado na página 40.

MCMINN, P. Search-based software test data generation: A survey: Research articles. **Softw. Test. Verif. Reliab.**, John Wiley and Sons Ltd., Chichester, UK, v. 14, n. 2, p. 105–156, jun. 2004. ISSN 0960-0833. Citado na página 23.

McMinn, P. Search-based software testing: Past, present and future. In: **IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops**. [S.l.: s.n.], 2011. p. 153–163. Citado na página 24.

MILLER, B.; GOLDBERG, D. Genetic algorithms, tournament selection, and the effects of noise. **Complex Syst.**, v. 9, 1995. Citado na página 52.

MIRHOSSEINI, S. M.; HAGHIGHI, H. A search-based test data generation method for concurrent programs. **International Journal of Computational Intelligence Systems**, v. 13, p. 1161–1175, 2020. ISSN 1875-6883. Disponível em: <https://doi.org/10.2991/ijcis.d.200805.003>. Citado na página 24.

MOHI-ALDEEN, S. M.; DERIS, S.; MOHAMAD, R. Systematic mapping study in automatic test case generation. In: **New Trends in Software Methodologies, Tools and Techniques**. [S.l.]: IOS Press Ebooks, 2014, (Frontiers in Artificial Intelligence and Applications). p. 703–720. Citado na página 45.

MYERS, G. J.; SANDLER, C.; BADGETT, T. **The Art of Software Testing**. 3rd. ed. [S.l.]: Wiley Publishing, 2011. ISBN 1118031962, 9781118031964. Citado nas páginas 29 e 39.

NICHOLS, B.; BUTTLAR, D.; FARRELL, J. P. **Pthreads programming - A POSIX standard for better multiprocessing.** [S.l.]: O'Reilly, 1996. ISBN 978-1-56592-115-3. Citado na página 30.

NIKRAVAN, E.; PARSA, S. Path-oriented random testing through iterative partitioning (ip-prt). **Turkish Journal of Electrical Engineering & Computer Sciences**, The Scientific and Technological Research Council of Turkey, v. 27, n. 4, p. 2666–2680, 2019. Citado na página 23.

NISTOR, A.; LUO, Q.; PRADEL, M.; GROSS, T. R.; MARINOV, D. Ballerina: Automatic generation and clustering of efficient random unit tests for multithreaded code. In: **34th International Conference on Software Engineering (ICSE)**. [S.l.: s.n.], 2012. ISSN 0270-5257. Citado na página 24.

PLEROU, A.; VLAMOU, E.; PAPADOPOULOS, V. Fuzzy genetic algorithms: Fuzzy logic controllers and genetics algorithms. **Global Journal For Research Analysi.**, v. 5, n. 11, 2017. Citado na página 52.

PRADO, R. R.; SOUZA, P. S. L.; DOURADO, G. G. M.; SOUZA, S. R. S.; ESTRELLA, J. C.; BRUSCHI, S. M.; LOURENCO, J. Extracting static and dynamic structural information from java concurrent programs for coverage testing. In: **Latin American Computing Conference (CLEI)**. [S.l.: s.n.], 2015. p. 1–8. Citado nas páginas 44 e 94.

PRATIHAR, D. K. **Soft Computing: Fundamentals and Applications**. 1st. ed. [S.l.]: Alpha Science International, Ltd, 2013. ISBN 1842658638. Citado na página 52.

PRESSMAN, R. **Software Engineering: A Practitioners Approach**. 7. ed. New York, NY, USA: McGraw Hill, Inc., 2010. Citado na página 37.

QUINN, M. J. **Parallel Programming in C with MPI and OpenMP**. [S.l.]: McGraw-Hill Education Group, 2003. ISBN 0071232656. Citado nas páginas 32 e 33.

RAI, D.; TYAGI, K. Bio-inspired optimization techniques: A critical comparative study. **SIGSOFT Softw. Eng. Notes**, Association for Computing Machinery, New York, NY, USA, v. 38, n. 4, p. 1–7, jul. 2013. ISSN 0163-5948. Disponível em: <https://doi.org/10.1145/2492248.2492271>. Citado na página 74.

RAPPS, S.; WEYUKER, E. Selecting software test data using data flow information. **IEEE Transactions on Software Engineering**, SE-11, n. 4, p. 367–375, April 1985. ISSN 0098-5589. Citado na página 39.

RüNGER, G.; RAUBER, T. **Parallel Programming - for Multicore and Cluster Systems; 2nd Edition.** [S.l.]: Springer, 2013. ISBN 978-3-642-37800-3. Citado na página 31.

SCALABRINO, S.; GRANO, G.; NUCCI, D. D.; OLIVETO, R.; LUCIA, A. D. Search-based testing of procedural programs: Iterative single-target or multi-target approach? In: **Search Based Software Engineering**. [S.l.: s.n.], 2016. Citado na página 23.

Schimmel, J.; Molitorisz, K.; Jannesari, A.; Tichy, W. F. Automatic generation of parallel unit tests. In: **2013 8th International Workshop on Automation of Software Test (AST)**. [S.l.: s.n.], 2013. p. 40–46. Citado na página 60.

_____. Combining unit tests for data race detection. In: **2015 IEEE/ACM 10th International Workshop on Automation of Software Test**. [S.l.: s.n.], 2015. p. 43–47. Citado nas páginas 60, 61 e 62.

SHULL, F.; MENDONçA, M. G.; BASILI, V.; CARVER, J.; MALDONADO, J. C.; FABBRI, S.; TRAVASSOS, G. H.; FERREIRA, M. C. Knowledge-sharing issues in experimental software engineering. **Empirical Software Engineering**, Kluwer Academic Publishers, v. 9, n. 1-2, p. 111–137, 2004. ISSN 1382-3256. Citado na página 89.

SILVA, J. D. P.; SOUZA, S. R.; SOUZA, P. S. L. Geração automática de dados de teste para programas concorrentes com uso de meta-heurísticas. In: **8th Brazilian Workshop on Systematic and Automated Software Testing (SAST)**. [S.l.: s.n.], 2014. p. 71–80. Citado na página 46.

SOMMERVILLE, I. **Software Engineering**. 9. ed. Harlow, England: Addison-Wesley, 2010. ISBN 978-0-13-703515-1. Citado na página 35.

SOUZA, P. S.; SOUZA, S. R.; ZALUSKA, E. Structural testing for message-passing concurrent programs: an extended test model. **Concurrency and Computation: Practice and Experience**, v. 26, n. 1, p. 21–50, 2014. ISSN 1532-0634. Citado nas páginas 40 e 43.

SOUZA, P. S.; SOUZA, S. S.; ROCHA, M. G.; PRADO, R. R.; BATISTA, R. N. Data flow testing in concurrent programs with message passing and shared memory paradigms. **Procedia Computer Science**, v. 18, p. 149–158, 2013. ISSN 1877-0509. Citado nas páginas 42 e 44.

SOUZA, S.; VERGILIO, S.; SOUZA, P. S. L. Introdução ao teste de software. In: _____. [S.l.]: Elsevier, 2007. v. 1, cap. Teste de Programas Concorrentes, p. 231–249. Citado nas páginas 34 e 41.

SOUZA, S. R. S.; SOUZA, P. S. L.; BRITO, M. A. S.; SIMAO, A. S.; ZALUSKA, E. J. Empirical evaluation of a new composite approach to the coverage criteria and reachability testing of concurrent programs. **Software Testing, Verification and Reliability**, 2015. ISSN 1099-1689. Citado nas páginas 23 e 29.

SOUZA, S. R. S.; VERGILIO, S. R.; SOUZA, P. S. L.; aO, A. S. S.; HAUSEN, A. C. Structural testing criteria for message-passing parallel programs. **Concurrency and Computation: Practice and Experience**, John Wiley & Sons, Ltd., v. 20, n. 16, p. 1893–1916, 2008. ISSN 1532-0634. Citado na página 42.

Steenbuck, S.; Fraser, G. Generating unit tests for concurrent classes. In: **2013 IEEE Sixth International Conference on Software Testing, Verification and Validation**. [S.l.: s.n.], 2013. p. 144–153. ISSN 2159-4848. Citado nas páginas 57 e 58.

SUN, C. ai; ZHAO, Y.; PAN, L.; HE, X.; TOWEY, D. A transformation-based approach to testing concurrent programs using UML activity diagrams. **Software: Practice and Experience**, Wiley-Blackwell, v. 46, n. 4, p. 551–576, apr 2015. Disponível em: <https://doi.org/10.1002/spe.2324>. Citado na página 70.

TANENBAUM, A. S. **Modern Operating Systems**. 3rd. ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2007. ISBN 9780136006633. Citado nas páginas 22, 31 e 32.

THIERENS, D. Selection schemes, elitist recombination, and selection intensity. In: **Proceedings of the 7th International Conference on Genetic Algorithms**. [S.l.]: Morgan Kaufmann, 1998. p. 152–159. Citado na página 52.

TIAN, T.; GONG, D. Evolutionary generation of test data for path coverage of message-passing parallel programs. **China journal of computers**, v. 36, n. 11, p. 2212–2223, 2013. Citado nas páginas 24 e 71.

_____. Evolutionary generation approach of test data for multiple paths coverage of message-passing parallel programs. **Chinese Journal of Electronics**, v. 23, p. 291, 2014. Citado nas páginas 24 e 41.

_____. Test data generation for path coverage of message-passing parallel programs based on co-evolutionary genetic algorithms. **Automated Software Engg.**, Kluwer Academic Publishers, Hingham, MA, USA, v. 23, n. 3, p. 469–500, set. 2016. ISSN 0928-8910. Disponível em: <http://dx.doi.org/10.1007/s10515-014-0173-z>. Citado nas páginas 24, 64, 65 e 71.

VERGILIO, S. R.; MALDONADO, J. C.; JINO, M. Infeasible paths in the context of data flow based testing criteria: Identification, classification and prediction. **Journal of the Brazilian Computer Society**, v. 12, n. 1, p. 73–88, Feb 2006. Citado na página 115.

_____. Introdução ao teste de software. In: _____. [S.l.]: Elsevier, 2007. v. 1, cap. Geração de Dados de Teste, p. 269–313. Citado na página 46.

VILELA, R. F. **Evidências sobre o uso de técnicas de geração automática de dados de teste em programas concorrentes**. Dissertação (Mestrado) — University of São Paulo, 2016. Citado na página 24.

VILELA, R. F.; PINTO, V. H. S. C.; COLANZI, T. E.; SOUZA, S. R. S. Bio-inspired optimization of test data generation for concurrent software. In: NEJATI, S.; GAY, G. (Ed.). **Search-Based Software Engineering**. Cham: Springer International Publishing, 2019. p. 121–136. ISBN 978-3-030-27455-9. Citado nas páginas 25, 68, 69, 76 e 107.

VILELA, R. F.; SOUZA, P. S. L.; DELAMARO, M. E.; SOUZA, S. R. S. Evidências sobre configurações de algoritmos genéticos para geração automática de dados de teste. In: **Proceedings of XIX Ibero-American Conference on Software Engineering**. [S.l.: s.n.], 2016. Citado na página 101.

WAZLAWICK, R. S. **Engenharia de software: conceitos e práticas**. 1. ed. Rio de Janeiro: Elsevier, 2013. Citado na página 40.

WIEGAND, R. P.; LILES, W. C.; JONG, K. A. D. An empirical analysis of collaboration methods in cooperative coevolutionary algorithms. In: **Proceedings of the 3rd Annual Conference on Genetic and Evolutionary Computation**. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001. (GECCO'01), p. 1235–1242. ISBN 1-55860-774-9. Disponível em: <http://dl.acm.org/citation.cfm?id=2955239.2955458>. Citado na página 64.

WILHELMSTÖTTER, F. **Jenetics**. 2020. Disponível em: <http://jenetics.io>. Citado na página 94.

WOHLIN, C.; RUNESON, P.; HöST, M.; OHLSSON, M. C.; REGNELL, B.; WESSLÉN, A. **Experimentation in Software Engineering: An Introduction**. Norwell, MA, USA: Kluwer Academic Publishers, 2000. ISBN 0-7923-8682-5. Citado na página 89.

WONG, W.; LEI, Y.; MA, X. Effective generation of test sequences for structural testing of concurrent programs. In: **Proceedings 10th IEEE International Conference on Engineering of Complex Computer Systems**. [S.l.: s.n.], 2005. p. 539–548. Citado na página 22.

WU, L. L.; KAISER, G. E. **Constructing Subtle Concurrency Bugs Using Synchronization-Centric Second-Order Mutation Operators**. [S.l.], 2011. Citado na página 41.

YUAN, X.; YANG, J. Effective concurrency testing for distributed systems. In: . New York, NY, USA: Association for Computing Machinery, 2020. ISBN 9781450371025. Disponível em: <https://doi.org/10.1145/3373376.3378484>. Citado na página 21.

ZADEH, L. Fuzzy sets. **Information and Control**, v. 8, n. 3, p. 338–353, 1965. ISSN 0019-9958. Disponível em: <https://www.sciencedirect.com/science/article/pii/S001999586590241X>. Citado na página 53.


ICMC
USP
SÃO CARLOS