UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
COORDENADORIA DO CURSO DE ENGENHARIA DE SOFTWARE

MATHEUS DEON BORDIGNON

TESTE DE MUTAÇÃO PARA PROGRAMAS CONCORRENTES EM ELIXIR

TRABALHO DE CONCLUSÃO DE CURSO

DOIS VIZINHOS

2019

MATHEUS DEON BORDIGNON

TESTE DE MUTAÇÃO PARA PROGRAMAS CONCORRENTES EM ELIXIR

Trabalho de Conclusão de Curso apresentado como requisito parcial à obtenção do título de Bacharel em Engenharia de Software, da Universidade Tecnológica Federal do Paraná.

Orientador: Prof. Dr. Rodolfo Adamshuk Silva

DOIS VIZINHOS

2019

TERMO DE APROVAÇÃO

Teste de Mutação para Programas Concorrentes em Elixir

por

Matheus Deon Bordignon

Este Trabalho de Conclusão de Curso foi apresentado em 26 de Novembro de 2019 como requisito parcial para a obtenção do título de Bacharel em Engenharia de Software. O(a) candidato(a) foi arguido(a) pela Banca Examinadora composta pelos professores abaixo assinados. Após deliberação, a Banca Examinadora considerou o trabalho aprovado.

Rodolfo Adamshuk Silva
Presidente da Banca

Francisco Carlos Monteiro Souza
Membro Titular

Rafael Alves Paes de Oliveira
Membro Titular

* A Folha de Aprovação assinada encontra-se na Coordenação do Curso

AGRADECIMENTOS

Agradeço a Deus pelo dom da vida, por tudo que tenho e por tudo que sou. Sua presença é essencial em minha vida e faz ser mais forte a cada dia.

Agradeço a minha família, Alceu, Rosane, Lucilene e Mônica, pelo apoio incondicional sempre, pelas orações, pelo incentivo e pelo amor. Obrigado por tudo que fizeram e fazem por mim, por entenderem minhas ausências nesse período e por sempre estarem a disposição para tudo que eu precisar.

Agradeço ao meu orientador, Prof. Rodolfo pela atenção e apoio desde a primeira conversa, pela disponibilidade, ajuda e, principalmente, pela paciência que teve comigo. Obrigado pelos conselhos e pelas cobranças sempre que necessárias.

Agradeço aos amigos e colegas de curso, pela ajuda e convivência durante este período, Andrei, Burdella, Cunha, Guilherme e Veloso. Agradeço de modo geral aos professores do curso pelos ensinamentos e conhecimentos compartilhados.

RESUMO

Bordignon, M. D. TESTE DE MUTAÇÃO PARA PROGRAMAS CONCORRENTES EM ELIXIR. 103 f. Trabalho de Conclusão de Curso – Coordenadoria do Curso de Engenharia de Software, Universidade Tecnológica Federal do Paraná. Dois Vizinhos, 2019.

A capacidade de processamento dos computadores mostra-se cada vez mais insuficiente e incentiva o uso de programação concorrente para o desenvolvimento de aplicações que reduzam o tempo computacional. O Elixir é uma linguagem que suporta o desenvolvimento de aplicações concorrentes e distribuídas de forma dinâmica e moderna. A utilização da programação concorrente difere-se da programação convencional, adicionando características como a comunicação, sincronização e o não determinismo aos programas. As atividades de teste, que buscam garantir a qualidade e corretude dos programas, tornam-se mais complexas devido a estas características. O teste de mutação é um critério de teste que se baseia nos enganos cometidos por programadores durante o desenvolvimento do software e apresenta uma alta eficácia para revelar defeitos. Este trabalho tem por objetivo a definição de operadores de mutação para funções concorrentes em Elixir. Para isso, um *benchmark* de programas concorrentes em Elixir foi construído e validado para auxiliar na atividade de teste de software. Utilizando o *benchmark*, foi definida a taxonomia de falhas em Elixir, explorando as falhas produzidas pela funções concorrentes do *Kernel* e do módulo *Task*. Considerando as falhas da taxonomia e explorando enganos típicos de programadores durante o desenvolvimento, foi definido um conjunto de operadores de mutação para explorar aspectos de concorrência. Também, foi proposto um experimento para avaliar algumas características de qualidade dos operadores de mutação definidos e os resultados mostraram que o operador de mutação *ReplSpawn* gerou o maior número de mutantes, enquanto o operador *DelReceive* demonstrou mais completude na abrangência da taxonomia de falhas. No experimento também foi calculada a taxa de inclusão dos operadores de mutação de deleção em relação aos operadores *DelSpawn* e *DelTaskStart*.

Palavras-chave: Elixir, Programação Concorrente, Teste de mutação

ABSTRACT

Bordignon, M. D. MUTATION TESTING FOR CONCURRENT PROGRAMS IN ELIXIR. 103 f. Trabalho de Conclusão de Curso – Coordenadoria do Curso de Engenharia de Software, Universidade Tecnológica Federal do Paraná. Dois Vizinhos, 2019.

The processing capacity of computers has been insufficient and encourages the use of concurrent programming for the development of applications that reduce computational time. Elixir is a language that supports the development of concurrent and distributed applications in a dynamic and modern way. The use of concurrent programming differs from sequential programming, adding features such as communication, synchronization, and non-determinism to programs. Testing activities seek to guarantee the quality and correctness of the programs and they become more complex due to these characteristics presents in concurrent programs. Mutation testing is a test criterion based on mistakes made by programmers during software development and is highly effective in revealing defects. The aim of this work is the definition of mutation operators for concurrent functions in Elixir. For this, a benchmark of concurrent programs in Elixir was built and validated to aid in software testing activity. Using the benchmark, the fault taxonomy in Elixir was defined, exploiting the faults produced by the concurrent functions of the *Kernel* and the *Task* module. Considering the faults of the taxonomy and exploiting typical programmer mistakes during development, a set of mutation operators was defined to exploit concurrents aspects. Also, an experiment was proposed to evaluate some quality characteristics of the defined mutation operators and the results showed that the *ReplSpawn* mutation operator generated the largest number of mutants, while the *DelReceive* operator demonstred more completeness in the scope of fault taxonomy. The inclusion rate of the deletion mutation operators in relation to the *DelSpawn* and *DelTaskStart* operators was also calculated.

Keywords: Elixir, Concurrent Programming, Mutation Testing

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ALGORITMOS

LISTA DE SIGLAS

DEMT	*Deterministic Execution Mutation Testing*
GFC	Gráfo de Fluxo de Controle
GQM	*Goal/Question/Metric*
LoC	Linha de Código
MIMD	*Multiple Instruction Stream, Multiple Data Stream*
MISD	*Multiple Instruction Stream, Single Data Stream*
MPI	*Messagem Passing Interface*
OO	Orientado à Objetos
ORRN	*Relational Operator Replacement*
PID	*Process Identifier*
PVM	*Parallel Virtual Machine*
RIP	*Reachability, Infection and Propagation*
SIMD	*Single Instruction Stream, Multiple Data Stream*
SISD	*Single Instuction Stream, Single Data Stream*
SSDL	*Statement Deletion* ou Eliminação de Comandos
STRI	*Trap on IF Condition*
TDD	*Teste Driven-Development*

SUMÁRIO

1 INTRODUÇÃO

Com o aumento da procura por processamento e desempenho, diversos paradigmas de programação vêm sendo utilizados por programadores, inclusive no mesmo código. Um paradigma de programação representa a forma como o programador irá escrever seu código e geralmente são divididos em quatro categorias: imperativa, funcional, lógica e orientada à objetos (SEBESTA, 2012). A evolução nos paradigmas de programação foi impulsionada pela necessidade de separar e gerenciar artefatos, conceitos ou recursos de interesse de um ou mais envolvidos no desenvolvimento de software (SIMMONDS, 2012).

Há uma gama de linguagens de programação funcional e muitas podem ser escolhidas. Porém, para o desenvolvimento do projeto, optou-se pela linguagem Elixir, criada pelo brasileiro José Valim. Inspirada na linguagem de programação também funcional Erlang, o Elixir auxilia a resolução de problemas de escalabilidade, tolerância a falhas e alta concorrência (DAVI, 2017).

A fim de garantir a qualidade e corretude de programas concorrentes, testes devem ser aplicados no programa base durante todo o ciclo de vida do software. Existem diversas técnicas para realizar o teste de uma aplicação, como a técnica funcional ou de caixa preta, a técnica estrutural ou também conhecida como caixa branca, e a técnica de teste baseada em defeitos, escolhida para aplicação neste trabalho.

O teste de mutação é um critério da técnica baseada em defeitos e tem como objetivo melhorar o conjunto de casos de teste inicial. Inicialmente, o programa é executado e nele é aplicado o conjunto de casos de teste para assim verificar sua corretude. Caso algum problema seja identificado, o programa deve ser corrigido nessa etapa e novamente submetido aos testes. Após ser aprovado pelos casos de teste, começa o processo de identificação de operadores de mutação. Continuamente, conjuntos de operadores de mutação vem sendo definidos para cada linguagem. Eles representam erros passíveis de serem cometidos por programadores durante o desenvolvimento do software. Com os operadores mutantes definidos, é possível gerar os mutantes do programa base, ou seja,

programas semelhantes ao programa base, porém com modificações sintáticas. Em seguida, inicia-se a execução dos mutantes e, é essencial que todos os mutantes compilem e sejam passíveis de execução. Para cada mutante, é aplicado o mesmo conjunto de casos de teste aplicados ao programa base e na identificação de um erro ou falha, o mutante é considerado morto. Caso contrário, duas opções são aceitas: melhorar os casos de teste para que estes consigam detectar o mutante ou considerar o mutante equivalente ao programa base (SILVA, 2013).

O principal desafio na execução dos testes para programas concorrentes é o não determinismo, visto que a partir de uma mesma entrada, o mesmo programa pode produzir resultados corretos diferentes. Na tentativa de resolver este problema, algumas técnicas foram definidas como a abordagem DEMT proposta por Carver (1993) para execução determinística dos mutantes. Nessa abordagem, são geradas sequências de sincronização a partir do programa base e um conjunto de entradas, para, em seguida, executar cada mutante com determinada entrada e verificar se produz a mesma sequência de sincronização do programa base. Em Silva, Souza e Souza (2012) foram definidas duas abordagens de execução dos mutantes, uma baseando na DEMT (*Deterministic Executation Mutation Testing*) e outra de forma não determinística onde o resultado obtido pelo mutante era comparado com os resultados obtidos no programa original, sem considerar sequências de sincronização. Apesar do problema do não determinismo e do alto custo de execução quando comparado as demais técnicas, o teste de mutação é amplamente aceito na comunidade do teste de software que o considera um dos critérios mais eficazes para a detecção de problemas e defeitos (DELAMARO; JINO; MALDONADO, 2017).

1.1 CONTEXTUALIZAÇÃO

A linguagem Elixir, embora recente, tem aumentado o número de adeptos a sua comunidade e recebe cada vez mais contribuições (PET-SI, 2018). Utilizando o paradigma funcional e herdando do Erlang a capacidade de resolução de problemas de concorrência, distribuição, tolerância a falhas e alta disponibilidade *hot swap*, a linguagem mostrasse promissora e estimula programadores iniciantes a conhecê-la.

O teste de mutação vem sendo amplamente pesquisado e utilizado em razão de sua capacidade de aplicação a diferentes artefatos do software. Além dos programas que se deseja testar, a segunda entrada necessária para a aplicação do teste é o conjunto operadores de mutação, ou seja, defeitos e alterações colocadas propositalmente no código que estão suscetíveis de ocorrer por descuido do desenvolvedor.

Os mutantes são gerados a partir dos operadores de mutação e de um programa, considerado correto e aprovado por um conjunto de casos de teste. Neste ponto, é importante lembrar que os mutantes não devem conter erros de compilação para que sejam passíveis de execução. As próximas tarefas são a execução dos mutantes e à aplicação dos casos de teste neles.

A prática de teste para programas concorrentes é considerada mais complexa quanto a execução para programas convencionais, principalmente pela característica não determinística dos programas concorrentes, ou seja, sua possibilidade de geração de diferentes resultados a partir de uma mesma entrada.

Os problemas relacionadas à sincronização e comunicação (ditos imprevisíveis) são difíceis de reproduzir. Linguagens emergentes sofrem com a ausência de critérios e ferramentas de teste. Por essa razão, o modo de projetar aplicações chamado "*let it crash*" é utilizado nas comunidades de linguagens emergentes. Esse tipo de mentalidade incentiva a programação do fluxo principal (conhecido também como caminho feliz ou *happy-path*), na qual concentra-se primeiro em cenários sem nenhuma condição de erro (estado incorreto do sistema), permitindo concentração no propósito da aplicação e tratar os caminhos de exceção mais tarde (após eles ocorrerem) (ALBUQUERQUE; CAIXINHA, 2018).

1.2 MOTIVAÇÃO

No contexto de teste de mutação para programas concorrentes, existem poucas iniciativas e trabalhos. No artigo de Silva e Souza (2018), os autores citam como desafios na área de teste de mutação a definição de operadores de mutação e a execução destes, já que ambas as tarefas seriam facilitadas com uma ferramenta que ainda é inexistente para diversas linguagens emergentes, como o Elixir. Outra questão na execução dos mutantes é relacionada a execução determinística dos mutantes, na qual, torna-se necessário guardar as sequências de sincronização utilizadas.

Embora cresça a utilização do Elixir, na literatura não foram encontrados trabalhos relacionados ao Elixir em nenhuma das etapas pertencentes ao teste, situação que motiva a definição de operadores de mutação para programas concorrentes em Elixir, bem como sua execução.

1.3 OBJETIVOS

1.3.1 OBJETIVO GERAL

O objetivo geral deste trabalho de conclusão de curso é a definição de operadores de mutação para programas concorrentes em Elixir através da instanciação da técnica de teste de mutação. A abordagem utiliza mutantes gerados a partir de programas base para avaliar e melhorar sua a qualidade do conjunto de casos de teste inicial. A partir da primeira execução, novos casos de teste são inseridos, melhorando a qualidade do conjunto de casos de teste.

Dada a ausência de teste para programas concorrentes em Elixir, ressalta-se a contribuição desse trabalho no estado da arte, sendo este pioneiro na aplicação de teste de software para programas concorrentes em Elixir. A definição de tais critérios pode incentivar e motivar programadores em Elixir a testar seus códigos e melhorar sua qualidade.

1.3.2 OBJETIVOS ESPECÍFICOS

Alguns objetivos específicos foram definidos para atingir o objetivo geral. O primeiro objetivo é conseguir um *Benchmark* de programas concorrentes em Elixir, para os estudos iniciais. A partir da definição do *benchmark*, o segundo objeto consiste na identificação das falhas que podem ocorrer nas funções concorrentes do Elixir. O terceiro objetivo específico consiste na definição dos operadores de mutação para a linguagem Elixir, considerando a taxonomia de falhas identificada anteriormente.

1.4 ESTRUTURA DA MONOGRAFIA

O seguinte trabalho está dividido em cinco capítulos. No Capítulo 1, é feita a apresentação do tema, relatando o atual contexto da área e as motivações que levam o autor a desenvolver a ideia. No Capítulo 2, são apresentadas as principais definições e conceitos dentro da área abrangida pelo trabalho. No Capítulo 3 são apresentadas as contribuições deste trabalho. No Capítulo 4 é apresentado o experimento realizado sobre os operadores de mutação definidos. Por fim, o Capítulo 5 realiza as considerações finais à respeito do trabalho.

2 ASPECTOS CONCEITUAIS

Engenharia de Software, disciplina da engenharia, compreende a reunião de todos os aspectos da produção de software, abrangendo desde os estágios iniciais da especificação do sistema até sua manutenção, quando este já está em uso. Seu principal objetivo é apoiar o desenvolvimento profissional de software, baseando-se em técnicas que apoiam especificação, projeto e evolução de programas (SOMMERVILLE, 2016).

A área de engenharia de software vem ganhando cada vez mais destaque, principalmente pela sua importância em capacitar indivíduos a desenvolver sistemas complexos dentro do prazo e com alta qualidade. A base para isso consiste em processos, ou seja, a metodologia que deverá ser seguida para obter sucesso nas entregas de tecnologia de engenharia de software (PRESSMAN, 2011).

O processo de Software é uma sequência de atividades levada em consideração durante a produção de um software. Existem quatro atividades fundamentais comuns a todos os processos de software: (1) A especificação de software compreende as definições do software a ser produzido, levantando requisitos e restrições; (2) O desenvolvimento engloba todo o processo de programação do software; (3) Na validação de software, são feitas verificações para garantir que este satisfaça os desejos do cliente. Por fim, (4) a evolução de software reúne as modificações e mudanças realizadas para cumprir requisitos do cliente e do mercado (SOMMERVILLE, 2016). O foco deste trabalho está na atividades de validação e verificação de software.

2.1 TESTE DE SOFTWARE

Embora produzam resultados impressionantes, sistemas robustos podem trazer enormes problemas para os desenvolvedores (PRESSMAN, 2011). Problemas referentes a sincronização, manipulação e processamento de dados são apenas alguns dos problemas passíveis de acontecer sem o devido tratamento. Para controlar esses problemas, verificando que os softwares estejam corretos e diminuindo riscos, existe o teste, que pode ser definido

como um conjunto de tarefas que podem ser planejadas com antecedência e executadas sistematicamente. Teste muitas vezes exige mais trabalho do projeto do que qualquer outra ação da engenharia de software, já que procura assegurar que o programa faça o que foi projetado (PRESSMAN, 2011). O software deve ser previsível e consistente, sem surpresas para os usuários (MYERS et al., 2004).

Para falar de teste de software, é necessário ter discernimento sobre Verificação e Validação, atividades diferentes mas que, executadas juntas, podem assegurar qualidade. Verificação compreende ao conjunto de tarefas que garantem que o software execute corretamente suas funções e atividades. O processo de verificação envolve a análise do produto desenvolvido, certificando-se de que este atende aos requisitos funcionais e não funcionais. Através da verificação, procura-se responder a pergunta: "Estamos criando o produto corretamente?". A validação reúne o conjunto de tarefas que confirmam que o software foi desenvolvido segundo os requisitos do cliente e portanto cumpra com seu propósito. Com a validação, busca-se responder a pergunta: "Estamos criando o produto certo?"(PRESSMAN, 2011).

Outros conceitos que serão bastante abordados e devem ser esclarecidos são *Engano, Defeito, Erro e Falha*. Engano refere-se a ação humana que produz um defeito. Defeito por sua vez, pode ser considerado uma definição incorreta de dados, geralmente gerada por engano humano. A existência de um defeito pode gerar um erro durante a execução do programa, ou seja, um estado inconsistente ou ação inesperada do programa. Este erro gerado pode ocasionar uma falha que é definido como resultado diferente do resultado esperado (DELAMARO; JINO; MALDONADO, 2017). Na Figura 1 são apresentadas as diferenças entre os conceitos de defeito, erro e falha (NETO, 2007).

Figura 1: Defeito x Erro x Falha

Fonte: Adaptada de NETO (2007).

Existem diferentes maneiras de testar um software e assegurar-se de que este tenha qualidade. Idealmente, um programa P, o qual deseja-se assegurar a qualidade, deveria ser testado com todas as possíveis entradas e sendo avaliadas todas as possíveis saídas. Porém, isso se torna impraticável devido ao número de entradas (ou combinação de entradas) que um programa pode ter. Desta forma, aplica-se o teste de subdomínios, no qual algumas entradas do domínio de entrada são escolhidas e testadas. Criar subdomínios é agrupar elementos de entrada que possuem características semelhantes e escolher uma para utilizar como entrada de teste. Para saber se valores diferentes estão no mesmos subdomínio, utilizam-se regras. Essas regras são chamadas de critérios de teste. Para selecionar os critérios, técnicas de teste são utilizadas. Existem três técnicas de teste principais: a funcional, estrutural e a baseada em defeitos. Cada técnica tem suas características e diferenças e as características que definem cada uma são os requisitos de teste (DELAMARO, 2012). Nas subseções a seguir serão abordadas cada uma das técnicas.

2.1.1 TESTE FUNCIONAL

Teste funcional ou também conhecido como teste de caixa preta é uma técnica de teste em que o objetivo principal é verificar se a aplicação realiza o objetivo para que foi desenvolvida, ou seja, os requisitos são o foco do teste funcional. Ele é popularmente denominado teste de caixa preta, pois durante sua execução, não são considerados detalhes de implementação. Portanto, é de suma importância que os requisitos estejam bem especificados, afinal, eles serão a base para qualificação. Especificações falhas ou mesmo incompletas tornam difícil a aplicação de teste funcional.

O teste funcional é considerado eficiente na detecção de defeitos por sua capacidade de aplicação em todas as fases de teste e a variedade de linguagens e paradigmas de programação aptas a aplicação dos testes, já que detalhes de implementação não são considerados. Porém, esta se torna sua principal desvantagem pois, para entradas de dados muito grandes ou infinitas, o tempo de atividade do teste se torna inviável, deixando o teste impraticável.

Os principais critérios conhecidos da técnica de teste funcional são o particionamento em classes de equivalência, a análise do valor limite, o teste funcional sistemático, o grafo causa-efeito e a técnica combinatorial. No particionamento em classes de equivalência, busca-se reduzir a quantidade de dados de entrada afim de torná-la mais viável para as atividades de teste. O processo consiste em dividir as possíveis entradas em conjuntos,

denominados classes de equivalência. As classes devem ter características únicas e, podem conter dados de entrada válidos e inválidos (DELAMARO; JINO; MALDONADO, 2017).

Com a divisão em classes, é possível assumir que determinado elemento de uma classe é considerado uma representação da classe, portanto presume-se que elementos da mesma classe produzam resultados iguais. O critério auxilia a reduzir o tamanho do domínio de entrada, porém, a técnica não provê meios de determinação dos dados de teste ou de combinação entre eles que pudesse abranger diferentes classes de equivalência com mais eficiência (ROPER, 1995).

A análise do valor limite complementa a técnica do particionamento, porém, os elementos de cada classe, que serão utilizados no teste, não são escolhidos ao acaso, mas selecionados os elementos limites de cada classe. A técnica baseia-se nesse princípio, pois a grande maioria dos erros ocorre nas extremidades das classes. O critério apresenta os mesmos benefícios e o mesmo problema que o particionamento. Somando as técnicas de particionamento de dados em classes com a análise do valor limite, tem-se o teste funcional sistemático. A técnica sugere testar elementos dentro e no limite de uma classe.

Embora minimizados, os problemas ainda persistem sobre a técnica. Outra grande desvantagem dos três critérios apresentados é a deficiência na exploração de combinações de entrada. Resolvendo este problema, existe o critério do grafo causa-efeito em que há combinação dos valores de entrada e análise do resultado obtido. Primeiramente todas as causas (dados de entrada) e efeitos (dados de saída) são identificados e mapeados em um grafo. Em seguida, transforma-se o grafo em uma tabela de decisão na qual os casos de teste são obtidos das regras da tabela. O principal problema deste critério é o crescimento exponencial da tabela de decisão ao acrescentar uma causa (dado de entrada) ao programa.

Ainda existe o critério combinatorial, que representa uma versão do grafo causa-efeito porém, nele buscam-se explorar as diversas possibilidades de combinações. Devido a essa característica, seu principal problema é a geração dos dados de entrada que se torna uma tarefa complexa e de natureza combinatorial. Após uma breve descrição sobre teste funcional e compreensão dos principais critérios, é possível tirar conclusões de suas vantagens e desvantagens. Necessitar apenas das especificações do produto para gerar os casos de teste é o principal benefício, tornando-o praticável em toda linguagem e paradigma de programação. Sua carência é justamente a dificuldade em assegurar que partes críticas e essenciais do código foram testadas.

2.1.2 TESTE ESTRUTURAL

Teste estrutural, popularmente conhecido como teste de caixa-branca, é o teste executado considerando-se a estrutura de controle do projeto procedimental, ou seja, sua implementação. Durante o teste, são analisadas as expressões lógicas, variáveis, pares de definições e condições.

Os critérios do teste estrutural são dividas em: baseados na complexidade, baseados no fluxo de controle e baseados no fluxo de dados (DELAMARO; JINO; MALDONADO, 2017). É importante ressaltar a importância do GFC (Grafo de Fluxo de Controle) que é utilizado pela maioria dos critérios. No Algoritmo **??** é apresentado um exemplo de função em Elixir que calcula o máximo divisor comum de dois números e na Figura 2 é possível visualizar o GFC referente ao algoritmo.

1 Função em Elixir

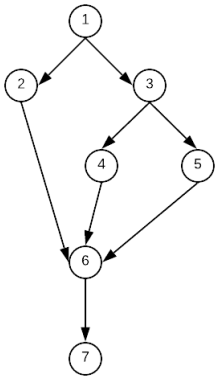
```
defmodule mdc(numero1, numero2){

        if numero1 == numero2 do                        # Nó 1
                resposta = numero1                      # Nó 2
        else                                            # Nó 3
                if numero1 < numero2 do                 # Nó 3
                        resposta = numero2 - numero1    # Nó 4
                else                                    # Nó 5
                        resposta = numero1 - numero2    # Nó 5
                end                                     # Nó 6
        end                                             # Nó 6
        IO.puts(resposta)                               # Nó 7
end
```

Beneficiando-se de informações referentes a complexidade do programa para derivar os requisitos de teste, tem-se a categoria das técnicas baseadas na complexidade. O critério de McCabe (1976) (ou teste de caminho básico) representa a categoria. Seu objetivo é determinar o número de caminhos independentes do programa, ou seja, os fluxos de dados existentes para testá-los.

O critério baseado no fluxo de controle determina quais estruturas são realmente necessárias, baseando-se apenas nas características de controle da execução do programa, procurando exercitar os elementos do GFC. Existem três critérios conhecidos desta classe. O critério de teste Todos-Nós defende a execução de pelo menos uma vez cada comando do programa, ou seja, que o programa passe por todos os vértices do GFC. Já o critério Todas-Arestas, defende a exploração de cada aresta do GFC ao menos uma vez, conduzindo

Figura 2: Exemplo de Grafo de Fluxo de Controle



Fonte: Autoria própria.

a execução de cada fluxo de controle. Por fim, a técnica Todos-Caminhos soma as duas técnicas anteriores e, portanto, todos os caminhos do programa são executados (DELAMARO; JINO; MALDONADO, 2017).

Finalizando a apresentação dos critérios de teste estrutural, o baseado em fluxo de dados analisa o fluxo de dados para derivar os requisitos de teste, ou seja, são analisados o comportamento das variáveis do programa. Existem duas famílias de critérios de fluxo de dados. Os critérios de Rapps e Weyuker (1982), Rapps e Weyuker (1985) propõem alguns conceitos e definições como o Grafo Def-Uso - que é uma versão aprimorada do GFC, contendo informações a respeito do fluxo de dados do programa que servirão de definição para os requisitos de teste. Os critérios Potenciais-usos de Maldonado et al. (1991), requisitam associações independentes entre uma referência em uso a uma definição de variável. Seu objetivo é realizar testes sobre as referências enquanto elas não são utilizadas para certificar-se de que seu valor não foi alterado.

Existem desvantagens que dificultam a automatização do processo de validação de software, como a inexistência de um teste de propósito geral e a dificuldade em utilizar o mesmo teste em programas diferentes. Ao optar por teste estrutural, fica-se sujeito a algumas deficiências como a existência de elementos não executáveis, caminhos ausentes ou funcionalidades não implementadas. Apesar disto, é indiscutível a eficiência do teste estrutural pois este, além de abranger todo o programa, pode servir como complementação para outras técnicas e tem resultados relevantes para atividades de manutenção, depuração e avaliação da confiabilidade de software (DELAMARO; JINO; MALDONADO, 2017).

2.1.3 TESTE BASEADO EM DEFEITOS

Teste baseado em defeitos, compreende a técnica de teste que considera os erros mais frequentes durante o desenvolvimento de software para geração dos casos de teste. Como visto anteriormente, defeitos são implementações incorretas realizadas no código fonte de um programa ou aplicação. Defeitos são os principais causadores de problemas em programas e muitas vezes podem passar despercebidos tanto no desenvolvimento quanto no teste.

O principal critério da técnica baseada em defeitos é o teste de mutação. Seu processo consiste em testar determinado programa, gerando uma vizinhança de programas mutantes do programa original. Não é possível assegurar a corretude de um programa somente através de software e portanto busca-se elevar o grau de coerência, gerando programas mutantes que produzam saídas diferentes das saídas do programa original, indicando que este está correto e não possui nenhum dos possíveis erros. Assim, o critério busca detectar erros reais através da geração de erros artificiais.

Existem duas hipóteses que servem de base para a definição e embasamento do critério de teste de mutação. A primeira diz respeito a ideia de que um programador competente produz um programa correto ou muito próximo ao ideal. Através deste, é possível criar mutantes com defeitos simples que representam situações plausíveis de engano por parte de programadores. Esta hipótese é conhecida como Programador Competente (DEMILLO; LIPTON; SAYWARD, 1978).

A segunda hipótese diz respeito ao efeito de acoplamento, ou seja, casos de teste qualificados a identificar erros simples são capazes de identificar erros mais complexos. Unindo as duas hipóteses, alcançá-se o critério de teste de mutação, em que para testar um determinado programa, geram-se cópias deste com erros simples passíveis de ocorrência e, a partir dos testes destes mutantes, é possível detectar erros complexos no programa base (DELAMARO; JINO; MALDONADO, 2017).

Há quatro passos para aplicação do teste de mutação. Tudo começa com a geração dos mutantes, seguido pela execução do programa em teste. Após essas duas atividades, realiza-se a execução dos mutantes e, por fim, estes são analisados. Na geração dos mutantes, constrói-se a vizinhança de um programa, isto é, o conjunto de programas modificados e propositalmente incorretos. O conjunto de mutantes definirá o sucesso da aplicação do critério de teste e, portanto, deverá ser abrangente e com baixa cardinalidade, para que seja possível verificar a adequação de um conjunto de casos de teste.

Na geração de mutantes, busca-se introduzir defeitos através de desvios sintáticos que produzam consequências semânticas. Este processo força o programador a criar casos de teste capazes de detectar os defeitos introduzidos nos mutantes e a deficiência na detecção destes defeitos ou incapacidade de mostrar o defeito no mutante, torna altamente provável a existência de um erro (AGRAWAL et al., 1999).

Embora aplicados a diferentes linguagens e paradigmas, não há uma prática direta para definição dos operadores de mutação e, portanto, utiliza-se a experiência no uso para detecção dos enganos mais comuns para projeção dos operadores. Existem três principais grupos de operadores de mutação, os de SSDL (*statement deletion* ou eliminação de comandos), os de ORRN (*relational operator replacement* ou troca de operador relacional) e, os STRI (*trap on IF condition* ou armadilha em condição de IF). O Algoritmo 2 apresenta um exemplo de utilização de operador SSDL. Na primeira linha é declarada a variável *valor* valendo 3 e em seguida, é enviado para o processo atual este valor. Nas linhas 4, 5 e 6, é realizado o *receive* no processo atual, que quando recebe o valor, soma 1 na variável *valor*. Por fim na linha 8 é impresso o valor. Na parte direita da figura, é possível visualizar o operador SSDL onde as linhas 4, 5 e 6, utilizadas pela função *receive*, são eliminadas e ao executar o algoritmo, este não imprimirá o valor corretamente.

2 Exemplo de operador SSDL

```
    Original
1   valor = 3
2   send(self(), valor)
3
4   receive do
5     valor -> valor = valor + 1
6   end
7
8   IO.puts valor
```

```
     Mutante 1
1   valor = 3
2   send(self(), valor)
3
4   # Operador SSDL
5
6
7
8   IO.puts valor
```

O Algoritmo 3 apresenta um exemplo do operador ORRN, em que na segunda

linha o operador lógico == foi alterado pelo operador lógico >=. O Algoritmo 4 apresenta um exemplo do operador STRI, em que a condição do *IF* foi alterada para sua negação.

3 Exemplo de operador ORRN

```
    Original
1  defmodule Maior(numero1, numero2) do
2    if numero1 == numero2 do
3      resposta = "Numeros iguais"
4    else
5      if numero1 > numero2 do
6        resposta = numero1
7      else
8        resposta = numero2
9      end
10   end
11 end

    Mutante 1
1  defmodule Maior(numero1, numero2) do
2    if numero1 >= numero2 do
3      resposta = "Numeros Iguais"
4    else
5      if numero1 > numero2 do
6        resposta = numero1
7      else
8        resposta = numero2
9      end
10   end
11 end
```

Os operadores SSDL atuam removendo trechos ou linhas de código simulando possíveis esquecimentos de programadores. Ainda, estes servem para comprovar o efeito de cada comando na resposta do programa. Já os operados ORRN representam as mudanças de operadores relacionais simulando equívocos que podem passar despercebidos durante o desenvolvimento. Por fim, os operados STRI são responsáveis por adicionar armadilhas nas condições *IF* e, tem três principais operadores: (1) definição da condição como *true*, (2) definição da condição como *false* e, (3) negação da condição.

A segunda etapa da aplicação do teste de mutação é a execução do programa a ser testado em que se analisa seu comportamento e suas respostas. Após a execução do programa, inicia-se a execução dos mutantes. Durante o processo, cada programa mutante é comparado ao original. Programas mutantes que produzem resultados diferentes evidenciam que aquele determinado erro ou resultado incorreto não acontece no programa

4 Exemplo de operador STRI

```
      Original
1  if 1 + 1 == 2 do
2    IO.puts "Calculo certo"
3  else
4    IO.puts "Calculo errado"
5  end

      Mutante 1
1  if !(1 + 1 == 2) do
2    IO.puts "Calculo certo"
3  else
4    IO.puts "Calculo errado"
5  end
```

original e, portanto, o mutante é considerado morto.

Finalizando a etapa de aplicação, a análise dos mutantes vivos constitui as etapas de encerramento do teste e verificação dos mutantes que sobreviveram. Um dos principais dados analisados é o *score* de mutação - valor obtido da divisão do número de mutantes mortos pelo número total de mutantes. Quanto mais próximo de 1, melhor. O segundo ponto a ser analisado diz respeito ao exame dos mutantes sobreviventes e validação como equivalentes ao programa original ou não. Um mutante é dito como equivalente se para todas as entradas de teste, o mutante dará a mesma resposta que o programa original. Desta forma é impossível identificar uma entrada que possa matar esses mutantes.

O principal desafio da técnica é tempo de execução e custo computacional principalmente causado pela geração de mutantes, tarefa que pode tornar-se exaustiva e extensa visto o número de possibilidades proporcionados pelos operadores. Assim, torna-se evidente a necessidade de ferramentas que auxiliem desde a criação dos casos de teste, execução dos mutantes e análise dos resultados da execução.

2.2 PROGRAMAÇÃO CONCORRENTE

Programação concorrente refere-se ao conceito de construir um programa que contenha no mínimo dois processos (ou *threads*) que possam ser executados paralelamente e/ou concorrentemente, interagindo na solução de problemas (ANDREWS, 1991). Enquanto a programação sequencial convencional visa a utilização de primitivas que são executadas sequencialmente, como definições e uso de variáveis, desvios condicionais e incondicionais,

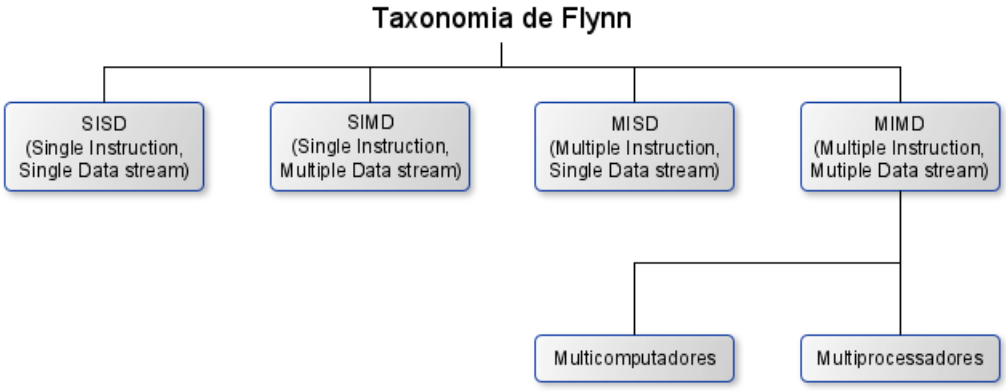
estruturas de repetição e chamadas de funções/sub-rotinas/métodos, a programação concorrente procura meios de executar o máximo de tarefas ao mesmo tempo, melhorando o desempenho da aplicação e aumentando a utilização dos recursos computacionais disponíveis (DELAMARO; JINO; MALDONADO, 2017).

As três etapas básicas para a aplicação da programação concorrente são a representação do conjunto de tarefas que serão executados concorrentemente, a geração e finalização destas tarefas no formato de processos ou *threads* (que serão comentados posteriormente), e a coordenação e gerência das interações entre os processos enquanto estes estiverem executando juntos (GOTTLIEB; ALMASI, 1989).

2.2.1 ARQUITETURAS PARALELAS

A arquitetura de um sistema representa o modo como o hardware está organizado, desde quantidade e comunicação entre os processadores até análise do acesso à memória. Para facilitar a compreensão das arquiteturas, diversas taxonomias foram criadas e dentre as mais famosas é lembrada a taxonomia de Flynn (FLYNN, 1972). Baseando-se nos conceitos de fluxo de dados e fluxo de instruções, a taxonomia identificou quatro modelos de processamento, SISD (*Single Instuction Stream, Single Data Stream*), SIMD (*Single Instruction Stream, Multiple Data Stream*), MISD (*Multiple Instruction Stream, Single Data Stream*) e MIMD (*Multiple Instruction Stream, Multiple Data Stream*). A Figura 3 apresenta a taxonomia de Flynn.

Figura 3: Taxonomia de Flynn



Fonte: Adaptada de Stallings e Midorikawa (2002).

A arquitetura SISD compreende a um único fluxo de instruções atuando sobre um único conjunto de dados, semelhando-se ao processamento sequencial da máquina de Von

Neumann. A arquitetura permite a execução paralela por meio de *pipeline* onde diferentes instruções executam diferentes estágios do *pipeline*. A arquitetura SIMD compreende em um único fluxo de instruções atuando sobre múltiplos conjunto de dados. Os principais exemplos da arquitetura são os processadores vetoriais e matriciais.

Na arquitetura MISD, múltiplos fluxos de instruções atuam sobre um único conjunto de dados e devido a esta característica existem poucas aplicações comerciais. Recentemente, algumas iniciativas estão explorando o potencial dessa arquitetura (TRACY et al., 2015). Por fim, a arquitetura MIMD, presente na maioria dos sistemas paralelos atuais, compreende a múltiplos fluxos de instruções atuando sobre múltiplos conjuntos de dados, permitindo que diferentes processadores executem diferentes códigos em diferentes conjuntos de dados. Por ser a classe mais abrangente, foi necessário classificá-las de acordo com a organização da memória, separando-as em multiprocessadores e multicomputadores.

Multiprocessadores compreende a união de vários processadores que possuem acesso a memória compartilhada em um sistema paralelo. Os processadores ficam fisicamente muito próximos e estão conectados por meio de uma rede interconectada. Geralmente, a comunicação entre os processadores é realizada através de leitura e escrita na memória compartilhada. Os multicomputadores compreendem a múltiplos processadores em um sistema paralelo, porém sem acesso a memória compartilhada. Diferente da classificação anterior, nesta os processadores ficam fisicamente distantes e a comunicação entre eles é realizada por passagem de mensagens ou por meio de um espaço de endereçamento comum. No contexto deste trabalho, serão estudados programas concorrentes para serem executados em máquinas MIMD de memória distribuída (multicomputadores).

O objetivo da programação concorrente é desenvolver aplicações concorrentes em que diferentes processos atuem concorrentemente para resolução de um problema (SILVA, 2013). Processos são programas em execução e são divididos em duas categorias. Os processos que começaram sua execução e em determinado instante de tempo ainda não finalizaram, são denominados processos concorrentes (GOTTLIEB; ALMASI, 1989). Processos concorrentes competem por memória, processadores e dispositivos de Entrada e Saída. Por outro lado, existem os processos paralelos, que representam uma forma de processos concorrentes pois há garantia que eles estejam executando em diferentes processadores no mesmo intervalo de tempo (DELAMARO; JINO; MALDONADO, 2017). Um processo pode ter uma ou mais linhas de execução que são denominadas *threads*.

Como mencionado anteriormente, processos podem comunicar-se de maneira diferente dependendo da arquitetura utilizada. A seguir, são descritos os principais

paradigmas de programação relacionados a comunicação e sincronização entre processos.

2.2.2 PARADIGMAS DE PROGRAMAÇÃO

Paradigma de programação representa o modo como acontece a comunicação e sincronização entre os processos. Os dois paradigmas de programação concorrente são a memória compartilhada e a passagem de mensagem.

Memória compartilhada representa o paradigma de programação em que existe um espaço de endereçamento compartilhado, ou seja, diferentes processos conseguem ter acesso as mesmas variáveis. Como é possível perceber, será necessário ter um mecanismo de segurança de dados. Isso é necessário já que diferentes processos podem acessar a mesma região crítica, como por exemplo uma variável que pode ser lida e modificada. Dentre as maneiras de controlar o acesso a região crítica, citam-se os semáforos e as barreiras (STEEN; TANENBAUM, 2009; GRAMA et al., 2003).

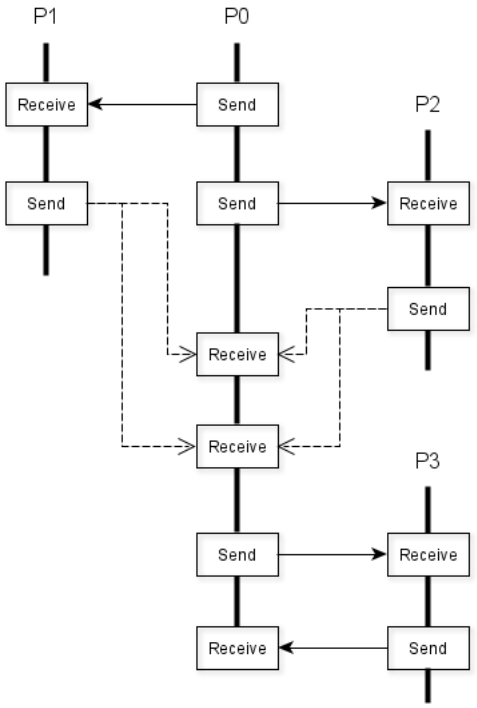
Na sincronização entre processos realizada com semáforos, define-se uma variável global visível para os processos que será controlada pelo sistema operacional e manipulada através de operações conhecidas como *down* e *up*. Antes de acessar uma região crítica, o processo verifica por meio do *down* se o valor da variável é maior que zero. Se for maior que zero, o valor é decrementado e o processo executa sua ação e caso o valor for zero, indica que outro processo está acessando a região crítica, e portanto o processo é colocado para dormir. A operação *up* é utilizada pelo processador para despertar um processo para que este possa realizar suas operações na região crítica (TANENBAUM; FILHO, 1995).

Na sincronização entre processos realizada com barreira, aplica-se uma trava a determinados processos até que todos os processos tenham chegado até barreira. Uma barreira pode ser implementada de diferentes formas como por exemplo um contador ou uma variável de condição (GRAMA et al., 2003).

Por sua vez, o paradigma de programação baseado em passagem de mensagem é principalmente utilizado para comunicação entre processos que não utilizam um mesmo endereçamento compartilhado. Existem dois elementos base utilizados para a passagem de mensagem, *send* e *receive*, que embora possam variar sintaticamente, dependendo da linguagem, tem o mesma semântica. *Send* compreende ao envio de uma mensagem de um processo a outro, enquanto *receive* é utilizado pelo processo destino para receber a mensagem (SILVA, 2013). A Figura 4 apresenta um exemplo de como acontece a comunicação entre processos por passagem de mensagens. No exemplo são exibidos

quatro processos que se comunicam utilizando as primitivas *send* e *receive* para realizarem determinada tarefa. As setas contínuas representam o fluxo de mensagens entre os processos e as setas tracejadas representam um possível fluxo de mensagens e não se pode ter certeza de qual fluxo irá ocorrer durante a execução do sistema. Nesse exemplo, os *receives* são chamados de não determinísticos, pois não se tem a garantia de qual sincronização irá ocorrer.

Figura 4: Exemplo de como acontece a comunicação por passagem de mensagens.



Fonte: Adaptada de Silva (2013).

Características como a universalidade, expressividade, a fácil depuração e o desempenho são algumas vantagens de utilizar passagem de mensagens. A universidade compreende a ideia de que a passagem de mensagens pode ser realizada por uma comunicação de rede e, portanto, compatível com a grande maioria dos processadores atuais. A expressividade refere-se a fácil utilização e controle dos dados mesmo em sistemas complexos. Fácil depuração como o nome já diz, facilita a detecção de erros e anomalias no código. Por fim, a utilização da passagem de mensagem traz um ganho de desempenho em relação ao uso da memória compartilhada já que a explícita associação entre processos e dados permite que o compilador e gerenciador de cache funcionem perfeitamente (GROPP et al., 1999).

Dependendo da linguagem utilizada, há variações no processo de comunicação entre

os processos. Em uma comunicação síncrona (bloqueante), há bloqueio e consequente espera por parte dos dos processos que estão trocando mensagens. Um *send* bloqueante aguarda até que o receptor execute um *receive*, da mesma forma que, um *receive* bloqueante bloqueia o receptor (processo atual) até que o emissor envie uma mensagem. Na comunicação assíncrona (não-bloqueante), ambos os métodos de troca de mensagens (*send* e *receive*) não ficam bloqueados e não bloqueiam nenhuma tarefa. A utilização da comunicação assíncrona pode levar a problemas de perda de mensagens ou dados incorretos já que após o envio de uma mensagem (*send*), está pode sofrer inúmeras alterações antes que o *receive* a receba. Para solucionar este problema há a opção de se utilizar um *buffer* que servirá para armazenar mensagens, e protegê-las de modificações indevidas, até que está seja recebida.

2.3 ELIXIR

A linguagem de programação Elixir foi criada pelo brasileiro José Valim em 2012. Valim que trabalhava com a linguagem *Ruby*, encontrava dificuldades com performance em seus projetos e percebeu que o *Ruby* não era apto a lidar com questões de simultaneidade e concorrência. Após análise e pesquisas, Valim viu que o *Erlang* apresenta suporte à construção de aplicações distribuídas, porém, notou que a linguagem tinha uma sintaxe complexa que provocava uma baixa curva de aprendizado. Neste contexto, ele desenvolveu o Elixir, que através de uma sintaxe mais agradável, auxilia o programador a resolver os problemas de concorrência, distribuição e tolerância a falhas semelhante ao Erlang (DAVI, 2017).

Base de inspiração para Valim, o Erlang foi criado na década de 80 por Joe Armstrong, Mike Williams e Robert Virding, desenvolvedores da Ericsson, empresa multinacional de telecomunicação. Inicialmente, eles buscavam algo já existente para desenvolver sua próxima geração de sistemas de Telecom, porém, percebendo a carência de uma linguagem adequada as suas necessidades, acabaram criando o Erlang - uma linguagem de alto nível habilitada a resolver problemas de concorrência, distribuição, tolerância a falhas e alta disponibilidade com código de *hot swap* (ARMSTRONG, 2013).

Dentre os principais paradigmas de programação, o Elixir utiliza o paradigma funcional, ou seja, durante o desenvolvimento busca-se a criação e utilização de funções menores que, reunidas, são capazes de resolver problemas complexos. O paradigma funcional originou-se a partir do cálculo Lambda, proposto por Alonzo Church na década de 30, que compreende um conjunto de técnicas para definição e aplicação de recursão em

funções (HUDAK, 1989).

Nas linguagens imperativas, tem-se a ideia de uma única estrutura com instruções sequenciais para manipulação de valores em variáveis. Por sua vez, as linguagens funcionais utilizam conceitos de imutabilidade e funções como base para seu desenvolvimento. O paradigma de programação OO (Orientado à Objetos), utiliza-se de classes e objetos para modelar o problema de uma forma mais próxima da realidade e, pode ser utilizado juntamente com o paradigma funcional.

A linguagem Elixir foi escolhida para ser utilizada durante este trabalho pelas oportunidades oferecidas, como o suporte a programação concorrente e paralela, e a aplicação do paradigma funcional. O Elixir soma os conceitos de seu paradigma funcional com as heranças obtidas do Erlang, e assim possibilita a resolução de problemas de concorrência e distribuição de forma mais simples. Além disso, o crescimento da comunidade e o ganho de mercado, motivam e favorecem a aplicação do Elixir.

Antes de iniciar a apresentação das funções utilizadas no desenvolvimento de programas concorrentes, torna-se necessário definir alguns termos da programação em Elixir. A linguagem tem cinco tipos básicos de dados: (1) inteiros, (2) pontos flutuantes, (3) booleanos, (4) átomos e (5) strings. Os tipos inteiros e pontos flutuantes são utilizados para números, sendo a diferença entre eles, a presença ou não de casas decimais. O tipo booleano serve para representar dados lógicos através de *true*, *false* e *nil*, sendo que tudo é verdadeiro com exceção de *false* e *nil*. Átomos são constantes cujo nome representa seu valor e *strings* são utilizadas para a representação de palavras.

Elixir contém três grupos de operadores: matemáticos, booleanos e de comparação. Os operadores matemáticos são utilizados para realização de operações matemáticas como a soma (+), subtração (-), multiplicação (*) e divisão (/). Para operações lógicas, a linguagem fornece os três operadores lógicos básicos: || representando *OR*, && representando *AND* e ! representando a negação. Por fim, ainda existem os operadores relacionais: maior (>), menor (<), maior ou igual (>=), menor ou igual (<=), igual (==), propriamente igual (===), diferente (!=) e propriamente diferente (!==).

2.3.1 FUNÇÕES CONCORRENTES

No Elixir, a comunicação entre processos é feita por troca de mensagens em vez de variáveis compartilhadas, o que permite que a linguagem não tenha de gerenciar o estado dessas variáveis (ALMEIDA, 2018). O Elixir possui funções para resolver problemas que

envolvem paralelismo e concorrência presentes em seis módulos: *Kernel*, *Task*, *Process*, *Agent*, *Genserver* e *Task.Supervisor*. Este trabalho considera apenas as funções do *Kernel* e do módulo *Task* (Tabela 1) por serem os mais utilizados e difundidos pela comunidade de desenvolvedores. Os números após as funções, como *spawn*/1 e *self*/0, indicam a aridade, ou seja, o número de argumentos que as funções podem receber.

Tabela 1: Funções de Programação Concorrente em Elixir em Kernel e Task

Função	Parâmetro	Descrição
spawn/1 spawn/3	função módulo, função, argumentos	Criação de processos
spawn_link/1 spawn_link/3	função módulo, função, argumentos	Criação de processos com uma ligação bidirecional
spawn_monitor/1 spawn_monitor/3	função módulo, função, argumentos	Criação de processos com uma ligação unidirecional
send/2 receive/1	PID destino, mensagem mensagem	Envio de mensagem Recebimento de mensagem
self/0		Identificador
Task.async/1 Task.async/3	função módulo, função, argumentos	Inicia uma tarefa que deve ser aguardada
Task.async_stream/3 Task.async_stream/5	enumerable, função, opções enumerable, módulo, função, argumentos, opções	Executa a função concorrentemente em cada valor do *enumerable*
Task.await/2	tarefa, timeout	Aguarda uma resposta da tarefa e a retorna
Task.child_spec/1	argumento	Retorna uma especificação para iniciar uma tarefa sob um supervisor
Task.shutdown/2	tarefa, shutdown	Desvincula e desliga a tarefa e, em seguida, procura uma resposta
Task.start/1 Task.start/3	função módulo, função, argumentos	Inicia uma tarefa
Task.start_link/1 Task.start_link/3	função módulo, função, argumentos	Inicia um processo vinculado ao processo atual
Task.yield/2 Task.yield_many/2	tarefa, timeout tarefas, timeout	Bloqueia temporariamente o processo atual, aguardando a resposta da tarefa

Fonte: Autoria própria.

2.3.1.1 SPAWN E SELF

A função *spawn* é utilizada para a criação de processos. Ela pode ser utilizada com um ou três parâmetros. Em sua forma mais simples, uma função é esperada como parâmetro, e o retorno, representará o PID do processo criado. Para a utilização mais avançada, três parâmetros são necessários: (1) o módulo que a função executada está definida, (2) a função e (3) os argumentos da função.

A função *self* serve para apresentar o PID do processo atual. No Algoritmo 5 é criada uma função *soma* e em seguida são apresentados exemplos de chamadas das funções *spawn* e *self*.

5 Exemplo das funções *spawn* e *self*

```
soma = fn (a, b) -> a + b end
spawn(soma)

spawn(Modulo01, soma, [1, 2])

pid_atual = self()
```

2.3.1.2 SPAWN_LINK

Spawn_link cria um processo e automaticamente cria uma associação entre o processo criado e o processo da chamada, assim o processo criado será notificado quando o processo da chamada falhar. Essa função é utilizada para controle de erros, pois vinculando dois processos, quando um dos dois falhar, o outro também será interrompido e não ficará ocioso. Para explicação da função *spawn_link* foram definidos dois algoritmos. Após a execução do Algoritmo 6, será obtido como saída as palavras *antes* e *depois*, pois, em seu código é criada uma conexão com um processo sem erros. No Algoritmo 7 somente a palavra *antes* será impressa, pois, é criada uma associação entre o processo atual (responsável por imprimir as palavras) e um processo que contém um erro de lógica, 1 == 2, que será finalizado e finalizará o processo atual também devido ao *spawn_link*.

6 Exemplo da função *spawn_link*

```
IO.puts "antes"
spawn_link (fn() -> :ok end)
Process.sleep 100
IO.puts "depois"
```

7 Exemplo da função *spawn_link* com falha

```
IO.puts "antes"
spawn_link (fn() -> 1 == 2 end)
Process.sleep 100
IO.puts "depois"
```

2.3.1.3 SPAWN_MONITOR

A função *spawn_monitor* é utilizada para monitorar um processo sem a necessidade de vinculá-lo a outro. O processo que solicitou monitoramento será informado quando o

processo monitorado encerrou. No Algoritmo 8, o processo atual irá ser notificado que o processo a ele associado falhou.

8 Exemplo da função *spawn_ monitor*

```
spawn_monitor (fn() -> 1 = 2 end)
```

2.3.1.4 SEND E RECEIVE

As funções *send* e *receive* são utilizadas para troca de mensagens entre processos. Para enviar uma mensagem à um processo, é utilizada a função *send* passando como parâmetros, o PID do processo destino e a mensagem que se deseja enviar. As mensagens enviadas caem na caixa de correio do processo e não são perdidas, ou seja, é possível enviar uma segunda mensagem para um processo mesmo que este ainda não tenha recebido a primeira. A função *send* é não bloqueante, pois não espera um retorno do processo destino.

Para receber a mensagem enviada pelo *send*, é utilizada a função *receive*. A função *receive* irá ler as mensagens da caixa de correio na mesma ordem em que foram enviadas. Além disso, a função *receive* interrompe o processo atual até que receba uma mensagem ou até atingir determinado tempo limite. A função *receive* pode operar de forma determinística, isto é, levando em consideração a ordem do envio de mensagens. Para este, é necessário utilizar o *match* _. O Algoritmo 9 apresenta um exemplo da utilização da função *send* para enviar um átomo ao processo atual, e da função *receive* para receber a mensagem e imprimir uma frase na tela.

9 Exemplo das funções *send* e *receive*

```
send(self(), :Ola_Mundo)

receive do
    :Ola_Mundo -> IO.Puts("Processo recebeu mensagem Ola Mundo")
end
```

2.3.1.5 TASKS

As *Tasks* ou Tarefas, são processos destinados a realizar uma mesma ação específica durante toda a sua vida. Tarefas são utilizadas principalmente para conversão de código sequencial em código concorrente de forma assíncrona, ou seja, que retornará a resposta quando estiver concluída e não interromperá nenhum processo ou tarefa. Por meio de

Tarefas, é possível executar diversas tarefas não dependentes em paralelo. O Algoritmo 10 apresenta um exemplo do uso das funções *Task.async* e *Task.await*, no qual, um processo é criado, vinculado e monitorado pelo chamador (*tarefa*), e será executado em segundo plano. Quando o processo finalizar, uma mensagem será enviada ao chamador. Para receber a mensagem, é utilizada a função *Task.wait*.

10 Exemplo das funções *Task.async* e *Task.await*

```
tarefa = Task.async (fn -> fazer_algo() end)

resposta = Task.await(tarefa)
```

Sempre que utilizar *Task.async* espera-se receber uma resposta, e por isso, será necessário chamar *Task.await* em algum momento. Quando a intenção é somente executar uma tarefa, sem importância a sua resposta, utiliza-se *Task.start* e/ou *Task.start_link* passando a tarefa a ser executada como parâmetro. *Task.start_link* cria uma tarefa e automaticamente cria uma associação entre a mesma e o processo da chamada, assim o processo criado será notificado quando a tarefa falhar. O Algoritmo 11 apresenta um exemplo das funções *Task.start* e *Task.start_link*, que tem a finalidade de imprimir uma mensagem para o usuário.

11 Exemplo das funções *Task.start* e *Task.start_link*

```
tarefa = Task.start(fn -> IO.puts "Ola mundo" end)

tarefa_vinculada = Task.start_link(fn -> IO.puts "Ola mundo" end)
```

Além da função *Task.await*, o módulo *Task* ainda conta com mais duas funções para captura de respostas de tarefas. *Task.yield* trabalha de forma semelhante a *Task.await*, diferindo-se apenas por ser não-bloqueante. *Task.yield_many* serve para obter respostas de uma lista de tarefas.

O módulo *Task* ainda conta com as funções *Task.async_stream*, *Task.shutdown* e *Task.child_spec*. *Task.async_stream* serve para criar um conjunto de tarefas assíncronas a partir de uma única chamada. *Task.shutdown* é utilizada para finalização de tarefas, enquanto, *Task.child_spec* é utilizada para monitoramento e supervisão de tarefas.

2.4 TESTE PARA PROGRAMAÇÃO CONCORRENTE

Após conhecer as principais características de programas concorrentes e os principais tipos de teste de software, é possível explorar a utilização de testes em programas concorrentes. Um grande fator que influencia os testes para programas concorrentes é sua natureza não determinística. Programas sequencias tem característica determinística, isto é, independente do número de execuções de um mesmo programa, o mesmo valor de entrada produzirá a mesma saída. Programas concorrentes tem característica não determinística, possibilitando que a execução de um mesmo programa concorrente com o mesmo valor de entrada, produza diferentes resultados. Assim, quando relacionados a programas concorrentes, o teste visa a identificação de erros relacionados a comunicação, ao paralelismo e a sincronização, e portanto, os casos de teste são gerados a partir dessas categorias (SILVA, 2013). Apesar disso, os demais tipos de erro também devem ser considerados e tratados, afinal estes além de sustentar o programa incorreto, ainda podem provocar erros de comunicação, paralelismo e sincronização comentados anteriormente. Na Tabela 2 é possível visualizar os principais tipos de erros que podem ser cometidos no contexto de programação concorrente.

Embora existam diferentes critérios de teste para programação concorrente, como os critérios da técnica estrutural, o foco deste trabalho é trabalhar teste de mutação para programas concorrentes.

Como dito anteriormente, programas concorrentes tem comportamento não determinístico e isto se torna o principal problema durante a análise do comportamento dos mutantes. Em programas sequenciais, se o comportamento de um mutante for diferente do comportamento de seu original, este mutante é considerado morto. Porém, em programas concorrentes não é possível agir de tal forma e o ideal seria comparar o conjunto de resultados possíveis de um mutante com o conjunto de resultados possíveis do programa original, processo que em geral não é possível devido ao alto número de entradas mesmo para programas considerados simples.

Bradbury, Cordy e Dingel (2006) propuseram operadores de mutação para programas desenvolvidos em Java. Neste trabalho, foi importante notar a elaboração de grupos de operadores de mutação para Java relacionados à concorrência. Os grupos continham operadores relacionados aos métodos concorrentes, um grupo modificava os parâmetros, outro modificava as chamadas dos métodos e o outro modificava as palavras-chave dos métodos. Havia ainda um último grupo que modificava regiões críticas do código.

Tabela 2: Principais Tipos de Erros em Programas Concorrentes

Tipo de Erro	Descrição
Deadlock	Acontece quando duas ou mais threads bloqueiam uma à outra.
Livelock	Ocorre quando uma thread não consegue terminar a execução ou entrar em uma região crítica por excesso de trabalho ou falta de velocidade.
Starvation	Ocorre quando uma thread nunca é escalonada para ser executada.
Contagem incorreta de threads	Inicialização incorreta do número de threads requeridas para completar uma ação ou uma inicialização incorreta do número de permissões no semáforo.
Violação de Atomicidade	Causado pela execução concorrente de várias threads violando a atomicidade de uma região do código.
Código Desprotegido	Ocorre quando uma região crítica fica desprotegida, permitindo que várias threads alterem o mesmo dado.
Inconsistência de Dados	Ocorre quando diferentes threads possuem visões diferentes de uma mesma variável.
Erro de Sincronização	Ocorre quando uma sincronização não esperada entre threads ou processos ocorre.
Erro de Decomposição	Relacionado com a decomposição dos dados e funções para geração dos processos concorrentes.
Erro em Passagem de Mensagem	Uso de uma função errada gerando um comportamento não esperado (por exemplo, uso do receive bloqueante em vez de receive não bloqueante).
Mensagens Perdidas	Ocorre quando uma mensagem enviada para um processo nunca é recebida.
Erro de Comunicação	Ocorre quando uma mensagem errada é enviada ou quando o envio é feito para um processo errado

Fonte: Delamaro, Jino e Maldonado (2017).

Carver (1993) propôs o procedimento DEMT (*Deterministic Execution Mutation Testing*). Neste procedimento, inicialmente o programa em teste é executado de forma não determinística a fim de gerar sequências de sincronização aleatórias. Em seguida, os mutantes são gerados e executados de forma determinística, a partir de um conjunto de dados relacionado a uma sequência de sincronização. Dessa forma, o mutante é considerado morto caso não consiga executar a sequência de sincronização.

No trabalho de Silva, Souza e Souza (2012) foram definidos operados de mutação

para o teste de programas concorrentes em MPI (*Messagem Passing Interface*). Os operadores definidos exercitam as funcionalidades de comunicação e sincronização de processos concorrentes e foram divididos em três grupos. No primeiro grupo, os autores definiram os mutantes aplicados em funções coletivas, no segundo grupo, os aplicados a funções ponto a ponto e no terceiro grupo, os que podem ser aplicados a todas as funções MPI. Para execução dos mutantes, foram definidos dois procedimentos: o primeiro baseado na técnica DEMT, onde os operadores eram executados de forma determinística a partir de uma entrada de teste e comparava-se o resultado com a sequência de sincronização relacionada. Já para o segundo critério, não eram consideradas bases de sincronização, o resultado da execução do mutante era comparado com o conjunto de resultados possíveis do programa original. A partir dos dois procedimentos, os autores puderam classificar os mutantes vivos e mortos.

No trabalho de Giacometti, Souza e Souza (2003) foram definidos operadores de mutação para a validação de aplicações paralelas em PVM (*Parallel Virtual Machine*). Os operadores definidos exploram características relacionadas ao paralelismo e a comunicação, e procuram modelar erros no fluxo de dados entre as tarefas, erros no empacotamento de mensagens e erros no paralelismo e sincronização de tarefas. Neste trabalho foram definidos quinze operadores de mutação.

2.4.1 TESTE EM ELIXIR

Assim como em todas as linguagens, o Elixir também necessita de funções que auxiliem a verificar e validar a corretude e qualidade do software desenvolvido. A linguagem conta com uma ferramenta para apoio ao teste de unidade, o ExUnit (VALIM, 2019). Teste de unidade compreende testar a menor parte testável de um programa, como por exemplo, uma função em linguagens funcionais. O ExUnit fornece várias estruturas para a realização de testes unitários em Elixir, principalmente utilizando a técnica TDD (Test Driven-Development) que promove a sequência de visualizar o erro, corrigir e atualizar o código e, por fim, rodar o teste e ver que o erro foi corrigido. No Algoritmo 12 é apresentado um exemplo de teste criado através da ferramenta ExUnit.

O foco deste trabalho é a execução de teste de mutação em Elixir. Por esse motivo, foi realizada um estudo para verificar possíveis aplicações do teste de mutação para programas em Elixir. Como resultado, foi encontrada uma ferramenta para apoio ao teste de mutação em Elixir (POLO, 2017) que gera dois tipos de operadores: ORRN e STRI.

12 Exemplo de teste criado na ferramenta ExUnit

```
defmodule AppTest do
  use ExUnit.Case
  doctest App

  test "teste simples" do
    assert 1 + 1 == 2
  end
end
```

2.4.1.1 OPERADOR ORRN

A ferramenta desenvolvida por Polo, utiliza-se de operadores ORRN para gerar os mutantes, de forma que um operador $<$ (menor) é alterado para $>$ (maior), $>=$ (maior ou igual), $<=$ (menor ou igual), $==$ (igual) e $!=$ (diferente), produzindo assim cinco mutantes. O processo é realizado para todos os seis operadores relacionais de forma semelhante.

A segunda definição de operadores que a ferramenta realiza é sobre operadores lógicos *AND* e *OR*, que, por serem binários, necessitam de dois elementos para realizar o cálculo. Para o operador *AND* a ferramenta gera cinco alterações no código: (1) definindo o primeiro argumento da condição como *true*, (2) definindo o primeiro argumento da condição como *false*, (3) definindo o segundo elemento da condição como *true*, (4) definindo o segundo elemento da condição como *false* e, (5) alterando o *AND* para *OR*. O mesmo processo é realizado para o operador *OR*, com exceção do último caso em que altera-se o operador para *AND*.

2.4.1.2 OPERADOR STRI

Além disso, a ferramenta também define operadores mutantes STRI, para criar armadilhas nas condições *IF* do código fonte, onde um mutante é gerado definindo o argumento do *IF* como *true*, outro mutante é gerado definindo o argumento como *false* e por último, o argumento do *IF* é alterado para sua negação.

3 TESTE DE MUTAÇÃO PARA PROGRAMAS CONCORRENTES EM ELIXIR

Neste capítulo serão apresentados artefatos desenvolvidos para apoiar o teste de mutação para programas concorrentes em Elixir. Um *benchmark* de programas concorrentes em Elixir é apresentado na Seção 3.2, mostrando seu processo de desenvolvimento e validação. Na Seção 3.3, é apresenta a taxonomia de falhas em Elixir, desenvolvida através de um estudo e análise das funções concorrentes da linguagem. A partir desta taxonomia, foram definidos quinze operadores de mutação que são apresentados na Seção 3.4, contendo além de sua descrição, um exemplo de sua utilização.

3.1 INTRODUÇÃO

A proposta deste trabalho é definir operadores de mutação para programas concorrentes em Elixir. Para a validação dos operadores de mutação, mutantes foram gerados para um conjunto de programas em um *benchmark* preestabelecido. Tendo em vista o crescimento da linguagem Elixir e a diversidade de funções aplicadas à concorrência, operadores tornam-se uma ótima escolha para simular erros na utilização destas funções e auxiliar o teste de mutação a detectar erros e falhas nos programas. Na Figura 5, é apresentada a visão da aplicação do teste de mutação à programas concorrentes em Elixir. Essa imagem apresenta uma instanciação do processo do teste de mutação para programas concorrentes em Elixir e representa o contexto desse trabalho.

A Figura apresenta 5 atividades necessárias para a aplicação do teste de mutação para programas concorrentes em Elixir: (1) Definição do *benchmark*, (2) Geração dos mutantes, (3) Execução dos mutantes, (4) Análise dos mutantes equivalentes e (5) Geração de sequências de sincronização. Cada uma das atividades são descritas a seguir:

1. **Definição do *benchmark***: Inicialmente são necessárias três entradas para aplicação do teste: (1) o programa a ser testado, (2) os operados de mutação da linguagem, (3) e o conjunto de casos de teste do programa. O *benchmark* compreende ao conjunto

de programas que serão submetidos ao teste. Os programas embora diferentes, devem seguir uma lógica comum representando o contexto de aplicação que se deseja testar.

2. **Geração dos mutantes:** Na etapa de geração dos mutantes, será produzido um conjunto de mutantes do programa original que servirá como entrada para realização da próxima tarefa de execução dos mutantes. Com o *benchmark* de Elixir em mãos, é necessária a definição dos operadores de mutação para programas concorrentes. Não foi encontrado nenhum trabalho na área ou alguma taxonomia de erros na linguagem Elixir e nem a definição de operadores de mutação para funções concorrentes em Elixir. A contribuição deste trabalho encontra-se nessa atividade, na qual foram definidos operadores de mutação para programas concorrentes em Elixir.

3. **Execução dos mutantes:** Durante a execução do mutante são gerados três artefatos, o conjunto de mutantes vivos, o conjunto de mutantes mortos e o score de mutação. Esses resultados definirão quais ações deverão ser aplicadas, já que um *score* de mutação igual ou próximo a 1 determina o fim do teste. Caso o score de mutação não fique próximo ou igual a um, será necessário realizar pelo menos uma das duas ações subsequentes.

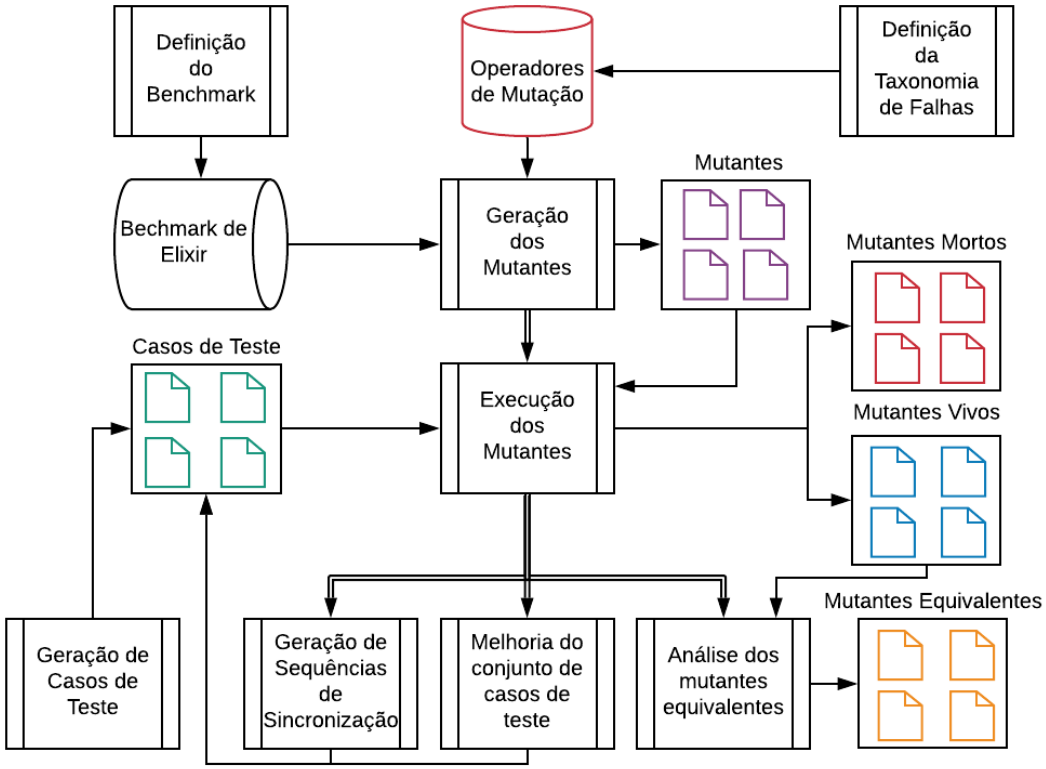
4. **Análise dos mutantes equivalentes:** Esta tarefa diz respeito a analisar e detectar mutantes equivalentes ao programa original para em seguida descartá-los do teste.

5. **Geração de sequências de sincronização:** A geração de novas sequências de sincronização é utilizada como forma de reduzir o impacto do não determinismo. Tanto o programa original como os programas mutantes podem produzir resultados diferentes a partir de uma mesma entrada de dados e por isso o ideal seria comparar o conjunto de resultados do programa mutante com o conjunto de resultados do programa original com uma mesma entrada de dados e, caso algum apresentasse comportamento diferente, este era distinguído do programa original (DELAMARO; JINO; MALDONADO, 2017). Esta abordagem, em geral, não é possível devido as possibilidades infinitas de sequências de sincronização e, em razão disso, alguns autores propuseram técnicas para que a execução do teste de mutante seja possível.

3.2 BENCHMARK DE PROGRAMAS CONCORRENTES

No contexto de testes de programas concorrentes, *benchmarks* de programas são usados para: (1) comparar abordagens distintas de maneira justa e uniforme, (2)

Figura 5: Geração de Teste de Mutação em Programas Concorrentes em Elixir



Fonte: Autoria Própria.

determinar se os modelos podem representar os programas a serem testados, (3) avaliar se os critérios podem revelar defeitos e guiar a seleção de dados de teste e (4) verificar se ferramentas de teste podem gerenciar adequadamente códigos-fonte variados (DOURADO et al., 2016).

Benchmarks de programas concorrentes vem sendo desenvolvidos para auxiliar na atividade de teste. Alguns dos principais são os apresentados em Eytani e Ur (2004), Rungta e Mercer (2009) e Dourado et al. (2016). Embora os *benchmarks* contribuam para o teste de programas concorrentes, os *benchmarks* citados consideram exclusivamente programas concorrentes na linguagem Java, considerando o paradigma orientado a objetos.

Como uma tentativa de melhorar os esforços de teste de software, esta seção apresenta um *benchmark* de programas para o suporte da atividade de teste de software para programas concorrentes em Elixir (BORDIGNON; SILVA, 2019).

3.2.1 BENCHMARK PROPOSTO

O *benchmark* proposto é composto por onze programas em Elixir (Tabela 3), classificados utilizando Linha de Código (LoC). Os programas 1 a 8 estão no grupo de pequeno porte e os programas 9 a 11 estão no grupo de médio porte. Os programas de pequeno porte são aconselháveis para testar modelos, critérios e ferramentas em estágios iniciais, no qual a verificação da viabilidade delas é avaliada.

Os programas de médio porte, possuem maior número de linhas de código (superior a 100) e visam validar ferramentas em estágios avançados de desenvolvimento, simulando soluções de problemas mais complexos.

Tabela 3: Programas concorrentes em Elixir

Nº	**Nome do Arquivo**	**Descrição do Problema**	**LoC**	**Referência**
1	Exemplo Spawn_Monitor	Maior valor e monitor de tarefas	15	Autoria Própria
2	Yield_many	Gerenciador de Tarefas	21	Ziegelmayer (2019)
3	Elixir Study	Tarefa com operação de soma	22	Amboni (2017)
4	Events	Vida de um processo	26	Code (2019)
5	Agents and Tasks in Elixir	Execução de tarefas sem resposta	28	Tatarintsev (2017)
6	Synchronous Task Stream	Fluxo de Tarefas Síncronas	55	Costa (2019)
7	Pangram	Pangrama	56	Kemp (2018)
8	Parallel Letter Frequency	Frequência Letra Paralelo	76	Chetty (2018)
9	17-dining-philosophers	Jantar dos Filósofos	100	Morgan (2015)
10	Parallel Letter Frequency	Frequência Letra Paralelo	184	Defrang (2014)
11	Elixir-sorting	QuickSort e MergeSort Paralelo	481	Perrone (2018)

Fonte: Autoria própria.

No Programa 1, as funções *spawn_ monitor/1* e *Task.child_ spec/1* foram exploradas com dois módulos, um responsável por criar um processo monitorado que recebe uma lista numérica e imprime o maior valor, e outro que permite ao usuário criar tarefas e monitorá-las. O Programa 2 cria dez tarefas que duram de um a dez segundos e retorna o número de segundos pelo qual elas ficaram ociosas após determinado período de tempo. Cada tarefa ficará ociosa pelo tempo calculado através da fórmula: Identificação da tarefa * 1000 milissegundos. O programa também encerra as tarefas ociosas a mais de 5 segundos, portanto, as cinco primeiras tarefas serão impressas, enquanto, as cinco últimas serão encerradas.

O Programa 3 cria uma tarefa associada ao processo atual que irá efetuar uma soma e imprimir o resultado. O Programa 4 cria um processo que será monitorado pelo processo atual, exibindo diferentes status do processo durante sua vida, como ao nascer,

dormir e enviar mensagens. O Programa 5 contém três módulos e promovem a utilização das funções *Task.start/1* e *Task.start/3* para executar tarefas que não esperam respostas, como a impressão de uma frase indicando que um e-mail foi enviado. No Programa 6, cinco módulos fornecem funções executar três diferentes processos: (1) limitar o tempo de execução de diferentes tarefas, (2) checar a concorrência máxima de tarefas e (3) imprimir listas ordenadas.

O Programa 7 apresenta a solução do problema do Pangrama, criando várias tarefas para checar se a frase inserida é um pangrama. Nos programas 8 e 10, são apresentadas soluções para o cálculo da frequência de cada letra em determinada frase de maneira paralela. O Programa 9 apresenta a solução do problema Jantar dos Filósofos, criando cinco processos que simulam os filósofos. No Programa 11 tem-se as implementações do QuickSort e MergeSort executadas de maneira paralela, onde, em cada método, são criados dois processos que paralelamente ordenarão determinada lista numérica.

3.2.2 CASOS DE TESTE

Além do desenvolvimento do *benchmark*, tornou-se necessário o desenvolvimento do conjunto de casos de teste para cada programa. Para realizar este procedimento, utilizou-se o critério de partição de equivalência em conjunto com um estudo do código fonte dos programas. A partir desses processos, foram definidas entradas válidas e inválidas dos programa para serem utilizadas como entradas para os programas. A Tabela 4 apresenta o número de casos de teste criados para cada programa do *benchmark*. O Conjunto de casos de teste pode ser consultado no Apêndice A.

Tabela 4: Dados de teste dos programas do Benchmark

Programa	**Casos de teste**
Exemplo Spawn_Monitor	4
Yield_Many	1
Elixir Study	4
Events	1
Agents and Tasks in Elixir	2
Synchronous Task Stream	9
Pangram	2
Parallel Letter Frequency	2
17-dining-philosophers	1
Parallel Letter Frequency	2
Elixir Sorting	6

Fonte: Autoria própria.

3.2.3 AVALIAÇÃO DO BENCHMARK

O objetivo desta avaliação é verificar a habilidade do *Benchmark* em abranger todas as funções de programação do Elixir presente no *Kernel* e no módulo *Task*. Esse objetivo partiu da necessidade de criar um *benchmark* conciso que possa ser utilizado na condução de estudos experimentais na área de teste de software para programas concorrentes em Elixir, como a definição de critérios, modelos e ferramentas de teste. O *benchmark* contempla as funções do *Kernel* e do módulo *Task* por estas serem as mais utilizadas (DAVI, 2017).

Para isso, foi feito um estudo nos códigos dos programas do *benchmark* e feita uma associação entre os programas e as funções. A Tabela 5 apresenta as associações entre as funções concorrentes e os programas presentes no *Benchmark*. Os números na tabela correspondem à quantidade de vezes que a função está presente no programa. Os números em itálico representam as funções únicas em cada programa. É possível observar que o Programa 9 apresenta o maior número de funções (13). Já o Programa 1 apresenta o menor número de funções (3), e foi criado para englobar a função *Task.child_spec/1*.

A construção do *benchmark* contribui e incentiva o desenvolvimento de mais trabalhos e pesquisas na área de programação concorrente na linguagem Elixir. Além disso, após a avaliação, o *benchmark* mostra-se qualificado a validação de critérios e ferramentas de teste de software, fornecendo programas de diferentes complexidades, classificados de acordo com o número de linhas de comando e, contendo todas as funções concorrentes do *Kernel* e do módulo *Task*.

O *benchmark* possui como característica principal englobar todas as funções concorrentes em Elixir presentes no *Kernel* e no módulo *Task*. Além disso, o *benchmark* buscou explorar as diferentes aridades de algumas funções, como a *spawn*, *spawn_monitor* e *spawn_link*. O *benchmark* escontra-se disponível para download em `https://coens.dv.utfpr.edu.br/rodolfoa/pesquisa/elixir/`.

3.3 TAXONOMIA DE FALHAS

Para a definição de técnicas e critérios de teste no contexto de programas concorrentes, dois pontos importantes devem ser considerados: 1) os tipos de erros que devem ser evidenciados pelos critérios de teste; e 2) como representar o programa concorrente para obter as informações necessárias para os critérios de teste (DELAMARO; JINO; MALDONADO, 2017). No contexto de sistemas computacionais tolerantes a falhas,

Tabela 5: Relação das funções com os programas do *benchmark*

Função/Programa	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**Total**
spawn/1	0	0	0	0	0	0	0	0	0	***1***	0	1
spawn/3	0	0	0	0	0	0	0	0	***5***	0	0	5
spawn_link/1	0	0	0	0	0	0	0	0	0	0	***2***	2
spawn_link/3	0	0	0	0	0	0	0	0	***1***	0	0	1
spawn_monitor/1	***1***	0	0	0	0	0	0	0	0	0	0	1
spawn_monitor/3	0	0	0	***1***	0	0	0	0	0	0	0	1
self/0	0	0	0	0	0	0	0	0	1	1	3	5
send/2	0	0	0	1	0	0	0	0	3	3	2	9
receive/1	0	0	0	2	0	0	0	0	3	2	2	9
Task.async/1	0	1	0	0	0	0	1	0	0	0	0	2
Task.async/3	0	0	0	0	0	0	0	***1***	0	0	0	1
Task.async_stream/3	0	0	0	0	0	***3***	0	0	0	0	0	3
Task.async_stream/5	0	0	0	0	0	***2***	0	0	0	0	0	2
Task.await/1	0	0	0	0	0	0	0	***1***	0	0	0	1
Task.child_spec/1	***1***	0	0	0	0	0	0	0	0	0	0	1
Task.shutdown/2	0	1	0	0	0	0	1	0	0	0	0	2
Task.start/1	1	0	0	0	1	0	0	0	0	0	0	2
Task.start/3	0	0	0	0	***1***	0	0	0	0	0	0	1
Task.start_link/1	0	0	***1***	0	0	0	0	0	0	0	0	1
Task.start_link/3	0	0	***1***	0	0	0	0	0	0	0	0	1
Task.yield/2	0	0	0	0	0	0	***1***	0	0	0	0	1
Task.yield_many/2	0	***1***	0	0	0	0	0	0	0	0	0	1
Total	3	3	2	4	2	5	3	2	13	7	9	53

Fonte: Autoria própria.

a técnica de teste baseada em defeitos é aplicada por meio do critério de injeção de falhas (BEIZER, 1990). Para que esse critério seja aplicado e visando atender ao primeiro ponto importante no teste de programas concorrente, uma taxonomia de falhas faz-se necessária.

Diferentes taxonomias de defeitos/erro/falha foram definidas na literatura para linguagens de programação como Ada (OFFUTT; VOAS; PAYNE, 1996), MPI (VETTER; SUPINSKI, 2000; LUECKE et al., 2003; KRAMMER et al., 2003; DESOUZA et al., 2005; SAMOFALOV et al., 2005; PEDERSEN, 2006), C e C ++ (LU et al., 2008), e SystemC (SEN, 2009).

Farchi, Nir e Ur (2003) apresentam uma taxonomia de defeitos para programas Java *multithreading*. Embora a taxonomia tenha sido desenvolvida para a linguagem Java, ela pode ser aplicada em outros contextos independente da linguagem de programação. Os defeitos são categorizados com relação a (1) Código desprotegido: (a) Operação não atômica dada como atômica, (b) Acesso em dois estágios, (c) *Lock* errado ou inexistente e (d) Bloqueio de dupla checagem; (2) *Interleavings*: (a) Uso de *sleep* para sincronização e

(b) Perda de *notify*; e (3) Bloqueio ou morte de *thread*: (a) Bloqueio em região crítica e (b) *Thread* órfã.

Levando em consideração os trabalhos de Vetter e Supinski (2000), Krammer et al. (2003) e Luecke et al. (2003), DeSouza et al. (2005) apresentam uma taxonomia de erros para programas desenvolvidos em MPI. A taxonomia apresentada divide os erros em três categorias principais: (1) Sincronização sendo dividido em (a) *Deadlock*: Padrão e Dependente do tempo e (b) Condição de corrida: Interface e Entre processos; (2) Incompatibilidade, dividida em (a) Tipo de chamada, (b) Argumentos e (c) Tamanho; e (3) Recursos, classificado em (a) Alocação, (b) Inicialização e (c) Desalocação.

Lopez et al. (2018) apresentam uma taxonomia de erros de concorrência em programas baseados em atores. A taxonomia leva em consideração programas desenvolvidos em Erlang, Actor-Foundry, Scala e Java Script. Os autores apresentam um catálogo de defeitos concorrentes coletados da literatura e os classificam na taxonomia definida. A taxonomia de erros é composta por duas categorias. A primeira é denominada como Falta de Progresso, na qual estão os erros de (1) Deadlock de comunicação, (2) Deadlock comportamental e (3) Livelock. A segunda categoria é denominada Violação de Protocolo de Mensagem na qual estão os erros de (1) Violação da ordem de mensagem, (2) Combinação incorreta de mensagem e (3) Inconsistência de memória.

Embora existam alguns artigos e trabalhos, não há uma definição do processo de definição de taxonomia de falhas/defeitos. Porém, ao observar os trabalhos relacionados, as abordagens utilizadas podem ser divididas em duas categorias: estática e dinâmica.

Na categoria estática estão as abordagens que realizam um estudo preliminar na linguagem e nos possíveis defeitos que o programador pode inserir nos programas concorrentes e como esses defeitos podem ocasionar em falhas. Nessa categoria, os possíveis defeitos são obtidos a partir do conhecimento dos autores da abordagem em relação à linguagem de programação e como pequenas mudanças sintáticas podem alterar a semântica da aplicação.

Na categoria dinâmica estão os trabalhos em que um estudo é feito levando em consideração a análise de código com defeito desenvolvido por programadores. Nessa categoria estão trabalhos em que os autores aplicam um experimento para coletar informações de defeitos reais inseridos por programadores ao desenvolver aplicações concorrentes. Com base na análise de defeitos nos códigos, a taxonomia de defeitos/falhas é desenvolvida.

A Taxonomia de falhas desenvolvida para Elixir está inserida na categoria

estática de definição de taxonomia de falhas, na qual somente a sintaxe da linguagem de programação é considerada e a taxonomia de falhas é gerada a partir da análise dos autores sobre as funções e os possíveis enganos que podem ser cometidos pelos programadores ao desenvolver aplicações concorrentes em Elixir. Cada defeito do conjunto de defeitos definidos foi semeado em códigos e executou-se para que as falhas pudessem ser obtidas.

3.3.1 DEFINIÇÃO DA TAXONOMIA

Para a definição da taxonomia de falhas para programas concorrentes em Elixir, o processo mostrado na Figura 6 foi seguido. O processo divide-se em 5 etapas:

1. Identificação das funções concorrentes
2. Agrupamento de funções com semântica semelhante
3. Identificação dos enganos
4. Definição de defeitos
5. Execução e coleta de falhas

Cada uma das etapas é apresenta com mais detalhes a seguir.

3.3.1.1 IDENTIFICAÇÃO DA FUNÇÕES CONCORRENTES

A partir da análise da documentação da linguagem de programação Elixir [1], identificou-se os módulos que apresentam funções de programação concorrente disponíveis na linguagem. Como resultado, encontrou-se que funções que permitem a programação concorrente estão distribuídas em seis módulos: *Kernel*, *Task*, *Process*, *Agent*, *Genserver* e *Task.Supervisor*.

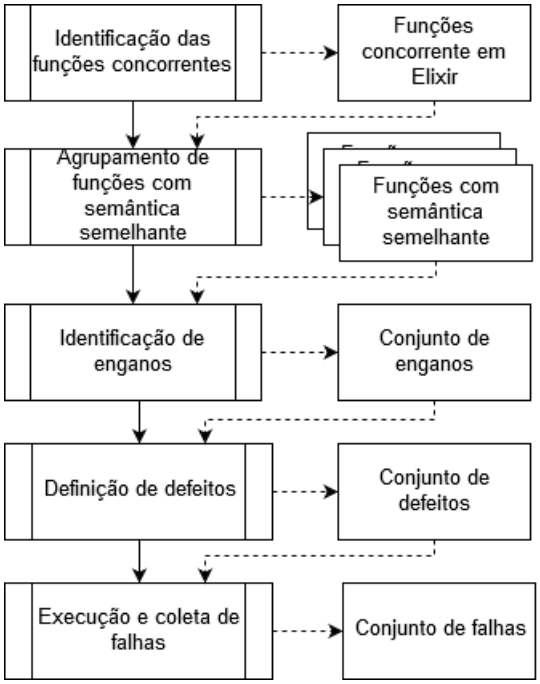
Para a definição da taxonomia foram consideradas apenas as funções do *Kernel* e do módulo *Task* (Tabela 1) por serem os mais utilizados e difundidos pela comunidade de desenvolvedores.

3.3.1.2 AGRUPAMENTO DE FUNÇÕES COM SEMÂNTICA SEMELHANTE

O segundo passo consistiu no agrupamento de funções com semântica semelhante. Duas funções são consideradas de semântica semelhante quando possuem objetivos similares

[1] https://elixir-lang.org/docs.html

Figura 6: Processo de definição da taxonomia de falhas para funções concorrentes em Elixir



Fonte: Autoria própria.

no contexto da programação, diferenciando-se entre si, por exemplo, em como uma ação é realizada pelo programa. O objetivo desse passo é identificar funções que possuem o mesmo objetivo, porém que se diferem entre si por alguma característica especial.

A Tabela 6 apresenta o agrupamento das funções. Por exemplo, um conjunto de funções agrupadas no módulo *Kernel* é composto por funções de criação de processos *spawn/1*, *spawn/3*, *spawn_link/1*, *spawn_link/3*, *spawn_monitor/1* e *spawn_monitor/3*. Essas funções estão no mesmo grupo pois realizam a criação de processos e são semelhantes semanticamente. Porém, sintaticamente elas se diferem primeiramente pela aridade, sendo que uma recebe um parâmetro e a outra recebe três parâmetros. Além disso, as funções *spawn*, *spawn_link* e *spawn_monitor* se diferem entre si pois a função *spawn_link* mantém um vínculo de ligação com o processo pai (que cria o processo) e o *spawn_monitor* o processo criado é monitorado pelo processo pai.

3.3.1.3 IDENTIFICAÇÃO DOS ENGANOS

Após a definição dos grupos de funções com semântica semelhantes, realizou-se uma análise sintática das funções de cada grupo, observando os parâmetros de cada função e destacando as mudanças sintáticas entre elas. Por exemplo, as funções de *spawn/1*,

Tabela 6: Agrupamento de funções com semântica semelhante no Kernel e Task

Grupo	**Função**
Funções de Criação de Processos no Kernel	spawn / 1 spawn / 3 spawn_link / 1 spawn_link / 3 spawn_monitor / 1 spawn_monitor / 3
Função de Envio de Mensagem no Kernel	send / 2
Função de Recebimento de Mensagem no Kernel	receive
Funçõe de criação de tarefas no módulo Task	Task.start / 1 Task.start / 3 Task.start_link / 1 Task.start_link / 3 Task.async / 1 Task.async / 3 Task.async_stream / 3 Task.async_stream / 5
Funções de respostas de tarefas no módulo Task	Task.await / 1 Task.yield / 2 Task.yield_many / 2
Função de encerramento de tarefas no módulo Task	Task.shutdown / 2
Função de monitoramento e supervisão de tarefas no módulo Task	Task.child_spec / 1

Fonte: Autoria própria.

spawn_link/1 e *spawn_monitor/1* diferem-se entre si sintaticamente por possuir um sufixo *_link*, *_monitor* ou por não possuir sufixo:

spawn(fn -> do_something() end)

spawn_link(fn -> do_something() end)

spawn_monitor(fn -> do_something() end)

A partir dessa análise, identificou-se possíveis enganos que os programadores podem cometer ao utilizar cada grupo de função que possuem sintaxe semelhante e semântica diferente, porém parecida, como é o caso das funções de criação de processos *spawn*.

Como resultado, obteve-se um conjunto de possíveis enganos que um programador pode cometer ao programar utilizando as funções concorrentes em Elixir dos módulos *Kernel* e *Task*. Esse conjunto de enganos servem como base para a definição da taxonomia de defeitos.

3.3.1.4 DEFINIÇÃO DE DEFEITOS

A taxonomia de defeitos apresentada abaixo simula os enganos facilmente cometidos por programadores durante o desenvolvimento de programas concorrentes em Elixir utilizando as funções presentes nos módulos *Kernel* e *Task*.

Cinco categorias de defeitos foram definidas: (1) Defeitos por troca de função, (2) Defeitos por ausência de função, (3) Defeitos por ausência de parâmetro (4) Defeitos por adição de parâmetro e (5) Defeitos por troca de parâmetro. Os defeitos por troca de função correspondem as alterações nas chamadas das funções. Os defeitos por ausência de função simulam esquecimentos cometidos por programadores na chamada de funções. As diferentes aridades de algumas funções do Elixir podem levar a enganos durante o desenvolvimento de software. Os defeitos causados pela ausência ou por adição de parâmetros entram nesse contexto. Os defeitos por troca de parâmetros estão relacionados ao uso da palavra-chave __MODULE__ que pode ser utilizada quando a função a ser executada está no módulo atual de execução.

Os defeitos identificados para o módulo *Kernel* podem ser observados na Tabela 7 e os defeitos identificados para o módulo *Task* podem ser observados na Tabela 8. A primeira coluna da tabela representa a categoria dos defeitos. A segunda coluna apresenta a função alvo na qual deverá ser inserido o defeito e a última coluna apresenta a função com o defeito inserido. Por exemplo, a primeira linha da Tabela 7 apresenta o defeito da Troca de função *spawn/1* por *spawn_link/1*.

Essa taxonomia de defeitos é utilizada para semear defeitos em programas para que a taxonomia de falhas seja definida, como mostrado a seguir.

3.3.1.5 EXECUÇÃO E COLETA DE FALHAS

O último passo consistiu na semeadura de defeitos em código concorrente Elixir, a execução desses códigos e a coleta das falhas encontradas. Nesse passo, buscou-se simular as possíveis falhas que podem ser causadas pela inserção dos defeitos definidos no passo anterior. Como resultado, foram obtidas falhas que foram agrupadas em *Exceções da linguagem Elixir* e *Falhas relacionadas à programação concorrente*. As Tabelas 9 e 10 apresentam as falhas obtidas no *Kernel* e módulo *Task* respectivamente.

O grupo *Exceções da linguagem Elixir* contém três falhas: (1) *Argument Error*, (2) *Function Clause Error* e, (3) *Match Error*. Tais falhas são lançadas pela própria máquina virtual do Erlang (que roda os processos em Elixir).

Tabela 7: Categoria de defeitos do módulo Kernel

Categoria	Função	Defeito
Troca de função	spawn/1	spawn_link/1
	spawn/1	spawn_monitor/1
	spawn/3	spawn_link/3
	spawn/3	spawn_monitor/3
	spawn_link/1	spawn/1
	spawn_link/1	spawn_monitor/1
	spawn_link/3	spawn/3
	spawn_link/3	spawn_monitor/3
	spawn_monitor/1	spawn/1
	spawn_monitor/1	spawn_link/1
	spawn_monitor/3	spawn/3
	spawn_monitor/3	spawn_link/3
Deleção de função	spawn/1	//
	spawn/3	//
	spawn_link/1	//
	spawn_link/3	//
	spawn_monitor/1	//
	spawn_monitor/3	//
	self/0	//
	send/2	//
	receive/1	//
Deleção de parâmetro	spawn/3	spawn/1
	spawn_link/3	spawn_link/1
	spawn_monitor/3	spawn_monitor/1
	Receive/2	Receive/1
	Receive/2	Receive/1
Adição de parâmetro	Receive/1	Receive/2
Troca de parâmetro	Spawn/3	__MODULE__
	Spawn_link/3	__MODULE__
	Spawn_monitor/3	__MODULE__

Fonte: Autoria própria.

1. ***Argument Error:*** Esta falha é obtida durante a utilização de funções com parâmetros incorretos. Como exemplo é possível citar a função *spawn* que espera como parâmetro uma função, a qual será executa. Caso seja passado um número, uma string ou qualquer outro dado que não seja uma função, será obtido *Argument Error*.

2. ***Function Clause Error:*** Esta falha é lançada quando uma função é executada com parâmetros que não a possibilitam ser executada. Embora seu conceito seja muito semelhante ao *Argument Error*, a grande diferença é que no *Argument Error* a função tenta executar seu objetivo com o valor recebido como parâmetro e dispara

Tabela 8: Categoria de defeitos do módulo Task

Categoria	Função	Defeito
Troca de Função	async/1	start/1
	async/1	start_link/1
	async/3	start/3
	async/3	start_link/3
	start/1	start_link/1
	start/1	async/1
	start/3	start_link/3
	start/3	async/3
	start_link/1	start/1
	start_link/1	async/1
	start_link/3	start/3
	start_link/3	async/3
	await/2	yield/2
	await/2	yield_many/2
	yield/2	await/2
	yield/2	yield_many/2
	yield_many/2	await/2
	yield_many/2	yield/2
Deleção de Funções	async/1	//
	async/3	//
	async_stream/3	//
	async_stream/5	//
	await/2	//
	child_spec/1	//
	shutdown/2	//
	start/1	//
	start/3	//
	start_link/1	//
	start_link/3	//
	yield/2	//
	yield_many/2	//
Deleção de Parâmetros	async/3	async/1
	async_stream/5	async_stream/3
	start/3	start/1
	start_link/3	start_link/1
	await/2	await/1
	yield/2	yield/1
	yield_many/2	yield_many/1
Troca de Argumento	Async/3	__MODULE__
	Start/3	__MODULE__
	Start_link/3	__MODULE__

Fonte: Autoria própria.

o erro quando o processo não é concluído. *Function Clause Error* acontece quando a função verifica os parâmetros recebidos e em caso de dados incorretos, nem inicia seu processo. Um erro de *Function Clause Error* pode-se obtido ao chamar a função

Tabela 9: Falhas do módulo Kernel para cada categoria de defeitos

Categoria	Falha
Troca de função	Finalização precoce
	Deadlock comportamental
	Match Error
Deleção de função	Finalização Precoce
	Deadlock comportamental
Deleção de parâmetro	Argument Error
	Deadlock comportamental
Adição de parâmetro	Finalização precoce
Troca de parâmetro	Dados incorretos

Fonte: Autoria própria.

Tabela 10: Falhas do módulo Task para cada categoria de defeitos

Categoria	Falha
Troca de Função	Function Clause Error
	Finalização precoce
	Argument Error
	Deadlock comportamental
	Dados incorretos
Deleção de Funções	Function Clause Error
	Intercalação de mensagem incorreta
	Finalização precoce
	Dados incorretos
Deleção de Parâmetros	Function Clause Error
	Finalização precoce
	Intercalação de mensagem incorreta
Troca de parâmetro	Dados incorretos

Fonte: Autoria própria.

Task.yield_many passando como parâmetro somente uma tarefa, sendo que a mesma espera uma lista de tarefas como parâmetro.

3. ***Match Error:*** Esta falha acontece na tentativa de corresponder um dado a uma variável sendo que ambos não são compatíveis. Em Elixir esta falha pode ser obtida, por exemplo, ao atrelas a uma tupla de tamanho dois o retorno da função *spawn*, já que está retorna somente um valor.

O grupo *Falhas relacionadas à programação concorrente* contém erros relacionados ao comportamento do programa e resultado gerado. Nesta categoria estão presentes: (1) Dados incorretos, (2) *Deadlock* comportamental, (3) Finalização precoce e (4) Intercalação de mensagem incorreta.

1. ***Dados Incorretos:*** Esta falha está associada ao resultado incorreto produzido pelo programa. Um exemplo desta falha pode ser simulado com a substituição do primeiro parâmetro da função *spawn* pela tag $\_\_MODULE\_\_$ em um programa solução de calculadora com diversos módulos para realizar cada operação. Caso a chamada da função *spawn* para o cálculo da soma seja alterada conforme comentado, pode ser obtido qualquer valor diferente da soma esperada.

2. ***Deadlock Comportamental:*** Esta falha está relacionada ao comportamento do sistema que geralmente encontra-se travado. O problema acontece durante a troca de mensagens entre processos, sendo a causa do problema, um processo esperando uma mensagem que nunca lhe será enviada. Um exemplo de *Deadlock comportamental* pode ser gerado ao realizar a substituição da função *spawn_link* por *spawn* e induzir um erro de tipos incorretos de dados no processo. Nesta simulação, *spawn_link* iria encerrar ambos os processos ao lançar a exceção de tipo de dados, enquanto com *spawn*, somente o processo que originou o erro lançaria a exceção e o processo pai ficaria travado esperando a resposta.

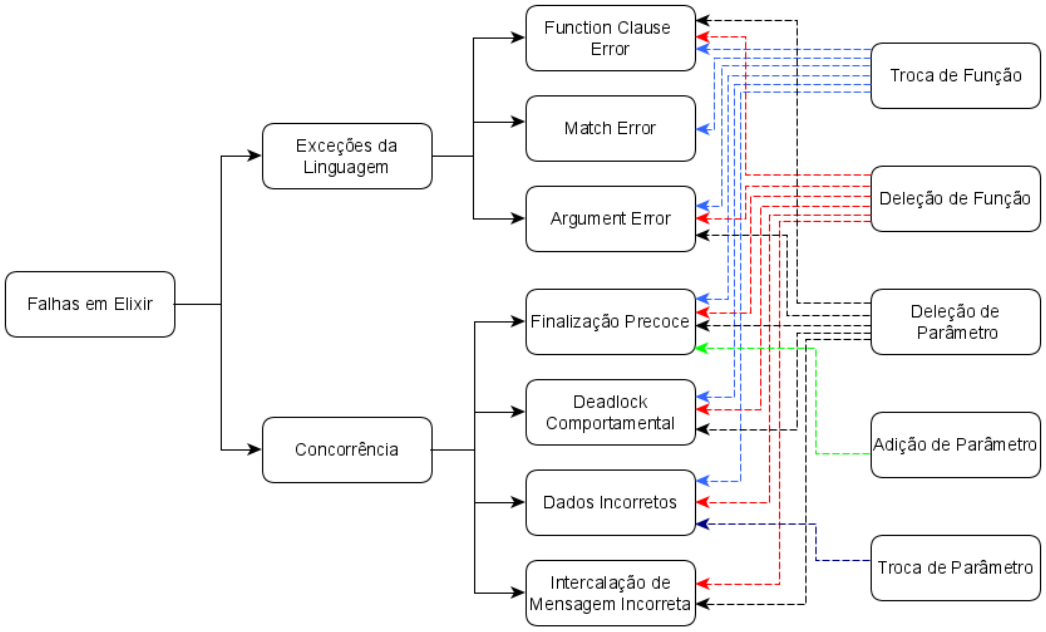
3. ***Finalização Precoce:*** Os erros de finalização precoce ocorrem quando o programa termina a execução sem ter terminado o seu processamento por completo. Por exemplo, ao utilizar a função *spawn* para a criação de processo, o processo criador não é finalizado caso o processo criado tenha algum problema, caso que não ocorre quando a função *spawn_link* é utilizada. Com isso, ao trocar essas funções, o processo criador é finalizado antes de completar a sua execução completa. Outro exemplo do erro é a deleção de uma função *spawn* que continha um retorno assíncrono, já que ao executar o programa sem a função, nada é executado e o programa finaliza sua execução.

4. ***Intercalação de Mensagem Incorreta*** Este erro acontece quando o resultado produzido pelo programa difere do esperado, como por exemplo ao executar um programa que executa soma, substração e multiplicação de valores e, ao invés de obter o retorno nesta ordem, obter primeiro a subtração, seguido pela soma e pela multiplicação. O problema está associado a funções de recebimento de mensagens, como por exemplo a substituição da função *Task.await* por *Task.yield*, já que enquanto a primeira aguarda a mensagem, *Task.yield* somente verifica se a mensagem chegou e não fica bloquada. Tal mudança pode impactar em *Intercalação de Mensagem Incorreta*.

A Figura 7 apresenta as categorias da taxonomia de falhas e os defeitos relacionados

a elas.

Figura 7: Visão geral das falhas obtidas



Fonte: Autoria própria.

3.3.2 APLICAÇÃO DA TAXONOMIA DE FALHAS

Para a aplicação da taxonomia de falhas em programas concorrentes em Elixir, o *benchmark* de programas concorrentes em Elixir, apresentado na seção anterior, foi utilizado como base para a semeadura dos defeitos e coleta das falhas.

Para cada programa presente no *benchmark* (Tabela 3), foi realizada uma busca manual por cada função de concorrência e semeado um defeito para cada ocorrência da função na taxonomia de defeitos (Tabela 7 e Tabela 8). Como resultado, um conjunto de programas defeituosos foi gerado, como mostram a Tabela 11 e a Tabela 12. Nessas tabelas, cada linha apresenta o número de programas defeituosos gerados para cada uma das categorias da taxonomia de defeitos.

Para a execução dos programas, um conjunto de dados de teste foi selecionado observando as condições de Alcançabilidade, Infecção e Propagação (R.I.P. - *Reachability, Infection and Propagation*) que o dado de teste possuía com relação ao defeito semeado. Como resultado, foi definido um conjunto de casos de teste que é adequado a encontrar as falhas definidas na taxonomia de falhas. A Tabela 13 apresenta a quantidade de dados de teste presente no conjunto final de caso de teste para cada um dos programas do

Tabela 11: Defeitos semeados nas funções do módulo Kernel

Programa	**Categoria**					**Total**
	Troca de função	Deleção de função	Deleção de parâmetro	Adição de parâmetro	Troca de parâmetro	
Exemplo Spawn_Monitor	1	1	0	0	0	2
Yield_Many	0	0	0	0	0	0
Elixir Study	0	0	0	0	0	0
Events	0	3	2	2	0	7
Agents and Tasks in Elixir	0	0	0	0	0	0
Synchronous Task Stream	0	0	0	0	0	0
Pangram	0	0	0	0	0	0
Parallel Letter Frequency	0	0	0	0	0	0
17-dining-philosophers	6	11	4	3	6	30
Parallel Letter Frequency	1	6	3	2	0	12
Elixir Sorting	4	6	2	2	0	14
Total	14	29	11	9	6	65

Fonte: Autoria própria.

Tabela 12: Defeitos semeados nas funções do módulo Task

Programa	**Categoria**					**Total**
	Troca de função	Deleção de função	Deleção de parâmetro	Adição de parâmetro	Troca de parâmetro	
Exemplo Spawn_Monitor	1	0	0	0	0	1
Yield_Many	0	1	1	0	0	2
Elixir Study	2	2	0	0	0	4
Events	0	0	0	0	0	0
Agents and Tasks in Elixir	2	2	0	0	0	4
Synchronous Task Stream	0	0	0	0	0	0
Pangram	1	2	1	0	0	4
Parallel Letter Frequency	1	0	0	0	0	1
17-dining-philosophers	0	0	0	0	0	0
Parallel Letter Frequency	0	0	0	0	0	0
Elixir Sorting	0	0	0	0	0	0
Total	7	8	2	0	0	16

Fonte: Autoria própria.

benchmark.

Por fim, com a execução dos programas defeituosos com os dados de teste, é possível encontrar as falhas nos programas, como apresentadas na taxonomia de falhas. A última coluna da Tabela 13 apresenta as falhas apresentadas pelos programas defeituosos de cada um dos programas do *benchmark*.

Tabela 13: Dados de teste e falhas encontradas

Programa	**Dados de teste**	**Falhas encontradas**
Exemplo Spawn_Monitor	4	Finalização Precoce
Yield_Many	1	Dados incorretos
Elixir Study	4	Deadlock Comportamental, Finalização Precoce
Events	1	Function Clause Error, Deadlock Comportamental, Finalização Precoce
Agents and Tasks in Elixir	2	Finalização Precoce
Synchronous Task Stream	9	-
Pangram	2	Dados incorretos
Parallel Letter Frequency	2	Dados incorretos
17-dining-philosophers	1	Argument Error, Dados incorretos, Deadlock Comportamental, Intercalação de mensagem incorreta
Parallel Letter Frequency	2	Deadlock Comportamental, Finalização Precoce
Elixir Sorting	6	Argument Error, Deadlock Comportamental, Finalização Precoce

Fonte: Autoria própria.

3.4 OPERADORES DE MUTAÇÃO

A partir da Taxonomia de Falhas desenvolvida, criaram-se operadores de mutação para apoiar o teste de mutação para programas concorrentes em Elixir. Tais operador exploram as funções concorrentes presentes no Kernel e no módulo Task. Para cada operador é apresentado uma explicação de como o operador funciona e qual seu objetivo. Na sequência, é apresentado um exemplo de como a mutação irá ocorrer no programa.

Os operadores de mutação foram divididos em cinco categorias: (1) Troca de função, (2) Deleção de função, (3) Deleção de parâmetro de função, (4) Adição de parâmetro em função e, (5) Troca de parâmetro em função. Os operadores foram divididos em tais categorias conforme o tipo de alteração realiza pelo operador. A categoria "Troca de função" apresenta operadores que realizam a substituição entre funções concorrentes com objetivos e aplicações semelhantes. A categoria "Deleção de função" opera removendo determinada função concorrente do programa. A categoria "Deleção de parâmetro de função" trabalha transformando funções de aridades maiores para suas versões em aridades menores. A categoria "Adição de parâmetro em função" compreende a adição de um novo parâmetro a determinada função concorrente. Por fim, a categoria "Troca de parâmetro em função" explora funções com aridades altas, substituindo determinado parâmetro por alguma palavra chave da linguagem. A Tabela 14 apresenta os operadores de mutação

para programas concorrentes em Elixir.

Tabela 14: Operadores de Mutação para Programas Concorrentes em Elixir

Categoria	Operador	Descrição
Troca de Função	ReplSpawn	Troca de função de criação de processos no Kernel.
Troca de Função	ReplTaskStart	Troca de função de criação de tarefas no módulo Task.
Troca de Função	ReplTaskAnswer	Troca de função de respostas de tarefas no módulo Task.
Deleção de Função	DelSpawn	Deleção de função de criação de processos no Kernel.
Deleção de Função	DelSend	Deleção de função de envio de mensagens no Kernel.
Deleção de Função	DelReceive	Deleção de função de recebimento de mensagens no Kernel.
Deleção de Função	DelTaskStart	Deleção de função de criação de tarefas no módulo Task.
Deleção de Função	DelTaskAnswer	Deleção de função de respostas de tarefas no módulo Task.
Deleção de Função	DelShutdown	Deleção de função de encerramento de tarefas no módulo Task.
Deleção de Função	DelChildSpec	Deleção de função de monitoramento e supervisão de tarefas no módulo Task.
Deleção de Parâmetro de Função	DelParameterReceive	Deleção de parâmetros na função de recebimento de mensagens entre processos no Kernel.
Deleção de Parâmetro de Função	DelTimeoutTaskAnswer	Deleção de parâmetros nas funções de respostas de tarefas no módulo Task.
Adição de Parâmetro em Função	AddAfterReceive	Adição de parâmetro na função de recebimento de mensagens no Kernel.
Troca de Parâmetro em Função	ReplModuleSpawn	Troca de parâmetros nas funções de criação de processos no Kernel.
Troca de Parâmetro em Função	ReplModuleTaskCreate	Troca de parâmetros nas funções de criação de tarefas no módulo Task.

Fonte: Autoria própria.

3.4.1 OPERADORES DE MUTAÇÃO DE TROCA DE FUNÇÃO

3.4.1.1 REPLSPAWN - TROCA DE FUNÇÃO DE CRIAÇÃO DE PROCESSOS NO KERNEL

Este operador explora as substituições entre as funções *spawn*, *spawn_link* e *spawn_monitor*, responsáveis pela criação de processos no Kernel. Nas ocorrências de uma dessas funções, será criado um mutante para cada uma das outras, com base na Tabela 15.

Na tabela, é possível nota a ausência da troca de *spawn* por *spawn_monitor*

Tabela 15: Mutações criadas pelo operador ReplSpawn

Original	Mutantes
spawn	spawn_link
spawn_link	spawn spawn_monitor
spawn_monitor	spawn_link

Fonte: Autoria Própria.

e vice-versa. Tais mudanças foram ignoradas pois, não foram identificados problemas significativos na troca. O Algoritmo 13 apresenta um exemplo de aplicação do operador *ReplSpawn*.

13 Exemplo de aplicação do operador *ReplSpawn*

```
Original: spawn_link(fn -> :ok end)

Mutante1: spawn(fn -> :ok end)

Mutante2: spawn_monitor(fn -> :ok end)
```

Pode-se observar que tanto o programa original quanto os programa mutantes executam a mesma função, no entanto, a função *spawn_link* presente no programa original foi substituída por *spawn* no Mutante1 e *spawn_monitor* no Mutante2. Utilizando este operador podem ser obtidos falhas de *deadlock* comportamental e finalização precoce devido a característica de propagação e ligação que a função *spawn_link* cria entre o processo criador e o processo criado. Este link faz com que caso um problema seja identificado no processo criado, esse problema será propagado para o processo criador.

Caso a função spawn ou spawn_link esteja atreladas a uma variável (onde o retorno da função seja atribuído a uma variável) não poderá ser feita a troca com spawn_monitor e o contrário também é aplicado. Vale ressaltar que a troca entre spawn e spawn_link pode ser realizado mesmo quando as funções estiverem atreladas a uma variável, pois o tipo de retorno é o mesmo.

3.4.1.2 REPLTASKSTART - TROCA DE FUNÇÃO DE CRIAÇÃO DE TAREFAS NO MÓDULO TASK

Este operador realizará a a substituição da função *Task.start* por *Task.start_link* e vice-versa, responsáveis pela criação de tarefas no módulo Task. O Algoritmo 14 apresenta um trecho de código de aplicação do operador no qual há a substituição da função *Task.start*

por *Task.start_link* no Mutante1.

14 Exemplo de aplicação do operador *ReplTaskStart*

```
Original: Task.start(fn -> :ok end)

Mutante1: Task.start_link(fn -> :ok end)
```

A aplicação deste operador poderá obter falhas de *deadlock* e finalização precoce devido a característica de propagação e ligação da função *Task.start_link*.

Neste operador foi desconsiderada a função *Task.async* devido a sua característica de sincronização ser diferente das funções *Task.start* e *Task.start_link*.

3.4.1.3 REPLTASKANSWER - TROCA DE FUNÇÃO DE RESPOSTAS DE TAREFAS NO MÓDULO TASK

O presente operador efetuará a substituição da função *Task.await* por *Task.yield* e vice-versa, funções responsáveis pela obtenção de respostas de tarefas assíncronas no módulo *Task*. A substituição demonstra importância devido a diferença de comportamento entre as duas funções, sendo que, enquanto a função *Task.await* aguarda a tarefa executar para obter a resposta, a função *Task.yield* apenas verifica se esta já executou e não fica aguardando por ela. O Algoritmo 15 apresenta um trecho de código de aplicação do operador.

15 Exemplo de aplicação do operador *ReplTaskAnswer*

```
Original: tarefa = Task.async(fn -> :ok end)
          Task.await(tarefa)

Mutante1: tarefa = Task.async(fn -> :ok end)
          Task.yield(tarefa)
```

No exemplo, uma tarefa é criada através da função *Task.async* e posteriormente recebida através das funções *Task.await* (Original) e *Task.yield* (Mutante1). Com a aplicação deste mutante é possível simular falhas de finalização precoce, substituindo *Task.yield* por *Task.await*, e processamento de dados incorretos, substituindo *Task.await* por *Task.yield*. Tais falhas tornam-se possíveis devido a característica bloqueante da função *Task.await*.

3.4.2 OPERADORES DE MUTAÇÃO DE DELEÇÃO DE FUNÇÃO

3.4.2.1 DELSPAWN - DELEÇÃO DE FUNÇÃO DE CRIAÇÃO DE PROCESSOS NO KERNEL

Este operador tem por objetivo deletar ocorrências das funções *spawn*, *spawn_link* e *spawn_monitor*. Para cada ocorrência de função de criação de processos será criado um mutante sem a chamada da função. A mudança só será aplicada em situações que o retorno da função não esteja atrelado a uma variável para garantir a geração de mutantes passíveis de execução. No Algoritmo 16 é possível observar um exemplo de aplicação do operador.

16 Exemplo de aplicação do operador *DelSpawn*

```
Original: def escrevaOlaMundo() do
             spawn(fn -> IO.puts "Ola Mundo!" end)
          end

Mutante1: def escrevaOlaMundo() do

          end
```

Pode-se observar que o Mutante1 não apresenta a função *spawn* e portanto, não conseguirá escrever a mensagem "Ola Mundo!" em sua execução. A aplicação desse operador poderá simular falhas de finalização precoce, como por exemplo, na retirada de uma função *spawn* que criaria um processo responsável por iniciar a aplicação.

3.4.2.2 DELSEND - DELEÇÃO DE FUNÇÃO DE ENVIO DE MENSAGENS NO KERNEL

Este operador tem como objetivo a deleção da função de envio de mensagens entre processos no Kernel. Para cada ocorrência desta função, será criado um mutante sem a chamada da mesma. O Algoritmo 17 apresenta um exemplo de aplicação do operador.

No exemplo apresentado acima é possível observar que o Mutante1 não executará nada na chamada da função *enviar_mensagem* devido a ausência da função *send*. Através deste operador será possível simular falhas de *deadlock comportamental* já que algum *receive* posterior a execução da função *send* não receberá mensagem alguma.

17 Exemplo de aplicação do operador *DelSend*

```
Original: def enviar_mensagem(destino, mensagem) do
             send(destino, mensagem)
          end

Mutante1: def enviar_mensagem(destino, mensagem) do

          end
```

3.4.2.3 DELRECEIVE - DELEÇÃO DE FUNÇÃO DE RECEBIMENTO DE MENSAGENS NO KERNEL

Este operador tem como objetivo a retirada da função de recebimento de mensagem do Kernel. Para cada ocorrência da função, será criado um mutante que não possuirá esta chamada. No Algoritmo 18 é apresentado um exemplo de aplicação do operador *DelReceive*.

18 Exemplo de aplicação do operador *DelReceive*

```
Original: def receber_mensagem() do
             receive do
                :ok -> IO.puts "Recebeu a mensagem"
             end
          end

Mutante1: def receber_mensagem() do

          end
```

Pode-se observar no Mutante1 a ausência da função *receive*, assim, tal processo que executa a função *receber_mensagem* nunca receberá a mensagem esperada. Aplicando-se o mutante podem ser obtidos falhas de finalização precoce, já que os dados que deveriam ser obtidos não são obtidos e a aplicação encerra.

3.4.2.4 DELTASKSTART - DELEÇÃO DE FUNÇÃO DE CRIAÇÃO DE TAREFAS NO MÓDULO TASK

Este operador tem por objetivo a deleção de chamadas das funções *Task.start* e *Task.start_link*, responsáveis pela criação de tarefas no módulo Task. Para cada ocorrência da função, será criado um mutante que não possuirá a chamada. A mudança só será aplicada em situações em que o retorno da função não esteja atrelado a uma variável para

garantir a geração de mutantes passíveis de execução. No Algoritmo 19 é apresentado um exemplo de aplicação deste operador.

19 Exemplo de aplicação do operador *DelTaskStart*

```
Original: def soma(a, b) do
             Task.start(fn -> IO.puts a + b end)
          end

Mutante1: def soma(a, b) do

          end
```

Pode-se observar que o Mutante1 não contém a chamada da função *Task.start*, portanto, a chamada da função *soma* nunca executará ação alguma. Aplicando-se este operador, pode-se obter falhas de finalização precoce devido a ausência da função responsável pela criação de tarefas.

3.4.2.5 DELTASKANSWER - DELEÇÃO DE FUNÇÃO DE RESPOSTAS DE TAREFAS NO MÓDULO TASK

Este operador de mutação tem por objetivo a deleção de ocorrências das funções de obtenção de respostas de tarefas no módulo Task: *Task.await*, *Task.yield* e *Task.yield_many*. Para cada ocorrência da função será criado um mutante que não possuirá esta chamada. A mudança só será aplicada em situações em que o retorno da função não esteja atrelado a uma variável para garantir a geração de mutantes passíveis de execução. O Algoritmo 20 apresenta um exemplo de aplicação do operador.

20 Exemplo de aplicação do operador *DelTaskAnswer*

```
Original: def subtracao(a, b) do
             tarefa = Task.async(fn -> a - b end)
             IO.puts Task.await(tarefa)
          end

Mutante1: def subtracao(a, b) do
             tarefa = Task.async(fn -> a - b end)

          end
```

Pode-se observar a ausência da função *Task.await* no Mutante1, assim embora seja criada uma tarefa que efetua a subtração de dois números, nunca será obtido este

valor. Tal operador de mutação pode ocasionar em falhas de intercalação de mensagens incorretas, como por exemplo em um programa com três funções *Task.await*, a retirada da primeira chamada produzirá um resultado diferente da original.

3.4.2.6 DELSHUTDOWN - DELEÇÃO DE FUNÇÃO DE ENCERRAMENTO DE TAREFAS NO MÓDULO TASK

Este mutante tem por objetivo a retirada da função *Task.shutdown*, responsável pelo encerramento de tarefas no módulo Task. Para cada ocorrência da função, será criado um mutante que não possuirá a chamada. O Algoritmo 21 apresenta um exemplo de aplicação do operador.

Pode-se perceber a ausência da função *Task.shutdown* no Mutante1. A ausência desta função fará com que as tarefas que estourarem o tempo limite de 5 segundos não sejam encerradas. Este operador poderá simular dados incorretos, onde uma tarefa que deveria ser encerrada continua viva devido a ausência da função que realizaria seu encerramento.

3.4.2.7 DELCHILDSPEC - DELEÇÃO DE FUNÇÃO DE MONITORAMENTO E SUPERVISÃO DE TAREFAS NO MÓDULO TASK

Este operador tem por objetivo a retirada da função *Task.child_spec*, utilizada no monitoramento e supervisão de tarefas no módulo Task. Para cada ocorrência da função, será criado um mutante que não possuirá esta chamada. A mudança só será aplicada em situações que o retorno da função não esteja atrelado a uma variável para garantir a geração de mutantes passíveis de execução. No Algoritmo 22 é apresentado um exemplo de aplicação deste operador na qual pode-se observar a retirada da função *Task.child_spec* no Mutante1. A aplicação deste operador pode simular falhas de finalização precoce no programa.

3.4.3 OPERADORES DE MUTAÇÃO DE DELEÇÃO DE PARÂMETRO DE FUNÇÃO

3.4.3.1 DELPARAMETERRECEIVE - DELEÇÃO DE PARÂMETRO NA FUNÇÃO DE RECEBIMENTO DE MENSAGENS ENTRE PROCESSOS NO KERNEL

Este operador possui a finalidade de deletar um parâmetro (*match*) da função de recebimento de mensagens no Kernel, sendo ele um *match* ou um *after* responsável pelo tempo limite da função. Para ambas as ocorrências, será criado um mutante sem a presença deste *match*. O Algoritmo 23 apresenta um exemplo de aplicação do operador.

21 Exemplo de aplicação do operador *DelShutdown*

```
Original: tasks =
            for i <- 1..10 do
                Task.async(fn ->
                   Process.sleep(i * 1000)
                   i
                end)
              end

              tasks_with_results = Task.yield_many(tasks, 5000)
              results =
                Enum.map(tasks_with_results, fn {task, res} ->
                  res || Task.shutdown(task, :brutal_kill)
                end)

              for {:ok, value} <- results do
                IO.inspect(value)
              end

Mutante1: tasks =
              for i <- 1..10 do
                 Task.async(fn ->
                    Process.sleep(i * 1000)
                    i
                 end)
              end

              tasks_with_results = Task.yield_many(tasks, 5000)
              results =
                Enum.map(tasks_with_results, fn {task, res} ->
                  res
                end)

              for {:ok, value} <- results do
                IO.inspect(value)
              end
```

22 Exemplo de aplicação do operador *DelChildSpec*

```
Original: def monitorar(tarefa) do
             Task.child_spec(tarefa)
          end

Mutante1: def monitorar(tarefa) do

          end
```

23 Exemplo de aplicação do operador *DelParameterReceive*

```
Original: send(self(), :olamundo)
          receive do
            :olamundo -> IO.puts "Ola mundo!"
            :helloworld -> IO.puts "Hello World!"
          end

Mutante1: send(self(), :olamundo)
          receive do

            :helloworld -> IO.puts "Hello World!"
          end
```

É possível observar que no Mutante1, a função *receive* só contém um *match* (*:helloworld*), estando ausente o primeiro *match* (*:olamundo*). Tal mudança pode simular falhas de *deadlock* comportamental devido a característica bloqueante do *receive*. A retirada do *match* de um *receive* levará ao *deadlock* comportamental pois quando executado, o *receive* nunca obterá a resposta correspondente aquele *match*.

3.4.3.2 DELTIMEOUTTASKANSWER - DELEÇÃO DE PARÂMETRO NAS FUNÇÕES DE RESPOSTAS DE TAREFAS NO MÓDULO TASK

Este operador tem como objetivo a deleção do parâmetro de tempo limite (*timeout*) das funções *Task.await*, *Task.yield* e *Task.yield_many*. Para cada ocorrência da função com *timeout* definido, será criado um mutante com a chamada da função mas sem a presença do tempo limite. O Algoritmo 24 apresenta um exemplo de aplicação deste operador.

24 Exemplo de aplicação do operador *DelTimeoutTaskAnswer*

```
Original: tarefa = Task.asycn(fn ->
                      Process.sleep(7000)
                      :ok
                   end)
          Task.await(tarefa, 10000)

Mutante1: tarefa = Task.async(fn ->
                      Process.sleep(7000)
                      :ok
                   end)
          Task.await(tarefa)
```

Pode-se perceber a ausência do parâmetro de *timeout* na chamada da função *Task.await* no Mutante1. Tal mudança poderá simular falhas de finalização precoce, para chamadas de *Task.await* e problemas de intercalação de mensagens incorretas para chamadas de *Task.yield* e *Task.yield_many*.

3.4.4 OPERADORES DE MUTAÇÃO DE ADIÇÃO DE PARÂMETRO EM FUNÇÃO

3.4.4.1 ADDAFTERRECEIVE - ADIÇÃO DE PARÂMETRO NA FUNÇÃO DE RECEBIMENTO DE MENSAGENS NO KERNEL

Este operador tem como objetivo a adição do parâmetro de tempo limite (*after*) à função *receive*. Para cada ocorrência da função com a ausência do parâmetro de tempo limite, será criado um mutante com a mesma chamada da função, no entanto, com a presença de um tempo limite. Para tempo limite será utilizado 5 segundos sendo que este é o tempo padrão utilizando nas funções de recebimento do módulo Task como *Task.await*, *Task.yield* e *Task.yield_many*. O Algoritmo 25 apresenta um exemplo de aplicação do operador.

Pode-se perceber a presença do parâmetro de tempo limite (*after*) na chamada da função *receive* no Mutante1. Tal mudança fará com que a mensagem nunca seja recebida já que a função *enviar* espera 10 segundos para enviar a mensagem, enquanto o parâmetro *after* inserido no *receive* é acionado após 5 segundos. Aplicando-se o operador, pode-se obter falhas de finalização precoce devido ao estouto do tempo limite configurado no mutante.

3.4.5 OPERADORES DE MUTAÇÃO DE TROCA DE PARÂMETRO EM FUNÇÃO

3.4.5.1 REPLMODULESPAWN - TROCA DE PARÂMETRO NAS FUNÇÕES DE CRIAÇÃO DE PROCESSOS NO KERNEL

Este operador tem como objetivo a substituição do primeiro parâmetro de módulo nas funções de criação de processos no Kernel, pela palavra chave $\_\_MODULE\_\_$ que em Elixir compreende ao módulo atual. Para cada ocorrência da função com aridade três, será criado um mutante na qual o primeiro parâmetro da função será a palava $\_\_MODULE\_\_$. No Algoritmo 26 é apresentado um exemplo de aplicação deste operador.

Pode-se notar a presença da palavra $\_\_MODULE\_\_$ como primeiro parâmetro da função *spawn* no Mutante1. Essa mudança de parâmetro fará com que o resultado seja a soma dos dois números e não a subtração que era esperada. Tal mutante pode produzir

25 Exemplo de aplicação do operador *AddAfterReceive*

```
Original: def enviar(destino) do
             Process.sleep(10000)
             send(destino, :ok)
          end

          def principal() do
             processo_atual = self()
             spawn(__MODULE__, :enviar, [processo_atual])

             receive do
                :ok -> IO.puts "Recebeu mensagem"
             end
          end

Mutante:  def enviar(destino) do
             Process.sleep(10000)
             send(destino, :ok)
          end

          def principal() do
             processo_atual = self()
             spawn(__MODULE__, :enviar, [processo_atual])

             receive do
                :ok -> IO.puts "Recebeu mensagem"
             after
                5000 -> Process.exit(self(), :kill)
             end
          end
```

26 Exemplo de aplicação do operador *ReplModuleSpawn*

```
Original: defmodule App do
             def run(a, b) do
                spawn(Subtracao, calculo, [a, b])
             end

             def calculo(a, b) do
                IO.puts a + b
             end
          end

          defmodule Subtracao do
             def calculo(a, b) do
                IO.puts a - b
             end
          end

Mutante1: defmodule App do
             def run(a, b) do
                spawn(__MODULE__, calculo, [a, b])
             end

             def calculo(a, b) do
                IO.puts a + b
             end
          end

          defmodule Subtracao do
             def calculo(a, b) do
                IO.puts a - b
             end
          end
```

dados incorretos quando o módulo atual de chamada da função possuir uma função com mesmo nome a qual está sendo executada.

3.4.5.2 REPLMODULETASKCREATE - TROCA DE PARÂMETRO NAS FUNÇÕES DE CRIAÇÃO DE TAREFAS NO MÓDULO TASK

Este operador tem como objetivo a substituição do primeiro parâmetro de módulo nas funções de criação de tarefas no módulo Task - *Task.async*, *Task.start* e *Task.start_link* - pela palavra chave $\_\_MODULE\_\_$ que em Elixir compreende ao módulo atual. Para cada ocorrência da função com aridade três, será criado um mutante na qual o primeiro parâmetro da função será a palava $\_\_MODULE\_\_$. O Algoritmo 27 apresenta um exemplo de aplicação do operador.

Pode-se notar a presença da palavra $\_\_MODULE\_\_$ como primeiro parâmetro da função *Task.async* no Mutante1. Essa mudança irá produzir como resultado da execução do programa, o valor de uma soma sendo que o esperado era o valor da subtração. Tal mutante pode produzir dados incorretos quando o módulo atual de chamada da função possuir uma função com mesmo nome a qual está sendo executada.

3.4.6 COMPARAÇÃO COM OPERADORES DE OUTRAS LINGUAGENS

Os operadores de mutação definidos neste trabalho foram comparados com os operadores de mutação definidos para MPI (SILVA; SOUZA; SOUZA, 2012) e PVM (GIACOMETTI; SOUZA; SOUZA, 2003). Essa comparação foi possível pela característica comum das três linguagens (Elixir, MPI e PVM) em utilizar o paradigma de passagem de mensagens na comunicação entre os processos. A Tabela 16 apresenta a comparação dos operadores em Elixir com os operadores em MPI e PVM.

No total, quatro operadores definidos para Elixir tem semelhança com operadores definidos em MPI e PVM. O operador *ReplTaskAnswer* possui semelhança com o operador *ReplAwait*, no qual ambos funcionam como receptores de mensagens de tarefas. *DelSend*, *DelReceive* e *DelShutdown* possuem versões correspondentes em MPI e PVM. *DelSend* é semelhante aos operadores *DelSend* e *DelSendPVM* de MPI e PVM respectivamente. Os três operadores exploram a deleção da função de envio de mensagens entre os processos. *DelReceive* é semelhante aos operadores *DelRecv* e *DelRcvPVM*, que exploram a deleção da função de recebimento de mensagens. *DelShutdown* é o operador que explora a deleção da função de finalização de tarefas e, portanto, pode ser comparado com os operadores *DelFinTask* para MPI e PVM, que também apresentam o mesmo comportamento.

27 Exemplo de aplicação do operador *ReplModuleTaskCreate*

```
Original: defmodule App do

             use Task

             def run(a, b) do
                tarefa = Task.async(Subtracao, calculo, [a, b])
                IO.puts Task.await(tarefa)
             end

                 def calculo(a, b) do
                    a + b
                 end
              end

              defmodule Subtracao do
                 def calculo(a, b) do
                    IO.puts a - b
                 end
              end

Mutante1: defmodule App do

             use Task

             def run(a, b) do
                tarefa = Task.async(__MODULE__, calculo, [a, b])
                IO.puts Task.await(tarefa)
             end

             def calculo(a, b) do
                IO.puts a + b
             end
          end

          defmodule Subtracao do
             def calculo(a, b) do
                IO.puts a - b
             end
          end
```

Tabela 16: Comparação dos operadores em Elixir com os operadores em MPI e PVM

Operadores Elixir	Operadores MPI	Operadores PVM
ReplSpawn	-	-
ReplTaskStart	-	-
ReplTaskAnswer	ReplAwait	-
DelSpawn	-	-
DelSend	DelSend	DelSndPVM
DelReceive	DelRecv	DelRcvPVM
DelTaskStart	-	-
DelTaskAnswer	-	-
DelShutdown	DelFinTask	DelFinTask
DelChildSpec	-	-
DelParameterReceive	-	-
DelTimeoutTaskAnswer	-	-
AddAfterReceive	-	-
ReplModuleSpawn	-	-
ReplModuleTaskCreate	-	-

Fonte: Autoria própria.

3.5 CONSIDERAÇÕES FINAIS

Neste capítulo foram abordadas diferentes etapas para aplicação do teste de mutação a programas concorrentes em Elixir. Primeiramente foi apresentado o *benchmark* construído para suporte a atividade de teste, o qual, composto por onze programas, abrange todas as funções concorrentes do *Kernel* e do módulo *Task*. Em seguida, foi apresentada a definição da taxonomia de falhas, desenvolvida a partir de cinco etapas: (1) Identificação das funções concorrentes, (2) Agrupamento de funções com semântica semelhante, (3) Identificação dos enganos, (4) Definição de defeitos e (5) Execução e coleta de falhas. Por fim, foram apresentados os operadores de mutação definidos, apresentando a definição e uma simples aplicação para cada operador.

4 EXPERIMENTO

Neste capítulo é apresentado o experimento realizado para avaliar os operadores de mutação para programas concorrentes em Elixir definidos neste trabalho.

4.1 INTRODUÇÃO AO PROBLEMA

Este experimento busca validar a qualidade dos operadores de mutação definidos na Seção 3.4 do Capítulo 3. A linguagem Elixir não apresenta nenhum critério ou ferramenta de apoio ao teste de programas concorrentes e, devida essa carência, operadores de mutação foram definidos.

Um experimento foi definido para validar características de qualidade dos operadores, pois se tinha controle sobre os operadores de mutação definidos. Este experimento tem como objetivo avaliar algumas características importantes na qualificação de operadores de mutação como o número de mutantes gerados, a completude na abrangência de falhas em relação à taxonomia e, a taxa de inclusão dentre os operadores de deleção de função.

4.2 CARACTERIZAÇÃO DO ESTUDO

O experimento buscará qualificar os operadores de mutação definidos neste trabalho, avaliando quais operadores geram maior número de mutantes e quais contemplam mais categorias presentes na taxonomia de falhas. Entende-se que um melhor operador de mutação gera mutantes mais difíceis de serem mortos, aumentando a qualidade do conjunto de casos de teste. O experimento também buscará verificar a taxa de inclusão entre os operadores de mutação de deleção.

4.3 DEFINIÇÃO DOS OBJETIVOS

A realização do experimento baseou-se no paradigma Goal/Question/Metric (GQM) (CALDIERA; ROMBACH, 1994), sendo assim organizado em cinco partes (1) Objeto de estudo, (2) Propósito, (3) Perspectiva, (4), Foco qualitativo e, (5) Contexto.

- **Objetos de estudo:** Os objetos de estudo são os operadores de mutação para programas concorrentes em Elixir.
- **Propósito:** Avaliar variáveis independentes importantes na qualificação de operadores de mutação, verificando qual operador gera mais mutantes, qual operador demonstra mais completude em relação à taxonomia e o percentual de inclusão dentre os operadores de mutação de deleção.
- **Perspectiva:** Este experimento é executado do ponto de vista de um testador.
- **Foco Qualitativo:** O foco do experimento é validar a qualidade dos operadores de mutação.
- **Contexto:** Este experimento foi realizado utilizando três elementos: (1) Treze operadores de mutação para programação concorrente em Elixir, (2) um *benchmark* de onze programas e, (3) a taxonomia de falhas em Elixir.

O experimento pode ser sumarizado pelo template (WOHLIN et al., 2000) abaixo:

Analisar	.....	operadores de mutação
Com o propósito de	.....	avaliar
Com respeito à	.....	completude à taxonomia de falhas, quantidade de mutantes gerados e percentual de inclusão dos operadores de deleção de função
Do ponto de vista do	.....	testador

Fonte: Autoria própria.

4.4 DESIGN DO EXPERIMENTO

4.4.1 QUESTÕES

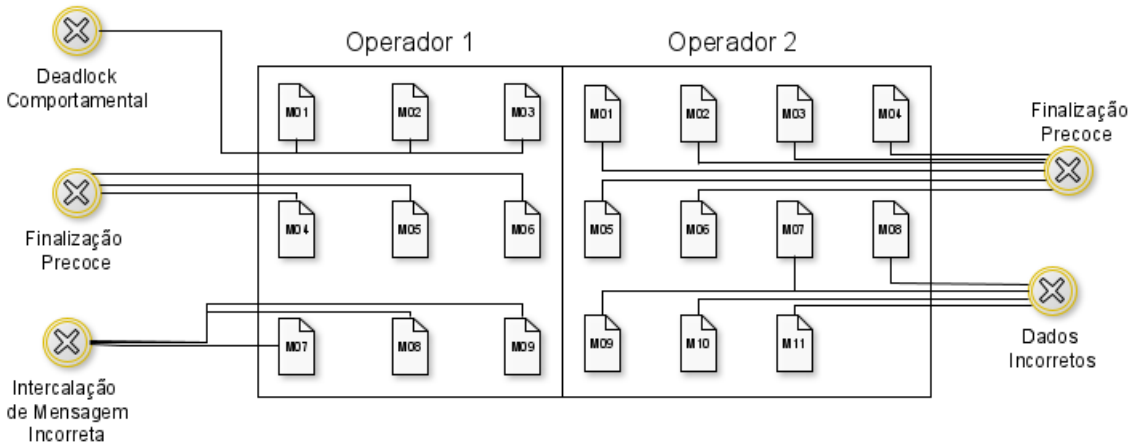
Neste experimento, foram definidas quatro questões de pesquisa para avaliar a qualidade dos operadores de mutação definidos, bem como se são efetivos para a aplicação do teste de mutação. As quatro questões são descritas abaixo:

1. **QP1)** Qual operador de mutação gera mais mutantes?
2. **QP2)** Qual operador de mutação demonstra mais completude em relação à taxonomia de falhas?
3. **QP3)** Qual operador de mutação de deleção de função apresenta a maior taxa de inclusão em relação ao Operador de Mutação de Deleção de função de Criação de Processos no Módulo Kernel (DelSpawn)?
4. **QP4)** Qual operador de mutação de deleção de função apresenta a maior taxa de inclusão em relação ao Operador de Mutação de Deleção de função de Criação de Tarefas no Módulo Task (DelTaskStart)?

A primeira questão explorada no experimento é o número de mutantes gerados por cada operador. Geralmente, operadores de mutação geram um grande número de mutantes e esse é o principal problema do teste de mutação por exigir grande demanda computacional na execução dos mutantes. Através do experimento, será possível conhecer qual operador gera o maior número de mutantes, embora essa medida não garanta que um operador é o melhor ou o pior.

A Figura 8 apresenta uma visão alto nível do fenômeno que procura-se observar nas questões 1 e 2 desse experimento. Na figura, embora o Operador 1 produza um menor número de mutantes (9 mutantes em relação aos 11 produzidos pelo Operador 2) e com diferentes tipos de falhas, o Operador 2 gera uma falha exclusiva (Dados Incorretos).

Figura 8: Questões de Pesquisa 1 e 2



Fonte: Autoria Própria.

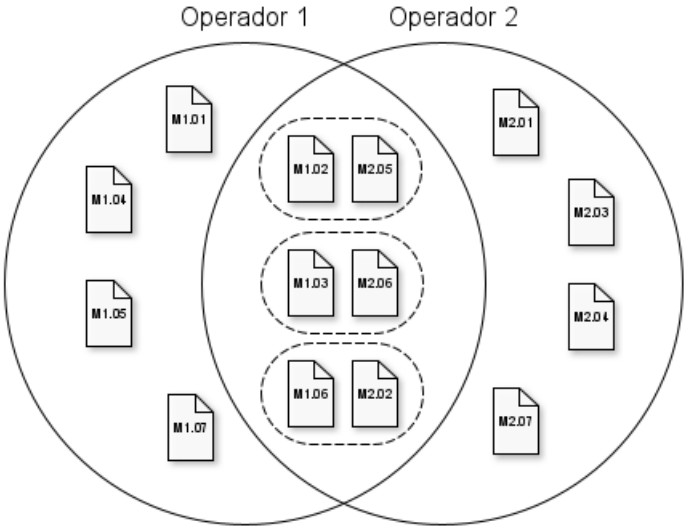
A segunda questão explorada no experimento é a completude na abrangência de falhas de cada operador. Ao utilizar um operador de mutação, espera-se que este seja capaz de produzir diferentes tipos de falhas. A taxonomia de falhas em Elixir, apresentada na Seção 3.3, será utilizada na condução do experimento. É importante destacar que uma maior abrangência das falhas não garante que determinado operador de mutação é o melhor para ser utilizado. Na Figura 8, embora o Operador 01 produza um maior número de falhas (três falhas em relação as duas produzidas pelo Operador 2), o Operador 2 consegue gerar uma falha exclusiva (Dados Incorretos).

Além dessas duas características, o experimento ainda fará uma análise interna da categoria Operadores de Mutação de Deleção de Função. Esta análise buscará descobrir a taxa de inclusão dentre os operadores da categoria. A taxa de inclusão dentre dois operadores de uma mesma categoria auxilia a identificar operadores que produzem mutantes semelhantes, falhas iguais e pode ajudar o testador a escolher um operador de mutação que inclui outros operadores e não gerar mutantes semanticamente semelhantes. A Figura 9 apresenta um exemplo da geração de mutantes de dois operadores, Operador 1 e Operador 2. Na figura, o tracejado representa mutantes que com sintática parecida e que produziram mesma falha. A taxa de inclusão do Operador 2 em relação ao Operador 1 é de aproximadamente 43 %, já que do total de sete mutantes produzidos pelo Operador 1, três mutantes do Operador 2 produziram mutantes semelhantes. No experimento serão avaliadas as taxas de inclusão dos operadores de mutação de deleção de função em relação aos operadores *DelSpawn* e *DelTaskStart*. Tais operadores foram escolhidos por produzirem a deleção de funções de criação de processos e tarefas.

4.5 SELEÇÃO DE VARIÁVEIS

- **Medida necessária QP1:** Número de mutantes gerados por cada operador de mutação, a partir da aplicação dos operadores de mutação no *benchmark*.
- **Medida necessária QP2:** Taxa de abrangência de falhas de cada operador de mutação, a partir da execução não-determinística dos mutantes gerados na QP1 e a taxonomia de falhas.
- **Medida necessária QP3:** Taxa de inclusão de cada operador de mutação de deleção de função com relação ao Operador de Mutação de Deleção de Função de Criação de Processos no Módulo Kernel (DelSpawn).
- **Medida necessária QP4:** Taxa de inclusão de cada operador de mutação de

Figura 9: Questões de Pesquisa 3 e 4



Fonte: Autoria Própria.

deleção de função com relação ao Operador de Mutação de Deleção de Função de Criação de Tarefas no Módulo Task (DelTaskStart).

4.6 OBJETOS DO EXPERIMENTO

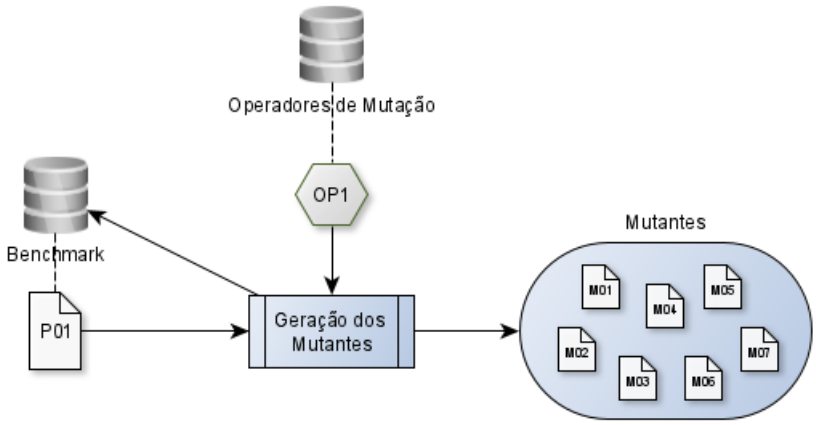
O objeto desse experimento é um *benchmark* de programas concorrentes em Elixir apresentado na Seção 3.2. O *Benchmark* definido conta com onze programas concorrentes e abrange em sua totalidade as funções concorrentes do *Kernel* e do módulo *Task*.

4.7 CONDUÇÃO DO EXPERIMENTO

A condução do experimento foi dividida em três etapas, sendo que cada etapa representa os passos necessários para execução de cada parte do experimento.

1. **Geração de Mutantes:** Nesta etapa, cada operador de mutação será selecionado em conjunto com o *benchmark* de programas concorrentes. Em seguida, para cada programa, será aplicado o operador gerando o máximo de mutantes distintos sintaticamente possíveis. A Figura 10 ilustra o fluxo de trabalho desta etapa na qual, para cada programa (P) no *benchmark*, será aplicado cada um dos operadores de mutação presentes no conjunto de operadores de mutação, gerando um conjunto de mutantes.

Figura 10: Condução do Experimento: Geração dos Mutantes



Fonte: Autoria Própria.

A partir desta primeira etapa, será possível analisar qual operador de mutação gera o maior número de mutantes com base no *benchmark*.

2. **Associação entre os Operadores de Mutação e a Taxonomia de Falhas:** Para melhor descrever esta etapa, foi elaborado um roteiro com três atividades: (1) Execução não-determinística, (2) Análise e, (3) Abrangência. A Figura 11 ilustra a aplicação desta etapa do experimento.

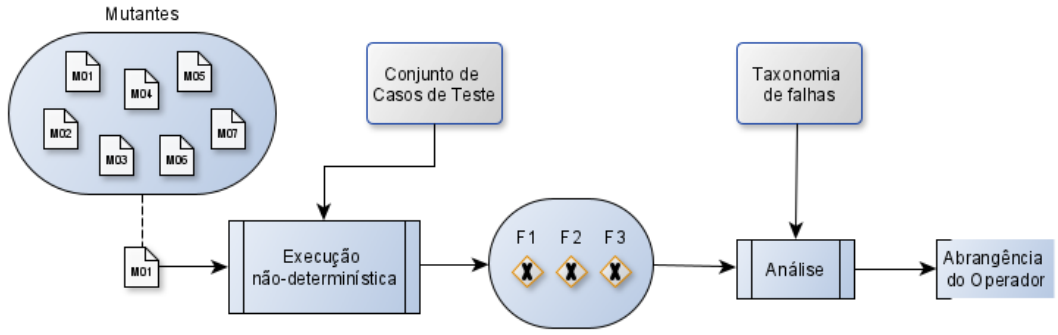
   (a) Execução não-determinística: Nesta atividade, cada mutante produzido anteriormente será selecionado em conjunto com seu programa original, para em seguida serem executados de maneira não-determinística, a fim de simular e forçar possíveis falhas.

   A execução não-determinística não levará em consideração a ordem de execução dos processos nos programas. Para cada programa e seus respectivos mutantes, será aplicado o conjunto de casos de teste inicial. Embora não seja ideal a aplicação desta técnica, não há nenhuma ferramenta de suporte a execução determinística dos programas em Elixir

   (b) Análise: Nesta atividade, serão associadas as falhas obtidas na atividade anterior com a taxonomia de falhas desenvolvida no Capítulo 3. Serão analisados o número de falhas que cada operador gerou em relação ao número total de falhas que ele poderia gerar e o número total de falhas presentes na taxonomia.

   (c) Abrangência: A partir dos dados obtidos na atividade anterior, serão calculadas as taxas e elaborada a tabela de resultados.

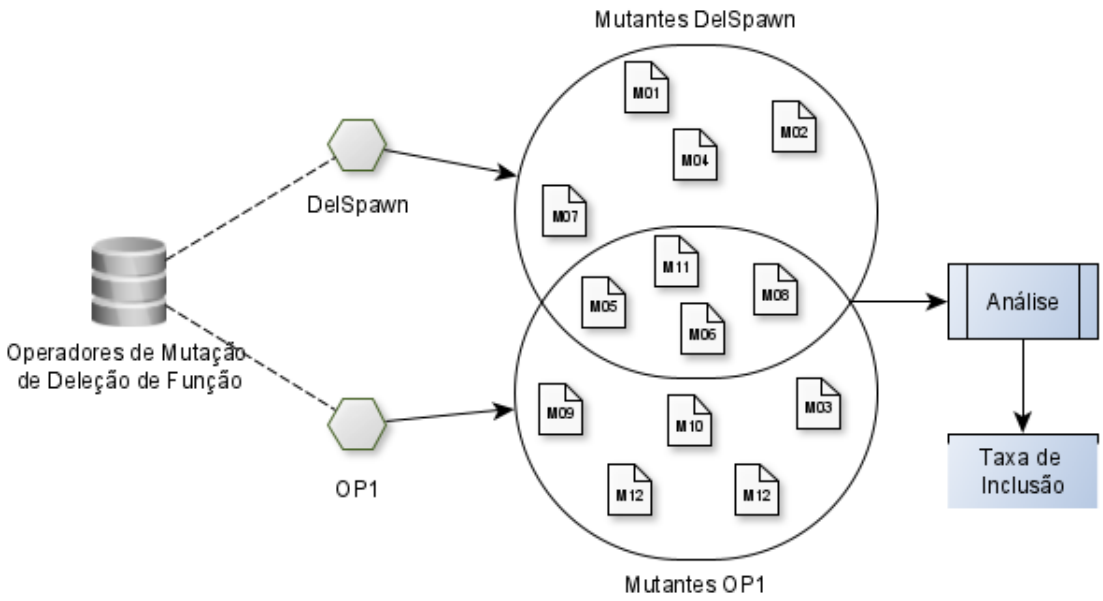
Figura 11: Condução do Experimento: Análise da completude de abrangência de cada operador



Fonte: Autoria Própria.

3. **Análise da taxa de inclusão dentre os operadores de mutação de deleção:** Nesta etapa, serão analisadas as taxas de inclusão dos operadores de deleção de função em relação aos operadores *DelSpawn* e *DelTaskStart*. As Figuras 12 e 13 ilustram a aplicação desta etapa.

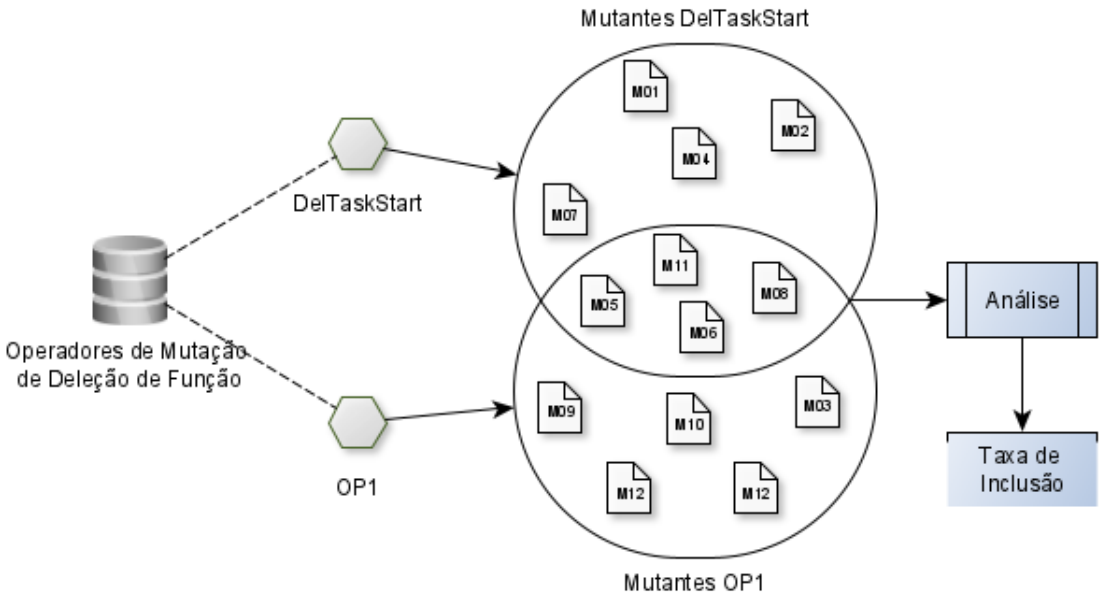
Figura 12: Condução do Experimento: Análise da taxa de inclusão em relação ao Operador *DelSpawn*



Fonte: Autoria Própria.

Primeiramente, serão selecionados os mutantes gerados pelo operador *DelSpawn* obtidos na Etapa 1. Em seguida, para cada operador de deleção de função, os mutantes gerados serão comparados semanticamente. Por exemplo, a deleção de uma

Figura 13: Condução do Experimento: Análise da taxa de inclusão em relação ao Operador *DelTaskStart*



Fonte: Autoria Própria.

função *spawn*, através do operador *DelSpawn*, que continha dentro dela a chamada de uma função *send*, pode produzir um mutante semelhante ao mutante produzido pelo operador *DelSend*, já que em ambos, a função *send* não é executada. Analisando e comparando as falhas geradas, será possível identificar o número de mutantes produzidos pelo operador já pertencentes ao conjunto de mutantes produzidos pelo operador *DelSpawn*. Tal processo será aplicado da mesma forma para o operador *DelTaskStart* e, após as duas análises, será calculada a taxa de inclusão de cada operador.

4.8 COLETA E ANÁLISE DOS DADOS

Após a realização do experimento seguindo as etapas definidas, foram obtidos dados que devem ser interpretados para responder as quatro questões de pesquisa.

- **QP1)** Qual operador de mutação gera mais mutantes?

A Tabela 17 apresenta o número total de mutantes gerados de cada operador. Nessa tabela é possível ver que o operador *ReplSpawn* produziu a maior quantidade de mutantes, sendo doze no total. Essa tendência no operador *ReplSpawn*, em gerar um maior

número de mutantes, é gerada pela possibilidade do operador, em cada função *spawn*, *spawn_ link* e *spawn_ monitor*, gerar dois mutantes. O Operador *DelParameterReceive* também pode gerar um grande número de mutantes, já que para cada *match* de cada função *receive* é criado um mutante.

Tabela 17: Números de mutantes gerados por operador

Operador	**Nº Mutantes Produzidos**	**Programas Utilizados**
ReplSpawn	12	01; 04; 09; 10; 11
ReplTaskStart	5	01; 03; 05
ReplTaskAnswer	2	07; 08
DelSpawn	9	01; 04; 09; 10; 11
DelSend	9	04; 09; 10; 11
DelReceive	9	04; 09; 10; 11
DelTaskStart	4	01; 03; 05
DelTaskAnswer	1	07
DelShutdown	2	02; 07
DelChildSpec	0	
DelParameterReceive	11	04; 09; 10; 11
DelTimeoutTaskAnswer	2	02; 07
AddAfterReceive	9	04; 09; 10; 11
ReplModuleSpawn	6	09
ReplModuleTaskCreate	0	

Fonte: Autoria própria.

- **QP2)** Qual operador de mutação demonstra mais completude em relação à taxonomia de falhas?

A Tabela 18 apresenta os resultados da execução dos mutantes do *benchmark* produzidos na QP1. Três operadores atingiram 100 % de abrangência das falhas que ele era capaz de produzir, sendo eles, *DelSpawn*, *DelReceive* e *AddAfterReceive*. Já em relação as falhas possíveis da taxonomia, o operador *DelReceive* foi o que demonstrou maior completude, atingindo aproximadamente 57 % de abrangência.

Tabela 18: Completude de abrangência dos operadores

Operador	Falhas Obtidas	Falhas possíveis operador		Falhas possíveis taxonomia	
		Falhas	Abrang.	Falhas	Abrang.
ReplSpawn	FP	DC; FP; ME	33,33%	AE; DC; DI; FP; FCE; IMI; ME	14,29%
ReplTaskStart	DC	AE; DC; DI; FP; FCE	20%	AE; DC; DI; FP; FCE; IMI; ME	14,29%
ReplTaskAnswer	DI	AE; DC; DI; FP; FCE	20%	AE; DC; DI; FP; FCE; IMI; ME	14,29%
DelSpawn	DI; DC; FP	DI; DC; FP	100%	AE; DC; DI; FP; FCE; IMI; ME	42,86%
DelSend	DC; IMI	DC; FP; IMI	66,67%	AE; DC; DI; FP; FCE; IMI; ME	28,57%
DelReceive	AE; DC; FP; IMI	AE; DC; FP; IMI	100%	AE; DC; DI; FP; FCE; IMI; ME	57,14%
DelTaskStart	FP	DI; FP; FCE; IMI	25%	AE; DC; DI; FP; FCE; IMI; ME	14,29%
DelTaskAnswer	DI	DI; FP; FCE; IMI	25%	AE; DC; DI; FP; FCE; IMI; ME	14,29%
DelShutdown	DI	DI; FP; FCE; IMI	25%	AE; DC; DI; FP; FCE; IMI; ME	14,29%
DelChildSpec		DI; FP; FCE; IMI	0%	AE; DC; DI; FP; FCE; IMI; ME	0%
DelParameterReceive	DC; IMI	AE; DC; IMI	66,67%	AE; DC; DI; FP; FCE; IMI; ME	28,57%
DelTimeoutTaskAnswer		FP; FCE; IMI	0%	AE; DC; DI; FP; FCE; IMI; ME	0%
AddAfterReceive	FP	FP	100%	AE; DC; DI; FP; FCE; IMI; ME	14,29%
ReplModuleSpawn		DI	0%	AE; DC; DI; FP; FCE; IMI; ME	0%
ReplModuleTaskCreate		DI	0%	AE; DC; DI; FP; FCE; IMI; ME	0%

- **QP3)** Qual operador de mutação de deleção de função apresenta a maior taxa de inclusão em relação ao Operador *DelSpawn*?

A Tabela 19 apresenta a taxa de inclusão dos operadores de mutação de deleção de função em relação ao operador *DelSpawn*. Somente o operador *DelSend* produziu mutantes semelhantes aos do Operador *DelSpawn*, sendo dois ao total. Esses dois mutantes tiveram uma sintaxe muito semelhante e produziram a mesma falha. Como somente o operador *DelSend* produziu mutantes com comportamento semelhante aos mutantes do operador *DelSpawn*, o mesmo é considerado o operador de deleção de função com maior taxa de inclusão em relação ao operador *DelSpawn*.

Tabela 19: Taxa de inclusão dos Operadores de Mutação de Deleção de Função em relação ao Operador DelSpawn

Operador	**Nº Mutantes DelSpawn**	**Nº Mutantes Operador**	**Nº Mutantes Inclusos**	**% Abrangência**
DelSend	9	9	2	22,22%
DelReceive	9	9	0	0%
DelTaskStart	9	4	0	0%
DelTaskAnswer	9	1	0	0%
DelShutdown	9	2	0	0%
DelChildSpec	9	0	0	0%

Fonte: Autoria própria.

- **QP4)** Qual operador de mutação de deleção de função apresenta a maior taxa de inclusão em relação ao Operador *DelTaskStart*?

A Tabela 20 apresenta a taxa de inclusão dos operadores de mutação de deleção de função em relação ao operador *DelTaskStart*. Nenhum mutante produzido pelos operadores de deleção de função estava incluso nos mutantes produzidos pelo operador *DelTaskStart*, dessa forma, todos os operadores apresentaram a mesma taxa de inclusão.

Além das quatro questões de pesquisa acima respondidas, foi calculado o *score* de mutação dos porgramas do *benchmark*. A Tabela 21 apresenta o *score* de mutação dos programas do *benchmark*. O *score* de mutação dos programas mostra a qualidade e adequação dos casos de teste, já que os mesmos foram capazes de matar todos os mutantes. Na tabela é possível observar os mutantes equivalentes de cada programa que foram identificados manualmente.

Tabela 20: Taxa de inclusão dos Operadores de Mutação de Deleção de Função em relação ao Operador DelTaskStart

Operador	Nº Mutantes DelSpawn	Nº Mutantes Operador	Nº Mutantes Inclusos	% Abrangência
DelSpawn	4	9	0	0%
DelSend	4	9	0	0%
DelReceive	4	9	0	0%
DelTaskAnswer	4	1	0	0%
DelShutdown	4	2	0	0%
DelChildSpec	4	0	0	0%

Fonte: Autoria própria.

Tabela 21: *Score* de mutação dos programas do benchmark

Programa	Nº Mutantes Produzidos	Nº Mutantes Mortos	Nº Mutantes Equivalentes	Score de Mutação
Exemplo Spawn_Monitor	3	2	1	100%
Yield_Many	2	1	1	100%
Elixir Study	4	4	0	100%
Events	7	5	2	100%
Agents and Tasks in Elixir	4	2	2	100%
Synchronous Task Stream	0	-	-	-
Pangram	4	2	2	100%
Parallel Letter Frequency	1	1	0	100%
17-dining-philosophers	30	17	13	100%
Parallel Letter Frequency	12	12	0	100%
Elixir Sorting	14	10	4	100%

Fonte: Autoria própria.

4.9 AMEAÇAS À VALIDADE

Após a realização de um experimento, torna-se necessário a validação dos resultados, isto é, a garantia que os resultados obtidos possam ser considerados válidos para a população do interesse (neste caso, a comunidade acadêmica e científica) (WOHLIN et al., 2000). Wohlin et al. definiram quatro ameaças diferentes à validade dos estudos empíricos: construção, interno, externo e de conclusão. Neste experimento, foram identificadas ameaças de construção, interna e externa.

A validade de construção diz respeito à relação entre a teoria e o que é observado (WOHLIN et al., 2000). Na construção dos casos de teste dos programas do *benchmark*, equívocos podem ter sido cometidos, e casos de teste úteis na detecção de uma falha podem ter sido esquecidos. Para mitigar esse risco foi utilizada a técnica de partição de equivalência para encontrar valores válidos e inválidos para os programas, através de análise do código-fonte dos programas e conhecimento obtido da linguagem.

As ameaças a validade interna validam a relação entre o tratamento e a saída, isto é, se há confiabilidade entre os resultados esperados e os resultados obtidos. A execução não-determinística dos programas e dos mutantes é uma ameaça relacionada neste contexto, já que, o não determinismo pode estar presentes no programas. Para mitigar este risco calculou-se o *score* de mutação dos programas (Tabela 21), demonstrando que a execução não-determinística foi capaz de detectar todos os mutantes.

As ameaças a validade externa estão relacionadas à generalização dos resultados (WOHLIN et al., 2000). Uma ameaça neste experimento é a representatividade dos programas do *benchmark*. Para mitigar esse risco, buscou-se construir o *benchmark* com programas de diversas fontes e de diversos tamanhos. Além disso, no *benchmark* procurou-se abrangir a totalidade das funções concorrentes do *Kernel* e do módulo *Task.*.

4.10 CONSIDERAÇÕES FINAIS

Este capítulo apresentou um experimento para validar características úteis na qualificação de operadores de mutação. O experimento buscou responder quatro questões de pesquisa onde o Operador *ReplSpawn* demonstrou maior potencialidade na criação de mutantes enquanto o operador *DelReceive* demonstrou maior completude na abrangência de falhas da taxonomia de falhas. Também foi visto que o operador *DelSend* apresenta maior taxa de inclusão em relação ao operador *DelSpawn*, enquanto que para o operador *DelTaskStart*, todos os demais operadores da categoria apresentaram a mesma taxa. O experimento mostra algumas características de qualidade dos operadores definidos e auxilia a validação dos mesmos para aplicação.

5 CONCLUSÃO

O teste de mutação é uma técnica de teste que vem se mostrando muito eficaz na detecção de erros e falhas em programas. A técnica explora enganos cometidos por programadores durante o desenvolvimento de software. A programação concorrente difere-se da programação convencional por adicionar características como a comunicação e a sincronização entre processos. Estas caraterísticas tornam o processo de teste de software mais complexo. Muitas linguagens de programação suportam à concorrência, dentre elas, destaca-se o Elixir, uma linguagem emergente que adiciona o paradigma de programação funcional à programação concorrente, resolvendo problemas de paralelismo e concorrência de forma moderna e dinâmica.

O presente trabalho apresentou a proposta de aplicação do teste de mutação à programas concorrentes em Elixir, visando obter um conjunto de operadores de mutação e gerar mutantes a partir de um *benchmark* pré-determinado para avaliar a qualidade dos operadores. Primeiramente, foi definido um *benchmark* de programas concorrentes em Elixir, em sua maioria, representam soluções para problemas clássicos da programação concorrente. Os programas foram coletados de diversos repositórios e se mostram aptos ao teste de software

Após isso, foi realizada a definição da taxonomia de falhas em Elixir. Nesta etapa foram estudadas as funções concorrentes do *Kernel* e do módulo *Task* para em seguida agrupar funções com semântica semelhante. Feito isso, foram realizados inúmeros testes para simular e definir as falhas.

Em seguida, foi definido um conjunto de operadores de mutação para programas concorrentes baseando-se em enganos cometidos por programadores durante o desenvolvimento do software. Foram definidos quinze operadores de mutação, divididos em quatro grupos (Troca de função, Deleção de função, Deleção de parâmetro de função, Adição de parâmetro de função e Troca de parâmetro de função).

Por fim, foi aplicado um experimento com a finalidade de avaliar algumas

características dos operadores de mutação definidos. No experimento, revelou-se que o operador *ReplSpawn* gerou o maior número de mutantes, enquanto o operador *DelReceive* apresentou maior completude na abrangência das falhas da taxonomia. No experimento também calculou-se a taxa de inclusão dos operadores de mutação de deleção de função em relação aos operador *DelSpawn* e *DelTaskStart*.

5.1 CONTRIBUIÇÕES

Pode-se destacar como principais contribuições deste trabalho:

1. Instanciação de um processo a ser seguido para a aplicação do teste de mutação em programas concorrentes em Elxir.
2. Definição de um conjunto de operadores de mutação para programas concorrentes em Elixir.
3. Construção e validação de um *benchmark* de programas concorrentes em Elixir para suporte à atividade de teste de software.
4. A definição da taxonomia de falhas em Elixir.
5. Validação do operador de mutação que gera o maior número de mutantes.
6. Validação do operador de mutação com maior completude em relação à taxonomia de falhas.
7. Validação do operador de mutação com maior taxa de inclusão em relação ao operador *DelSpawn*.
8. Validação do operador de mutação com maior taxa de inclusão em relação ao operador *DelTaskStart*.

5.2 DIFICULDADES E LIMITAÇÕES

Durante a execução do trabalho, algumas dificuldades ocorreram. A primeira dificuldade foi relacionada ao aprendizado da linguagem Elixir, tanto por ser um primeiro contato com a linguagem, como pela utilização do paradigma funcional. Para resolver esse problema, foram pesquisados muitos livros e documentação sobre a linguagem. Outro problema relacionado a linguagem foi a ausência de um ambiente de desenvolvimento integrado (IDE) para Elixir.

Durante a criação do *benchmark* de programas concorrentes, houveram problemas na coleta de programas. A ideia inicial era utilizar programas que solucionassem os problemas clássicos da programação concorrente, porém, não foram encontrados programas que contemplassem todas as funções concorrentes do *Kernel* e do módulo *Task*. Com isso, foi necessário adotar outros programas que utilizassem as funções.

Outro problema ocorreu durante a definição da taxonomia de falhas. Nenhuma das falhas obtidas tinha documentação adequada por parte da linguagem, sendo que as falhas relacionadas a programação concorrente nem são disponibilizadas. Para contornar esse problema, foi estudado o código e criados dados de teste que forçassem a falha a ocorrer.

Durante a execução dos mutantes foi encontrada dificuldade. A linguagem não conta com nenhuma ferramenta para suporte a execução determinística dos programas, assim optou-se por executá-los de forma não-determinística.

5.3 TRABALHOS FUTUROS

Como trabalho futuro, seria importante a implementação de uma ferramenta para geração dos mutantes utilizando os operadores de mutação definidos. Um segundo ponto, está em desenvolver operadores de mutação para os demais módulos da linguagem como *Process*, *Agent*, *Task.Supervisor* e *Gen-Server*.

Além disso, seria interessante desenvolver um procedimento para a análise dos mutantes, levando em consideração a característica não-determinística dos programas.

REFERÊNCIAS

AGRAWAL, H. et al. Design of mutant operators for the c programming language. 10 1999.

ALBUQUERQUE, A.; CAIXINHA, D. **Mastering Elixir**. [S.l.]: Packt Publishing, 2018.

ALMEIDA, U. **Learn functional programming with elixir: New Foundations for a New World.** [S.l.]: Pragmatic Bookshelf, 2018. 200 p.

AMBONI, L. **Elixir Study**. 2017. Disponível em: `https://github.com/luizamboni/elixir-study/blob/master/otp/5-task/module-task.exs`. Acesso em: 17 Jul. 2019.

ANDREWS, G. R. **Concurrent programming: principles and practice**. [S.l.]: Benjamin/Cummings Publishing Company San Francisco, 1991.

ARMSTRONG, J. **Programming Erlang: software for a concurrent world**. [S.l.]: Pragmatic Bookshelf, 2013.

BEIZER, B. **Software Testing Techniques (2Nd Ed.)**. New York, NY, USA: Van Nostrand Reinhold Co., 1990. ISBN 0-442-20672-0.

BORDIGNON, M. D.; SILVA, R. A. **Benchmark de programas concorrentes em Elixir para suporte à validação de critérios e ferramentas de teste**. Dois Vizinhos, PR: UTFPR, 2019.

BRADBURY, J. S.; CORDY, J. R.; DINGEL, J. Mutation operators for concurrent java (j2se 5.0). In: IEEE. **Second Workshop on Mutation Analysis (Mutation 2006-ISSRE Workshops 2006)**. [S.l.], 2006. p. 11–11.

CALDIERA, V. R. B.-G.; ROMBACH, H. D. Goal question metric paradigm. **Encyclopedia of software engineering**, v. 1, p. 528–532, 1994.

CARVER, R. Mutation-based testing of concurrent programs. In: IEEE. **Proceedings of IEEE International Test Conference-(ITC)**. [S.l.], 1993. p. 845–853.

CHETTY, J. **Parallel Letter Frequency**. 2018. Disponível em: `https://exercism.io/tracks/elixir/exercises/parallel-letter-frequency/solutions/d823090d277942b9820411b258e77b5c`. Acesso em: 17 Jul. 2019.

CODE, R. **Events**. 2019. Disponível em: `https://rosettacode.org/wiki/Events#Elixir`. Acesso em: 13 Jul. 2019.

COSTA, D. **Elixir Findings: Synchronous Task Stream**. 2019. Disponível em: `https://medium.com/@dinojoaocosta/elixir-findings-asynchronous-task-streams-7f6336227ea`. Acesso em: 16 Jul. 2019.

DAVI, T. **Elixir: Do zero à concorrência**. [S.l.]: Casa do Código, 2017. 121 p.

DEFRANG, X. **Parallel Letter Frequency**. 2014. Disponível em: `https://github.com/xavier/exercism-assignments/tree/master/elixir/parallelletter-frequency`. Acesso em: 20 Jun. 2019.

DELAMARO, M.; JINO, M.; MALDONADO, J. **Introdução ao teste de software**. [S.l.]: Elsevier Brasil, 2017.

DELAMARO, M. E. **Introdução ao teste de software: técnicas e ferramentas - Conceitos básicos**. 2012. Disponível em: <http://disciplinas.stoa.usp.br/course/view.php?id=35>. Acesso em: 28/12/2016.

DEMILLO, R.; LIPTON, R.; SAYWARD, F. Hints on test data selection: Help for the practicing programmer. **Computer**, v. 11, p. 34 – 41, 05 1978.

DESOUZA, J. et al. Automated, scalable debugging of MPI programs with intel message checker. In: **Workshop on Software engineering for high performance computing system applications**. New York, NY, USA: ACM, 2005. p. 78–82.

DOURADO, G. et al. A suite of java message-passing benchmarks to support the validation of testing models, criteria and tools. **Procedia Computer Science**, v. 80, p. 2226–2230, 12 2016.

Eytani, Y.; Ur, S. Compiling a benchmark of documented multi-threaded bugs. In: **18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.** [S.l.: s.n.], 2004. p. 266–.

Farchi, E.; Nir, Y.; Ur, S. Concurrent bug patterns and how to test them. In: **Proceedings International Parallel and Distributed Processing Symposium**. [S.l.: s.n.], 2003.

FLYNN, M. J. Some computer organizations and their effectiveness. **IEEE transactions on computers**, IEEE, v. 100, n. 9, p. 948–960, 1972.

GIACOMETTI, C.; SOUZA, S.; SOUZA, P. Teste de mutação para a validação de aplicações concorrentes usando pvm. **REIC, Eletrônica de Iniciação científica**, v. 2, 2003.

GOTTLIEB, A.; ALMASI, G. **Highly parallel computing**. [S.l.]: Benjamin/Cummings Redwood City, CA, 1989.

GRAMA, A. et al. **Introduction to parallel computing**. [S.l.]: Pearson Education, 2003.

GROPP, W. D. et al. **Using MPI: portable parallel programming with the message-passing interface**. [S.l.]: MIT press, 1999.

HUDAK, P. Conception, evolution, and application of functional programming languages. **ACM Computing Surveys (CSUR)**, ACM, v. 21, n. 3, p. 359–411, 1989.

KEMP, Z. **Pangram soluction**. 2018. Disponível em: `https://exercism.io/tracks/elixir/exercises/pangram/solutions/2f9b102f8612419eb21df0aa56a9519c`. Acesso em: 17 Jul. 2019.

KRAMMER, B. et al. MARMOT: An MPI analysis and checking tool. In: **Parallel Computing**. [S.l.: s.n.], 2003.

LOPEZ, C. T. et al. A study of concurrency bugs and advanced development support for actor-based programs. In: _____. **Programming with Actors: State-of-the-Art and Research Perspectives**. Cham: Springer International Publishing, 2018. p. 155–185.

LU, S. et al. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In: **13th International Conference on Architectural Support for Programming Languages and Operating Systems**. New York, NY, USA: ACM, 2008. p. 329–339.

LUECKE, G. R. et al. MPI-CHECK: a tool for checking Fortran 90 MPI programs. **Concurrency and Computation: Practice and Experience**, v. 15, n. 2, p. 93–100, 2003.

MALDONADO, J. C. et al. Critérios potenciais usos: Uma contribuição ao teste estrutural de software. **(Publicação FEE)**, [sn], 1991.

MCCABE, T. J. A complexity measure. **IEEE Transactions on software Engineering**, IEEE, n. 4, p. 308–320, 1976.

MORGAN, T. **17-dining-philosophers**. 2015. Disponível em: `https://github.com/seven1m/30-days-of-elixir/blob/master/17-dining-philosophers.exs`. Acesso em: 13 Jul. 2019.

MYERS, G. J. et al. **The art of software testing**. [S.l.]: Wiley Online Library, 2004.

NETO, A. Introdução a teste de software. **Engenharia de Software Magazine**, v. 1, p. 54–59, 2007.

OFFUTT, A. J.; VOAS, J.; PAYNE, J. **Mutation operators for Ada**. [S.l.], 1996.

PEDERSEN, J. B. Classification of programming errors in parallel message passing systems. In: **CPA**. [S.l.: s.n.], 2006. p. 363–376.

PERRONE, P. **ExSorting**. 2018. Disponível em: `https://github.com/pedroperrone/elixirsorting`. Acesso em: 15 Jun. 2019.

PET-SI. **ELIXIR: uma linguagem de programação brasileira em sistemas distribuídos do mundo**. 2018. Disponível em: <http://www.each.usp.br/petsi/jornal/?p=2459>. Acesso em: 20 Jun. 2019.

POLO, J. **Mutante testing for Elixir**. 2017. Disponível em: <https://github.com/JordiPolo/mutation>. Acesso em: 15 jun. 2019.

PRESSMAN, R. S. **Engenharia de Software: Uma abordagem profissional**. 7^{th}. ed. New Work, USA: McGraw-Hill Education, 2011. 780 p.

RAPPS, S.; WEYUKER, E. J. Data flow analysis techniques for test data selection. In: IEEE COMPUTER SOCIETY PRESS. **Proceedings of the 6th international conference on Software engineering**. [S.l.], 1982. p. 272–278.

RAPPS, S.; WEYUKER, E. J. Selecting software test data using data flow information. **IEEE transactions on software engineering**, IEEE, n. 4, p. 367–375, 1985.

ROPER, M. **Software testing**. [S.l.]: McGraw-Hill, Inc., 1995.

RUNGTA, N.; MERCER, E. G. Clash of the titans: Tools and techniques for hunting bugs in concurrent programs. In: **Proceedings of the 7th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging**. New York, NY, USA: ACM, 2009. (PADTAD '09), p. 9:1–9:10. ISBN 978-1-60558-655-7.

SAMOFALOV, V. et al. Parallel computing: Current and future issues of high-end computing. **Proceedings of the International Conference**, 2005.

SEBESTA, R. W. **Concepts of programming languages**. [S.l.]: Boston: Pearson,, 2012.

SEN, A. Mutation operators for concurrent SystemC designs. In: **10th International Workshop on Microprocessor Test and Verification**. Washington, DC, USA: IEEE Computer Society, 2009. p. 27–31.

SILVA, R.; SOUZA, S.; SOUZA, P. Mutation testing for concurrent programs in mpi. In: **13th Latin American Test Workshop, Quito, Ecuador**. [S.l.: s.n.], 2012. p. 69–74.

SILVA, R. A. **Teste de mutação aplicado a programas concorrentes em MPI**. Tese (Doutorado) — Universidade de São Paulo, 2013.

SILVA, R. A.; SOUZA, S. d. R. S. d. Achievements, challenges and opportunities on mutation testing of concurrent programs. In: **Escola Regional de Engenharia de Software - ERES**. [S.l.]: UTFPR, 2018.

SIMMONDS, D. M. The programming paradigm evolution. **IEEE Computer**, v. 45, n. 6, p. 93–95, 2012.

SOMMERVILLE, I. **Software Engineering**. 10^{th}. ed. Scotland: Pearson Addison-Wesley, 2016. 816 p.

STALLINGS, W.; MIDORIKAWA, E. T. **Arquitetura e organização de computadores: projeto para o desempenho**. [S.l.]: Prentice Hall, 2002.

STEEN, M. V.; TANENBAUM, A. S. **Sistemas Distribuídos: princípios e paradigmas**. [S.l.]: São Paulo: Prentice Hall, 2009.

TANENBAUM, A. S.; FILHO, N. M. **Sistemas operacionais modernos**. [S.l.]: Prentice-Hall, 1995.

TATARINTSEV, V. **Agents and Tasks in Elixir**. 2017. Disponível em: `https://whatdidilearn.info/2017/12/24/agents-and-tasks-in-elixir.html`. Acesso em: 17 Jul. 2019.

TRACY, I. T. et al. Nondeterministic finite automata in hardware-the case of the levenshtein automaton. **Architectures and Systems for Big Data (ASBD), in conjunction with ISCA**, 2015.

VALIM, J. **ExUnit**. 2019. Disponível em: <https://hexdocs.pm/ex_unit/ExUnit.html>.

VETTER, J. S.; SUPINSKI, B. R. Dynamic software testing of mpi applications with umpire. In: **2000 ACM/IEEE conference on Supercomputing**. Washington, DC, USA: IEEE Computer Society, 2000.

WOHLIN, C. et al. **Experimentation in Software Engineering: An Introduction**. Norwell, MA, USA: Kluwer Academic Publishers, 2000. ISBN 0-7923-8682-5.

ZIEGELMAYER, F. **Task Yield_Many**. 2019. Disponível em: `https://hexdocs.pm/elixir/Task.html#yield_many/2`. Acesso em: 16 Jul. 2019.

APÊNDICE A – CONJUNTO DE CASOS DE TESTE PARA OS PROGRAMAS DO *BENCHMARK*

A.1 INTRODUÇÃO

O Seguinte documento contém os casos de teste desenvolvidos para os programas do *benchmark* de programas concorrentes em Elixir (BORDIGNON; SILVA, 2019). O conjunto de casos de teste foi construído a partir da técnica de particionamento de equivalência em conjunto com um estudo do código fonte dos programas, buscando valores válidos e inválidos.Para cada programa, é apresentada uma descrição do algoritmo, seguido pelas entradas definidas e os resultados esperados.

A.2 PROGRAMA 01 - EXEMPLO SPAWN_MONITOR

O programa contém dois módulos (*Spawn_ monitor* e *Child_ Spec*) sendo que o módulo *Spawn_ monitor* serve para calcular o maior valor de uma lista, enquanto o módulo *Child_ Spec* cria uma tarefa e apresenta as informações da mesma para o usuário.

A.2.1 CASOS DE TESTE

1.1 Spawn_monitor.calcular([1, 7, 4, 200, -1])

Resultado: 200

1.2 Spawn_monitor.calcular(4)

Resultado: Deadlock comportamental (Ocorre a falha pois foi passado como parâmetro um tipo inválido, cujo programa não trata).

1.3 Spawn_monitor.calcular(4)

Resultado: Deadlock comportamental (Ocorre a falha pois foi passado como parâmetro um tipo inválido, cujo programa não trata).

1.4 Spawn_monitor.calcular([])

Resultado: Deadlock comportamental (Ocorre a falha pois foi passado como parâmetro uma lista vazia, cujo programa não trata).

1.5 Child_Spec.criaTarefas(:tarefa1)

Resultado: #PID<0.163.0>

Tarefa criada

#PID<0.163.0>

A.3 PROGRAMA 02 - YIELD_MANY

O programa trabalha como um gerenciador de tarefas. Primeiramente são criadas 10 tarefas sendo que cada uma fica ociosa por um tempo (a tarefa 1 fica ociosa por 1 segundo, a tarefa 2 fica ociosa por 2 segundos, ...). Em seguida, é executado um *loop* para obter as respostas das tarefas, sendo que as que não terminaram sua execução são finalizadas.

A.3.1 CASOS DE TESTE

2.1 App.comecar()

Resultado: 1

2

3

4

5

[1, 2, 3, 4, 5]

A.4 PROGRAMA 03 - ELIXIR STUDY

O programa conta com dois módulos que realizam o mesmo processo de receber um número, somar 2 a ele e imprimir o mesmo para o usuário.

A.4.1 CASOS DE TESTE

3.1 TaskModule1.start_link(7)

Resultado: 9

3.2 TaskModule1.start_link("a")

Resultado: Finalização Precoce

3.3 TaskModule3.start_link(11)

Resultado: 13

3.4 TaskModule3.start_link([])

Resultado: Finalização Precoce

A.5 PROGRAMA 04 - EVENTS

O programa cria um processo e realiza algumas trocas de mensagens, sempre apresentado ao usuário informações do ciclo de vida deste processo. Antes de cada mensagem de retorno é apresentado o horário.

A.5.1 CASOS DE TESTE

4.1 Events.main()

Resultado: 23:02:40 => Program start

23:02:40 => Program sleeping

23:02:40 => Task start

23:02:41 => Program signalling event

23:02:41 => Task resumed

:task_is_down

A.6 PROGRAMA 05 - AGENTS AND TASKS IN ELIXIR

O programa conta com três módulos que executam uma mesma ação. Após a execução, é criada uma tarefa que imprime uma mensagem na tela.

A.6.1 CASOS DE TESTE

5.1 App.task_start1()

Resultado: Notification has been sent

5.2 Task3.start()

Resultado: Notification has been sent

A.7 PROGRAMA 06 - SYNCHRONOUS TASK STREAM

O programa conta com cinco módulos que executam *loops* de 1 a 10 criando tarefas e as imprimindo para o usuário. Para alguns dados de teste o retorno pode variar conforme o Hardware disponível.

A.7.1 CASOS DE TESTE

6.1 Concurrent3.process(3)

Resultado: 3, 1573513516825

6.2 Concurrent3.process(7)

Resultado: 7, 1573513596850

6.3 Concurrent5.process(5)

Resultado: 5, 1573513630532

6.4 Concurrent5.process(8)

Resultado: 7, 1573513669179

6.5 OrdemNumerica3.process(3)

Resultado: [1, 2, 3, 4]

6.6 OrdemNumerica3.process(7)

Resultado: [1, 2, 3, 4, 5, 6, 7, 8]

6.7 Processor3.process(4)

Resultado: 4

6.8 Processor3.process(8)

Resultado: 8

6.9 Processor5.executar(9)

Resultado: [exit: :timeout,
exit: :timeout,
exit: :timeout,
exit: :timeout,
exit: :timeout,
ok: 6,
ok: 7,
ok: 8,
ok: 9,
ok: 10]

A.8 PROGRAMA 07 - PANGRAM

O programa contém a solução para o *Pangram*, ou seja, é passada uma frase e a mesma é verificada se contém todas as letras do alfabeto.

A.8.1 CASOS DE TESTE

7.1 Pangram.pangram?("abcdefghijklmnopqrstuvwxyz")

Resultado: 26

7.2 Pangram.pangram?("abcde")

Resultado: nil

A.9 PROGRAMA 08 - PARALLEL LETTER FREQUENCY

Programa que calcula a frequência de letras da frase informada.

A.9.1 CASOS DE TESTE

8.1 Frequency.frequency([], 2)

Resultado: %{}

8.2 Frequency.frequency(["mutation testing for concurrent programs"], 2)

Resultado: %{"a"=> 2,
"c"=> 2,
"e"=> 2,
"f"=> 1,
"g"=> 2,
"i"=> 2,
"m"=> 2,
"n"=> 4,
"o"=> 4,
"p"=> 1,
"r"=> 5,
"s"=> 2,
"t"=> 5,
"u"=> 2}

A.10 PROGRAMA 09 - 17-DINING-PHILOSOPHERS

Programa que simula o problema do jantar dos filósofos. O programa pode produzir diferentes resultados, tendo característica não-determinística.

A.10.1 CASOS DE TESTE

9.1 Table.simulate()

Resultado: 1 philosopher waiting: Aristotle
2 philosopher waiting: Kant, Aristotle
Aristotle is eating (count: 1)

Kant is eating (count: 1)

3 philosophers waiting: Spinoza, Kant, Aristotle

4 philosophers waiting: Marx, Spinoza, Kant, Aristotle

Spinoza is eating (count: 1)

[...]

A.11 PROGRAMA 10 - PARALLEL LETTER FREQUENCY

Programa que calcula a frequência de letras da frase informada.

A.11.1 CASOS DE TESTE

10.1 Frequency.frequency([], 2)

Resultado: %{}

10.2 Frequency.frequency(["mutation testing for concurrent programs"], 2)

Resultado: %{"a"=> 2,

"c"=> 2,

"e"=> 2,

"f"=> 1,

"g"=> 2,

"i"=> 2,

"m"=> 2,

"n"=> 4,

"o"=> 4,

"p"=> 1,

"r"=> 5,

"s"=> 2,

"t"=> 5,

"u"=> 2}

A.12 PROGRAMA 11 - ELIXIR SORTING

Programa que ordena uma matriz através dos métodos de *QuickSort* e *MergeSort*.

A.12.1 CASOS DE TESTE

11.1 QuickSort.parallel([1, 100, -5, 0, 14, 7])

Resultado: [-5, 0, 1, 7, 14, 100]

11.2 MergeSort.parallel([1, 100, -5, 0, 14, 7])

Resultado: [-5, 0, 1, 7, 14, 100]

11.3 QuickSort.parallel([])

Resultado: []

11.4 MergeSort.parallel([])

Resultado: []

11.5 lista = Benchmark.random_list_with_size(10000000)

QuickSort.parallel(lista)

Resultado: [1 , 2 , 7 , 8 , 8 , 9 , 10, 11, 11, 11, 12, 12, 13, 14, 14, 15, 18, 18, 19,
19, 22, 23, 23, 24, 25, 25, 25, 26, 26, 30, 31, 31, 31, 33, 33, 38, 39, 40,
40, 41, 41, 43, 44, 45, 46, 47, 50, 50, 51, 52, ...]

11.6 lista = Benchmark.random_list_with_size(10000000)

MergeSort.parallel(lista)

Resultado: [1 , 2 , 7 , 8 , 8 , 9 , 10, 11, 11, 11, 12, 12, 13, 14, 14, 15, 18, 18, 19,
19, 22, 23, 23, 24, 25, 25, 25, 26, 26, 30, 31, 31, 31, 33, 33, 38, 39, 40,
40, 41, 41, 43, 44, 45, 46, 47, 50, 50, 51, 52, ...]