Taxonomia de Falhas em Programas Concorrentes em Elixir

Matheus Deon Bordignon
Universidade Tecnológica Federal do Paraná - UTFPR
Dois Vizinhos, PR, Brasil
matheusbordignon@alunos.utfpr.edu.br

Rodolfo Adamshuk Silva
Universidade Tecnológica Federal do Paraná - UTFPR
Dois Vizinhos, PR, Brasil
rodolfoa@utfpr.edu.br

ABSTRACT

Computer processing capacity is becoming increasingly insufficient and it encourages the use of concurrent programming to develop applications that reduce the computing time. Due to features such as communication, synchronization and non-determinism, concurrent programs may present concurrency-related errors. This paper presents a defect taxonomy for Elixir concurrent programs considering the functions present on Kernel and Task modules. Defect patterns were identified from the insertion of small disturbances into concurrent functions present in a benchmark of concurrent Elixir programs. The association between entered defects and concurrent programming errors has resulted in defect taxonomy for concurrent Elixir programs. The defined taxonomy will be used to support the definition of criteria and testing tools for concurrent Elixir programs.

KEYWORDS

Taxonomia de Falhas, Elixir, Programação Concorrente, Teste de Software

1 INTRODUÇÃO

A capacidade de processamento dos computadores mostra-se cada vez mais insuficiente e incentiva o uso de programação concorrente e paralela para desenvolver aplicações que reduzam o tempo computacional. A 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 [1]. Enquanto a programação sequencial convencional utiliza 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 [2].

Para criar um software utilizando os conceitos de programação concorrente, três etapas básicas são necessárias: (1) representação do conjunto de tarefas que serão executados concorrentemente, (2) geração e finalização destas tarefas no formato de processos ou *threads* e (3) coordenação e gerência das interações entre os processos enquanto estes estiverem executando juntos [3].

Para a realização das etapas 2 e 3, ferramentas de programação concorrente fazem-se necessárias, como linguagens e bibliotecas que suportam a programação concorrente. Essas linguagens e bibliotecas podem ser classificadas de acordo com o paradigma de desenvolvimento a ser utilizado. O 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.

No paradigma de memória compartilhada, existe um espaço de endereçamento compartilhado, no qual diferentes processos podem fazer operações de leitura e escrita nessas variáveis. Como é possível perceber, será necessário ter um mecanismo de segurança de dados, pois diferentes processos podem acessar a mesma região crítica, por exemplo uma variável que pode ser lida e modificada. Dentre as maneiras de controlar o acesso à região crítica, citam-se os semáforos, barreiras e monitores [4, 5].

O paradigma baseado em passagem de mensagem é utilizado para comunicação entre processos que não utilizam um mesmo endereçamento de memória compartilhado. Existem dois elementos base utilizados para a passagem de mensagem: *send* e *receive*, que embora possam variar sintaticamente, dependendo da linguagem, tem a mesma semântica. *Send* compreende no envio de uma mensagem de um processo a outro, enquanto *receive* é utilizado pelo processo destino para receber a mensagem [5].

Uma linguagem ou biblioteca de programação pode implementar um ou ambos paradigmas de programação. Atualmente, diferentes linguagens emergentes de programação dão suporte à programação concorrente, como Go, Elixir, Ruby, Python entre outras. O Elixir, em especial, foi criada pelo brasileiro José Valim em 2012 e é uma linguagem que suporta o desenvolvimento de aplicações concorrentes e distribuídas de forma dinâmica e moderna. O Elixir, através de uma sintaxe agradável, auxilia o programador a resolver os problemas de concorrência, distribuição e tolerância a falhas semelhante ao Erlang [6].

Um fator que influencia os testes para programas concorrentes é sua natureza não determinística. Programas sequenciais têm característica determinística, isto é, independentemente do número de execuções de um mesmo programa, o mesmo valor de entrada produzirá a mesma saída. Programas concorrentes têm 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 à comunicação, ao paralelismo e à sincronização e, portanto, os casos de teste são gerados a partir dessas categorias. Apesar disso, os demais tipos de erro também devem ser considerados e tratados, pois além de sustentar o programa incorreto, ainda podem provocar erros de comunicação, paralelismo e sincronização comentados anteriormente.

A detecção de falhas na execução em Elixir é menos frequente em comparação com as linguagens de programação mais comuns. Os problemas relacionados à sincronização e comunicação (ditos imprevisíveis) são difíceis de reproduzir. Por essa razão, o modo de projetar aplicações chamado "*Let it crash*" é utilizado nas comunidades Elixir e Erlang. 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 no sistema), permitindo concentração no propósito da aplicação e tratar os caminhos de exceção mais tarde (após eles ocorrerem) [7].

Uma forma de aumentar a disponibilidade, confiabilidade de programas em Elixir é refinar possíveis reações a falhas que ocorrem durante o funcionamento do sistema. Em geral, isso pode ser feito por meio de funções que monitoram o funcionamento de processos e reiniciam um processo quando um problema ocorre como, por exemplo, um *deadlock*. De qualquer maneira, a detecção de falhas e o conhecimento sobre essas falhas são essenciais.

Um dos problemas ao testar programas concorrentes em linguagens emergentes como o Elixir é a falta de estudos sobre as falhas que podem ocorrer pela utilização incorreta das funções de programação concorrente. Por meio de um processo de 5 passos, foi possível definir uma taxonomia de falhas para programas concorrentes em Elixir que utilizam os módulos *Kernel* e *Task*.

O artigo está organizado da seguinte forma: Visão geral e as definições são apresentadas na Seção 2. Na Seção 3 são discutidos os trabalhos relacionados. A Seção 4 apresenta a taxonomia de falhas que é seguida por um exemplo de aplicação da taxonomia na Seção 5. Finalmente, as conclusões e perspectivas sobre trabalhos futuros são apresentadas na Seção 6.

2 VISÃO GERAL E DEFINIÇÕES

2.1 Programação concorrente em Elixir

O Elixir é uma linguagem de programação que segue o paradigma funcional de programação. Herdando do Erlang a capacidade de resolução de problemas de concorrência, distribuição, tolerância a falhas e alta disponibilidade *hot swap*, a linguagem vem ganhando mais adeptos e vem sendo utilizada em grandes empresas [8].

No Elixir, a comunicação entre processos é feita somente por troca de mensagens, o que permite que a linguagem não tenha que gerenciar o estado (seguro ou não seguro) dessas variáveis [9]. O Elixir possui funções utilizadas para resolver problemas que envolvem paralelismo e concorrência presentes em seis módulos: *Kernel*, *Task*, *Process*, *Agent*, *Genserver* e *Task.Supervisor*. Tabela 1 apresenta as funções dos módulos *Kernel* e *Task*, os quais serão explorados na taxonomia. Os números após as funções (como *spawn*/1) indicam a aridade, ou seja, o número de argumentos que as funções podem receber.

2.2 Teste de software

Embora sistemas robustos produzam resultados impressionantes, eles podem trazer enormes problemas para os desenvolvedores. Problemas referentes à sincronização, manipulação e processamento de dados são apenas alguns dos problemas passíveis de acontecer sem o devido tratamento. Para controlar esses problemas, verificando a corretude dos softwares e diminuindo riscos, existe o teste, que pode ser definido como um conjunto de tarefas que podem ser planejadas com antecedência e executadas sistematicamente.

No contexto de teste de software, Engano, Defeito, Erro e Falha são tratados de maneira distinta. Engano refere-se a ação humana que produz um defeito. Defeito é 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 pode ocasionar uma falha que é o resultado diferente do resultado esperado [2]. O termo defeito será utilizando para problemas sintáticos cometidos pelos desenvolvedores. Já os termos erro e falha serão utilizados para o resultado final que pode ser observado ou inferenciado (por exemplo *deadlock* que não é uma resposta do sistema, mas sim o estado em que ele se encontra).

Table 1: Funções de Programação Concorrente em Elixir nos módulos Kernel e Task

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

3 TRABALHOS RELACIONADOS

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 de modo a obter as informações necessárias para os critérios de teste [2]. No contexto de sistemas computacionais tolerantes a falhas, a técnica de teste baseada em defeitos é aplicada por meio do critério de injeção de falhas [10]. 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 [11], MPI [12–16], C e C ++ [17] e SystemC [18]. Farchi et al. [19] 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 [12], Krammer et al. [14] e Luecke et al. [13], DeSouza et al. [15] apresentam uma taxonomia de erros para programas desenvolvidos em MPI. A taxonomia apresentada divide os erros em três categorias principais: (1) Sincronização subdividido em (a) *Deadlock*: Padrão e Dependente do tempo e (b) Condição de corrida: Interface e Entre processos; (2) Incompatibilidade, subdividido em (a) Tipo de chamada, (b) Argumentos e (c) Tamanho; e (3) Recursos, subdividido em (a) Alocação, (b) Inicialização e (c) Desalocação.

Lopez et al. [20] 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. A taxonomia é composta por duas categorias. A primeira é denominada 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. Como validação da taxonomia, um catálogo de defeitos concorrentes coletados da literatura foi classificado segundo a taxonomia definida.

4 TAXONOMIA DE FALHAS

Com base nos artigos apresentados na Seção 3, não há uma definição de processo para a definição de uma 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 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.

Este trabalho está inserido na categoria estática de definição de taxonomia de defeitos, na qual somente a sintaxe da linguagem de programação é considerada e a taxonomia de falha 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.

O trabalho de Lopez et al. [20] é o mais próximo da taxonomia proposta neste artigo, uma vez que programas na linguagem Erlang foram utilizados no processo de definição da taxonomia. O Elixir é executado pela máquina virtual do Erlang e possui características semelhantes ao Erlang. Desta forma, algumas categorias de falhas foram extraídas dessa taxonomia. Porém, como a taxonomia proposta neste artigo é somente para Elixir, não foi possível utilizar todas as categorias da taxonomia de Lopez et al. [20] e houve a necessidade de adicionar novas categorias.

Para a definição da taxonomia de falhas para programas concorrentes em Elixir, o processo mostrado na Figura 1 foi seguida. O processo divide-se em 4 etapas:

(1) Identificação das funções concorrentes
(2) Agrupamento de funções com semântica semelhante
(3) Definição de defeitos
(4) Execução e coleta de falhas

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

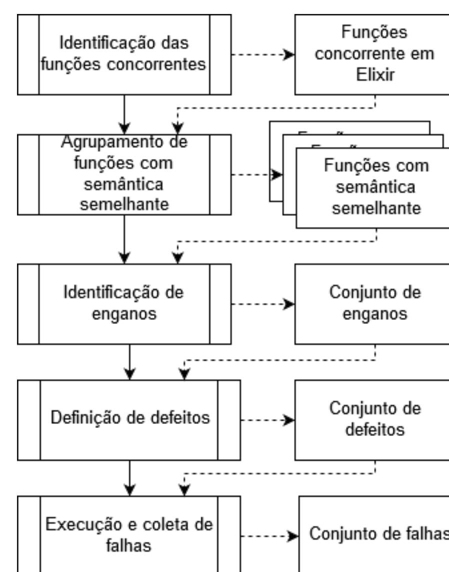

Figure 1: Processo de definição da taxonomia de falhas para funções concorrentes em Elixir

4.1 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 nos módulos: *Kernel*, *Task*, *Process*, *Agent*, *Genserver* e *Task.Supervisor*. Este trabalho considera apenas as funções do *Kernel* e no módulo *Task* (Tabela 1) por serem os mais utilizados e difundidos pela comunidade de desenvolvedores.

4.2 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 no contexto da programação, diferenciando-se entre si, por exemplo, em como uma ação

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

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 2 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 uma vez que elas realizam a criação de processos e são semelhantes semanticamente. Porém, sintaticamente elas diferem-se 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* diferem-se entre si pois a função *spawn_link* mantém um vínculo de ligação com o processo pai (que cria o processo) e a função *spawn_monitor* faz com que o processo criado seja monitorado pelo processo pai.

Table 2: Agrupamento de funções com semântica semelhante nos módulos 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

4.3 Definição de 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*, *spawn_link/1* e *spawn_monitor/1* diferem-se entre si sintaticamente por possuir um sufixo *_link*, *_monitor* ou por não possuir sufixo como visto a seguir:

```
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 serve como base para a definição da taxonomia de defeitos mostrada na próxima seção.

4.4 Definição de defeitos

A taxonomia de defeitos apresentada abaixo simula os enganos que podem ser 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 parâmetros, (4) Defeitos por adição de parâmetro e (5) Defeitos por troca de parâmetro. Os defeitos por troca de função correspondem às 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 3 e os defeitos identificados para o módulo *Task* podem ser observados na Tabela 4. 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 3 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 na próxima seção.

4.5 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 5 e 6 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 na qual os processos em Elixir são executados.

(1) ***Argument Error:*** Esta falha é obtida durante a utilização de funções com parâmetros incorretos. Um exemplo é a função *spawn* que espera como parâmetro a função que será executa.

Table 3: 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__

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 diferença é que no *Argument Error* a função tenta executar seu objetivo com o valor recebido como parâmetro e dispara 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 *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. Esta falha pode ser obtida, por exemplo, ao atrelar a uma tupla de tamanho dois o retorno da função *spawn*, já que está retorna somente um valor. Por exemplo, na linha de código: *{pid,ref} = spawn(fn − > task() end)}*, o retorno da função *spawn* é o identificador do processo criado. No exemplo, caso este valor fosse atrelado a uma tupla de tamanho dois, um *Match Error* seria obtido.

Table 4: 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__

Table 5: 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

O grupo *Falhas relacionadas à programação concorrente* contém falhas relacionadas ao comportamento do programa e resultado

Table 6: 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

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 pode ser simulado com a substituição do primeiro parâmetro da função *spawn* pela tag *__MODULE__* em um programa 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:*** Este erro está relacionado ao comportamento do sistema que encontra-se travado. O problema acontece durante a troca de mensagens entre processos, no qual um processo fica 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*, o processo filho lançaria a exceção e o processo pai ficaria travado esperando a resposta.
(3) ***Finalização Precoce:*** Erros de finalização precoce ocorrem quando o programa termina a execução sem ter terminado por completo o seu processamento. 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. Isso não ocorre quando a função *spawn_link* é utilizada. Ao trocar essas funções, o processo criador é finalizado antes de completar a sua execução. 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*** Esta falha acontece quando o resultado produzido pelo programa difere do esperado. Por exemplo, um programa deve retornar o valor da soma de dois números seguido da multiplicação entre eles, porém retorna primeiro a multiplicação e depois a soma. 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 bloqueada.

A Figura 2 apresenta as categorias da taxonomia de falhas e os defeitos relacionados a elas.

5 APLICAÇÃO DA TAXONOMIA DE FALHAS

Nesta seção, é apresentada uma aplicação da taxonomia de falhas definida para programas concorrentes em Elixir a qual engloba as funções dos módulos *Kernel* e *Task*. O objetivo desse exemplo é mostrar como a taxonomia de falhas pode ser utilizada como suporte ao teste de programas concorrentes em Elixir. Para isso, os seguintes passos foram realizados: (1) Seleção do conjunto de programas, (2) Inserção manual dos defeitos, (3) Execução dos programas defeituosos e (4) Coleta das falhas.

Para a aplicação da taxonomia de falhas em programas concorrentes em Elixir, um *benchmark* de programas concorrentes em Elixir foi utilizado como base para a semeadura dos defeitos e coleta das falhas. O *benchmark* utilizado foi definido em Bordignon e Silva [21] e representa as funções do *Kernel* e do módulo *Task* em Elixir (Tabela 7). O *benchmark* é composto por 11 programas de diferentes tamanhos (medidos por LoC) nos quais busca-se explorar as diferentes aridades de algumas funções, como a *spawn*, *spawn_monitor* e *spawn_link*.

Table 7: Programas concorrentes em Elixir [21]

Id	Nome do Arquivo	Descrição do Problema	LoC
P01	Exemplo Spawn_Monitor	Maior valor e monitor de tarefas	15
P02	Yield_many	Gerenciador de Tarefas	21
P03	Elixir Study	Tarefa com operação de soma	22
P04	Events	Vida de um processo	26
P05	Agents and Tasks in Elixir	Execução de tarefas sem resposta	28
P06	Synchronous Task Stream	Fluxo de Tarefas Síncronas	55
P07	Pangram	Pangrama	56
P08	Parallel Letter Frequency	Frequência Letra Paralelo	76
P09	17-dining-philosophers	Jantar dos Filósofos	100
P10	Parallel Letter Frequency	Frequência Letra Paralelo	184
P11	Elixir-sorting	QuickSort e MergeSort Paralelo	481

Para cada programa presente no *benchmark* (Tabela 7), foi realizada uma busca manual por cada função de concorrência e semeado um defeito para cada ocorrência da função na taxonomia de defeitos (Tabela 3 e Tabela 4). Como resultado, um conjunto de programas defeituosos foi gerado, como mostram a Tabela 8 e a Tabela 9. 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

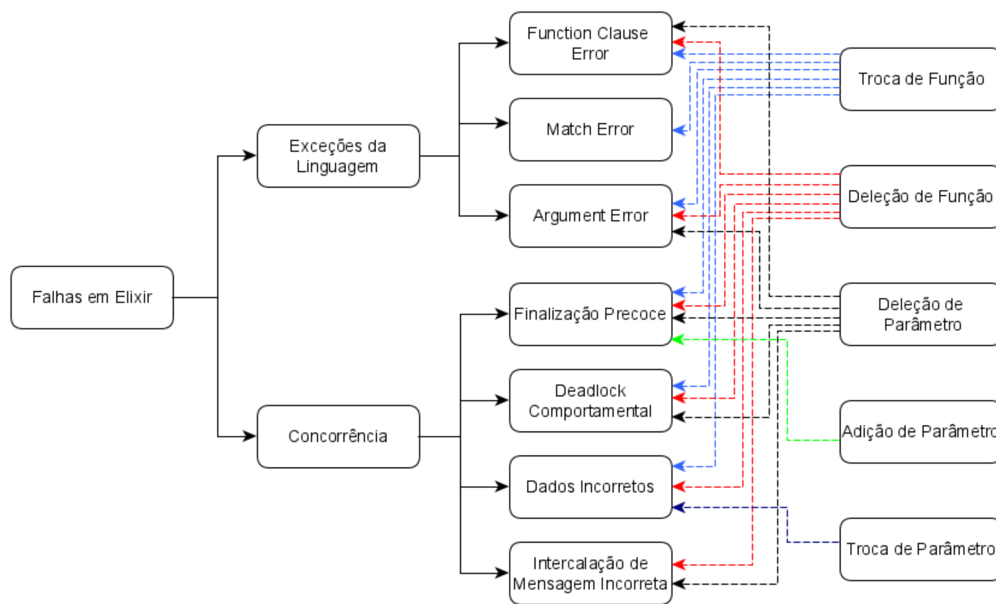

Figure 2: Visão geral das falhas obtidas

Table 8: 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

Table 9: 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

para encontrar as falhas definidas na taxonomia de falhas. A Tabela 10 apresenta a quantidade de dados de teste presente no conjunto final de casos de teste para cada um dos programas do *benchmark*.

Por fim, a partir da execução dos programas defeituosos com os dados de teste, é possível encontrar as falhas nos programas, como apresentado na taxonomia de falhas. A última coluna da Tabela 10 apresenta as falhas apresentadas pelos programas defeituosos de cada um dos programas do *benchmark*.

Table 10: 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 Erro, Deadlock Comportamental, Finalização Precoce
Agents and Tasks in Elixir	2	Finalização Precoce
Synchronous Task Stream	9	-
Pangram	22	Dados incorretos
Parallel Letter Frequency	1	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

6 CONCLUSÃO

O Elixir é uma linguagem de programação que suporta o desenvolvimento de aplicações concorrentes e distribuídas de forma dinâmica e moderna. Ao desenvolver aplicações concorrentes, deve-se levar em consideração características como a comunicação, sincronização e o não determinismo intrínsecas à programação concorrente. Nesse cenário, as atividades de verificação, validação e teste precisam considerar essas características durante sua aplicação.

Como uma forma de auxiliar as atividades de verificação, validação e teste de programas concorrentes em Elixir, este artigo apresentou uma taxonomia de falhas englobando as funções presentes nos módulos *Kernel* e *Task*. A taxonomia foi dividida em dois grupos. O grupo *Exceções da linguagem Elixir* contém três falhas: (1) *Argument Error*, (2) *Function Clause Error* e, (3) *Match Error*. 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.

Por mais que existam taxonomias de defeitos para programas concorrentes, a taxonomia apresentada nesse trabalho abrange as características da linguagem de programação Elixir. A taxonomia de falhas pode ser utilizada para examinar mecanismos de detecção de falhas e servir como suporte para a definição de critérios e ferramentas de teste de programas concorrentes em Elixir.

Como trabalhos futuros, espera-se expandir a taxonomia de falhas considerando defeitos nos outros módulos do Elixir não abrangidos nesse trabalho. Além disso, com base na taxonomia de defeitos definida, será criado um conjunto de operadores de mutação para programas concorrentes em Elixir que irá embasar a aplicação do critério de teste de mutação da técnica de teste baseada em defeitos.

REFERENCES

[1] Gregory R. Andrews. *Concurrent programming: principles and practice.* Benjamin/Cummings Publishing Co., San Francisco, USA, 1991.
[2] Simone R. S. Souza, Paulo S. L. Souza, Silvana M. Melo, Rodolfo A. Silva, and Silvia R. Vergilio. Teste de programas concorrentes. In Marcio E. Delamaro, Mario Jino, and Jose Maldonado, editors, *Introdução ao Teste de Software*, volume 2, pages 261–296. Elsevier Editora Ltda, Rio de Janeiro, BR, 2016.
[3] Allan Gottlieb and George S. Almasi. *Highly parallel computing.* Benjamin/Cummings, Redwood City, CA, 1989.
[4] Maarten Van Steen and Andrew S. Tanenbaum. *Sistemas Distribuídos: princípios e paradigmas.* Prentice Hall, São Paulo, BR, 2009.
[5] Ananth Grama, Vipin Kumar, Anshul Gupta, and George Karypis. *Introduction to parallel computing.* Pearson Education, Essex, UK, 2003.
[6] Tiago Davi. *Elixir: Do zero à concorrência.* Casa do Código, São Paulo, BR, 2017.
[7] André Albuquerque and Daniel Caixinha. *Mastering Elixir.* Packt Publishing, Birmingham, UK, 2018.
[8] Sean Callan, Tajinder Chumber, and George Mantzouranis. A collection of companies using elixir in production. Disponível em: https://elixir-companies.com/en, 2019. Acesso em: 13 Jul. 2019.
[9] Ulisses Almeida. *Learn functional programming with elixir: New Foundations for a New World.* Pragmatic Bookshelf, USA, 2018.
[10] Boris Beizer. *Software Testing Techniques.* Van Nostrand Reinhold Co., New York, USA, 1990.
[11] A. Jefferson Offutt, Jeff Voas, and Jeff Payne. Mutation operators for Ada. Technical report, George Mason University, 1996.
[12] Jeffrey S. Vetter and Bronis R. Supinski. Dynamic software testing of MPI applications with umpire. In *2000 ACM/IEEE conference on Supercomputing*, pages 1–10, Washington, DC, USA, 2000. IEEE Computer Society.
[13] Glenn R. Luecke, Hua Chen, James Coyle, Jim Hoekstra, Marina Kraeva, and Yan Zou. MPI-CHECK: a tool for checking Fortran 90 MPI programs. *Concurrency and Computation: Practice and Experience*, 15(2):93–100, 2003.
[14] Bettina Krammer, K. Bidmon, Matthias Stefan Müller, and Michael M. Resch. MARMOT: An MPI analysis and checking tool. In G. R. Joubert, W. E. Nagel, F. J. Peters, and W. V. Walter, editors, *Parallel Computing*, volume 13, pages 493–500. Elsevier, Dresden, GER, 2003.
[15] Jayant DeSouza, Bob Kuhn, Bronis R. de Supinski, Victor Samofalov, Sergey Zheltov, and Stanislav Bratanov. Automated, scalable debugging of MPI programs with intel message checker. In *Workshop on Software engineering for high performance computing system applications*, pages 78–82, New York, USA, 2005. ACM.
[16] Jan B. Pedersen. Classification of programming errors in parallel message passing systems. In *Communicating Process Architectures*, pages 363–376, Edinburgh, Sld, 2006. IOS.
[17] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 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*, pages 329–339, New York, NY, USA, 2008. ACM.
[18] Alper Sen. Mutation operators for concurrent SystemC designs. In *10th International Workshop on Microprocessor Test and Verification*, pages 27–31, Washington, DC, USA, 2009. IEEE Computer Society.
[19] Eitan Farchi, Yarden Nir, and Shmuel Ur. Concurrent bug patterns and how to test them. In *Proceedings International Parallel and Distributed Processing Symposium*, pages 1–7, Nice, FR, 2003. IEEE.
[20] Carmen Torres Lopez, Stefan Marr, Elisa Gonzalez Boix, and Hanspeter Mössenböck. *A Study of Concurrency Bugs and Advanced Development Support for Actor-based Programs*, pages 155–185. Springer International Publishing, Cham, 2018.
[21] Matheus D. Bordignon and Rodolfo A. Silva. Benchmark de programas concorrentes em elixir para suporte à validação de critérios e ferramentas de teste. In *II Simpósio de Engenharia de Software*, Dois Vizinhos, PR, 2019. UTFPR.