Teste estrutural para aplicações concorrentes em Erlang

Alexandre Ponce de Oliveira

SERVIÇO DE PÓS-GRADUAÇÃO DO ICMC-USP

Data de Depósito:

Assinatura: _____

Alexandre Ponce de Oliveira

Teste estrutural para aplicações concorrentes em Erlang

Tese apresentada ao Instituto de Ciências Matemáticas e de Computação - ICMC-USP, como parte dos requisitos para obtenção do título de Doutor em Ciências - Ciências de Computação e Matemática Computacional. EXEMPLAR DE DEFESA.

Área de Concentração: Ciências de Computação e Matemática Computacional

Orientador: Prof. Dr. Paulo Sérgio Lopes de Souza

USP – São Carlos
Março de 2017

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

O48t Oliveira, Alexandre Ponce de
 Teste Estrutural para Aplicações Concorrentes em
Erlang / Alexandre Ponce de Oliveira; orientador
Paulo Sérgio Lopes de Souza. -- São Carlos, 2017.
 143 p.

 Tese (Doutorado - Programa de Pós-Graduação em
Ciências de Computação e Matemática Computacional) --
Instituto de Ciências Matemáticas e de Computação,
Universidade de São Paulo, 2017.

 1. Linguagem Erlang. 2. Teste estrutural. 3.
Critérios de Teste. 4. Modelo de teste. 5.
ValiErlang. I. Souza, Paulo Sérgio Lopes de,
orient. II. Título.

Alexandre Ponce de Oliveira

Structural testing for concurrent applications in Erlang

Doctoral dissertation submitted to the Instituto de Ciências Matemáticas e de Computação - ICMC-USP, in partial fulfillment of the requirements for the degree of the Doctorate Program in Computer Science and Computational Mathematics. EXAMINATION BOARD PRESENTATION

Concentration Area: Computer Science and Computational Mathematics

Advisor: Prof. Dr. Paulo Sérgio Lopes de Souza

USP – São Carlos
March 2017

À Deus pela minha vida e plena saúde.
Aos meus pais, Antônio e Sueli.
À minha esposa Carolina.
Aos meus filhos, Gabriel e Rafaela.

AGRADECIMENTOS

Agradeço primeiramente a Deus, por ter me concedido saúde, me guiado e dado forças para o cumprimento desta longa etapa de minha vida.

Agradeço a minha esposa Carolina pelo amor, carinho e compreensão durante toda essa jornada. Seu companheirismo, incentivo e auxílio foram fundamentais para que eu chegasse até aqui. Agradeço também aos meus filhos Gabriel e Rafaela, que foram a minha inspiração durante todo o trabalho.

Agradeço a minha família, em especial meus pais, Antônio e Sueli, pelo apoio incondicional durante todos os obstáculos enfrentados até hoje e também por sempre acreditar em seus filhos.

Agradeço ao meu orientador, professor Paulo Sérgio Lopes de Souza, pelo profissionalismo, dedicação, competência, sabedoria, paciência, amizade e todo ensinamento transmitido ao longo desses mais de cinco anos, isso foi fundamental para o início, desenvolvimento e conclusão deste projeto. Agradeço também a professora Simone do Rocio Senger de Souza pelas dicas e todo apoio oferecido durante o desenvolvimento do projeto.

Por fim, agradeço a todos os professores do ICMC, pelos ensinamentos transmitidos durante as disciplinas. E também, a todos os colegas do LaSDPC, pela companhia, amizade e momentos de bate-papo para tornar o ambiente de estudos mais produtivo e agradável.

RESUMO

OLIVEIRA, A. P. **Teste Estrutural para Aplicações Concorrentes em Erlang**. 2017. 143 f. Tese (Doutorado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação (ICMC/USP), São Carlos – SP.

As atividades de validação, verificação e teste contribuem para melhorar a qualidade dos programas, independentemente do paradigma de programação utilizado. **Erlang** é um exemplo de linguagem funcional, e aspectos como: dados imutáveis, *higher-order functions*, *lazy evaluation* e *pattern matching* impõem restrições à atividade de teste estrutural de software, as quais requerem uma atenção especial do testador. A linguagem **Erlang** foi criada para o desenvolvimento de aplicações concorrentes, em tempo real e com tolerância a falhas. A aplicação da atividade de teste de software torna-se necessária para aplicações desenvolvidas em **Erlang**. Um mapeamento sistemático realizado identificou os trabalhos relacionados e também a identificação de falhas típicas encontradas em programas **Erlang**. O resultado do mapeamento evidenciou a falta de propostas que considerem as principais características de **Erlang**, incluindo uma ferramenta de teste que dê suporte à aplicação prática da atividade de teste. Esta lacuna foi considerada significativa. Foi proposto um conjunto de critérios de teste estruturais para verificar a cobertura de códigos em aplicações sequenciais e distribuídas de programas **Erlang**. Os critérios de teste exploram as possíveis falhas relacionadas à comunicação entre funções, comunicação entre processos, sincronização, concorrência, recursividade e tolerância a falhas. A definição dos critérios contou com o suporte de um modelo de teste para a obtenção das informações sobre o fluxo de controle, fluxo de dados e fluxo de comunicação de programas **Erlang**. O modelo estabeleceu 15 tipos de nós, 9 tipos de arestas e 5 tipos de usos de variáveis. Para apoiar a aplicação dos critérios, uma ferramenta de teste, chamada **Valierlang**, também foi implementada. A **ValiErlang** é composta por 5 módulos que realizam as seguintes etapas: análise estática, instrumentação do código fonte, geração do GFC, definição dos elementos requeridos, execução do código instrumentado, geração do rastro de execução e avaliação dos elementos cobertos e elementos não executáveis. Foi realizado um estudo experimental para verificar a aplicabilidade dos critérios de teste por meio da **ValiErlang**. Neste estudo foram utilizados seis programas com diferentes características, todas essenciais aos programas em **Erlang**. Com base nos resultados obtidos, foi possível comprovar que a abordagem contribui com o testador devido ao sucesso na aplicação dos critérios e também a eficácia dos critérios em revelar defeitos. E as etapas para o testador de software realizar a aplicação do teste estrutural, tem o apoio ferramental da **ValiErlang**.

Palavras-chave: Linguagem funcional, linguagem **Erlang**, teste estrutural, modelo de teste, critérios de teste, **ValiErlang**.

ABSTRACT

OLIVEIRA, A. P. **Structural Testing for Concurrent Applications in Erlang**. 2017. 143 f. Tese (Doutorado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação (ICMC/USP), São Carlos – SP.

The validation, verification and test activities contribute to improve the quality of the programs, independently of the used programming paradigm. **Erlang** is an example of functional language and aspects such as: immutable data, *higher-order functions*, *lazy evaluation* e *pattern matching* impose restrictions to the software structural testing activity, which require a special attention by the tester. The **Erlang** language was created for the development of concurrent applications, in real time and with fault tolerance. The application of the software testing activity becomes necessary for applications developed in **Erlang**. An executed systematic mapping identified the related works and the identification of typical failures found in **Erlang** programs. The result of the mapping highlighted the lack of proposals that consider the main features of **Erlang**, including a testing tool that supports the practical application of the testing activity. This gap was considered significant. A set of structural testing criteria was proposed to verify the coverage of codes in sequential and distributed applications in **Erlang** programs. The test criteria explore the possible failure related to the communication between functions, communication between processes, synchronization, concurrence, recursion and fault tolerance. The definition of the criteria had the support of a test model to obtain information about control flux, data flux, and communication flux of **Erlang** programs. The model established 15 types of nodes, 9 types of edges and 5 types of variable uses. To support the application of criteria, a testing tool called **ValiErlang** was also implemented. **ValiErlang** is composed of 5 modules that execute the following stages: static analysis, source code instrumentation, CFG generation, definition of the required elements, instrumented code execution, execution trace generation and evaluation of the covered and non-executable elements. An experimental study was executed to verify the applicability of the testing criteria by **ValiErlang**. In this study six programs with different characteristics were used. All characteristics were essential to the **Erlang** programs. Based on the obtained results, it was possible to prove that the approach contributes with the tester because of the success in the criteria application and the efficiency of the criteria in revealing defects. The stages for the testes to execute the structural test application have the support of the **Erlang** tool.

Key-words: Functional language, **Erlang** language, structural testing, test model, testing criteria, **ValiErlang**.

LISTA DE ILUSTRAÇÕES

FIGURA 2.1 - EXEMPLO DE USO DE COMPREENSÕES DE LISTA.....	38
FIGURA 2.2 - EXEMPLO DE COMUNICAÇÃO ENTRE PROCESSOS.	39
FIGURA 2.3 - INTERPRETADOR DE COMANDOS <i>ERLANG</i>	44
FIGURA 2.4 – EXEMPLO DE COMPILAÇÃO E EXECUÇÃO DE UM PROGRAMA.....	44
FIGURA 2.5 – EXEMPLO DE RECURSÃO COM USO DE LISTA.	45
FIGURA 2.6 – EXEMPLO DE CRIAÇÃO DE LISTA A PARTIR DE UMA LISTA EXISTENTE.	46
FIGURA 2.7 – EXEMPLO DE CRIAÇÃO DE UM PROCESSO.	47
FIGURA 2.8 – EXEMPLO DE ENVIO DE MENSAGEM PARA UM PROCESSO.	47
FIGURA 2.9 – EXEMPLO DE TROCA DE MENSAGEM ENTRE DOIS PROCESSOS.	49
FIGURA 3.1 – (A) PROGRAMA QUE CALCULA O MÁXIMO DIVISOR COMUM (GCD) ENTRE DOIS VALORES; (B) GFC DO PROGRAMA GCD; (C) GRAFO DE DEFINIÇÃO E USO DO GCD.....	58
FIGURA 3.2 – ARQUITETURA DA FERRAMENTA VALIPAR.....	64
FIGURA 4.1 – EXEMPLO DE TROCA DE MENSAGENS ENTRE DOIS PROCESSOS.....	86
FIGURA 4.2 – PROGRAMA GCD.	91
FIGURA 4.3 – GRAFO DE FLUXO DE CONTROLE PARALELO DO PROGRAMA GCD EM <i>ERLANG</i>	92
FIGURA 5.1 – ARQUITETURA COMPLETA DA FERRAMENTA <i>VALIERLANG</i>	100
FIGURA 5.2 – EXEMPLO DE UM CÓDIGO FONTE A SUA CORRESPONDÊNCIA EM AST.	102
FIGURA 5.3 – EXEMPLO DE UM TRECHO DO ARQUIVO GRAVADO PELO MÓDULO PARSE.....	102
FIGURA 5.4 – ARQUITETURA PARCIAL DA <i>VALIERLANG</i> COM O MÓDULO INST.	103
FIGURA 5.5 – CFG PARCIAL DO PROGRAMA GCD GERADO PELA <i>VALIERLANG</i>	106
FIGURA 5.6 – PROGRAMA GCD INSTRUMENTADO.....	107
FIGURA 5.7 – PCFG PARCIAL DO PROGRAMA GCD.	108
FIGURA 5.8 – ARQUIVO COM OS ELEMENTOS REQUERIDOS CORRESPONDENTES AO CRITÉRIO TODOS-NÓS-SYNC.....	109
FIGURA 5.9 – RASTRO PARCIAL DO ARQUIVO GERADO PELO MÓDULO EXEC DA <i>VALIERLANG</i>	110

FIGURA 5.10 – INFORMAÇÕES SOBRE A COBERTURA DO CRITÉRIO TODOS-NÓS-SYNC E O ARQUIVO COM O SUMÁRIO DOS RESULTADOS.	112
FIGURA 5.11– INFORMAÇÕES SOBRE A COBERTURA DAS CHAMADAS RECURSIVAS.	112
FIGURA 5.12 – ESTRUTURA DE DIRETÓRIOS CRIADA PELA VALIERLANG	113
FIGURA 6.1 – FUNÇÃO <i>SERVER</i> DO PROGRAMA MESSENGER . É TRACE DE EXECUÇÃO DE UM DADO DE TESTE PARA OS CRITÉRIOS TODOS-NÓS-CALLREC E TODOS-NÓS-RETREC	120
FIGURA 6.2 – EXEMPLO DE COBERTURA PARA O CRITÉRIO TODOS-M-USOS	121
FIGURA 6.3 – EXEMPLO DE ERRO RELACIONADO COM CONDIÇÃO DE DISPUTA DO PROGRAMA GCD	121
FIGURA 6.4 – EXEMPLO DE ERRO DE DOMÍNIO DO PROGRAMA GCD	122
FIGURA 6.5 – EXEMPLO DE RECEBIMENTO DE MENSAGEM PELO PROCESSO MONITOR.	123
FIGURA 6.6 – EXEMPLO DE COBERTURA PARA OS CRITÉRIOS TODOS-NÓS-RECURSIVOS. .	124

LISTA DE ALGORITMOS

ALGORITMO 2.1 - EXEMPLO DE FUNÇÃO QUE UTILIZA <i>PATTERN MATCHING</i>	37
ALGORITMO 2.2 - EXEMPLO DE FUNÇÃO QUE UTILIZA <i>GUARDS</i>	37
ALGORITMO 2.3 - EXEMPLO DE CRIAÇÃO DE PROCESSOS EM <i>HOSTS</i> DIFERENTES.	40
ALGORITMO 2.4 - EXEMPLO DE CRIAÇÃO DE PROCESSOS COM <i>LINKS</i>	41
ALGORITMO 2.5 - EXEMPLO DE UM PROGRAMA EM <i>ERLANG</i>	44
ALGORITMO 2.6 - EXEMPLO DE UM PROGRAMA QUE UTILIZA LISTA E RECURSÃO.....	45
ALGORITMO 2.7 - EXEMPLO DE SINTAXE PARA RECEBIMENTO DE MENSAGENS.	48
ALGORITMO 2.8 - EXEMPLO DE TROCA DE MENSAGEM ENTRE DOIS.....	48
ALGORITMO 2.9 - EXEMPLO DE UM PROGRAMA QUE ENVOLVE DOIS PROCESSOS E UM ERRO DE EXISTÊNCIA.....	51

LISTA DE TABELAS

TABELA 3.1 – MÁQUINAS DE BUSCA UTILIZADAS.....	66
TABELA 3.2 – NÚMERO DE ARTIGOS SELECIONADOS POR MÁQUINA DE BUSCA.	68
TABELA 3.3 – CRITÉRIOS BASEADOS NAS DEFINIÇÕES E USO DAS VARIÁVEIS.....	71
TABELA 4.1 – CONJUNTO DE DEFINIÇÃO DE VARIÁVEIS DO PROGRAMA GCD.	93
TABELA 4.2 – CONJUNTOS DO MODELO DE TESTE PARA O PROGRAMA GCD.	94
TABELA 4.3 – ALGUNS ELEMENTOS REQUERIDOS PELOS CRITÉRIOS DE TESTE PROPOSTOS.	95
TABELA 5.1 – INFORMAÇÕES GERADAS NO PRIMEIRO ESTÁGIO DO MÓDULO INST.....	104
TABELA 6.1 – NÚMEROS DE ELEMENTOS COBERTOS E NÃO EXECUTÁVEIS POR ESTUDO DE CASO.	118
TABELA 6.2 – QUANTIDADE DE CASOS DE TESTE UTILIZADOS POR ESTUDO DE CASO.	119
TABELA 6.3 – COMPARATIVO ENTRE AS FERRAMENTAS DE TESTE AVALIADAS.	126
TABELA 6.4 – DESCRIÇÃO DOS CRITÉRIOS QUE SÃO UTILIZADOS PELAS FERRAMENTAS DE TESTE.	127

LISTA DE ABREVIATURAS E SIGLAS

ASCII : American Standard Code for Information Interchange

AST: Abstract Syntax Tree

API: Application Programming Interface

BEAM : Bogdans Erlang Abstract Machine

BIF : Built-In Function

BPEL: Business Process Execution Language

BR: Blocking Receive

BS: Blocking Send

CFBT: Complete Functional Binary Tree

CPU: Central Processing Unit

DS: Distinction Sequence

EC: Critérios de Exclusão

EDG: Erlang Dependency Graph

GCD: Greatest Common Divisor

GD: Grafos de Dependência

GFC: Grafo de Fluxo de Controle

GFCP: Grafo de Fluxo de Controle Paralelo

GFD: Grafo de Fluxo de Dados

IC: Critérios de Inclusão

JAM: Joe's Abstract Machine

LDAP: Lightweight Directory Access Protocol

MC/DC: Multiple Condition/Decision Coverage

MEF: Máquina de Estado Finito

MPI: Message Passing Interface

OTP: Open Telecom Platform

PCFG: Parallel Control Flow Graph

PVM: Parallel Virtual Machine

SDG: System Dependence Graph

SPG: Semantic Program Graph

TDD: Test Driven Development

TT: Transition Tour

UIO: Unique input/output

VM: Virtual Machine

VV&T: Validação, Verificação e Teste

SUMÁRIO

1 INTRODUÇÃO	25
1.1 AS LINGUAGENS FUNCIONAIS E O TESTE DE SOFTWARE	25
1.2 MOTIVAÇÃO.....	29
1.3 OBJETIVOS.....	30
1.4 METODOLOGIA.....	31
1.5 ESTRUTURA DA MONOGRAFIA.....	32
2 LINGUAGEM DE PROGRAMAÇÃO FUNCIONAL ERLANG	35
2.1 CONSIDERAÇÕES INICIAIS	35
2.2 ESPECIFICAÇÃO DA LINGUAGEM <i>ERLANG</i>	35
2.2.1 <i>História de Erlang</i>	36
2.2.2 <i>Características de Erlang</i>	37
2.2.3 <i>Tipos de Dados em Erlang</i>	42
2.2.4 <i>Interpretador de Comandos Erlang</i>	43
2.2.5 <i>Programação Concorrente em Erlang</i>	46
2.3 CONSIDERAÇÕES FINAIS.....	49
3 TESTE DE SOFTWARE	53
3.1 CONSIDERAÇÕES INICIAIS	53
3.2 CONCEITOS E TIPOS DE TESTE	53
3.2.1 <i>Teste estrutural</i>	57
3.3 TESTE ESTRUTURAL DE PROGRAMAS CONCORRENTES	60
3.4 MAPEAMENTO SISTEMÁTICO EM TESTE DE LINGUAGENS FUNCIONAIS	65
3.4.1 <i>Planejamento</i>	65
3.4.2 <i>Execução</i>	67
3.4.3 <i>Resultados do mapeamento sistemático</i>	68
3.5 FALHAS TÍPICAS EM <i>ERLANG</i>	75
3.5.1 <i>Erros de concorrência</i>	75
3.5.2 <i>Erros de tipo</i>	76

3.5.3 Erros de domínio	76
3.5.4 Erros de existência	77
3.6 CONSIDERAÇÕES FINAIS	77
4 CRITÉRIOS DE TESTE ESTRUTURAL PARA PROGRAMAS ERLANG.....	79
4.1 CONSIDERAÇÕES INICIAIS.....	79
4.2 MODELO DE TESTE ESTRUTURAL PARA PROGRAMAS ERLANG	79
4.3 CRITÉRIOS ESTRUTURAIS DE TESTE	88
4.3.1 Critérios de teste baseados em fluxo de controle e de comunicação	88
4.3.2 Critérios de teste baseados em fluxo de dados e passagem de mensagem.....	89
4.4 NOVOS CRITÉRIOS DE TESTE ESTRUTURAIS DE TESTE	89
4.5 UM EXEMPLO	90
4.6 REVELANDO FALHAS	93
4.7 CONSIDERAÇÕES FINAIS	98
5 VALIERLANG	99
5.1 CONSIDERAÇÕES INICIAIS	99
5.2 UMA VISÃO GERAL DA VALIERLANG.....	100
5.3 MÓDULO PARSE	101
5.4 MÓDULO INST	103
5.5 MÓDULO ELEM.....	107
5.6 MÓDULO EXEC.....	109
5.7 MÓDULO EVAL	111
5.8 CONSIDERAÇÕES FINAIS	112
6 ESTUDOS EXPERIMENTAIS	115
6.1 CONSIDERAÇÕES INICIAIS.....	115
6.2 APLICAÇÃO DOS CRITÉRIOS DE TESTES	115
6.3 ANÁLISE DE FALHAS TÍPICAS PELA VALIERLANG	121
6.4 TRABALHOS RELACIONADOS	124
6.5 CONSIDERAÇÕES FINAIS	127
7 CONCLUSÕES	129
7.1 CONSIDERAÇÕES FINAIS DO PROJETO	129
7.2 CONTRIBUIÇÕES.....	131

7.3 LIMITAÇÕES E TRABALHOS FUTUROS.....	133
7.4 PUBLICAÇÕES RESULTANTES.....	134
REFERÊNCIAS.....	137

1 INTRODUÇÃO

1.1 As linguagens funcionais e o teste de software

As aplicações computacionais atuam em diversas áreas do mercado. Essa diversidade estimulou a criação de vários paradigmas de linguagens de programação com características distintas (SEBESTA, 2012). Tal diversidade permite a categorização das linguagens de programação em imperativas, orientadas a objetos, lógicas e funcionais.

As linguagens imperativas foram projetadas em função da arquitetura de Von Neumann, onde dados e instruções são armazenados em memória e posteriormente são requisitados pela *Central Processing Unit* (CPU) para execução. Essa organização faz com que as instruções e dados precisem ser transportados da memória para a CPU e o resultado da computação seja posteriormente devolvido à memória (SEBESTA, 2012; VAREJÃO, 2004).

As linguagens imperativas podem ser caracterizadas pelo estado de um programa, o qual é mantido por variáveis. Tais variáveis representam células de memória que possuem um endereço e um valor associado. Os valores das variáveis são modificados através de instruções de atribuição e tais mudanças causam mudança no estado de um programa. Nas linguagens imperativas as iterações são bem eficientes, pois normalmente tem-se instruções e dados em células adjacentes na memória (SEBESTA, 2012). Alguns exemplos de linguagens imperativas são: **C**, **Pascal**, **Fortran**, **Basic**, **Algol**, **Cobol**, **Ada**, **Python** e **Assembly**.

Linguagens de programação orientadas a objetos têm como foco o uso de abstração de dados, onde o processamento com objetos de dados é encapsulado e o acesso a esses objetos é protegido. Tais linguagens utilizam também o conceito de herança que potencializa a reutilização de softwares e a vinculação dinâmica de métodos, flexibilizando o uso da herança (SEBESTA, 2012).

A linguagem orientada a objeto assemelha-se com os conceitos das linguagens imperativas, tais como: as expressões, as instruções de atribuição de valores e as instruções de controle (SEBESTA, 2012). Como exemplo de linguagens orientadas a objeto podem ser citadas **Smalltalk**, **C++** e **Java**.

Linguagens de programação lógicas são sistemas de notação simbólica para escrever instruções lógicas com algoritmos específicos e utilizar regras de inferência lógica para produzir resultados (SEBESTA, 2012; LOUDEN, 1993). A lógica simbólica pode ser utilizada para as três necessidades da lógica formal: para expressar proposições, para expressar as relações entre proposições, e descrever como novas proposições podem ser inferidas a partir de outras proposições que são assumidas como verdade. Os objetos em proposições lógicas de programação são representados por constantes ou variáveis. Uma constante é um símbolo que representa um objeto e uma variável é um símbolo que pode representar objetos diferentes em momentos diferentes (SEBESTA, 2012). **Prolog** é a linguagem lógica mais conhecida e utilizada (TATE, 2010).

Linguagens funcionais são baseadas em funções matemáticas que são a correspondência de um conjunto. Uma definição de função é descrita por uma expressão ou por uma tabela. As funções normalmente são aplicadas para um elemento específico do conjunto. Uma característica importante das funções matemáticas é o uso de recursão e expressões condicionais para controlar a ordem em que as expressões de correspondência são avaliadas (SEBESTA, 2012). Outras características igualmente importantes são: dados imutáveis, *higher-order functions* (funções de ordem superior), *lazy evaluation* e *pattern matching* (correspondência de padrão) (SEBESTA, 2009).

Ainda conforme Sebesta (2012), uma linguagem puramente funcional não utiliza variáveis, e sem variáveis não há iterações para efetuar repetições; isso é feito com o uso de recursão. Os programas são definições de funções e suas execuções consistem em avaliar as especificações das funções. As funções podem ser de ordem superior, i.e., usam funções como parâmetros e produzem uma função como resultado (WATT, 1990). As linguagens funcionais não têm efeitos colaterais (ou mudança de estado), pois definem um mesmo valor, dado o mesmo parâmetro - em outras palavras, têm transparência referencial.

Alguns exemplos de linguagens funcionais são: **Lisp**, **Scheme**, **Scala**, **ML**, **Miranda**, **Haskell** e **Erlang**. A seguir, estão descritas sucintamente importantes

linguagens de programação funcionais e, dada a importância de **Erlang** neste projeto de pesquisa, a sua especificação é apresentada com mais detalhes no capítulo 2 (ARMSTRONG et al., 1996).

A linguagem **Lisp** foi pioneira no paradigma funcional. Sua primeira versão era puramente funcional. Todas as linguagens criadas a partir de **Lisp** incluíram recursos das linguagens imperativas, como uso de variáveis, declarações de atribuição e iteração (SEBESTA, 2012). A linguagem **Lisp** é utilizada para cálculos simbólicos, lógica matemática e inteligência artificial (MCCARTHY et al., 1962).

Sebesta (2012) cita que existiam dois tipos de dados na versão original, que são os átomos e as listas. Os átomos são considerados constantes numéricas e símbolos em **Lisp**. As listas foram usadas como sua estrutura de dados, pois foram projetados para serem essenciais para o processamento de listas.

ML é uma linguagem de programação funcional que se difere de **Lisp** e seus dialetos em alguns aspectos significativos. Sua sintaxe é bem parecida com a linguagem Pascal. É fortemente tipada, ou seja, os erros de tipos são detectados em tempo de compilação (SEBESTA, 2012).

Scheme é uma linguagem criada a partir da linguagem **Lisp** mais simples e pequena. Caracteriza-se pelo tratamento dado às funções, como funções de alta ordem, onde tais funções podem ser os valores de expressões e elementos de listas e podem ser atribuídas a variáveis ou até mesmo serem passadas como parâmetros (SEBESTA, 2012).

Haskell é uma linguagem funcional pura. Por não ter variáveis e instruções de atribuição, ela não permite efeito colateral e não tem nenhuma característica das linguagens imperativas. Uma função que receber os mesmos parâmetros e entrada vai gerar sempre o mesmo resultado (TATE, 2000). **Haskell** tem semelhanças em relação à linguagem **ML** por utilizar uma sintaxe similar. Ela é fortemente tipada e usa o mesmo método de identificação de tipos. Existem duas características bem definidas em **Haskell**. A primeira delas é o uso de uma técnica de avaliação chamada preguiçosa (*lazy evaluation*). Nesta técnica nenhuma subexpressão ou função é avaliada até que o seu valor seja necessário. A segunda característica é que **Haskell** possui um método de definição de listas que permite a geração de um número indeterminado de números, por exemplo [2..], gera uma lista de inteiros a partir do número 2; essa característica é chamada de compreensões de lista (*list comprehensions*) (SEBESTA, 2012).

Em geral, aplicações desenvolvidas em linguagens funcionais são usadas em sistemas críticos, como redes de comutação telefônica; e devem fornecer alta qualidade, confiabilidade e eficiência. Neste contexto, atividades como validação, verificação e atividades de teste são necessárias e contribuem para melhorar a qualidade dos programas funcionais (BALAKRISHNAN; ANAND, 2009).

A atividade de teste de software visa encontrar defeitos não revelados que são responsáveis por erros durante a execução de programas. Para identificar e corrigir os defeitos existentes no software são utilizadas atividades de validação, verificação e teste. Essas atividades são apoiadas por técnicas que procuram garantir que o software corresponda às suas especificações (MYERS, 2004).

A validação avalia o software no final do desenvolvimento para verificar sua conformidade com a especificação. O conhecimento sobre o software é essencial para o sucesso da validação. A verificação determina em cada fase do processo de desenvolvimento se o software atende aos requisitos estabelecidos na fase anterior. A verificação é, geralmente, uma atividade mais técnica porque utiliza o conhecimento sobre os requisitos e também sobre as especificações do software (AMMANN; OFFUTT, 2008).

A atividade de teste consiste na execução de um programa inserindo dados de teste e comparando a saída de acordo com o resultado esperado na especificação. Caso o programa não atenda aos padrões, é considerado defeituoso. Os dados de teste são elementos do domínio de entrada que representam todas as possibilidades para os dados de entrada. Os casos de teste são a combinação de dados de teste com os resultados esperados (MYERS, 2004).

Os critérios dos testes são utilizados para representar os requisitos de teste a serem executados, e desta forma, guiar a seleção de dados de teste. Os critérios de teste são estabelecidos de acordo com a técnica de teste utilizada. Três técnicas de teste são amplamente utilizadas e conhecidas na literatura: estrutural, funcional e baseada em defeitos (BALAKRISHNAN; ANAND, 2009; MYERS, 2004; COPELAND, 2004).

O teste estrutural baseia-se no conhecimento da estrutura interna do programa, onde os caminhos lógicos são testados de acordo com os casos de teste disponíveis. Os critérios de teste estruturais usam o fluxo de controle e o fluxo de dados de um programa para derivar os requisitos de teste. O fluxo de controle representa as estruturas de programação usadas no código-fonte, como instruções

sequenciais, estruturas de desvio condicional e estruturas de repetição. O fluxo de dados representa as estruturas de dados utilizadas no programa, por meio do uso e definições de variáveis (MYERS, 2004).

A aplicação do teste de software deve ser ortogonal ao paradigma de programação usado: imperativo, funcional, lógico ou orientado a objetos; no entanto, a complexidade da atividade de teste está relacionada às características e dimensões do software a ser testado e também ao paradigma de programação utilizado.

Considerando este contexto de teste de software e linguagens funcionais, este trabalho tem como foco central a linguagem de programação funcional **Erlang**. **Erlang** foi projetada para programação concorrente, em tempo real e sistemas distribuídos tolerantes a falhas (ARMSTRONG et al., 2003). O objetivo principal da linguagem **Erlang** é melhorar a programação das aplicações telefônicas que são atualizadas em tempo de execução, e conseqüentemente, não permitiam a perda do serviço durante a atualização do código (CESARINI; THOMPSON, 2009).

A aplicação da atividade de teste para programas desenvolvidos em **Erlang** é importante e necessária em virtude de alguns tipos de erros que ocorrem e podem passar despercebidos por parte do programador. Por exemplo, erro de existência (CLOCKSin; MELLISH, 2003), ocasionados por chamadas de funções e processos inexistentes e também cláusulas para recebimento de mensagens inacessíveis.

O erro de existência (um exemplo é mostrado na seção 2.3) é tão sutil que muitos programadores **Erlang** não têm a consciência que pode ocorrer, e conseqüentemente, não são triviais de serem detectados. Esta situação ilustrada mostra a necessidade de teste de software para programas **Erlang**. Considerando este cenário e a carência de ferramentas que foi detectada em um estudo realizado e será mostrado no capítulo 3, vislumbrou-se a oportunidade em conduzir essa pesquisa no contexto de teste de software para a linguagem funcional **Erlang**.

1.2 Motivação

Considerando que a atividade de teste deve ser aplicada em todos os tipos de aplicações, independentemente do paradigma ao qual a aplicação foi desenvolvida, este trabalho tem como motivações:

- a) A carência de trabalhos que exploram a atividade de teste de uma forma

ampla para programas **Erlang**. Quando se diz ampla é no sentido de explorar as principais características da linguagem **Erlang** quanto aos aspectos relacionados ao fluxo de controle, fluxo de dados, fluxo de comunicação e sincronização.

- b) **Erlang** foi projetada para o uso da programação concorrente e este tipo de programação foi impulsionado pela demanda atual dos computadores com múltiplos núcleos. A aplicação do teste apresenta diferenças em relação à programação para códigos sequenciais, pois agrega um comportamento não determinístico ao programa desenvolvido, tornando este mais complexo e desafiador.
- c) Modelos, critérios e ferramentas de testes já existentes do paradigma imperativo, não são suficientes para aplicar no paradigma funcional. Existem muitas características, tais como, variáveis imutáveis, que exigem novas especificações para a aplicação de testes.
- d) E, por fim, a carência de modelos, critérios e ferramentas de teste estrutural, específicos para a linguagem **Erlang**. Tal carência fica mais evidente quando se observa a falta da atividade de teste para as principais características de **Erlang** empregadas em conjunto em um único programa, tais como: comunicação entre funções, comunicação entre processos, sincronização, concorrência, recursividade e tolerância a falhas.

1.3 Objetivos

Esse projeto de doutorado tem como objetivo principal a definição de critérios de teste estruturais para verificação dos códigos para aplicações sequenciais e distribuídas desenvolvidas na linguagem funcional **Erlang**. Os critérios de teste são baseados em cobertura de código e exploram as possíveis falhas relativas à comunicação entre funções, comunicação entre processos, sincronização, concorrência, recursividade e tolerância a falhas. Os critérios de testes têm como objetivo auxiliar na avaliação da adequação dos conjuntos de casos de teste e também nortear o testador para a seleção dos casos de teste que possam detectar defeitos. Neste contexto, o objetivo também é prover qualidade aos códigos Erlang por meio de tais critérios.

A partir deste objetivo central, um conjunto de objetivos específicos foi delineado:

- Levantamento de estudos relacionados à atividade de teste de software aplicada a linguagens funcionais de modo a demonstrar o atual estado da arte do contexto deste projeto de pesquisa.
- Especificação de um modelo de teste que estabeleça as principais características da linguagem **Erlang**, de modo a permitir a aplicação do teste estrutural. Isto é necessário porque a estrutura de codificação de linguagens funcionais difere do paradigma imperativo (não existe uma função principal, por exemplo).
- Implementação de uma ferramenta de teste para apoiar a efetiva aplicação dos critérios de teste definidos. O objetivo da ferramenta é validar os programas **Erlang** considerando aspectos sequenciais e concorrentes.

1.4 Metodologia

Para atingir o conjunto de elementos descritos como objetivos específicos e comprovar a tese proposta, as seguintes etapas metodológicas foram realizadas:

- a) Estudo minucioso da linguagem **Erlang** para entender em detalhes a sua sintaxe, semântica, processo de compilação e uso das aplicações e principalmente os tipos de erros existentes.
- b) Mapeamento sistemático que identificou os estudos disponíveis na literatura relacionados ao teste de software para linguagens funcionais. O mapeamento foi realizado em três etapas: planejamento, execução e apresentação dos resultados.
- c) O modelo de teste especificado foi baseado no modelo proposto por Souza et al. (2008b) e Sarmanho et al. (2008) que consideram o teste estrutural de programas concorrentes aplicado ao paradigma imperativo. Apesar do diferente paradigma, conceitos como definição de variável, uso predicativo e uso computacional são comuns e foram aproveitados.
- d) Uma família de critérios de teste estrutural foi definida para as principais características da linguagem **Erlang**, cujo objetivo é guiar a escolha de

- casos de teste para detectar erros relacionados ao fluxo de controle, fluxo de dados e passagem de mensagem.
- e) Desenvolvimento de uma ferramenta de teste, chamada **ValiErlang**, para oferecer suporte necessário à aplicação dos critérios de teste estrutural para programas **Erlang**. A ferramenta foi construída em cinco etapas, a primeira etapa foi desenvolvida em linguagem **Erlang** e as demais etapas foram desenvolvidas com o uso da linguagem **Java**.
 - f) Realização de estudos experimentais com diferentes programas para a aplicação dos critérios de teste por meio da ferramenta **ValiErlang**. A análise da cobertura realizada pela ferramenta contribui para comprovar a necessidade de testes para programas **Erlang** e a sua eficácia em revelar defeitos não tão simples de serem identificados sem o auxílio do teste proposto neste trabalho.

1.5 Estrutura da monografia

Esta tese está estruturada em sete capítulos. Neste primeiro capítulo é apresentada a introdução com as considerações iniciais, a motivação, os objetivos propostos, a metodologia para desenvolvimento de tese e informações sobre os grupos de pesquisa do trabalho.

O segundo capítulo apresenta a fundamentação teórica sobre linguagens funcionais. A linguagem **Erlang** é descrita com mais detalhes devido ao objetivo da tese. São apresentadas as principais características da linguagem, bem como exemplos de códigos e os processos de compilação e execução desses códigos.

No terceiro capítulo são descritos os conceitos básicos sobre teste de software com ênfase ao teste estrutural para programas sequencias e programas concorrentes. O detalhamento das etapas para a realização do mapeamento sistemático e os resultados também são descritos neste capítulo.

O quarto capítulo discorre sobre os conceitos básicos e o modelo de teste especificado para a definição dos critérios de teste. Os resultados da aplicação dos critérios de teste em um exemplo de programa **Erlang** também são apresentados.

O quinto capítulo apresenta de forma detalhada todos os aspectos da implementação da **ValiErlang**, que foi dividida em cinco módulos, o módulo *parse* que realiza a análise estática do programa e os módulos **Inst**, **Elem**, **Exec** e **Eval**

que realizam atividades distintas relacionadas ao teste estrutural de programas **Erlang**.

A aplicação dos critérios de testes em seis programas **Erlang** é apresentada no sexto capítulo. Considerações a respeito da análise realizada após a obtenção dos resultados dessa aplicação também são apresentadas.

No sétimo capítulo são apresentadas as considerações finais desta tese em conjunto com as contribuições, limitações e as publicações realizadas. Os trabalhos futuros também são discutidos e poderão ser conduzidos como continuação da presente pesquisa.

2 LINGUAGEM DE PROGRAMAÇÃO FUNCIONAL ERLANG

2.1 Considerações iniciais

Este capítulo aborda as principais características da linguagem **Erlang**. A história da linguagem também é descrita brevemente. A maioria dos conceitos é exemplificada por meio de exemplos de códigos para uma melhor ilustração. Algumas características como: funções anônimas, *higher-order functions* e *records* não foram abordadas porque não fizeram parte do escopo deste projeto.

As principais características da linguagem descritas neste capítulo são: *pattern matching*, *guards*, comunicação entre processos (primitivas *send* e *receive*), criação de processos em diferentes hosts, tolerância a falhas e recursão. Este capítulo também descreve os tipos de dados em **Erlang** e como efetuar a compilação e execução dos programas no interpretador de comandos.

Este capítulo tem como objetivo transmitir ao leitor as características de **Erlang** e como fazer uso da linguagem por meio de alguns exemplos de códigos. Espera-se que ao ler este capítulo o leitor possa entender a linguagem e o seu funcionamento. Este conhecimento é importante para os próximos capítulos que abordarão estudos relacionados ao teste de software para **Erlang** e também a especificação de critérios de teste estruturais baseados em um modelo com as principais características descritas neste capítulo.

2.2 Especificação da linguagem Erlang

Erlang é uma linguagem funcional desenvolvida pela *Ericsson Computer Science Laboratory*, a partir de 1986. **Erlang** teve como objetivo principal o desenvolvimento de aplicações na área de telecomunicações com características de distribuição, em grande escala e em softwares de controle de tempo real

(ARMSTRONG, 1996). A comunicação entre processos não utiliza memória compartilhada, é realizada através de trocas de mensagens (ARMSTRONG, 2007b).

2.2.1 História de Erlang

De acordo com ARMSTRONG (2007a), a linguagem **Erlang** começou como um meta-interpretador escrito em **Prolog** que adicionou o recurso de execução paralela de processos, habilidades para concorrência e melhorou o tratamento de erros.

Esses novos recursos fizeram a linguagem crescer e ser nomeada como **Erlang**, cujo nome é uma homenagem ao matemático dinamarquês Agner Krarup **Erlang** (1878-1929), criador da teoria de processos estocásticos em equilíbrio estatístico que foi muito utilizada na indústria de telecomunicações (ARMSTRONG et al., 1996).

A linguagem **Erlang** foi influenciada pela combinação de linguagens de sistemas concorrentes como **Ada**, **Modula** e **Chill**, linguagens funcionais como **ML** e **Miranda** e a linguagem lógica **Prolog** (ARMSTRONG, 2007a; CESARINI; THOMPSON, 2009).

Por volta de 1988, foi construída uma máquina abstrata chamada de *Joe's Abstract Machine* (JAM), a qual contava com primitivas para concorrência e manipulação de exceções. Nesta época o grupo cresceu para três pessoas, Mike Williams escreveu a máquina virtual para executar os códigos gerados pela JAM, Joe ARMSTRONG escreveu o compilador e Robert Virding escreveu as bibliotecas de suporte. Em 1990 Claes Vikström se juntou ao grupo e melhorou o desempenho da distribuição de processos para a linguagem (ARMSTRONG, 2007a).

Em 1992, em função de um problema de desempenho, Bogumil Husman desenvolveu a *Bogdans Erlang Abstract Machine* (BEAM), a qual substituiu a JAM em 1997 e tornou-se o sistema base para desenvolvimento de novos produtos em **Erlang** (ARMSTRONG, 1996; ARMSTRONG, 2007a).

Para o desenvolvimento de aplicações completas era necessário um ambiente de desenvolvimento completo. Em 1996 foi lançado um *framework* que contemplava todo o ambiente de execução, desde a codificação dos programas até sua execução, chamado de *Open Telecom Platform* (OTP). Em 1998 este *framework* tornou-se *open source* (ARMSTRONG, 2007a).

2.2.2 Características de Erlang

A principal vantagem de **Erlang** em relação às outras linguagens funcionais é que, desde a sua construção, a linguagem tem como características o desenvolvimento de aplicações concorrentes, distribuídas e em tempo real com características de alta disponibilidade e tolerantes a falhas (ARMSTRONG et al., 1996).

A seguir são detalhadas as principais características de **Erlang** encontradas em (ARMSTRONG et al., 1992; CESARINI; THOMPSON, 2009).

Construção de alto nível: **Erlang** é uma linguagem declarativa e, por princípio, tenta descrever o que deve ser calculado, ao invés de dizer como o valor é calculado. A definição de uma função usa a correspondência de padrão (*pattern matching*) para selecionar um entre diferentes casos e extrair componentes a partir de estruturas de dados complexas. **Erlang** não só combina padrão sobre os dados de alto nível, mas também uma sequência de *bits* que permitem funções de manipulação de protocolos.

O código descrito no Algoritmo 2.1 tem a definição de uma forma (no caso, um quadrado ou um círculo) e, de acordo com o tipo de forma que a função recebe, corresponde à cláusula correta da função e retorna a área correspondente.

Algoritmo 2.1 - Exemplo de Função que utiliza *Pattern Matching*.

```
area({quadrado, Lado}) -> Lado * Lado;  
area({circulo, Raio}) -> match:pi() * Raio * Raio.
```

Guards são muito utilizados em **Erlang** e considerados como uma extensão de *pattern matching*. *Guards* são condições avaliadas e que precisam ser satisfeitas para continuar a executar o conteúdo de uma cláusula.

O Algoritmo 2.2 mostra a aplicação de *guards* em um programa que calcula o fatorial de um número.

Algoritmo 2.2 - Exemplo de Função que utiliza *Guards*.

```
fatorial(Num) when Num == 0 -> 1;  
fatorial(Num) when Num > 0 -> Num * fatorial(Num - 1).
```

No código do Algoritmo 2.2 a primeira condição (*guard*) só é executada quando o valor da variável *Num* for igual a zero, para valores maiores que zero é executada a segunda condição.

A funcionalidade conhecida como "Compreensões de lista" combina os geradores de lista com os filtros, retornando outra lista que contém os elementos da primeira, após a aplicação dos filtros. A Figura 2.1 mostra alguns exemplos de compreensões de lista.

Figura 2.1 - Exemplo de uso de compreensões de lista.

A screenshot of an Erlang shell window titled "Erlang". The window has a menu bar with "File", "Edit", "Options", "View", and "Help". Below the menu bar is a toolbar with icons for file operations and a help icon. The main text area shows the following text:

```
Erlang R15B (erts-5.9) [smp:4:4] [async-threads:0]

Eshell V5.9 (abort with ^G)
1> Lista = [1,2,3,4,5,6,7,8,9,10].
[1,2,3,4,5,6,7,8,9,10]
2> [X || X <- Lista, X < 4].
[1,2,3]
3> [X || X <- Lista, X rem 2 == 0].
[2,4,6,8,10]
4> [X || X <- Lista, X rem 2 /= 0].
[1,3,5,7,9]
5> █
```

Fonte: Elaborada pelo autor.

O primeiro exemplo da Figura 2.1 faz um filtro para mostrar apenas os elementos da lista que são menores que 4 (comando 2), no segundo e terceiro exemplos mostram, respectivamente, os números pares (comando 3) e números ímpares (comando 4).

Processos Concorrentes: A concepção de **Erlang** é fundamentada pelo conceito de concorrência. A linguagem **Erlang** não fornece *threads* para compartilhar memória, portanto cada processo **Erlang** é executado em seu próprio espaço de memória e possui sua própria *heap* e *stack*. Assim processos não interferem uns com os outros, e consequentemente não há condição de disputa.

Os processos em **Erlang** são considerados leves, pois a *Virtual Machine* (VM) **Erlang** não cria uma *thread* de sistema operacional para cada processo criado. A própria VM cria, escalona e gerencia os processos, independentemente do sistema operacional. Tal procedimento resulta na criação de processos na casa dos microssegundos.

Passagem de Mensagem: Processos em **Erlang** comunicam-se através de passagem de mensagens. Tais mensagens podem ser qualquer valor de dados em **Erlang**. A operação de envio (*send*) é bloqueante, pois o processo continua o processamento apenas quando for realizada a cópia da mensagem do *buffer* da aplicação para um *buffer* do sistema operacional. Cada processo possui uma caixa de mensagens que armazena as mensagens recebidas e são obtidas de forma seletiva e bloqueantes (Culler; Gupta; Singh, 1997). Em ambientes distribuídos a ordem de recebimento das mensagens depende das condições do ambiente da rede. A Figura 2.2 mostra um exemplo de envio de mensagem entre dois processos.

Figura 2.2 - Exemplo de comunicação entre processos.

Fonte: Elaborada pelo autor.

No exemplo da Figura 2.2, o processo cliente envia uma mensagem ao processo **calc_area** contendo o seguinte conteúdo: **{quadrado, Lado}**. O operador **!** é equivalente à primitiva *send*. O processo **calc_area**, por sua vez, executa a primitiva *receive* e fica aguardando o recebimento de uma mensagem com o conteúdo **{quadrado, Lado}**. Ao receber uma mensagem, o processo **calc_area** envia uma mensagem de volta ao processo cliente com o conteúdo **{resp, Lado*Lado}**.

Da mesma forma que na criação de processos, a troca de mensagens fica em torno dos microssegundos, pois todo o contexto envolvido resume-se a cópia de dados a partir do espaço de memória de um processo para o outro.

Desta forma, **Erlang** reduz o tempo para a criação de processos e para a troca de mensagens, em relação às linguagens que utilizam memória compartilhada, semáforos e *threads* no nível de sistema operacional.

Atualização de código com o sistema em execução: Aplicações de telecomunicações necessitam de alta disponibilidade e as atualizações não podem interromper seu funcionamento. Para tanto, **Erlang** permite a substituição do código fonte com o sistema em execução. O sistema *runtime* mantém uma tabela global contendo os endereços para todos os módulos carregados. Quando uma nova versão de módulo é atualizada, os novos processos utilizam essa versão atualizada, enquanto os processos anteriores continuam a utilizar a versão antiga até terminarem. A linguagem permite a execução simultânea de duas versões do módulo em um sistema, assim torna-se possível a melhoria ou correção de bugs de um sistema sem interrupção.

Ambiente Distribuído: As aplicações **Erlang** podem ser executadas em um ambiente distribuído de forma transparente. Uma instância de uma VM **Erlang** é denominada nó. Um ou mais computadores podem executar vários nós independentemente de sua arquitetura de hardware ou sistema operacional. Processos podem ser criados em nós de outras máquinas, pois é possível o registro do processo na VM **Erlang**. As mensagens podem ser passadas entre os processos independentemente se eles estão no mesmo ou em diferentes nós. O Algoritmo 2.3 contém um trecho de código que cria processos em dois nós que pertencem a *hosts* diferentes.

Algoritmo 2.3 - Exemplo de criação de processos em *hosts* diferentes.

```
No1 = 'node1@urano',  
No2 = 'node2@marte',  
global:register_name(pid0, spawn(No1, calc, soma, [8, 18])),  
global:register_name(pid1, spawn(No2, calc, sub, [10, 8])).
```

No código do Algoritmo 2.3 foram criados dois processos: um registrado com o nome *pid0* e o outro com *pid1*, respectivamente nos *hosts*, urano e marte.

Tolerância a Falhas: **Erlang** possui um conjunto de bibliotecas que suportam o conceito de processos supervisores e trabalhadores e detecção de exceções e mecanismos de recuperação. Os processos criam *links* entre si para receberem notificações, em forma de mensagens, se o processo remoto, ao qual este processo

foi ligado, terminar. Essa monitoração é possível mesmo se o processo remoto estiver em execução em uma máquina diferente da rede. Caso um processo falhe, o mesmo pode ser reiniciado pelo seu processo supervisor. O Algoritmo 2.4 mostra uma função que faz a criação de um novo processo (linha) por meio da primitiva *spawn_link*. Essa primitiva cria um *link* entre os processos.

Algoritmo 2.4- Exemplo de criação de processos com *links*.

```
1  start_pong() ->
2    No1 = 'node1@urano',
3    global:register_name(pong, spawn_link(No1, m_pong,
pong, [])),
4    process_flag(trap_exit, true),
5    receive
```

Como pode ser observado no Algoritmo 2.4, um processo criado por meio da primitiva *spawn_link* envia um sinal para o processo criador quando ele finaliza de forma normal ou anormal. Esse sinal pode ser convertido em forma de mensagem com o uso do *flag trap_exit* definido como *true* (linha 4). Neste caso, os sinais de saída (*exit*) são convertidos em mensagens e podem ser monitorados pelo processo criador, por meio da primitiva *receive* (linha 5). Para receber um sinal de *exit*, o qual significa a finalização normal do processo criado, o processo criador aguarda pela *tupla* *{EXIT, From, Reason}* conforma cláusula *receive* na linha 6.

No exemplo do Algoritmo 2.4 apenas uma mensagem foi mostrada na tela (linha 7). Entretanto, se a aplicação necessita de tolerância a falhas pode ser tomada alguma ação específica, por exemplo, criar um novo processo.

Gerência de Memória: A VM ***Erlang*** gerencia automaticamente a memória, desobrigando o programador de se preocupar com a alocação que é feita pelo sistema *runtime* e com a liberação de memória que é feita pelo coletor de lixo (*garbage collector*). A área de memória de cada processo é coletada separadamente, quando um processo termina, sua memória é simplesmente recuperada.

A coleta separada resulta em tempos de coleta de lixo menores, o que contribui para se alcançar o cumprimento de prazos que aplicações de tempo real

necessitam. Caso a memória de todos os processos seja coletada ao mesmo tempo, sem um coletor de memória distribuído que possa fazer a coleta de lixo de maneira incremental, o sistema poderá parar por um longo período.

2.2.3 Tipos de Dados em *Erlang*

Erlang, por não ser uma linguagem puramente funcional, é uma linguagem com definição dinâmica de tipos e com semântica funcional, em grande parte, livre de efeitos colaterais (i.e. mudança de estado). As variáveis não são declaradas, o valor é vinculado na primeira ocorrência e, uma vez atribuído um valor, este não se altera mais. São oito tipos de dados simples e dois tipos de dados compostos existentes em ***Erlang*** (ARMSTRONG, 2007b; CESARINI; THOMPSON, 2009):

Inteiros: podem ser positivos ou negativos e representar números em bases diferentes de 10. Para isso a notação *Base#Valor* é utilizada. É possível também expressar caracteres como valores *American Standard Code for Information Interchange* (ASCII), quando a notação *\$Caracter* é utilizada. Alguns exemplos de inteiros são: -18, 15, 0, 2#110, 16#CD e \$A.

Floats: representam os números em ponto flutuante no formato duplo de 64 *bits* usando o padrão *Institute of Electrical and Electronics Engineers* (IEEE) 754 adotado em 1985 (Stallings, 2010). Podem ser representados na faixa de -10^{308} até 10^{308} .

Átomos: são usados para indicar valores distintos, chamados também de constantes literais, podem ser iniciados com letra minúscula ou delimitados entre aspas simples. Na primeira forma podem ser usadas letras, números e os caracteres arroba, ponto e *underline*; na segunda forma podem ser usados quaisquer caracteres.

Referências: as referências são objetos únicos, usados para a comparação de igualdade. As referências servem, por exemplo, para identificar ou confirmar a transmissão de mensagens entre dois processos em rede. Para fazer uso das referências é utilizada a *Built-In Function* (BIF) denominada *make_ref()*. BIF são funções internas (nativas) de ***Erlang***.

Binários: um binário é uma sequência de 8 bits. Existem primitivas em ***Erlang*** para compor e decompor binários e também para permitir uma eficiente entrada e saída de dados binários.

Pids: é a identificação de um processo em **Erlang**. Processos são criados pela primitiva *spawn* e recebem um PID.

Portas - portas são usadas em **Erlang** para permitir a comunicação entre processos, mesmo que tais processos sejam escritos em outras linguagens de programação. As portas são criadas pela BIF *open_port()*.

Funs: são funções anônimas, em algumas linguagens são conhecidas como expressões lambdas. *Funs* tem a sintaxe: *Z = fun(X) -> 2*X end*. As *Funs* podem ser usadas como funções de ordem superior (*higher-order functions*); podem receber funções como parâmetros e produzir uma função como resultado.

Tuplas: é um tipo de dado composto de um conjunto fixo de tipos de dados, os quais não precisam ser todos do mesmo tipo. Uma *tupla* é escrita por chaves, {...}, e os elementos que a compõem são separados por vírgula. Exemplos de *tuplas*: {123, abc}, {abc, {123, abc}, 123} e {cor, 'Azul'}.

Listas: representam um dos principais tipos de dados de **Erlang**. São semelhantes às *tuplas*, entretanto, são usadas para armazenar um número variável de tipos de dados. São escritos entre colchetes, [...], e seus elementos são separados por vírgulas. A sintaxe [] denota uma lista vazia.

Quando se trabalha com listas é possível referir-se ao primeiro elemento da lista e o restante da lista, quando o primeiro elemento da lista foi removido. Por convenção é possível usar o primeiro elemento da lista como o cabeça da lista e o restante da lista como a cauda, [Head|Tail].

2.2.4 Interpretador de Comandos Erlang

A Figura 2.3 mostra a tela inicial do interpretador de comandos **Erlang**. Com este interpretador é possível declarar variáveis para os tipos de dados, compilar os programas escritos em **Erlang** e executar as funções dos programas compilados. Só não é possível escrever suas próprias funções.

De acordo com Armstrong et al. (1996), os programas **Erlang** são compostos por módulos, cada módulo possui um determinado número de funções, as funções podem ser instanciadas a partir de outros módulos se forem exportadas pela diretiva *export*.

O código do Algoritmo 2.5 é um exemplo de um programa em **Erlang** que efetua o cálculo do dobro de um número.

Figura 2.3 - Interpretador de Comandos *Erlang*.

Fonte: Elaborada pelo autor.

Algoritmo 2.5 - Exemplo de um programa em *Erlang*.

```
-module(exemplo).  
  
-export([double/1]).  
  
double(X) -> 2*X.
```

Para a escrita dos programas é necessário utilizar as diretivas *module* e *export* que são obrigatórias em todo código. Toda diretiva de compilação começa com o sinal de menos; o nome dado à diretiva *module* é o mesmo nome dado ao nome do arquivo.

No código mostrado no Algoritmo 2.5, o nome do arquivo é **exemplo.erl** e é compilado pelo interpretador de comandos conforme ilustra a Figura 2.4. No código existe uma função chamada *double* que recebe um número como parâmetro. Nota-se que na diretiva *export* a função *double* é exportada, sendo possível executar pelo interpretador de comandos. O número depois da barra (/) indica a quantidade de parâmetros que precisa ser passada para a função.

Figura 2.4 – Exemplo de compilação e execução de um programa.

Fonte: Elaborada pelo autor.

A Figura 2.4 mostra o processo de compilação e execução do programa. Para compilar basta usar o comando `c` e informar o nome do programa entre parênteses. Para executar é necessário o nome do programa seguido pelo caractere dois pontos (`:`) e o nome da função. Depois são informados entre parênteses os parâmetros necessários para o processamento da função.

Um recurso eficiente com o uso de listas é a recursão, um exemplo de recursão com o uso de lista é ilustrado pelo código do Algoritmo 2.6.

Algoritmo 2.6 - Exemplo de um programa que utiliza lista e recursão.

```
-module(fat).  
-export([f/1]).  
  
f([Head|Tail]) ->  
    Head * f(Tail);  
f([]) -> 1.
```

O código descrito no Algoritmo 2.6 tem o mesmo objetivo do código descrito no Algoritmo 2.2, a diferença é que além da chamada recursiva de função, ele utiliza também o recurso das listas. A execução do código pode ser visualizada pela Figura 2.5.

Figura 2.5 – Exemplo de recursão com uso de lista.

Fonte: Elaborada pelo autor.

O cálculo é feito da seguinte forma:

HEAD = 1 e TAIL = [2,3,4], retorna $1 * \text{fat}([2,3,4])$

HEAD = 2 e TAIL = [3,4], retorna $2 * \text{fat}([3,4])$

HEAD = 3 e TAIL = [4], retorna $6 * \text{fat}([4])$

HEAD = 4 e TAIL = [], retorna $24 * \text{fat}([])$

[], retorna $24 * 1$ e termina o cálculo.

Outro recurso eficiente é a criação de uma nova lista a partir de uma lista existente, o que é feito com o caractere | (barra vertical) e ilustrado pela Figura 2.6.

Figura 2.6 – Exemplo de criação de lista a partir de uma lista existente.

Fonte: Elaborada pelo autor.

Na Figura 2.6 pode-se observar que Lista2 foi criada com cinco elementos, sendo quatro de Lista e um elemento novo, o número 5.

2.2.5 Programação Concorrente em Erlang

Erlang foi concebida para permitir a programação concorrente. Cada atividade concorrente em **Erlang** é chamada de processo e a única forma de interação entre os processos é através de passagem de mensagens. Os processos são gerenciados pela VM **Erlang**, com isso, podem ser gerenciadas diretamente por **Erlang** e não pelo sistema operacional (ARMSTRONG et. al, 1996; CESARINI; THOMPSON, 2009).

De acordo com Armstrong et al. (1996), **Erlang** possui três primitivas básicas para programação concorrente. A primitiva *spawn* tem como objetivo criar novos processos; as primitivas *send* e *receive* são usadas, respectivamente para o envio e recebimento de mensagens entre os processos.

A primitiva *spawn*(Módulo, Função, Argumentos) é uma BIF que necessita

avaliar a função exportada de um módulo com uma lista de argumentos que são os seus parâmetros. A BIF `spawn/3` (existem também as BIFs `spawn/1`, `spawn/2`, `spawn/4`) retorna a identificação do processo, i.e., seu PID. A Figura 2.7 ilustra a criação de um processo.

Figura 2.7 – Exemplo de criação de um processo.

The image shows a screenshot of the Erlang R15B shell window. The title bar says 'Erlang'. The menu bar includes 'File', 'Edit', 'Options', 'View', and 'Help'. Below the menu bar is a toolbar with icons for file operations and a help icon. The main text area displays the following text:

```
Erlang R15B (erts-5.9) [smp:4:4] [async-threads:0]  
Eshell V5.9 (abort with ^G)  
1> Pid = spawn(exemplo,double,[5]).  
<0.33.0>  
2> Pid.  
<0.33.0>  
3> █
```

Fonte: Elaborada pelo autor.

Como pode ser observado na Figura 2.7, a variável `Pid` recebeu o número de identificação do processo criado, no caso, `<0.33.0>`.

A primitiva `!` (*send*) tem a seguinte sintaxe: `Pid ! Msg`, onde `Msg` contém a mensagem a ser enviada para o processo identificado por `Pid`. Cada processo possui uma caixa postal (um buffer) que armazena todas as mensagens recebidas na ordem de chegada. A Figura 2.8 mostra um exemplo de envio de mensagem.

Figura 2.8 – Exemplo de envio de mensagem para um processo.

The image shows a screenshot of the Erlang R15B shell window. The title bar says 'Erlang'. The menu bar includes 'File', 'Edit', 'Options', 'View', and 'Help'. Below the menu bar is a toolbar with icons for file operations and a help icon. The main text area displays the following text:

```
Erlang R15B (erts-5.9) [smp:4:4] [async-threads:0]  
Eshell V5.9 (abort with ^G)  
1> Pid = self().  
<0.31.0>  
2> Pid ! 'hello world'.  
'hello world'  
3> flush().  
Shell got 'hello world'  
ok  
4> █
```

Fonte: Elaborada pelo autor.

Na Figura 2.8 observa-se o envio da mensagem `'hello world'` para o processo `Pid` (comando 2). O interpretador de comandos **Erlang** também é um processo e pode ser usado para enviar ou receber mensagem. No exemplo foi utilizada a BIF

self() que retorna o PID do processo atual. Este PID do processo atual foi armazenado na variável *Pid* para receber mensagens (comando 1). Por fim, a BIF *flush()* é usada para descarregar as mensagens da caixa postal e mostrá-las no interpretador de comandos (comando 3).

A sintaxe da primitiva *receive* pode ser observada no Algoritmo 2.7.

Algoritmo 2.7 - Exemplo de sintaxe para recebimento de mensagens.

```
receive
    Cláusula1 -> Ação1;
    Cláusula2 -> Ação2;
    ...
end
```

No Algoritmo 2.7, a *Cláusula1* e *Cláusula2* são padrões que são comparados com as mensagens que estão chegando. Quando uma mensagem é recebida e uma correspondência é encontrada, a mensagem é removida da caixa postal e sua ação correspondente é executada. O processo que executou a primitiva de comunicação *receive* fica bloqueado até que uma mensagem seja recebida e esta seja comparada com alguma cláusula da primitiva *receive*.

Em Armstrong et al. (1996) tem um exemplo de um programa que troca mensagens entre dois processos. Este código é exibido no Algoritmo 2.8.

Algoritmo 2.8 - Exemplo de troca de mensagem entre dois processos (ARMSTRONG et al., 1996).

```
-module(echo) .
-export([start/0]).
start() ->
    spawn(echo, loop, []).
loop() ->
    receive
        {From, Message} ->
            From ! 'Hello World',
            loop()
    end.
```

O código do Algoritmo 2.8 possui duas funções. A função *start* cria um novo processo que instancia a função *loop*. A função *loop* usa a primitiva *receive* e aguarda o recebimento de mensagens. Ao receber uma mensagem de outro processo que contém um PID e uma mensagem (tupla {From, Message}), este processo *loop* envia uma mensagem de volta para o processo com a mensagem 'Hello World'. A Figura 2.9 mostra a troca de mensagens entre dois processos, de acordo com o código descrito no Algoritmo 2.8.

Figura 2.9 – Exemplo de troca de mensagem entre dois processos.

```
Erlang R15B (erts-5.9) [smp:4:4] [async-threads:0]
Eshell V5.9 (abort with ^G)
1> Pid = self().
<0.31.0>
2> Id = echo:start().
<0.34.0>
3> Id ! {Pid, hello}.
{<0.31.0>,hello}
4> flush().
Shell got 'Hello World'
ok
5>
```

Fonte: Elaborada pelo autor.

Na Figura 2.9 observa-se o processo *Pid* (comando 1) e a criação do processo *Id*, que executou a função *start* do módulo *echo* (comando 2). Em seguida, o processo *Pid* envia uma mensagem para o processo *Id* com sua identificação e a mensagem *hello* (comando 3). O processo *Id*, ao receber a mensagem, vai comparar com o padrão e retornar a mensagem 'Hello World' para o processo *Pid*. Por fim, a BIF *flush()* é usada para checar se a mensagem de retorno enviada pelo processo *Id* foi recebida (comando 4).

2.3 Considerações Finais

Este capítulo abordou de maneira resumida diferentes paradigmas de programação, apresentando suas definições e principais características. Na sequência foram apresentadas as características das principais linguagens funcionais, com ênfase na linguagem **Erlang**. Em relação à **Erlang** foram descritas

sua história, desde a criação pelos engenheiros da Ericsson até a linguagem tornar-se *open source*, suas principais características, detalhes de implementação e execução e exemplos de códigos com primitivas da linguagem utilizadas para a programação concorrente.

Erlang apresenta diversas vantagens em relação às demais linguagens de programação. Pode-se destacar a atualização de código com o sistema em operação; manipula com eficiência um grande número de processos; apropriado para desenvolvimento de aplicações distribuídas e de tempo real; e robustez para permitir a execução contínua de um sistema.

Em contraposição às suas vantagens, **Erlang** também apresenta as seguintes desvantagens: as aplicações **Erlang** não têm bom desempenho para aplicações que efetuam cálculos numéricos; e a linguagem não é indicada para o desenvolvimento de aplicações com interfaces gráficas, os recursos que as bibliotecas da linguagem fornecem estão muito aquém das demais linguagens.

Algumas características, como por exemplo: dados imutáveis, *higher-order functions*, *lazy evaluation* e *pattern matching* não estão presentes nas linguagens imperativas. Além dessas, existem outras características como tolerância a falhas, chamadas de funções, chamadas recursivas e comunicação entre processos que as aplicações desenvolvidas em **Erlang** apresentam e que diferem em alguns aspectos das linguagens imperativas. Neste contexto, existe a necessidade de aplicação das atividades de teste para detectar possíveis falhas e também contribuir para melhorar a qualidade dos programas funcionais.

Um exemplo relacionado a falhas pode ser demonstrado em relação ao erro de existência. Este tipo de erro pode estar relacionado com intercalação na execução dos processos envolvidos (*interleaving*) (Christakis; Gotovos; Sagonas (2013). O Algoritmo 2.9 ilustra esta situação.

Este programa exemplo tem um possível erro de existência se o processo gerado na linha 5 (envia uma mensagem na linha 9 e finaliza) terminar antes que o processo principal tente registrar sua identificação (linha 5). Este registro resultará em um erro, pois o processo que seria registrado já não existiria.

Como resultado desta exceção, o processo principal vai falhar e sair de forma anormal. Este erro é tão sutil que muitos programadores **Erlang** não têm a consciência que pode ocorrer, e conseqüentemente, não são triviais de serem detectados.

Algoritmo 2.9 - Exemplo de um programa que envolve dois processos e um erro de existência.

```
1 -module(test).  
2 -export([pong/0,ping/1]).  
3 pong() ->  
4     Self = self(),  
5     register(ping_pong, spawn(test,ping,[Self]));  
6     receive ping -> ok end.  
7  
8 ping(PongPID) ->  
9     PongPID ! ping.
```

O exemplo de erro exibido anteriormente e outros que serão abordados ao longo do projeto justificam e motivam a necessidade de aplicação das atividades de teste para melhorar a qualidade dos programas **Erlang**. O próximo capítulo apresenta os conceitos sobre teste de software com foco em teste estrutural para programas sequenciais, teste estrutural para programas concorrentes e um levantamento dos trabalhos relacionados com o teste de linguagens funcionais, a fim de contextualizar e fundamentar o estado da arte do tema pesquisado neste doutorado.

3 TESTE DE SOFTWARE

3.1 Considerações iniciais

Este capítulo apresenta os conceitos básicos sobre teste de software e seus tipos, trazendo uma especificação mais detalhada sobre o teste estrutural com critérios de teste baseados em fluxo de controle e fluxo de dados. É apresentada também uma família de critérios relacionados ao teste estrutural para programas concorrentes (SOUZA et al., 2008b).

Este capítulo também apresenta um mapeamento sistemático na área de teste para linguagens funcionais para identificar trabalhos relacionados. Este estudo é importante para identificar o contexto de contribuições científicas relacionadas ao projeto, além de fornecer uma visão geral para o teste de linguagens funcionais e *insights* que foram utilizados para orientar o desenvolvimento deste projeto.

Para finalizar são apresentadas falhas que são tipicamente encontradas em programas **Erlang**, tais falhas normalmente não são triviais de serem detectadas pelos desenvolvedores e espera-se que as ferramentas de teste auxiliem na detecção das mesmas durante o processo de teste. Os conceitos descritos neste capítulo são essenciais para a proposta deste projeto de doutorado, relacionada à especificação de critérios de teste estruturais baseados em um modelo de teste para prover qualidade aos códigos **Erlang** e uma ferramenta denominada **ValiErlang** para dar suporte aos critérios especificados.

3.2 Conceitos e tipos de teste

A atividade de teste de software é uma das fases da engenharia de software e tem como finalidade examinar o comportamento de programas por meio de sua execução, tentando encontrar defeitos nos mesmos. A complexidade da atividade de

teste está relacionada com as características e dimensões do sistema a ser testado (MYERS, 2004). Essa complexidade pode levar a diversos problemas normalmente ocasionados por erro humano. Diversas atividades, conhecidas como “Validação, Verificação e Teste” ou VV&T, são utilizadas para identificar e sanar os defeitos existentes nos softwares. Tais atividades são amparadas por técnicas que procuram garantir que o software esteja de acordo com sua especificação (DELAMARO; MALDONADO; JINO, 2007).

Validação é o processo que avalia o software no final do desenvolvimento para verificar a conformidade de tal software com a especificação. O conhecimento do mesmo é imprescindível para o sucesso da validação. Verificação determina, a cada fase do processo de desenvolvimento, se o software atende aos requisitos estabelecidos na fase anterior. A verificação é geralmente uma atividade mais técnica, pois usa o conhecimento sobre o requisito e também sobre a especificação do software (AMMANN; OFFUTT, 2008).

Os termos defeito, erro e falha possuem definições semelhantes, porém, distintas entre si (DELAMARO; MALDONADO; JINO, 2007). Por defeito (*fault*) entende-se o processo ou definição de dados incorretos que podem ocasionar um erro durante a execução de um programa, normalmente produzido por um engano (*mistake*) do programador. O erro (*error*) é um estado inconsistente ou inesperado do programa. Ele é gerado pela existência de um defeito e pode levar a uma falha. A falha (*failure*): ocorre quando o resultado produzido pela execução do programa ou modelo não está de acordo com o resultado esperado, ou seja, a falha ocorre quando o defeito se propaga para a saída.

A atividade de teste é dividida em fases para facilitar a execução e iniciá-la antes mesmo do sistema ser finalizado. Assim, o teste começa com unidades menores de um programa até abranger o mesmo por completo. As principais fases da atividade de teste são (DELAMARO; MALDONADO; JINO, 2007; PRESSMAN, 2011):

- Teste de unidade: foca os esforços na menor unidade do programa e identifica defeitos dentro de funções e métodos. Defeitos encontrados nesta fase diminuem o risco de encontrar defeitos nas demais fases;
- Teste de integração: é uma técnica utilizada detectar defeitos quando as unidades que compõem o programa são integradas, assim é possível saber se as unidades vão funcionar integradas e sem defeitos;

- Teste de sistema: engloba o sistema após a integração das unidades, seu propósito é verificar se as funcionalidades foram implementadas de acordo com os requisitos. O teste de sistema pode englobar aspectos de segurança e desempenho.

A atividade de teste consiste na execução de um programa inserindo dados de teste e comparando a saída de acordo com o resultado esperado na especificação. Caso não esteja de acordo, considera-se que o programa possui um defeito. Dados de teste são elementos do domínio de entrada que representam todas as possibilidades para os dados de entrada. Casos de teste são as combinações dos dados de teste e os resultados esperados (DELAMARO; MALDONADO; JINO, 2007).

Segundo Delamaro, Maldonado e Jino (2007), executar todos os casos de testes do domínio de entrada é impraticável, dada a sua quantidade. Para minimizar essa quantidade de entradas de teste, pode ser utilizado um subconjunto do domínio de entrada que é construído através de uma seleção aleatória de casos de teste ou através do estabelecimento de subdomínios que categorizam os casos de testes de acordo com os requisitos de teste. Os critérios de teste são utilizados para representar os requisitos de teste e, dessa forma, nortear a seleção de dados de teste.

Os critérios de teste são estabelecidos em função da técnica de teste empregada. Três técnicas de teste são amplamente utilizadas e conhecidas na literatura: estrutural, funcional, baseada em defeitos e baseados em modelos (COPELAND, 2004; DELAMARO; MALDONADO; JINO, 2007; MYERS, 2004; Pezzè e Young, 2008; PRESSMAN, 2002).

O teste estrutural utiliza o conhecimento da estrutura interna dos programas para gerar/selecionar dados de teste que cubram caminhos, comandos, desvios condicionais, definições e uso de variáveis. O teste estrutural também é conhecido como teste de caixa branca e, devido à importância do mesmo neste projeto, o mesmo será detalhado na Seção 3.2.1.

A técnica conhecida como teste funcional (ou caixa preta), baseia-se na especificação e requisitos do software, não requer o conhecimento das estruturas internas de programação e de dados e o objetivo da mesma é verificar se os requisitos funcionais do programa estão em conformidade com os objetivos que foram especificados. Exemplos de critérios pertencentes à técnica e o teste funcional são:

a) Particionamento em classes de equivalência: Neste critério de teste funcional, o domínio de entrada é dividido em classes válidas ou classes inválidas e apenas um caso de teste de cada uma das classes é testado. Assim, se um caso de teste de determinada classe revela um defeito, os demais elementos dessa classe também devem revelar o mesmo defeito. Essa abordagem consequentemente reduz o número de casos de teste que devem ser testados.

b) Análise do valor limite: este critério explora as condições limites das classes de equivalência. São selecionados casos de teste que testam esses limites, pois têm a maior probabilidade de encontrar defeitos;

c) Grafo de causa-efeito: explora a combinação dos dados de entrada identificando causas e efeitos na especificação. As causas correspondem às entradas ou qualquer evento que gere uma resposta do sistema em teste. Já os efeitos correspondem às saídas, mudanças de estado ou qualquer resposta.

O teste baseado em defeitos, também conhecido como teste de mutação, cria mutantes do programa original e introduz defeitos típicos do processo de implementação. Esses mutantes são chamados de mutantes e para cada mutante um subdomínio de entrada é determinado. O testador escolhe casos de teste que testam o comportamento do programa mutante em relação ao programa original. Caso a saída gerada pelo programa mutante seja diferente do programa original, é dito que o mutante está morto. Caso as saídas sejam iguais, o mutante está vivo. No primeiro caso, o dado de teste teve êxito para identificar o mutante. No segundo caso o mutante pode permanecer vivo por duas razões: o dado de teste não conseguiu detectar o defeito inserido ou os programas são equivalentes e nenhum dado de teste pode detectá-los. Esse teste tem como principal objetivo avaliar a qualidade dos casos testes escolhidos.

Teste baseado em modelos consiste em utilizar modelos para produzir casos de teste que possam revelar discrepâncias de comportamento. Um modelo pode ser extraído a partir do código do programa ou de sua especificação e para sua efetividade é necessário estar correto. Os modelos semiformais podem ser representados por diagramas de classes e objetos enquanto que modelos formais podem ser representados por gramáticas ou através de máquinas de estados finitos (MEF). Os modelos têm como objetivo permitir a geração de casos de teste e modelar o comportamento do sistema a ser testado. Alguns métodos de geração de casos de teste baseados em MEFs são: *Transiton Tour* (TT); *Unique input/output*

(UIO); *Distinction Sequence* (DS) ou Sequência de Distinção; e W.

3.2.1 Teste estrutural

O teste estrutural é baseado no conhecimento da estrutura interna do programa, onde os caminhos lógicos são testados de acordo com os casos de teste disponíveis. Os critérios de teste estrutural utilizam o fluxo de controle e o fluxo de dados de um programa para derivar os requisitos de teste. O fluxo de controle representa as estruturas de programação utilizadas no código fonte, tais como instruções sequenciais, estruturas de desvio condicionais e estruturas de repetição. O fluxo de dados representa as estruturas de dados utilizadas no programa, através das definições e usos de variáveis (MYERS, 2004).

O teste estrutural representa um programa P através de um Grafo de Fluxo de Controle (GFC) ($P = (N, E, s)$). Neste GFC os nós pertencentes a **N** representam um bloco de comandos sequenciais (blocos têm no mínimo um comando) e as arestas indicam prováveis fluxos de controle entre tais blocos através das arestas pertencentes a **E**. Um GFC possui apenas um nó de entrada (*s*) e um nó de saída (*o*), onde *s* e *o* pertencem a **N**. Um nó que possui várias arestas de saída representa um desvio predicativo (MYERS, 2004). A Figura 3.1 é composta por: (a) programa que realiza o cálculo do máximo divisor comum entre dois valores, denominado *Greatest Common Divisor* (GCD); (b) GFC referente ao programa GCD; e (c) o grafo de definição e uso do programa GCD.

A partir de GFC são definidos caminhos que podem ser encontrados no programa, as definições para caminho são (DELAMARO; MALDONADO; JINO, 2007):

caminho é composto por uma sequência finita de nós. Na Figura 3.1(b) os nós 5, 6, 2 e 7 formam um caminho;

caminho simples é um caminho onde todos os nós são diferentes com exceção do primeiro e do último nó. Por exemplo, os nós 2, 3, 4, 6 e 2 da Figura 3.1(b);

caminho livre de laço é quando todos os nós são diferentes sem exceção. Por exemplo, os nós 1, 2, 3, 4 e 6 da Figura 3.1(b);

caminho livre de definição é o caminho entre a definição e uso de uma variável. Um exemplo são os nós 1, 2 e 3 da Figura 3.1(c);

caminho completo acontece quando o primeiro nó de um caminho é o nó de entrada e o último é o nó de saída do grafo. Na Figura 3.1(b), um exemplo é representado pelos nós 1, 2 e 7.

Figura 3.1 – (a) Programa que calcula o máximo divisor comum (GCD) entre dois valores; (b) GFC do programa GCD; (c) Grafo de definição e uso do GCD.

Fonte: (Sarmanho, 2009).

Os critérios estruturais podem ser classificados em: critérios baseados na complexidade, critérios baseados em fluxo de controle e critérios baseados em fluxo de dados (DELAMARO; MALDONADO; JINO, 2007). Essa Seção descreve apenas os dois últimos critérios.

CrITÉRIOS baseados em fluxo de controle são caracterizados pelo controle da execução do programa. Existem algumas métricas de teste que podem ser definidas em relação ao fluxo de controle e ao fluxo de dados do GFC. Os critérios de fluxo de controle mais conhecidos no contexto de teste estrutural são: Todos-Caminhos, Todas-Arestas e Todos-Nós.

- Todos-Caminhos: define que todos os caminhos do programa precisam ser executados, é o critério mais custoso e é considerado impraticável;
- Todas-Arestas: exige que cada aresta do grafo, ou seja, cada desvio condicional do programa seja executado ao menos uma vez;
- Todos-Nós: requer que a execução do programa passe em cada vértice do GFC pelo menos uma vez.

O grafo de definição e uso (Def-Use Graph) ilustrado na Figura 3.1(c) é uma extensão do GFC com informações relativas ao fluxo de dados do programa, tal grafo caracteriza associações entre partes do programa é atribuído um valor a uma variável (definição da variável) e partes na qual esses valores são utilizados (uso da variável) (RAPPS; WEYUKER, 1985).

Os critérios baseados em fluxo de dados utilizam as associações entre a definição e uso de uma variável para a derivação dos casos de teste (Delamaro; Maldonado; e Jino, 2007). A definição de uma variável ocorre quando ela recebe um valor. Um uso é uma ocorrência de uma variável que não recebe um novo valor, sendo dividido em uso computacional (c-uso) e uso predicativo (p-uso). O c-uso impacta uma computação realizada. No p-uso afeta o fluxo de controle do programa, em estruturas de decisão como *while*, *for*, *repeat* ou *if*.

Weyuker (1990) propôs uma família de critérios baseados em fluxo de dados:

Todas-Definições: requer que cada definição de variável seja utilizada pelo menos uma vez, independentemente se for um c-uso ou p-uso;

Todos-Usos: requer que todas as associações entre uma definição de variável e seus posteriores c-usos e p-usos sejam executados pelos casos de teste, através de um caminho livre de definição. Os critérios que representam esta categoria são: todos os usos predicativos (todos-p-usos) e todos os usos computacionais (todos-c-usos).

Todos-du-Caminhos: requer que todos os caminhos livres de definição sejam executados entre uma definição de variável e seus posteriores c-usos e p-usos.

Maldonado (1991) propôs os critérios Potenciais-Usos, os quais requerem associações independentemente da ocorrência de um uso a uma variável. Caso haja um caminho livre de definição entre uma definição da variável e um nó ou aresta, é dito um potencial-uso da variável.

3.3 Teste estrutural de programas concorrentes

A atividade de teste para aplicações concorrentes são mais críticas que as aplicações sequenciais, isso acontece devido às características de não determinismo, condições de disputas, sincronização e comunicação. O não determinismo permite que execuções diferentes de uma aplicação concorrente com os mesmos dados de entrada possam gerar saídas diferentes e corretas (SOUZA et al., 2008b).

Os modelos e critérios de teste estruturais para aplicações concorrentes têm como objetivo identificar defeitos relacionados a tais características e são baseados na comunicação e sincronização de aplicações concorrentes, o que diferem dos critérios de teste dos programas sequenciais (SOUZA et al., 2007).

Souza et al. (2008b) propôs um modelo de teste que adaptam os critérios estruturais baseados em fluxo de controle e fluxo de dados para aplicações concorrentes e inserem novos critérios baseados no fluxo de comunicação e sincronização das aplicações concorrentes. Tal modelo considera um número fixo n e conhecido de processos paralelos que executa seu próprio espaço de memória. A comunicação é feita de duas formas: comunicação ponto-a-ponto, onde um processo envia mensagem para outro processo com o uso das primitivas *send* e *receive*; e comunicação coletiva, onde um processo pode enviar uma mensagem para todos os processos da aplicação ou para um grupo específico.

Para um conjunto de processos $Prog = p^0, p^1, \dots, p^{n-1}$ que constitui um programa paralelo, é construído para cada processo p um GFC seguindo os mesmos conceitos de um programa sequencial. Cada GFC de p possui um único nó de entrada e um único nó de saída, um nó n pode ou não estar associado a uma função de comunicação (*send* ou *receive*). As funções de comunicação possuem os seguintes parâmetros: *send*(s, d, m) e *receive*(s, d, m); isso significa que processo s envia (ou recebe) uma mensagem m para o (ou do) processo d .

Um Grafo de Fluxo de Controle Paralelo (GFCP) é construído a partir do GFC

de cada p existente e pelas arestas de comunicação dos processos paralelos. Um nó i em um processo p é representado pela notação n^p_i . Dois subconjuntos de N são definidos: N_s são os nós com funções de envio de mensagem e N_r são os nós que recebem mensagens. Para cada $n^p_i \in N_s$, um conjunto R^p_i é associado contendo possíveis nós que recebem a mensagem enviada pelo nó n^p_i . As arestas podem ser: arestas intraprocessos (E_i) que são arestas internas de um processo; e arestas interprocessos (E_s): que representam a comunicação entre processos.

Um caminho π no GFC de um processo p é chamado intraprocesso se não contém nenhuma aresta interprocessos. Um caminho π que contém pelo menos uma aresta interprocessos é chamado caminho interprocessos. Para programas escritos em ambiente de passagem de mensagem, um novo tipo de definição de variável é possível: a definição decorrente da execução de uma função de comunicação tal como um *receive*. O conjunto de variáveis definidas em um nó n^p_i é dado por $def(n^p_i)$. O uso de uma variável x ocorre quando x é referenciada em uma expressão. No contexto de programas concorrentes podem ocorrer três formas de uso de variáveis:

Uso computacional (c-uso) ocorre em uma computação relacionada a um nó n^p_i do GFCP;

Uso predicativo (p-uso) ocorre em uma condição (predicado), associado a comandos de fluxo de controle em arestas intraprocessos (n^p_i, n^p_j) do GFCP;

Uso comunicacional (s-uso) ocorre em comandos de sincronização (*send* e *receive*) em arestas interprocessos $(n^{p^1}_i, n^{p^2}_j) \in E_s$.

Com essas informações é possível caracterizar as três associações requeridas pelos critérios estruturais aplicados a programas concorrentes:

- Associação c-uso: dada pela tripla (n^p_i, n^p_j, x) em que $x \in def(n^p_i)$, e n^p_j possui um c-uso de x e existe um caminho livre de definição em relação a x de n^p_i para n^p_j ;
- Associação p-uso: dada pela tripla $(n^p_i, (n^p_j, n^p_k), x)$ em que $x \in def(n^p_i)$, e (n^p_j, n^p_k) possui um p-uso de x e existe um caminho livre de definição em relação a x de n^p_i para (n^p_j, n^p_k) ;
- Associação s-uso: dada pela tripla $(n^p_i, (n^{p^1}_j, n^{p^2}_k), x)$ em que $x \in def(n^{p^1}_i)$, e $(n^{p^1}_j, n^{p^2}_k)$ possui um s-uso de x e existe um caminho livre de definição em relação a x de $n^{p^1}_i$ para $(n^{p^1}_j, n^{p^2}_k)$.

As associações p-uso e c-uso são associações intraprocessos, onde a

definição e o uso de x ocorrem no mesmo processo p_1 . Uma associação s-uso é interprocesso pois requer a participação de outro processo p_2 . Tais associações permitem verificar erros de envio e recebimento de mensagens. Assim como ocorre com as associações s-usos e comunicação interprocessos, outros tipos de associações inteprocessos podem ser caracterizadas:

- Associação s-c-uso: dada pela quintupla $(n_i^{p_1}, (n_j^{p_1}, n_k^{p_2}), n_l^{p_2}, x_k^{p_1}, x^{p_2})$ tal que existe uma associação s-uso $(n_i^{p_1}, (n_j^{p_1}, n_k^{p_2}), x^{p_1})$ e uma associação c-uso $(n_k^{p_2}, (n_l^{p_2}, x^{p_2}))$;
- Associação s-p-uso: dada pela quintupla $(n_i^{p_1}, (n_j^{p_1}, n_k^{p_2}), (n_l^{p_2}, x_m^{p_2}), x^{p_1}, x^{p_2})$ tal que existe uma associação s-uso $(n_i^{p_1}, (n_j^{p_1}, n_k^{p_2}), x^{p_1})$ e uma associação c-uso $(n_k^{p_2}, (n_l^{p_2}, x_m^{p_2}))$.

Em Souza et al. (2008b) foram definidas duas famílias de critérios de teste estrutural para programas concorrentes em ambientes de passagem de mensagem: critérios baseados em fluxo de controle e no fluxo de comunicação e os critérios baseados em fluxo de dados e em passagem de mensagens.

Critérios baseados em fluxo de controle e no fluxo de comunicação

É possível testar cada GFC individualmente com o uso dos critérios todos-nos e todas-arestas, isso é possível devido à execução de um único processo. Para testar a comunicação presente no GFCP são definidos os seguintes critérios:

- Todos-nós-s: requer que todos os nós do conjunto N_s sejam executados pelo menos uma vez pelo conjunto de casos de teste;
- Todos-nós-r: requer que todos os nós do conjunto N_r sejam executados pelo menos uma vez pelo conjunto de casos de teste;
- Todos-nós: requer que todos os nós do conjunto N sejam executados pelo menos uma vez pelo conjunto de casos de teste;
- Todas-arestas-s: requer que todas as arestas do conjunto E_s sejam executadas pelo menos uma vez pelo conjunto de casos de teste;
- Todas-arestas: requer que todas as arestas do conjunto E sejam executadas pelo menos uma vez pelo conjunto de casos de teste;

O critério todos-caminhos do GFC de p (para $p = 0, \dots, n-1$) e todos-caminhos de GFCP equivale a executar todas as combinações possíveis dos caminhos nos GFC de p , tais critérios são impraticáveis devido ao problema de explosão de estados.

Critérios baseados em fluxo de dados e em passagem de mensagens

- Todos-defs: requer que para cada nó de n^p_i e cada x em $def(n^p_i)$ uma associação definição-uso (p-uso ou c-uso) com relação a x seja executada pelo conjunto de casos de teste;
- Todos-defs/s: requer que para cada nó de n^p_i e cada x em $def(n^p_i)$ uma associação s-c-uso ou s-p-uso com relação a x seja executada pelo conjunto de casos de teste. Se não existir associação s-c-uso ou s-p-uso em relação a x , é requerida qualquer outro tipo de associação intraproceto em relação a x ;
- Todos-c-usos: requer que todas as associações c-usos sejam executadas pelo conjunto de casos de teste;
- Todos-p-usos: requer que todas as associações p-usos sejam executadas pelo conjunto de casos de teste;
- Todos-s-usos: requer que todas as associações s-usos sejam executadas pelo conjunto de casos de teste;
- Todos-s-c-usos: requer que todas as associações s-c-usos sejam executadas pelo conjunto de casos de teste;
- Todos-s-p-usos: requer que todas as associações s-p-usos sejam executadas pelo conjunto de casos de teste.

Os critérios de teste propostos por Souza et al. (2008b) são aplicados na ferramenta ValiPar que contempla técnicas estáticas e dinâmicas de análise de cobertura de código para o teste estrutural de aplicações concorrentes. A ValiPar valida programas paralelos escritos em PVM (SOUZA et al., 2008a), MPI (HAUSEN et al., 2007), Pthreads (SARMANHO et al., 2008) e BPEL (ENDO et al., 2008).

A ValiPar utiliza o conceito de sessões de teste, assim é possível testar um determinado programa paralelo; parar a atividade de teste e retomá-la mais tarde. A

ferramenta permite criar sessões de teste, salvar e executar os dados de teste e avaliar a cobertura de um determinado critério de teste; os critérios estruturais implementados na ferramenta são: todos-nos, todos-nos-r, todos-nos-s, todas-arestas, todas-arestas-s, todos-c-usos, todos-p-usos e todos-s-usos.

A Figura 3.2 mostra a arquitetura da Valipar que possui quatro módulos principais:

ValiInst: extrai o fluxo de controle e de dados e efetua a instrumentação do programa. Neste módulo é obtido o PGFC que possui o fluxo de controle, definição e uso de variáveis e a comunicação e sincronização, respectivamente entre processos e threads.

Figura 3.2 – Arquitetura da ferramenta Valipar.

Fonte: (Souza et al., 2008b).

ValiExec: realiza a execução do programa paralelo de acordo os casos de teste fornecidos. Durante as execuções são armazenadas as entradas e as saídas do programa, os caminhos executados e as sequências de sincronização estabelecidas.

ValiElem: gera os elementos requeridos para os critérios de cobertura do teste com base no PGFC gerado pela ValiInst e define associações de definição e uso de variáveis para obter os elementos requeridos baseados no fluxo de dados.

ValiEval: avalia a cobertura alcançada pelos casos de teste, para isso, utiliza

os elementos requeridos e os compara aos caminhos executados.

3.4 Mapeamento sistemático em teste de linguagens funcionais

Esta seção apresenta os resultados do mapeamento sistemático realizado para recuperar os principais estudos relacionados com o teste de software para linguagens funcionais.

De acordo com Kitchenham e Charters (2007), o mapeamento sistemático permite identificar as evidências ou falta de evidências sobre um assunto específico. Assim é possível direcionar o foco de futuras pesquisas e identificar áreas para estudos primários a serem realizados.

O mapeamento sistemático conduzido neste trabalho foi realizado em três fases: planejamento, execução e apresentação dos resultados. O mapeamento sistemático teve como foco um estudo sobre teste de software dentro do contexto da linguagem funcional *Erlang*.

3.4.1 Planejamento

A fase de planejamento foi realizada de acordo com Kitchenham e Charters (2007), sendo elaboradas três questões para atender aos objetivos do mapeamento sistemático:

a) Quais problemas relacionados às linguagens funcionais dificultam a aplicação da atividade de teste dos seus programas desenvolvidos? O interesse é identificar os principais problemas do paradigma funcional que tornam a atividade de teste mais complexa. Neste ponto não se restringiu o mapeamento à Linguagem *Erlang*, permitindo que perspectivas diferentes fossem consideradas e pudessem contribuir com as investigações em curso.

b) Como são feitos os testes para aplicações escritas em linguagens funcionais? O objetivo é encontrar trabalhos que aplicam alguma metodologia de teste e saber quais critérios de teste são utilizados. Novamente, optou-se por não restringir a busca à *Erlang*.

c) Existem ferramentas que auxiliam os testes das aplicações escritas em linguagens funcionais? A identificação de ferramentas de teste que apoiam a atividade de teste é importante para saber se elas são capazes de identificar

defeitos.

Estratégia de Busca

Estudos preliminares são essenciais para a formulação da *string* de busca. Foram utilizados livros específicos de linguagens funcionais, pesquisas em conferências e workshops de linguagens funcionais para obtenção dos principais autores, artigos da área e termos relacionados com teste. O conjunto inicial de palavras-chaves definidas foram: “functional language”, “testing” e “tools”. Durante buscas preliminares novas palavras-chaves foram adicionadas e combinadas com o uso de operadores booleanos AND e OR para melhorar as buscas.

De acordo com os estudos preliminares foi estabelecida a seguinte *string* de busca:

((**Erlang** OR functional language OR lisp OR haskell) AND (software testing OR structural testing OR mutation testing OR functional testing OR blackbox OR whitebox OR tools OR test OR quickcheck OR dialyzer))

Para a obtenção dos trabalhos resultantes das *strings* de busca foram utilizadas três máquinas de busca, listadas na Tabela 3.1. As pesquisas foram realizadas em Março de 2016.

Tabela 3.1 – Máquinas de busca utilizadas.

Máquina de Busca	Endereço eletrônico
ACM Digital Library	http://dl.acm.org/
IEEE Xplore Digital Library	http://ieeexplore.ieee.org/Xplore/home.jsp
SCOPUS	http://www.scopus.com/

Fonte: Elaborada pelo autor.

Os critérios de inclusão (IC) foram definidos a fim de obter os estudos primários que possam responder as questões de pesquisa:

IC1: Apresentam métodos e modelos de teste para aplicações escritas em linguagens funcionais;

IC2: Propõem ferramentas e pesquisas para o contexto de linguagens funcionais;

IC3: Aplicação de estudos de casos, métodos e modelos em ferramentas de testes.

Seleção de Estudos

Os seguintes critérios de exclusão (EC) foram definidos para filtrar os estudos primários que não estão relacionados com as questões de pesquisa:

EC1: Não apresentam abordagens de teste relacionadas com linguagens funcionais;

EC2: Apresentam abordagens de teste relacionadas com hardware;

EC3: Apresentam um tutorial de uso de uma ferramenta de teste;

Extração dos Dados

Na fase de extração de dados foi definido um formulário para registrar informações obtidas a partir dos estudos primários (KITCHENHAM; CHARTERS, 2007). O formulário de extração de dados fornece informações, tais como: fonte de extração, o título, o ano e os autores. O procedimento para extração de dados ocorreu após a seleção dos estudos. Para cada estudo analisado foi escrito um resumo para facilitar a documentação das respostas para as perguntas da pesquisa e estão disponíveis¹ para consulta.

3.4.2 Execução

A fase de execução foi realizada com auxílio do software State of the Art through systematic Review (StArt). Foram selecionados um total 562 artigos de acordo com as *strings* de busca. Os totais estão na Tabela 3.2 e separados por máquinas de busca.

A fase preliminar (fase 1) consistiu na identificação dos estudos duplicados (feito automaticamente pelo software), dados não relevantes (por exemplo,

¹ <http://labes.icmc.usp.br/~alexandre/mapping.pdf>

proceedings) e leitura do título, resumo e palavras-chave. Depois da aplicação dos critérios de inclusão e exclusão permaneceram 50 estudos. Ainda na fase preliminar (fase 2), foram selecionados os estudos através da leitura da introdução e conclusão. Após a aplicação dos critérios, 28 estudos relevantes permaneceram.

Tabela 3.2 – Número de artigos selecionados por máquina de busca.

Máquinas de Busca	Qtde	Incluídos	Fase 1 Excluídos	Incluídos	Fase 2 Excluídos	Incluídos	Fase 3 Excluídos
ACM	75	22	53	11	11	9	2
IEEE	171	4	167	1	3	1	-
SCOPUS	316	24	292	16	8	11	5
Total	562	50	512	28	22	21	7

Fonte: Elaborada pelo autor.

Na seleção final (fase 3), os estudos foram inteiramente analisados (leitura completa). Com a revisão completa foram selecionados 21 estudos primários. De acordo com os estudos preliminares realizados foram incluídos mais 5 estudos. Tais estudos não foram indexados pelas máquinas de busca escolhidas ou foram publicados em *workshops* locais das respectivas universidades.

3.4.3 Resultados do mapeamento sistemático

Nesta seção são apresentadas as respostas para cada questão de pesquisa definidas na Seção 3.4.1.

Quais problemas relacionados às linguagens funcionais dificultam a aplicação da atividade de teste dos seus programas desenvolvidos?

Widera (2006) cita que na programação funcional surgem dois problemas para aplicação do teste estrutural: a geração de um GFC e a aplicação dos critérios de cobertura para um GFC. A geração de um GFC é mais complicada devido à existência de funções de ordem superior. Tais funções podem receber e enviar funções como parâmetros. A criação dinâmica de funções torna o fluxo de controle imprevisível e, com isso, o fluxo de dados pode influenciar no fluxo de controle. Tal influência necessita da análise do fluxo de dados durante a geração do GFC para determinar o conjunto de arestas que são cobertas por tais dados.

Widera (2004) explica que a aplicação dos critérios de cobertura para um GFC tem uma complexidade maior (quando comparada às linguagens imperativas) devido às estruturas de controle das linguagens funcionais como, por exemplo, no uso de *lazy evaluation*. A recursão, um recurso muito utilizado pelas linguagens funcionais, também necessita de uma análise do fluxo de dados mais específica às necessidades deste tipo de linguagem. Outro aspecto a ressaltar na aplicação dos critérios de cobertura é que a distinção do uso predicativo (p-uso) de uma variável do uso computacional (c-uso), não são apropriados pois suas estruturas condicionais utilizam *pattern matching* ao invés de predicados.

Neste mesmo contexto, Tóth e Bozó (2011) citam que o objetivo de um grafo de fluxo de dados (GFD) é determinar até que ponto do código a definição de uma variável pode alcançar. Widera (2004) também especifica que para a análise do fluxo de dados deve-se usar definição e alcance de variáveis ou invés da definição e uso de variáveis.

Isto se deve ao fato das variáveis serem imutáveis nas linguagens funcionais. Para o fluxo de dados é importante analisar um valor a partir de sua primeira definição até o seu último uso (WIDERA, 2006). Considerando que um valor pode passar por diversas avaliações e variáveis, torna-se importante fazer seu rastreamento (trace).

Como são feitos os testes para aplicações escritas em linguagens funcionais?

No contexto de especificação da linguagem e modelos de teste foram obtidos cinco estudos. Tóth e Bozó (2011) apresentam o *Semantic Program Graph* (SPG), um modelo para representar as informações semânticas e sintáticas de códigos **Erlang**. Podem ser construídos GFD, GFC e grafos de dependência (GD) a partir das regras do SPG. Tais grafos são analisados, e no caso do GFD, o objetivo é determinar até que ponto do código que as definições podem alcançar. O GFC representa todos os possíveis caminhos de execução ou avaliação das expressões do código fonte para cada dado de teste. Já o GD representa a dependência de dados ou de controle em funções recursivas e comunicação entre diferentes nós através de passagem de mensagens. O GD é usado para extrair partes do código fonte e detectar componentes que podem ser paralelizados de forma eficiente e sem altos custos de sincronização. Os grafos são integrados com a ferramenta RefactorErl, que analisa código fonte e faz a extração de partes dos códigos **Erlang**.

Em Tóth et al. (2010) os autores aplicam o mesmo modelo em teste de regressão com o objetivo de reduzir os casos de testes depois do código modificado. Para isso é utilizado um grafo de dependência de comportamento. Com isso poderiam ser selecionados apenas os casos de testes que são afetados pela mudança de comportamento; os demais não necessitam serem testados novamente. Em Tóth e Bozó (2012) o objetivo é o mesmo, entretanto, é utilizado um grafo de dependência de controle para selecionar os casos de teste.

Silva, Tamarit e Tomás (2012) fazem uma adaptação GD, para um grafo denominado **Erlang Dependency Graph** (EDG). EDG utiliza um modelo baseado nas características da linguagem funcional **Erlang**. O EDG representa a dependência de dados ou controle em chamadas de funções e funções recursivas. Criaram a ferramenta Slicerl que faz extração de partes de um programa **Erlang**, baseado no modelo proposto por este estudo.

Guo, Derrick e Walkinshaw (2009) definem um modelo para transformação de uma função ou um programa **Erlang** em uma estrutura de árvore que representa o programa original em teste, mantendo suas funcionalidades. Tal árvore é denominada *Complete Functional Binary Tree* (CFBT). Cada nó da árvore corresponde a um predicado do programa original, assim predicados que podem estar escondidos dentro de funções do programa original são detectados. Isso torna mais viável a definição dos dados de teste para fazer a cobertura de todos os nós.

Os cinco estudos apresentados não abordaram no modelo de teste os programas concorrentes, os quais representam a essência da linguagem **Erlang**. Tais estudos ficaram restritos apenas à especificação de um modelo de teste e à geração de grafos e também não especificaram critérios de testes.

Em relação aos problemas descritos na questão de pesquisa anterior foram obtidos três estudos. Widera (2004) descreve um modelo que abrange um subconjunto de funções **Erlang** e propõe uma definição de GFC a partir deste modelo. Tal modelo abrange apenas programas sequenciais **Erlang**. Em Widera (2005b) o modelo é estendido para abranger funções de ordem superior, como descrito anteriormente, o fluxo de controle de tais funções pode depender do fluxo de dados. Widera (2005a) complementa o modelo para atender programas concorrentes em **Erlang** através da geração de processos e passagem de mensagens. Um algoritmo para elaboração do grafo de fluxo concorrente também foi especificado.

Com relação à especificação de critérios de testes, dois estudos foram obtidos. Widera (2004) propõe um conjunto de critérios de cobertura baseados em fluxo de dados adaptados para programação funcional. Tais critérios baseiam-se nas associações de definição e uso das variáveis (*du-pair*). Widera (2005b) introduz o critério *du-chain* que é uma sequencia de vários *du-pair*. Tal critério é dividido em cinco critérios para atender aos requisitos da linguagem funcional (vide Tabela 3.3).

Um estudo foi obtido sobre a necessidade de rastreamento do fluxo de dados. Widera (2004) descreve as propriedades e implementação de um protótipo de um interpretador para GFC. A partir dos dados de teste o interpretador realiza um *trace* no código fonte e o objetivo é avaliar partes do GFC que foram cobertos pelos dados de teste. O estudo não avalia os critérios de cobertura, apenas faz uma comparação do tempo de execução dos dados de teste com e sem o interpretador em pequenos exemplos de códigos.

Tabela 3.3 – Critérios baseados nas definições e uso das variáveis.

Critérios de Cobertura	Sigla	Definição
Aliasing aware <i>du-chain</i>	<i>a-aware du-chain</i>	utilizado em cópia de variáveis
Structure aware <i>du-chain</i>	<i>s-aware du-chain</i>	utilizado em construção de listas e tuplas
Result aware <i>du-chain</i>	<i>r-aware du-chain</i>	utilizado em chamadas de função
Freeze aware <i>du-chain</i>	<i>f-aware du-chain</i>	utilizado em geração dinâmica de função (fun)
Message aware <i>du-chain</i>	<i>m-aware du-chain</i>	utilizado em envio e recebimento de mensagens

Fonte: Elaborada pelo autor.

Em Tasharofi et al. (2013) os autores apresentam uma abordagem para testar o comportamento não-determinístico de programas **Scala** baseados em atores (concorrentes). Esta abordagem utiliza três critérios de cobertura relacionados à primitiva de comunicação *receive* para programas concorrentes, um algoritmo para gerar escalonadores viáveis para aumentar a cobertura e uma técnica para execução determinística. Os autores do trabalho Le et al. (2014) apresentam novos operadores de mutação para programas funcionais e também descrevem **MuCheck**, uma ferramenta de teste de mutação para programas **Haskell**.

Para resumir, com esta questão de pesquisa nós observamos contribuições que exploram principalmente os testes estruturais para linguagens funcionais. Estes artigos apresentam proposições para representar e extrair informações relevantes para testes de programas funcionais.

Em geral, os estudos não apresentam, de uma forma completa, as técnicas e

o processo de teste. De fato, as poucas metodologias de teste disponíveis na literatura e pertencentes ao escopo deste trabalho, não descrevem o processo de extração de informações relevantes do programa, a avaliação de cobertura dos critérios propostos e também estudos experimentais que analisem a eficácia dos critérios de teste. Resumindo, não foram encontradas metodologias robustas que possam ser avaliadas, inclusive por meio de uma ferramenta de teste. Este gap foi considerado significativo para a condução das pesquisas neste projeto.

Existem ferramentas que auxiliam os testes das aplicações escritas em linguagens funcionais?

Em Nagy e Víg (2008) os autores apresentam resultados de uma pesquisa com desenvolvedores de sistemas **Erlang** para levantar e disseminar o uso das ferramentas que estão sendo utilizadas e os problemas enfrentados pela comunidade **Erlang** com tais ferramentas. A pesquisa teve como foco testes baseados em modelos e propriedades e desenvolvimento orientado a testes. Dentre as ferramentas citadas pelos participantes da pesquisa podem-se destacar, **Dialyzer**, **QuickCheck**, **EUnit**, **Wrangler** e **RefactorErl**. Os participantes citaram a dificuldade de uso das ferramentas devido à falta de tutoriais e exemplos muito simples na documentação. As ferramentas também falham na geração dos resultados, por exemplo, aplicando caso de testes em ferramentas baseadas em modelo é difícil para obter informações sobre a cobertura e o que foi realmente testado.

Dialyzer, que é uma ferramenta de análise estática que identifica discrepâncias de software, como erros de tipos definidos, teve seu código inacessível devido a algum erro de programação ou testes desnecessários. Neste contexto, Christakis e Sagonas (2011) apresentaram uma técnica para detectar erros de passagem de mensagem em **Erlang**, considerando a criação dinâmica de processos e comunicação baseada em passagem de mensagem assíncrona. A técnica utiliza análise estática com o auxílio da ferramenta **Dialyzer** para construir um grafo de comunicação a partir dos GFCs gerados pelo **Dialyzer**. Assim, o GFC de cada função corresponde a um vértice do grafo de comunicação. Tal grafo é percorrido para detectar eventuais erros de passagem de mensagens.

Quviq QuickCheck é uma ferramenta de testes baseada em propriedades. Podem ser especificadas propriedades, escritas em código fonte **Erlang**, que são

testados por casos de testes gerados aleatoriamente pela ferramenta. Arts, Castro e Hughes (2008) apresentam uma metodologia para testar tipos de dados de acordo com as definições dos usuários usando **Quviq QuickCheck**. Foi definido um modelo, e a partir deste modelo, verificar se os tipos de dados das aplicações atendem às especificações definidas no modelo.

As ferramentas **Wrangler** e **RefactorErl** têm como objetivo refatorar um programa. Refatoração é uma técnica para alterar o código fonte original, seja para otimizar ou para organizar, mantendo suas funcionalidades. Um programa refatorado não deve apresentar mais ou menos defeitos que o programa original. Podem ser feitas análises entre o código original e o código alterado do programa para certificar se as funcionalidades permanecem iguais.

Neste contexto, Li e Thompson (2007) utilizaram **Quviq QuickCheck** para automatizar o teste das refatorações realizadas pelo **Wrangler**. Para cada programa alterado foram escritas propriedades que deveriam ser satisfeitas e, caso elas não sejam cumpridos, há a indicação de erro na implementação ou propriedade. Para cada teste foram gerados 100 dados de teste automaticamente pela ferramenta.

Em Li e Thompson (2009, 2011) foi desenvolvida e integrada na ferramenta **Wrangler** uma técnica para detecção de trechos de códigos que são sintaticamente idênticos. Os autores utilizaram análise sintática e decomposição de código e o objetivo da técnica é remover trechos de códigos duplicados para melhorar a compreensão e manutenção do código.

Gotovos, Christakis e Sagonas (2011) desenvolveram a ferramenta Concuerror cujo objetivo é auxiliar o processo de *Test Driven Development* (TDD). Tal ferramenta detecta, dado um programa e um conjunto de teste, erros relacionados à concorrência que podem ocorrer durante a execução de testes; por exemplo, condição de disputa e *deadlocks*.

D'Oswaldo, Kochems e Ong (2012) apresentam o software **Soter**, um analisador de programa e verificador de módulos **Erlang**. **Soter** utiliza a combinação de análise estática para verificar as propriedades especificadas pelo usuário. **Soter** extrai um modelo abstrato a partir de um conjunto de propriedades e depois verifica se as propriedades estão satisfeitas. A técnica de abstração é precisa para verificar um conjunto de propriedades de segurança, como exclusão mútua e limitação do *mailbox*.

Widera (2006) cita a ferramenta **Cover** que é distribuída como uma biblioteca

de ferramenta de **Erlang. Cover** aplica teste de cobertura para códigos **Erlang** que analisa as linhas individuais do código fonte; por trabalhar diretamente no código fonte não aplica análise de fluxo de dados. Esta ferramenta, no entanto, fica restrita à cobertura de expressões sem considerar as correspondências entre as funções de um programa, por exemplo, chamadas de funções.

O teste concólico (acrônimo das palavras concreto e simbólico) é uma técnica de teste de software que combina a execução concreta de um programa (dada uma entrada específica) com execução simbólica (gerando novas entradas de teste que oferecem uma melhor cobertura do caminho do que a geração aleatória de casos de teste). Neste contexto, Vidal (2014) introduziu uma formalização da execução tanto concreta como simbólica para a linguagem funcional e concorrente **Erlang**, entretanto o procedimento de teste concólico foi apenas esboçado. Giantsios, Papaspyrou e Sagonas (2015) demonstram a aplicação do teste concólico para um subconjunto da linguagem **Erlang**, que suporta *pattern matching*, listas, recursões e *higher-order functions*. Além disso, apresenta o **CutEr**, uma ferramenta que implementa esta técnica de teste.

Taylor e Derrick (2015) apresentam uma ferramenta de análise de cobertura de *Multiple Condition/Decision Coverage* (MC/DC) para programas **Erlang**, denominada **Smother**. A ferramenta limita-se apenas à cobertura de MC/DC e não considera outros aspectos da linguagem **Erlang**, inclusive não considera as expressões contidas nas estruturas de decisões cobertas.

Os números da Tabela 3.2 mostram que a maioria dos artigos encontrados não satisfazem aos critérios de inclusão. Grande parte dos trabalhos relacionados ao teste em hardware, não estão relacionados diretamente com uma metodologia de teste ou apresentam um tutorial de uso de uma ferramenta. Outro fator interessante é que muitos artigos não estão relacionados com o objetivo do mapeamento, mas contêm nas palavras-chaves as *strings* de busca.

Com relação aos artigos selecionados pode-se verificar que alguns artigos abordam a técnica de refatoração; em Li e Thompson (2009; 2011) é utilizada uma técnica para detectar clone em programas e fazer refatoração; em Bozó e Tóth (2011) é avaliado o impacto da refatoração através de casos de testes.

Análise estática de código foi abordada em Tóth e Bozó (2011), nestes trabalhos foi feita a análise estática para fatiar programas e detectar funções que poderiam ser paralelizadas; em Silva, Tamarit e Tomás (2012) os autores utilizam a

análise estática para fatiar programas; Sagonas, Silva e Tamarit (2013) usam análise estática para detectar erros de tipo; em Christaki e Sagonas (2011) é utilizada análise estática para detectar erros de passagem de mensagens com auxílio do software **Dialyzer**.

Outros tipos de testes também foram abordados: teste baseado em especificações (D'OSUALDO; KOCHEMS e ONG, 2012); teste baseado em propriedades para checar tipos de dados (ARTS; CASTRO e HUGHES, 2008); teste unitário e geração aleatória de testes (ARTS; THOMPSON, 2010).

Gotovos; Christakis e Sagonas (2011) propuseram uma ferramenta para identificar erros de concorrência com vistas a auxiliar o TDD.

Trabalhos que abordam o teste estrutural estão descritos em: (GUO; DERRICK; WALKINSHAW, 2009) que propõem a transformação de um programa como uma árvore binária, e a partir da árvore, derivam-se dados de teste para fazer a cobertura de todos os nós; Widera (2004) propõe uma definição de GFC para programas sequenciais **Erlang**; Widera (2005b) propõe um conjunto de critérios de cobertura baseados em fluxo de dados para programas sequenciais; e Widera (2005a) apresenta uma definição de GFC para programas concorrentes em **Erlang**, mas não aborda critérios que envolvam vários processos e a comunicação entre eles. Tais trabalhos evidenciam uma carência em relação ao teste estrutural e indica a necessidade de contribuições nessa direção.

3.5 Falhas típicas em **erlang**

Esta seção tem como objetivo listar falhas típicas que ocorrem em programas Erlang. O estudo para identificação das falhas foi baseado em trabalhos relacionados, manual de referência de Erlang e relatórios de erros. A seguir tem uma descrição de cada uma das falhas, que baseado em (CLOCKSIN; MELLISH, 2003) e (ERLANG BUGS, 2016), foram classificadas em erros de concorrência, erros de tipo, erros de domínio e erros de existência.

3.5.1 Erros de concorrência

Condição de disputa - ocorre quando os acessos a um recurso (em uma aplicação concorrente) compartilhado não são escalonados ou sincronizados

corretamente (DO; AJAKIYE, 2009 ; CHRISTAKIS; SAGONAS, 2011). Por exemplo, a execução uma primitiva *send* de um processo ocorrer antes da execução da primitiva *receive* de outro processo.

Deadlocks - ocorre quando dois ou mais processos bloqueiam um ao outro (CRONQVIST, 2004; PAWEL, 2005; CHRISTAKIS; SAGONAS, 2011). Por exemplo, dois processos executando a primitiva *receive*.

3.5.2 Erros de tipo

Argumento errado - ocorre quando uma chamada de função com um argumento com um tipo de dados errado ou quando o argumento foi definido de maneira incorreta (ERICSSON AB, 2016; ERLANG BUGS, 2016; CHRISTAKIS; SAGONAS, 2011). Por exemplo, uma chamada é feita para uma função que recebe uma *string* e converte-a para um átomo, entretanto, é passado um número ao invés da *string*.

Argumento invertido - ocorre em uma definição de função em que dois de seus argumentos estão na ordem errada (DO; AJAKIYE, 2009). Por exemplo, inverter os argumentos dia e mês em uma função que calcula o dia da semana.

3.5.3 Erros de domínio

Processo runaway - ocorre quando um processo consome recursos sem fazer qualquer trabalho útil (CRONQVIST, 2004). Por exemplo, quando o processo entra em um *loop* infinito.

Omissão de guard - ocorre quando não é implementada uma proteção necessária para o funcionamento correta de uma função (DO; AJAKIYE, 2009). Por exemplo, uma função que calcula a divisão entre dois números. Neste caso, é preciso um *guard* para verificar se o segundo é igual a 0.

Falha na correspondência de guards, cláusulas case, if e receive - ocorre quando não há um predicado correspondente (*pattern matching*) quando se avalia

um *guard*, uma cláusula *case*, cláusula *if* ou cláusula *receive* (ERICSSON AB, 2016), (LINDAHL; SAGONAS, 2004), (CHRISTAKIS; SAGONAS, 2011).

Cláusulas *case*, *if* e *receive* inacessíveis - ocorre quando uma cláusula contém um padrão que nunca vai ser combinado. Não há um predicado correspondente (*pattern matching*) quando se avalia um *guard*, uma cláusula *case*, cláusula *if* ou cláusula *receive* (CHRISTAKIS; SAGONAS, 2011).

Falha na expressão de correspondência - ocorre quando o valor do lado direito de uma expressão de correspondência de padrão não coincide com o valor no lado esquerdo (ERICSSON AB, 2016). Por exemplo, comparar uma *tupla* composta por dois elementos com uma *tupla* composta por três elementos.

Falha na cláusula da função correspondente - ocorre quando o padrão do argumento de uma função não corresponde a nenhuma cláusula dentro dessa função (ERICSSON AB, 2016; ERLANG BUGS, 2016). Por exemplo, enviar o valor 100 como parâmetro de entrada de uma função (por meio de uma variável) e internamente utilizar cláusulas *if* ou *case* que combinam valores diferentes de 100.

3.5.4 Erros de existência

Função inexistente - ocorre quando uma chamada de função é feita para uma função que não existe (ERICSSON AB, 2016; LINDAHL; SAGONAS, 2004).

Processo inexistente - ocorre quando uma mensagem é enviada ou uma ligação é feita com um processo que não existe (ERICSSON AB, 2016).

3.6 Considerações finais

Este capítulo abordou os conceitos e tipos de teste de software com ênfase ao teste estrutural. Em seguida foi apresentado o contexto de teste estrutural para programas concorrentes. Em relação a teste de aplicações concorrentes foram apresentados o modelo de teste proposto por Souza et al. (2008b) e as características da ferramenta **Valipar**.

Na sequência foi apresentado um estudo sobre testes para linguagens funcionais por meio de um mapeamento sistemático apresentado. Como resultado, o mapeamento identificou alguns trabalhos que apresentaram modelos de teste, critérios de teste e ferramentas de teste para linguagens funcionais. Por fim, o capítulo apresentou as falhas que são tipicamente encontradas em programas **Erlang**. Tais falhas não são detectadas em tempo de compilação e pode não ser trivial sua detecção por parte do desenvolvedor e também pelas ferramentas de teste.

Através dos resultados obtidos neste mapeamento, observa-se uma falta de critérios de teste e ferramentas que apoiem a atividade de teste estrutural. Foi observada também uma carência de estudos experimentais para analisar e avaliar os critérios de teste no contexto de linguagens funcionais. Estudos experimentais são fundamentais para apresentar informações sobre custo de sua aplicação e também avaliar a qualidade dos critérios de teste.

É baseado nesta carência que o contexto deste projeto de doutorado está inserido, com foco no desenvolvimento de um modelo de teste e critérios de testes estruturais baseados em fluxo de controle, fluxo de comunicação e fluxo de dados, os quais são descritos no próximo capítulo.

Os capítulos desenvolvidos até aqui foram importantes para fundamentar a pesquisa. O próximo capítulo especifica um modelo de teste que abstrai as principais características de programas escritos em linguagem **Erlang**. A partir deste modelo, foi definida uma família de critérios de teste que servem para selecionar e avaliar casos de teste de forma a aumentar as possibilidades de provocar falhas que foram aqui listadas ou, quando isso não ocorre, estabelecer um nível elevado de confiança no software.

4 CRITÉRIOS DE TESTE ESTRUTURAL PARA PROGRAMAS *ERLANG*

4.1 Considerações iniciais

Este capítulo detalha o modelo de teste estrutural que representa as informações extraídas de programas **Erlang**, considerando aspectos relacionados ao fluxo de controle, fluxo de dados e fluxo de comunicação. Na sequência são descritos os critérios de teste propostos em Souza et al. (2008b), as adaptações necessárias para **Erlang** e também é proposta uma família de novos critérios que consideram características específicas de **Erlang**, tais como: comunicação entre funções, chamadas recursivas de funções e tolerância a falhas. Por fim, é mostrado um exemplo para ilustrar todas as informações descritas pelo modelo de teste e também a eficácia dos critérios propostos em relação à revelação de falhas.

Ao final deste capítulo, espera-se que o leitor entenda como as informações foram extraídas pelo modelo de teste e porque os critérios foram definidos. As informações apresentadas neste capítulo são a base para o desenvolvimento dos próximos capítulos que apresentarão, a ferramenta **ValiErlang** que apoia à aplicação dos critérios descritos e estudos experimentais com programas **Erlang** para analisar a cobertura dos critérios propostos.

4.2 Modelo de teste estrutural para programas **Erlang**

O modelo de teste proposto captura as informações sobre o fluxo de controle, fluxo de dados e fluxo de comunicação de programas **Erlang**. Tais informações são relativas à comunicação entre funções, comunicação entre processos por meio de passagem de mensagens, sincronização entre processos, concorrência e recursividade. Todas essas informações recuperadas do código fonte **Erlang** e

armazenadas no modelo de teste serão usadas como base a aplicação dos critérios de teste.

O modelo de teste considera um número n fixo e conhecido de processos concorrentes criados durante a inicialização da aplicação concorrente ou dinamicamente por meio da primitiva **spawn**. Cada processo p pode executar diferentes programas. Entretanto, cada processo possui seu próprio espaço de memória.

O programa concorrente é composto por um conjunto de n processos paralelos $\mathbf{Prog} = \{p^0, p^1, \dots, p^{n-1}\}$. Cada processo p tem seu próprio Grafo de Fluxo de Controle ($\mathbf{GFC}^p$), que é composto por um conjunto de nós $\mathbf{N}^p$ e um conjunto de arestas $\mathbf{E}^p$.

Cada processo p^i é composto por um número fixo e conhecido de funções f^i . Cada função f^i possui seu Grafo de Fluxo de Controle ($\mathbf{GFC}^{p,f}$) e contém um conjunto de nós $\mathbf{N}^{p,f}$ e um conjunto de arestas $\mathbf{E}^{p,f}$ que representam as declarações e os predicados da função. O $\mathbf{GFC}^{p,f} \subseteq \mathbf{GFC}^p$ e possui um único nó $\mathbf{N}_{start}$ de entrada e um único nó $\mathbf{N}_{end}$ de saída.

Como programas em **Erlang** normalmente são compostos por várias funções, o teste de unidade pode não ser eficiente. Desse modo, o uso do teste de integração pode obter resultados relevantes em relação ao módulo testado. O objetivo do teste interfunção é testar as interações entre as diversas funções que compõem um módulo.

O conjunto de arestas de sincronização entre as funções de p é representado por $\mathbf{E}_{sync}^p$. Os nós do $\mathbf{CFG}^{p,f}$ são representados por $n^{p,f}$, onde n corresponde ao nó, p indica o processo e f a função. Cada nó n no processo p corresponde a um conjunto de comandos executados sequencialmente ou podem ser associados com uma primitiva de comunicação (**send** ou **receive**).

Para representar os predicados de um processo p^i que são necessários para identificar por exemplo, cláusulas inacessíveis, quatro novos tipos de conjunto de nós são classificados: $\mathbf{N}_{if}$, $\mathbf{N}_{guard}$, $\mathbf{N}_{case}$ e $\mathbf{N}_{recv}$. Por exemplo, o conjunto $\mathbf{N}_{recv}$ contém as cláusulas da primitiva de comunicação **receive**. Esses nós são interligados por meio de arestas $\mathbf{E}^{p,f}$.

Para testar a comunicação e sincronização entre os processos e com base nas definições e grafos anteriores, é definido o Grafo de Fluxo de Controle Paralelo (*Parallel Control Flow Graph* - **PCFG**) do programa **Prog**. O **PCFG** é composto por

todos os CFG^p (formalmente, $GFC^p \subseteq GFCP$) e pelas arestas de comunicação e sincronização entre os processos concorrentes denominadas E_{sync} .

Além das arestas $E^{p,f}$ comuns que ligam os nós internos das funções, são necessárias arestas para interligar funções de um módulo e também arestas para interligar processos, considerando uma aplicação concorrente. Assim, os conjuntos de arestas de um $GFCP$ são classificados da seguinte forma:

- conjunto de arestas intrafunções ($E^{p,f}$): contém as arestas internas a uma função f , de modo que: $E^{p,f} = \{N_i^{p,f}, N_k^{p,f}\}$;
- conjunto de arestas interfunções (E^p_{sync}): contém as arestas que representam a comunicação entre duas funções de um processo p , de modo que $E^p_{sync} = \{N_i^{p,f1}, N_k^{p,f2} \mid N_i^{p,f1} \in N_{call}, N_k^{p,f2} \in N_{start}\}$;
- conjunto de arestas interprocessos (E_{sync}): contém as arestas que representam a comunicação entre os processos, ou seja, as arestas que ocorrem entre dois processos concorrentes, de modo que $E_{sync} = \{N_i^{p1,f1}, N_k^{p2,f2} \mid (N_i^{p1,f1} \in N_{sync}, N_i^{p1,f1} \supset 'BS' \in B, N_k^{p2,f2} \in N_{sync}, N_k^{p2,f2} \supset 'BR' \in B)\}$.

Em relação aos nós que são interligados pelas arestas E^p_{sync} , existe uma abordagem bem conhecida, proposta por Reps, Horwitz e Sagiv (1995) e por Harrold e Rothermel (1994), que divide cada nó que representa a comunicação entre duas funções em dois nós, de forma que um nó que representa a chamada para uma função e um nó que representa o retorno.

A recursão no contexto de **Erlang** e demais linguagens funcionais é muito utilizada. O modelo de teste foi inspirado na abordagem proposta em (REPS; HORWITZ; SAGIV, 1995; JAVED et al., 2012), na qual há dois tipos de nós: **call** e **return-site** e três tipos de arestas: **call-to-return-site**, **call-to-start**, **exit-to-return-site**. Entretanto, diferentemente das abordagens mencionadas anteriormente que não diferenciam os nós, o nosso modelo de teste define quatro novos tipos de nós para diferenciar as chamadas de funções das chamadas de funções recursivas:

- conjunto N_{call} e N_{return} : representam os nós que contêm, respectivamente, as chamadas de função e o retorno de uma função chamada. Esses nós são interligados por meio de arestas E^p_{sync} , pois são funções diferentes.
- conjunto $N_{callrec}$ e N_{retrec} : representam os nós que contêm, respectivamente, as chamadas de função recursiva e o retorno de uma função chamada. Esses nós são interligados por meio de arestas $E^{p,f}$, pois diferentes dos nós N_{call} e N_{return} ,

consideram funções iguais.

Para interligar os conjuntos de nós N_{call} , N_{return} , $N_{callrec}$ e N_{retrec} , o modelo define quatro novos tipos de arestas para considerar as chamadas de funções e as chamadas de funções recursivas:

- conjunto E_{call} : contém as arestas que representam as chamadas de diferentes funções, onde $E_{call} \subseteq E^p_{sync}$;
- conjunto E_{return} : contém as arestas que representam o retorno das chamadas de funções de diferentes funções, onde $E_{return} \subseteq E^p_{sync}$;
- conjunto E_{stack} : contém as arestas que representam as chamadas recursivas (empilhamento) de uma mesma função f , onde $E_{stack} \subseteq E^{p,f}$;
- conjunto de arestas de retorno da recursão ($E_{unstack}$): contém as arestas que representam o retorno das chamadas recursivas (desempilhamento) de uma mesma função f , onde $E_{unstack} \subseteq E^{p,f}$.

Para identificar os nós N_{call} , N_{return} , $N_{callrec}$ e N_{retrec} é necessário especificar um conjunto $L = \{ 'C', 'R', l=0, l=l+1, ..., l=l+n \}$ onde C indica uma chamada de função, R indica retorno de uma função e os números indicam a quantidade de chamadas recursivas de uma função. Com isso, é possível controlar o empilhamento e desempilhamento dessas chamadas recursivas. Assim, os nós N_{call} , N_{return} , $N_{callrec}$ e N_{retrec} são uma tripla $(n^{p,f}, f_i, l_j)$, onde o nó $n^{p,f}$ possui uma chamada para a função f_i ou retorno de uma função f_i e possui um número representado por $l_j \in L$.

Os valores contidos no conjunto L são usados pelos nós N_{call} , N_{return} , $N_{callrec}$ e N_{retrec} de acordo com a aresta associada da seguinte forma:

- A identificação C é usada quando o nó N_{call} está associado com uma aresta E_{call} , isso significa que o nó N_{call} realizou uma chamada de função;
- A identificação R é usada quando o nó N_{return} está associado com uma aresta E_{return} , significando que o nó N_{return} realizou um retorno de uma chamada de função;
- Os números de forma crescente, indicam que o nó $N_{callrec}$ está associado com uma aresta E_{stack} . Isso significa que o nó $N_{callrec}$ realizou uma chamada recursiva para a mesma função, a cada chamada recursiva o número é incrementado em um;
- Os números de forma decrescente, indicam que o nó N_{retrec} está associado com uma aresta $E_{unstack}$. Isso significa que o nó N_{retrec} realizou um retorno de uma

chamada recursiva para a mesma função, a cada retorno o número é decrementado em um.

Desta forma é possível identificar a quantidade de chamadas recursivas que ocorreram para a mesma função. Esta identificação permite controlar o empilhamento e o desempilhamento dessas chamadas recursivas e identificar os caminhos executados no **GFC^p**.

A comunicação entre processos em **Erlang** é realizada por meio de passagem de mensagem, não há compartilhamento de memória (maiores detalhes na seção 2.2.5). Este modelo considera programas concorrentes com passagem de mensagens ponto a ponto, pois a linguagem não tem suporte para comunicação coletiva. Entretanto, é possível se obter comunicação coletiva por meio de uma sequência de comunicações ponto a ponto.

O conjunto de nós N_{sync} representa um subconjunto de nós que contém primitivas de comunicação e/ou sincronização. Essas primitivas classificadas de acordo com a sua semântica. Para isso é considerado um conjunto $B = \{BS, BR\}$, representando as seguintes primitivas de comunicação/sincronização: **BS** = *send* bloqueante (*Blocking Send*) e **BR** = *receive* bloqueante (*Blocking Receive*). Um nó N_{sync} é uma *tupla* $(n_i^{p,f}, b_k)$, onde o nó $n_i^{p,f} \in N^{p,f}$ possui uma primitiva de comunicação/sincronização $b_k \in B$.

De acordo com Armstrong et al. (2003), **Erlang** permite a construção de sistemas robustos para detecção de falhas em processos. É possível criar *links* entre os processos e, a partir disso, receber mensagens caso o processo ao qual foi ligado terminar ou falhar. Esta detecção é possível mesmo em um ambiente distribuído. Assim, é possível construir mecanismos de recuperação de processos.

As funções para criação de links entre processos são *link()*, *unlink()* e *spawn_link()* (maiores detalhes na seção 2.2.2). Para representar as instruções para detecção de falhas, o modelo de teste necessita identificar tais funções, para isso, os seguintes tipos de nós são propostos:

- Conjunto N_{link} : representa os nós que fazem a ligação entre dois processos;
- Conjunto N_{unlink} : representam os nós que fazem a desvinculação entre dois processos.

A partir da criação de um processo através da função *spawn_link()*, o processo que criou e o novo processo estão ligados (considera-se também a função *link()*). Qualquer evento que desfça a ligação entre eles (por exemplo, finalização

do processo ou falha de rede), uma mensagem é enviada aos processos envolvidos. Em caso de problemas no processo, a mensagem pode ser enviada de qualquer ponto de execução do processo e não apenas quando ele finalizar.

Desta forma, um nó N_{link} é uma tripla $(n_i^{p,f}, p_j, p_k)$, onde o nó $n_i^{p,f} \in N^{p,f}$ possui uma função de ligação entre os processos p_j e p_k . Um nó N_{unlink} também é uma tripla $(n_i^{p,f}, p_j, p_k)$, onde o nó $n_i^{p,f} \in N^{p,f}$ possui uma função de desfaz ligação entre os processos p_j e p_k .

Para interligar o conjunto de N_{link} e N_{unlink} , o modelo define novos tipos de arestas:

- conjunto E_{link} : contém as arestas que representam as funções que fazem a ligação entre dois processos p_j e p_k ;
- conjunto E_{unlink} : contém as arestas que representam a função que desfaz a ligação entre dois processos p_j e p_k ;

Um caminho intrafunção $\pi^{p,f}$ no $GFC^{p,f}$ de uma função f é aquele que contém somente arestas $E^{p,f}$. O caminho $\pi^{p,f}$ é representado por: $\pi^{p,f} = (n_0^{p,f}, n_1^{p,f}, \dots, n_m^{p,f})$, onde $(n_i^{p,f}, n_{i+1}^{p,f}) \in E^{p,f}$. O caminho intrafunção é utilizado para rastrear a execução de programas que utilizam recursão e também das estruturas condicionais (*if* e *case*), as cláusulas de funções (*guards*) e da primitiva *receive*.

Um caminho interfunção Π^p no GFC^p é aquele que contém pelo menos um nó N_{call} e um nó N_{return} e é representado por $\Pi^p = (\pi^{p,f1}, \pi^{p,f2}, \pi^{p,f3}, \dots, \pi^{p,fn}, S^p)$, onde S^p é o conjunto de pares de sincronização entre funções, de forma que toda aresta $(n_i^{p,f1}, n_i^{p,f2}) \in S^p$ e faz parte do conjunto de arestas E_{sync}^p .

Um caminho interprocesso Π no $GFCP$ é aquele que contém pelo menos uma aresta interprocesso (E_{sync}) e é representado por $\Pi = (\Pi^0, \Pi^1, \dots, \Pi^i, S)$, onde S é o conjunto de pares de sincronização entre os processos que foram executados no caminho Π , tal que $S \in E_{sync}$.

Uma variável x é definida quando um valor é armazenado na sua posição de memória correspondente. Normalmente, as declarações de definição são comandos de entrada e de atribuição. Uma variável também é definida quando é passada como um parâmetro (referência) para uma função.

Neste contexto, em ambientes de passagem de mensagens é preciso considerar as primitivas de comunicação. Por exemplo, a primitiva *receive* define uma ou mais variáveis com o valor n recebido na mensagem; assim, isso é

considerado uma definição. O modelo adota a definição:

$$\text{def}(n^p) = \{ x \mid x \text{ is defined in } n^p \}$$

As variáveis em **Erlang** são imutáveis, ou seja, uma vez definida não é possível atribuir um novo valor. Neste caso, tem-se uma única definição para uma variável x . Se necessário mudar o valor de uma variável é necessário fazer uma cópia para outra variável. O uso de uma variável x ocorre quando x é referenciado em uma expressão ou algum tipo de computação. O modelo utiliza os seguintes tipos de uso que são usados em Souza et al. (2008b) e adaptados para **Erlang**:

- uso computacional (**c-uso**): ocorre em uma computação (por exemplo, em uma expressão aritmética), relacionado a um nó $N^{p,f}$ do $GFC^{p,f}$;
- uso predicativo (**p-uso**): ocorre em uma condição (predicado) associada a comandos de fluxo de controle, ou seja, arestas intrafunções $\{N_i^{p,f}, N_k^{p,f}\} \in E^{p,f}$;
- uso comunicacional (**m-uso**): ocorre em comandos de sincronização/comunicação por troca de mensagem em arestas interprocessos $\{N_i^{p1,f1}, N_k^{p2,f2}\} \in E_{sync}$.

Note que **p-uso** e **c-uso** são intrafunções, ou seja, a definição e o uso de x ocorrem na mesma função p . Essas associações são normalmente necessárias para a aplicação de critérios de teste tradicionais para cada processo separadamente. Um **m-use** supõe a existência de um segundo processo e é uma associação interprocesso.

Considerando este contexto, para detectar falhas de comunicação em intrafunções (mesma função), ou seja, em chamadas recursivas, e também em interfunções, ou seja, chamadas de funções (funções diferentes), o modelo propõe dois novos tipos de uso com base nas características da linguagem **Erlang**:

- uso funcional (**f-uso**): ocorre em chamada de função, relacionado a um nó N_{call} ou N_{return} em arestas interfunções $(n_i^{p,f1}, n_j^{p,f2}) \in E_{sync}^{p}$;
- uso recursivo (**r-uso**): ocorre em uma chamada recursiva, relacionado a um nó $N_{callrec}$ ou N_{retrec} em arestas de chamadas recursivas $(n_i^{p,f}, n_j^{p,f}) \in E_{stack}$ ou $E_{unstack}$;

Um dos pontos fortes da linguagem **Erlang** (ARMSTRONG et al., 2003) é o desenvolvimento de aplicações web, de roteamento de pacotes e telefônicas que manipulam e encaminham mensagens. Essas aplicações normalmente têm um fluxo

de informação a partir do processo de origem (transmissor da mensagem) até o processo destino (receptor da mensagem). A Figura 4.1 exemplifica esta abordagem.

Figura 4.1- Exemplo de troca de mensagens entre dois processos.

Fonte: Elaborada pelo autor.

Como pode ser observado na Figura 4.1, o caminho entre a transmissão e recepção da mensagem passa uma sequência de processos, que analisam a mensagem e encaminham para o próximo processo até o destino final.

Neste contexto é interessante verificar as mensagens durante seu caminho fim-a-fim, isso leva a uma sequência de envios e recebimentos de mensagens, e consequentemente, por uma sequência de definições e usos de variáveis. A diferença em relação às linguagens imperativas é que essa sequência de definições e usos é realizada por variáveis diferentes, ou seja, o fluxo da informação (cada mensagem enviada) é feita pela cópia da variável original.

Existem algumas abordagens que consideram testar uma sequência de definições e usos de variáveis para o teste intrafunções (AHO et al., 1986) e teste interfunções (HARROLD; SOFFA, 1994; WIDERA, 2005b). O modelo considera estas abordagens e estende para o teste interprocessos.

Aho et al. (1986) define um caminho de execução (ou apenas caminho) a partir do ponto p_i até o ponto p_n . Entre esses, uma sequência de pontos de p_{i+1} até p_{n-1} , de modo que para cada $i = 1, 2, \dots, N - 1$:

- p_i é o ponto que pertence a uma declaração em um nó $n^{p,f} \subset \text{def}$ e p_{i+1} é um ponto posterior em relação ao fluxo de controle a essa mesma declaração, considerando um caminho intrafunção; ou

- p_i é o fim de um bloco e p_{i+1} é o início de um bloco sucessor, se considerar o

caminho interfunções e interprocessos.

O objetivo do teste para este contexto é detectar se uma variável x , a partir de um ponto p_i , alcançou o ponto p_n . As falhas relativas ao não alcance podem ser: perda de pacotes, processos inexistentes, cláusulas inacessíveis ou interrupção no fluxo da informação (mensagem) que podem ser causados, por exemplo, por falhas na rede de interconexão.

Uma solução é utilizar a definição **def(x)** para identificar o fluxo de informação e assim rastrear o alcance das variáveis a partir de sua definição **f-def**, até um ponto p_n . O fluxo de informação é identificado pela definição de uma variável X , uso dessa variável X , definição de X' (cópia de X) e uso de X' . Resumindo, uma sequência de definições e usos.

A partir dessas informações é possível caracterizar as associações, que estabelecem a composição de pares de definições e usos de variáveis para serem testadas e que são requeridas pelos critérios estruturais e devem ser cobertas durante os testes. O conceito de caminho livre de definição não se aplica para **Erlang**, devido a sua característica de atribuição única para as variáveis (imutáveis).

O modelo utiliza as associações especificadas em Souza et al. (2008b) e adaptadas para **Erlang** de acordo com suas características:

- associação **c-uso**: dada pela tripla $(n_1^{p,f}, n_2^{p,f}, d)$, de forma que em $(n_1^{p,f}, d)$ possui uma **def** de d e em $n_2^{p,f}$ tem um **c-uso** de d .
- associação **p-uso**: dada pela tripla $(n_1^{p,f}, (n_2^{p,f}, n_3^{p,f}), d)$, de forma que em $(n_1^{p,f}, d)$ possui uma **def** de d e em $(n_2^{p,f}, n_3^{p,f})$ tem um **p-uso** de d .
- associação **m-uso**: dada pela tripla $(n_1^{p_1,f_1}, (n_2^{p_1,f_1}, n_i^{p_2,f_i}), d)$, de forma que em $(n_1^{p_1,f_1}, d)$ possui uma **def** de d e em $(n_2^{p_1,f_1}, n_i^{p_2,f_i})$ tem um **m-uso** de d .

As associações **c-uso** e **p-uso** capturam aspectos sequenciais do programa, pois são associações intrafunções, ou seja, a definição e uso da variável ocorrem na mesma função f . No caso da associação **m-uso** considera-se a existência de um segundo processo p_2 que irá receber a mensagem, caracterizando a comunicação/sincronização interprocessos. Isso é similar ao definido em Souza et al (2013). Uma exceção é o uso de variáveis globais, onde pode ocorrer um **c-uso** ou **p-uso** em funções diferentes ou até mesmo em processos diferentes. Considerando o uso de funções, recursão e processos, este modelo estabelece três novos tipos de associações:

- associação **f-uso**: dada pela tripla $(n_1^{p,f_1}, (n_2^{p,f_1}, n_i^{p,f_2}), d)$, tal que em $(n_1^{p,f_1},$

d) possui uma **def** de **d**, sendo em $(n_2^{p,f1}, n_i^{p,f2})$ um **f-uso** de **d**. Essa associação captura a comunicação entre funções de um mesmo processo.

- associação **r-uso**: dada pela tripla $(n_1^{p,f}, (n_2^{p,f}, n_3^{p,f}), d)$, tal que em $(n_1^{p,f}, d)$ possui uma **def** de **d**, sendo em $(n_2^{p,f}, n_3^{p,f})$ um **r-uso** de **d**. Essa associação captura o empilhamento ou desempilhamento de chamadas recursivas.

- associação **p-link**: dada pela tripla $(n_i^{pi,fi}, n_j^{pi,fj}, n_k^{pk,fk})$, tal que $n_i^{pi,fi} \in N_{link}$, em $n_j^{pi,fj}$ possui um nó n_{sync} com a primitiva $BR \in B$ e $n_k^{pk,fk}$ representa o primeiro nó (n_{start}) do processo p_k , isso representa que o processo criado iniciou sua execução.

A associação *p-link* representa a ligação entre dois processos e possibilita a detecção de falhas no processo criado. Para esta associação existem duas situações bem distintas, a primeira é a execução completa do processo criado e o recebimento de uma mensagem 'EXIT' pelo processo criador, esta situação é a finalização normal do processo criado.

A segunda é o recebimento de uma mensagem pelo processo criador e a não execução incompleta do processo criado, neste caso, a mensagem recebida pelo processo criador não será 'EXIT' e ocorreu algo errado e a ligação entre os dois processos foi desfeita.

4.3 Critérios estruturais de teste

Com base nas definições da seção anterior, este trabalho propõe dois conjuntos de critérios de teste estruturais para programas **Erlang**. O modelo considera os critérios propostos em Souza et al. (2008b) e que são tradicionalmente usados nas linguagens imperativas com adaptações necessárias para **Erlang**.

4.3.1 Critérios de teste baseados em fluxo de controle e de comunicação

- Todos-Nós: requer que todos os nós do conjunto **N** sejam exercitados pelo menos uma vez pelo conjunto de casos de teste.
- Todas-Arestas: requer que todas as arestas do conjunto **E** sejam exercitadas pelo menos uma vez pelo conjunto de casos de teste.
- Todos-Nós-Sync: requer que todos os nós do conjunto N_{sync} sejam exercitados pelo menos uma vez pelo conjunto de casos de teste.

- Todas-Arestas-Sync: requer que todas as arestas dos conjuntos E_{sync} e E_{sync}^p (para $p = 0..n_p-1$) sejam exercitadas pelo menos uma vez pelo conjunto de casos de teste.

4.3.2 Critérios de teste baseados em fluxo de dados e passagem de mensagem

- Todos-Definições: requer que para cada n_i^p e $x \in \mathbf{def}(n_i^p)$, um conjunto de casos de teste sejam exercitadas pelo menos uma vez.
- Todos-Usos: requer que todas as associações c-uso, p-uso e m-uso sejam exercitadas pelo menos uma vez pelo conjunto de casos de teste.

4.4 Novos critérios de teste estruturais de teste

Para satisfazer características específicas de **Erlang**, tais como, a comunicação entre funções, chamadas recursivas de funções e tolerância a falhas, são propostos novos critérios baseados em fluxo de controle, fluxo de dados e fluxo de comunicação:

- Todos-Nós-Inter-Funções: requer que todos os nós do conjunto N_{call} e N_{return} sejam exercitados pelo menos uma vez pelo conjunto de casos de teste.
- Todos-Nós-Recursivos: requer que todos os nós do conjunto $N_{callrec}$ e N_{retrec} sejam exercitados pelo menos uma vez pelo conjunto de casos de teste.
- Todos-Nós-Links: requer que todos os nós do conjunto N_{link} e N_{unlink} sejam exercitados pelo menos uma vez pelo conjunto de casos de teste.
- Todas-Arestas-Inter-Funções: requer que todas as arestas dos conjuntos E_{call} e E_{return} sejam exercitadas pelo menos uma vez pelo conjunto de casos de teste.
- Todas-Arestas-Recursivas: requer que todas as arestas dos conjuntos E_{stack} e $E_{unstack}$ sejam exercitadas pelo menos uma vez pelo conjunto de casos de teste.
- Todas-Arestas-Links: requer que todas as arestas dos conjuntos E_{link} e E_{unlink} sejam exercitadas pelo menos uma vez pelo conjunto de casos de teste.

Os critérios baseados em fluxo de dados e passagem de mensagens são os seguintes:

- Todos-f-Usos: requer que todas as associações **f-uso** sejam exercitadas pelo conjunto de casos de teste.

- Todos-r-Usos: requer que todas as associações **r-uso** sejam exercitadas pelo conjunto de casos de teste.
- Todos-p-link: requer que todas as associações **p-link** sejam exercitadas pelo conjunto de casos de teste.

4.5 Um exemplo

Para ilustrar as informações recuperadas do código fonte e armazenadas pelo modelo de teste em relação às chamadas de funções, chamadas recursivas e comunicação entre processos, considere o programa GCD (Figura 4.2). Este programa usa até quatro processos paralelos (p^m , p^0 , p^1 e p^2) para calcular o máximo divisor comum de três números.

Este programa GCD funciona da seguinte forma: o programa principal (p^m) cria três processos escravos (linha 5, 6 e 24), recebe três números e envia os dois primeiros para o primeiro processo escravo (p^0 - linha 10) e os dois últimos para o segundo processo escravo (p^1 - linha 11). Os processos escravos inicializam a execução pela função **calc()** e ficam bloqueados (primitiva **receive**) aguardando os dados do processo p^m (linha 38). Após o recebimento, os processos escravos calculam o valor e enviam para o programa principal p^m (linha 52) e terminam suas execuções. O cálculo pode envolver dois ou três processos escravos - isso depende dos valores de entrada. Os processos escravos utilizam recursão (linhas 47 e 50), e de acordo com os valores recebidos a função **calc()** pode ser instanciada várias vezes. O resultado do cálculo do programa GCD é mostrado pelo processo p^m (linhas 28 ou 35). Apesar de simples, esta aplicação possui as características necessárias para a aplicação dos critérios de teste, tais como: concorrência, chamadas de função e recursão. O processo p^m representa o processo principal e os processos p^0 , p^1 e p^2 , representam os processos escravos.

A Figura 4.3 mostra o GFCP que representa o programa GCD. Os números à esquerda do código fonte entre asteriscos (Figura 4.2) representam os nós no grafo. As arestas interprocessos estão representadas pelas setas em vermelho. As setas em azul representam as arestas interfunções. As arestas das chamadas recursivas e o retorno da recursão são representadas pelas setas em verde.

A Tabela 4.1 apresenta o conjunto **def(n_i^p)** de todas as variáveis definidas do programa GCD. A Tabela 4.2 contém os valores de todos os conjuntos introduzidos

na seção 4.1. A Tabela 4.3 apresenta alguns elementos requeridos pelos critérios de teste estruturais introduzidos nas seções 4.3 e 4.4.

Figura 4.2- Programa GCD.

```

1  -module(gcd2).
2  -export([start/3, wait/0]).
3  /*01*/ start(NumX, NumY, NumZ) ->
4  /*02*/     global:register_name(pidm,self()),
5  /*03*/     global:register_name(pid0, spawn(gcd2,wait,[])),
6  /*04*/     global:register_name(pid1, spawn(gcd2,wait,[])),
7  /*05*/     Pidm = global:whereis_name(pidm),
8  /*05*/     Pid0 = global:whereis_name(pid0),
9  /*05*/     Pid1 = global:whereis_name(pid1),
10 /*06*/     Pid0 ! {NumX, NumY, Pidm},
11 /*07*/     Pid1 ! {NumY, NumZ, Pidm},
12 /*08-09-10*/ waitx().
13 /*11*/ waitx() ->
14 /*12*/     receive
15 /*13*/         {X} ->
16 /*14*/             io:format("Receive X de Slave! ~n"),
17 /*15-16*/         waity(X)
18 /*17*/     end.
19 /*18*/ waity(X) ->
20 /*19*/     receive
21 /*20*/         {Y} ->
22 /*21*/             io:format("Receive Y de Slave! ~n"),
23 /*22*/             if (X > 1) and (Y > 1) ->
24 /*23*/                 P2 = spawn(gcd2, wait, []),
25 /*24*/                 P2 ! {X, Y, global:whereis_name(pidm)},
26 /*25-26*/                 fim();
27 /*27*/             true ->
28 /*28*/                 io:format("GCD = ~w. Fim ~w! ~n", [X, global:whereis_name(pidm)]),
29 /*28*/                 end
30 /*29*/     end.
31 /*30*/ fim() ->
32 /*31*/     receive
33 /*32*/         {Z} ->
34 /*33*/             io:format("Receive Z de Slave! ~n"),
35 /*33*/             io:format("GCD = ~w. Fim ~w! ~n", [Z, global:whereis_name(pidm)]),
36 /*34*/     end.
37 /*35*/ wait() ->
38 /*36*/     receive
39 /*37*/         {X, Y, Mpid} ->
40 /*38*/             io:format("Received X(~w) e Y(~w) de ~w(Master)! ~n", [X, Y, Mpid]),
41 /*39-40*/             calc(X, Y, Mpid);
42 /*41*/             ->
43 /*42*/             io:format("Fim ~w! ~n", [self()])
44 /*43*/     end.
45 /*44-45*/ calc(X, Y, Mpid) when X < Y ->
46 /*46*/     io:format("x(~w) < y(~w)~n ", [X, Y]),
47 /*47-48*/     calc(X, Y - X, Mpid);
48 /*49*/     calc(X, Y, Mpid) when X > Y ->
49 /*50*/     io:format("x(~w) > y(~w)~n ", [X, Y]),
50 /*51-52*/     calc(X - Y, Y, Mpid);
51 /*53*/     calc(X, Y, Mpid) when X == Y ->
52 /*54*/     Mpid ! {X},
53 /*55-56*/     io:format("x(~w) == y(~w)~n ", [X, Y]).

```

Fonte: Elaborada pelo autor.

Os dados de teste devem ser gerados de forma a exercitar cada elemento requerido possível. Por exemplo, considerando os dados de teste {NumX = 1, NumY = 2, NumZ = 1}, os caminhos executados são ($\Pi = \pi^m, \pi^0, \pi^1, s$), onde $\pi^m = \{1, 2, 3, 4, 5, 6, 7, 8, 11, 12, 13, 14, 15, 18, 19, 20, 21, 22, 27, 28, 29, 16, 17, 9, 10\}$; $\pi^0 = \{35,$

36, 37, 38, 39, 44, 45, 46, 47, 44, 45, 49, 53, 54, 55, 56, 48, 56, 40, 43}; $\pi^1 = \{35, 36, 37, 38, 39, 44, 45, 49, 50, 51, 44, 45, 49, 53, 54, 55, 56, 52, 56, 40, 43\}$; $S = \{(6^m, 36^0), (7^m, 36^1), (54^0, 12^m), (54^1, 19^m)\}$. O processo p^2 não executa porque o resultado já foi produzido por p^0 e p^1 .

Figura 4.3- Grafo de fluxo de controle paralelo do programa GCD em **Erlang**.

Fonte: Elaborada pelo autor.

Em função do *receive* não determinístico nos nós 12^m e 19^m , serão possíveis quatro arestas de sincronização: $(54^0, 12^m)$, $(54^0, 19^m)$, $(54^1, 12^m)$, $(54^1, 19^m)$ e apenas duas delas são exercidas para cada execução do caminho Π dependendo

do tempo de execução: $((54^0, 12^m)$ ou $(54^0, 19^m)$, $(54^1, 12^m)$ ou $(54^1, 19^m)$). Em cada execução do programa, é necessário determinar as arestas interprocessos que foram executadas. Este aspecto está relacionado com a avaliação dos casos de teste e foi considerado na implementação de **ValiErlang**, descrita na seção 5.

Tabela 4.1- Conjunto de definição de variáveis do programa GCD.

Processo P^m	Processos P^0, P^1, P^2
$\text{def}(n_1^{m,0}) = \{\text{NumX}, \text{NumY}, \text{NumZ}\}$ $\text{def}(n_2^{m,0}) = \{\text{pidm}\}$ $\text{def}(n_3^{m,0}) = \{\text{pid0}\}$ $\text{def}(n_4^{m,0}) = \{\text{pid1}\}$ $\text{def}(n_5^{m,0}) = \{\text{Pidm}, \text{Pid0}, \text{Pid1}\}$ $\text{def}(n_{13}^{m,1}) = \{X\}$ $\text{def}(n_{18}^{m,2}) = \{X\}$ $\text{def}(n_{20}^{m,2}) = \{Y\}$ $\text{def}(n_{23}^{m,2}) = \{P2\}$ $\text{def}(n_{32}^{m,3}) = \{Z\}$	$\text{def}(n_{37}^{0,0}) = \{X, Y, \text{Mpid}\}$ $\text{def}(n_{41}^{0,0}) = \{ _ \}$ $\text{def}(n_{44}^{0,1}) = \{X, Y, \text{Mpid}\}$ $\text{def}(n_{37}^{1,0}) = \{X, Y, \text{Mpid}\}$ $\text{def}(n_{41}^{1,0}) = \{ _ \}$ $\text{def}(n_{44}^{1,1}) = \{X, Y, \text{Mpid}\}$ $\text{def}(n_{37}^{2,0}) = \{X, Y, \text{Mpid}\}$ $\text{def}(n_{41}^{2,0}) = \{ _ \}$ $\text{def}(n_{44}^{2,1}) = \{X, Y, \text{Mpid}\}$

Fonte: Elaborada pelo autor.

4.6 Revelando falhas

A eficácia (em termos de revelação de falhas) dos critérios propostos pode ser ilustrada por alguns tipos de falhas (descritas na seção 3.5) que poderiam estar presentes no programa GCD (Figura 4.2) e mostrando como os critérios contribuem para revelar esses tipos de falhas. As situações de falha são baseadas também nos trabalhos de Howden, (1976) e Krawczyk e Wiszniewski, (1994) e consideradas em Souza et al., (2008b), que descrevem falhas típicas em programas tradicionais e programas paralelos.

Howden, (1976) introduz dois tipos de falhas em programas tradicionais: computação e falhas de domínio. O primeiro ocorre quando o resultado de uma computação para uma entrada do domínio do programa é diferente do resultado esperado.

O segundo ocorre quando um caminho diferente do esperado é executado. Por exemplo, no processo p^1 , substituindo o comando do nó 47^{p1} (uma chamada recursiva) $(X, Y - X, \text{Mpid})$ pelo comando incorreto $(X, Y + X, \text{Mpid})$ corresponde a uma falha de computação. Uma falha de domínio pode ser ilustrada mudando o *guard* (*when* $X < Y$) na aresta $(45^{p1}, 46^{p1})$ pelo predicado incorreto (*when* $X > Y$), tomando um caminho diferente durante a execução.

Tabela 4.2- Conjuntos do modelo de teste para o programa GCD.

n	4		
Prog	p^m, p^0, p^1, p^2		
p^m	$\{f^{m,0}, f^{m,1}, f^{m,2}, f^{m,3}\}$	p^0	$\{f^{0,0}, f^{0,1}\}$
p^1	$\{f^{1,0}, f^{1,1}\}$	p^2	$\{f^{2,0}, f^{2,1}\}$
N^m	$\{n_0^{m,0}, \dots, n_{10}^{m,0}\}, \{n_{11}^{m,1}, \dots, n_{17}^{m,1}\}, \{n_{18}^{m,2}, \dots, n_{29}^{m,2}\}$ e $\{n_{30}^{m,3}, \dots, n_{34}^{m,3}\}$		
N^0	$\{n_{35}^{0,0}, \dots, n_{43}^{0,0}\}$ e $\{n_{44}^{0,1}, \dots, n_{56}^{0,1}\}$		
N^1	$\{n_{35}^{1,0}, \dots, n_{43}^{1,0}\}$ e $\{n_{44}^{1,1}, \dots, n_{56}^{1,1}\}$		
N^2	$\{n_{35}^{2,0}, \dots, n_{43}^{2,0}\}$ e $\{n_{44}^{2,1}, \dots, n_{56}^{2,1}\}$		
N_{start}	$\{n_1^{m,0}, n_{11}^{m,1}, n_{18}^{m,2}, n_{30}^{m,3}\}, \{n_{35}^{0,0}, n_{44}^{0,1}\}, \{n_{35}^{1,0}, n_{44}^{1,1}\}$ e $\{n_{35}^{2,0}, n_{44}^{2,1}\}$		
N_{end}	$\{n_{10}^{m,0}, n_{17}^{m,1}, n_{29}^{m,2}, n_{34}^{m,3}\}, \{n_{43}^{0,0}, n_{56}^{0,1}\}, \{n_{43}^{1,0}, n_{56}^{1,1}\}$ e $\{n_{43}^{2,0}, n_{56}^{2,1}\}$		
N_{if}	$\{n_{22}^{m,2}, n_{27}^{m,2}\}$		
N_{guard}	$\{n_{45}^{0,1}, n_{49}^{0,1}, n_{53}^{0,1}\}, \{n_{45}^{1,1}, n_{49}^{1,1}, n_{53}^{1,1}\}$ e $\{n_{45}^{2,1}, n_{49}^{2,1}, n_{53}^{2,1}\}$		
N_{recv}	$\{n_{13}^{m,1}, n_{20}^{m,2}, n_{32}^{m,3}\}, \{n_{37}^{0,0}, n_{41}^{0,0}\}, \{n_{37}^{1,0}, n_{41}^{1,0}\}$ e $\{n_{37}^{2,0}, n_{41}^{2,0}\}$		
B	{BS, BR}, onde BS = 'blocking send' and BR = 'blocking receive'		
N_{sync}	$\{(n_6^{m,0}, 'BS'), (n_7^{m,0}, 'BS'), (n_{12}^{m,1}, 'BR'), (n_{19}^{m,2}, 'BR'), (n_{24}^{m,2}, 'BS'), (n_{31}^{m,3}, 'BR'), (n_{36}^{0,0}, 'BR'), (n_{54}^{0,1}, 'BS'), (n_{36}^{1,0}, 'BR'), (n_{54}^{1,1}, 'BS'), (n_{36}^{2,0}, 'BR'), (n_{54}^{2,1}, 'BS')\}$		
N_{call}	$\{(n_8^{m,0}, 'waitx', 'C'), (n_{15}^{m,1}, 'waity', 'C'), (n_{25}^{m,2}, 'fim', 'C'), (n_{39}^{0,0}, 'calc', 'C'), (n_{39}^{1,0}, 'calc', 'C'), (n_{39}^{2,0}, 'calc', 'C')\}$		
N_{return}	$\{(n_9^{m,0}, 'waitx', 'R'), (n_{16}^{m,1}, 'waity', 'R'), (n_{26}^{m,2}, 'fim', 'R'), (n_{40}^{0,0}, 'calc', 'R'), (n_{40}^{1,0}, 'calc', 'R'), (n_{40}^{2,0}, 'calc', 'R')\}$		
$N_{callrec}$	$\{(n_{47}^{0,1}, 'calc', '*'), (n_{51}^{0,1}, 'calc', '*'), (n_{47}^{1,1}, 'calc', '*'), (n_{51}^{1,1}, 'calc', '*'), (n_{47}^{2,1}, 'calc', '*'), (n_{51}^{2,1}, 'calc', '*')\}$ * não é possível obter esta informação estaticamente.		
N_{retrec}	$\{(n_{48}^{0,1}, 'calc', '*'), (n_{52}^{0,1}, 'calc', '*'), (n_{48}^{1,1}, 'calc', '*'), (n_{52}^{1,1}, 'calc', '*'), (n_{48}^{2,1}, 'calc', '*'), (n_{52}^{2,1}, 'calc', '*')\}$ * não é possível obter esta informação estaticamente.		
$E^{m,f}$	$\{(n_1^{m,0}, n_2^{m,0}), (n_2^{m,0}, n_3^{m,0}), (n_3^{m,0}, n_4^{m,0}), (n_4^{m,0}, n_5^{m,0}), (n_5^{m,0}, n_6^{m,0}), (n_6^{m,0}, n_7^{m,0}), (n_7^{m,0}, n_8^{m,0}), (n_8^{m,0}, n_9^{m,0}), (n_9^{m,0}, n_{10}^{m,0}), (n_{10}^{m,0}, n_{11}^{m,1}), (n_{11}^{m,1}, n_{12}^{m,1}), (n_{12}^{m,1}, n_{13}^{m,1}), (n_{13}^{m,1}, n_{14}^{m,1}), (n_{14}^{m,1}, n_{15}^{m,1}), (n_{15}^{m,1}, n_{16}^{m,1}), (n_{16}^{m,1}, n_{17}^{m,1}), (n_{17}^{m,1}, n_{18}^{m,2}), (n_{18}^{m,2}, n_{19}^{m,2}), (n_{19}^{m,2}, n_{20}^{m,2}), (n_{20}^{m,2}, n_{21}^{m,2}), (n_{21}^{m,2}, n_{22}^{m,2}), (n_{22}^{m,2}, n_{23}^{m,2}), (n_{23}^{m,2}, n_{24}^{m,2}), (n_{24}^{m,2}, n_{25}^{m,2}), (n_{25}^{m,2}, n_{26}^{m,2}), (n_{26}^{m,2}, n_{27}^{m,2}), (n_{27}^{m,2}, n_{28}^{m,2}), (n_{28}^{m,2}, n_{29}^{m,2}), (n_{29}^{m,2}, n_{30}^{m,3}), (n_{30}^{m,3}, n_{31}^{m,3}), (n_{31}^{m,3}, n_{32}^{m,3}), (n_{32}^{m,3}, n_{33}^{m,3}), (n_{33}^{m,3}, n_{34}^{m,3})\}$		
$E^{0,f}$	$\{(n_{35}^{0,0}, n_{36}^{0,0}), (n_{36}^{0,0}, n_{37}^{0,0}), (n_{37}^{0,0}, n_{41}^{0,0}), (n_{37}^{0,0}, n_{38}^{0,0}), (n_{38}^{0,0}, n_{39}^{0,0}), (n_{39}^{0,0}, n_{40}^{0,0}), (n_{40}^{0,0}, n_{43}^{0,0}), (n_{41}^{0,0}, n_{42}^{0,0}), (n_{42}^{0,0}, n_{43}^{0,0}), (n_{44}^{0,1}, n_{45}^{0,1}), (n_{45}^{0,1}, n_{46}^{0,1}), (n_{45}^{0,1}, n_{49}^{0,1}), (n_{49}^{0,1}, n_{50}^{0,1}), (n_{49}^{0,1}, n_{53}^{0,1}), (n_{53}^{0,1}, n_{54}^{0,1}), (n_{54}^{0,1}, n_{55}^{0,1}), (n_{55}^{0,1}, n_{56}^{0,1}), (n_{46}^{0,1}, n_{47}^{0,1}), (n_{48}^{0,1}, n_{56}^{0,1}), (n_{50}^{0,1}, n_{51}^{0,1}), (n_{52}^{0,1}, n_{56}^{0,1})\}$		
$E^{1,f}$	$\{(n_{35}^{1,0}, n_{36}^{1,0}), (n_{36}^{1,0}, n_{37}^{1,0}), (n_{37}^{1,0}, n_{41}^{1,0}), (n_{37}^{1,0}, n_{38}^{1,0}), (n_{38}^{1,0}, n_{39}^{1,0}), (n_{39}^{1,0}, n_{40}^{1,0}), (n_{40}^{1,0}, n_{43}^{1,0}), (n_{41}^{1,0}, n_{42}^{1,0}), (n_{42}^{1,0}, n_{43}^{1,0}), (n_{44}^{1,1}, n_{45}^{1,1}), (n_{45}^{1,1}, n_{46}^{1,1}), (n_{45}^{1,1}, n_{49}^{1,1}), (n_{49}^{1,1}, n_{50}^{1,1}), (n_{49}^{1,1}, n_{53}^{1,1}), (n_{53}^{1,1}, n_{54}^{1,1}), (n_{54}^{1,1}, n_{55}^{1,1}), (n_{55}^{1,1}, n_{56}^{1,1}), (n_{46}^{1,1}, n_{47}^{1,1}), (n_{48}^{1,1}, n_{56}^{1,1}), (n_{50}^{1,1}, n_{51}^{1,1}), (n_{52}^{1,1}, n_{56}^{1,1})\}$		
$E^{2,f}$	$\{(n_{35}^{2,0}, n_{36}^{2,0}), (n_{36}^{2,0}, n_{37}^{2,0}), (n_{37}^{2,0}, n_{41}^{2,0}), (n_{37}^{2,0}, n_{38}^{2,0}), (n_{38}^{2,0}, n_{39}^{2,0}), (n_{39}^{2,0}, n_{40}^{2,0}), (n_{40}^{2,0}, n_{43}^{2,0}), (n_{41}^{2,0}, n_{42}^{2,0}), (n_{42}^{2,0}, n_{43}^{2,0}), (n_{44}^{2,1}, n_{45}^{2,1}), (n_{45}^{2,1}, n_{46}^{2,1}), (n_{45}^{2,1}, n_{49}^{2,1}), (n_{49}^{2,1}, n_{50}^{2,1}), (n_{49}^{2,1}, n_{53}^{2,1}), (n_{53}^{2,1}, n_{54}^{2,1}), (n_{54}^{2,1}, n_{55}^{2,1}), (n_{55}^{2,1}, n_{56}^{2,1}), (n_{46}^{2,1}, n_{47}^{2,1}), (n_{48}^{2,1}, n_{56}^{2,1}), (n_{50}^{2,1}, n_{51}^{2,1}), (n_{52}^{2,1}, n_{56}^{2,1})\}$		
E^m_{sync}	$\{(n_8^{m,0}, n_{11}^{m,1}), (n_{15}^{m,1}, n_{18}^{m,2}), (n_{25}^{m,2}, n_{30}^{m,3}), (n_{34}^{m,3}, n_{26}^{m,2}), (n_{29}^{m,2}, n_{16}^{m,1}), (n_{17}^{m,2}, n_9^{m,0})\}$		
E^0_{sync}	$\{(n_{39}^{0,0}, n_{44}^{0,1}), (n_{56}^{0,1}, n_{40}^{0,0})\}$		
E^1_{sync}	$\{(n_{39}^{1,0}, n_{44}^{1,1}), (n_{56}^{1,1}, n_{40}^{1,0})\}$	E^2_{sync}	$\{(n_{39}^{2,0}, n_{44}^{2,1}), (n_{56}^{2,1}, n_{40}^{2,0})\}$
E_{sync}	$\{(n_6^{m,0}, n_{36}^{0,0}), (n_7^{m,0}, n_{36}^{1,0}), (n_{24}^{m,2}, n_{36}^{2,0}), (n_{54}^{0,1}, n_{12}^{m,1}), (n_{54}^{0,1}, n_{19}^{m,2}), (n_{54}^{0,1}, n_{31}^{m,3}), (n_{54}^{1,1}, n_{12}^{m,1}), (n_{54}^{1,1}, n_{19}^{m,2}), (n_{54}^{1,1}, n_{31}^{m,3}), (n_{54}^{2,1}, n_{12}^{m,1}), (n_{54}^{2,1}, n_{19}^{m,2}), (n_{54}^{2,1}, n_{31}^{m,3})\}$		
E_{call}	$\{(n_8^{m,0}, n_{11}^{m,1}), (n_{15}^{m,1}, n_{18}^{m,2}), (n_{25}^{m,2}, n_{30}^{m,3}), (n_{39}^{0,0}, n_{44}^{0,1}), (n_{39}^{1,0}, n_{44}^{1,1}), (n_{39}^{2,0}, n_{44}^{2,1})\}$		
E_{return}	$\{(n_{34}^{m,3}, n_{26}^{m,2}), (n_{29}^{m,2}, n_{16}^{m,1}), (n_{17}^{m,2}, n_9^{m,0}), (n_{56}^{0,1}, n_{40}^{0,0}), (n_{56}^{1,1}, n_{40}^{1,0}), (n_{56}^{2,1}, n_{40}^{2,0})\}$		
E_{stack}	$\{(n_{47}^{0,1}, n_{44}^{0,1}), (n_{51}^{0,1}, n_{44}^{0,1}), (n_{47}^{1,1}, n_{44}^{1,1}), (n_{51}^{1,1}, n_{44}^{1,1}), (n_{47}^{2,1}, n_{44}^{2,1}), (n_{51}^{2,1}, n_{44}^{2,1})\}$		
$E_{unstack}$	$\{(n_{56}^{0,1}, n_{48}^{0,1}), (n_{56}^{0,1}, n_{52}^{0,1}), (n_{56}^{1,1}, n_{48}^{1,1}), (n_{56}^{1,1}, n_{52}^{1,1}), (n_{56}^{2,1}, n_{48}^{2,1}), (n_{56}^{2,1}, n_{52}^{2,1})\}$		

Fonte: Elaborada pelo autor.

Tabela 4.3- Alguns elementos requeridos pelos critérios de teste propostos.

all-nodes	$n_0^{m,0}, \dots, n_{34}^{m,2}, n_{35}^{0,0}, \dots, n_{56}^{0,1}, n_{35}^{1,0}, \dots, n_{156}^{1,1}, n_{35}^{2,0}, \dots, n_{56}^{2,1}$
all-edges	$(n_0^{m,0}, n_1^{m,0}), \dots, (n_4^{m,0}, n_5^{m,0}), (n_0^{0,0}, n_1^{0,0}), \dots, (n_5^{0,0}, n_6^{0,0}), (n_0^{1,0}, n_1^{1,0}), \dots, (n_5^{1,0}, n_6^{1,0}), (n_0^{2,0}, n_1^{2,0}), \dots, (n_5^{2,0}, n_6^{2,0}),$
all-nodes-sync	$(n_6^{m,0}, 'BS'), (n_7^{m,0}, 'BS'), (n_{12}^{m,1}, 'BR'), (n_{19}^{m,2}, 'BR'), (n_{24}^{m,2}, 'BS'), (n_{31}^{m,3}, 'BR'), (n_{36}^{0,0}, 'BR'), (n_{54}^{0,1}, 'BS'), (n_{36}^{1,0}, 'BR'), (n_{54}^{1,1}, 'BS'), (n_{36}^{2,0}, 'BR'), (n_{54}^{2,1}, 'BS')$
all-edges-sync	$(n_6^{m,0}, n_{36}^{0,0}), (n_7^{m,0}, n_{36}^{1,0}), (n_{24}^{m,2}, n_{36}^{2,0}), (n_{54}^{0,1}, n_{12}^{m,1}), (n_{54}^{0,1}, n_{19}^{m,2}), (n_{54}^{0,1}, n_{31}^{m,3}), (n_{54}^{1,1}, n_{12}^{m,1}), (n_{54}^{1,1}, n_{19}^{m,2}), (n_{54}^{1,1}, n_{31}^{m,3}), (n_{54}^{2,1}, n_{12}^{m,1}), (n_{54}^{2,1}, n_{19}^{m,2}), (n_{54}^{2,1}, n_{31}^{m,3})$
all-defs	$(n_1^{m,0}, \text{NumX}), (n_1^{m,0}, \text{NumY}), (n_1^{m,0}, \text{NumZ}), (n_2^{m,0}, \text{pidm}), (n_3^{m,0}, \text{pid0}), \dots, (n_{37}^{0,0}, X), (n_{37}^{0,0}, Y), \dots, (n_{37}^{1,0}, X), (n_{37}^{1,0}, Y), \dots, (n_{37}^{2,0}, X), (n_{37}^{2,0}, Y), (n_{37}^{2,0}, \text{Mpid}), \dots$
all-nodes-inter-functions	$(n_8^{m,0}, 'waitx', 'C'), (n_9^{m,0}, 'waitx', 'R'), \dots, (n_{39}^{0,0}, 'calc', 'C'), (n_{40}^{0,0}, 'calc', 'R'), (n_{39}^{1,0}, 'calc', 'C'), (n_{40}^{1,0}, 'calc', 'R'), (n_{39}^{2,0}, 'calc', 'C'), (n_{40}^{2,0}, 'calc', 'R')$
all-nodes-recursive	$(n_{47}^{0,1}, 'calc', '**'), (n_{51}^{0,1}, 'calc', '**'), (n_{47}^{1,1}, 'calc', '**'), (n_{51}^{1,1}, 'calc', '**'), (n_{47}^{2,1}, 'calc', '**'), (n_{51}^{2,1}, 'calc', '**'), (n_{48}^{0,1}, 'calc', '**'), (n_{52}^{0,1}, 'calc', '**'), (n_{48}^{1,1}, 'calc', '**'), (n_{52}^{1,1}, 'calc', '**'), (n_{48}^{2,1}, 'calc', '**'), (n_{52}^{2,1}, 'calc', '**')$
all-edges-inter-functions	$(n_8^{m,0}, n_{11}^{m,1}), (n_{15}^{m,1}, n_{18}^{m,2}), (n_{25}^{m,2}, n_{30}^{m,3}), (n_{39}^{0,0}, n_{44}^{0,1}), (n_{39}^{1,0}, n_{44}^{1,1}), (n_{39}^{2,0}, n_{44}^{2,1}), (n_{34}^{m,3}, n_{26}^{m,2}), (n_{29}^{m,2}, n_{16}^{m,1}), (n_{17}^{m,2}, n_9^{m,0}), (n_{56}^{0,1}, n_{40}^{0,0}), (n_{56}^{1,1}, n_{40}^{1,0}), (n_{56}^{2,1}, n_{40}^{2,0})$
all-edges-recursive	$(n_{47}^{0,1}, n_{44}^{0,1}), (n_{51}^{0,1}, n_{44}^{0,1}), (n_{47}^{1,1}, n_{44}^{1,1}), (n_{51}^{1,1}, n_{44}^{1,1}), (n_{47}^{2,1}, n_{44}^{2,1}), (n_{51}^{2,1}, n_{44}^{2,1}), (n_{56}^{0,1}, n_{48}^{0,1}), (n_{56}^{0,1}, n_{52}^{0,1}), (n_{56}^{1,1}, n_{48}^{1,1}), (n_{56}^{1,1}, n_{52}^{1,1}), (n_{56}^{2,1}, n_{48}^{2,1}), (n_{56}^{2,1}, n_{52}^{2,1})$
all-c-uses	$(n_2^{m,0}, n_5^{m,0}, \text{pidm}), (n_3^{m,0}, n_5^{m,0}, \text{pid0}), (n_4^{m,0}, n_5^{m,0}, \text{pid1}), (n_5^{m,0}, n_6^{m,0}, \text{Pid0}), (n_5^{m,0}, n_7^{m,0}, \text{Pid1}), (n_{23}^{m,2}, n_{24}^{m,2}, P2), (n_{18}^{m,2}, n_{28}^{m,2}, X), (n_2^{m,2}, n_{28}^{m,2}, \text{pidm}), (n_{32}^{m,3}, n_{34}^{m,3}, Z), (n_2^{m,0}, n_{34}^{m,3}, \text{pidm}), (n_{37}^{0,0}, n_{38}^{0,0}, X), (n_{37}^{0,0}, n_{38}^{0,0}, Y), (n_{37}^{0,0}, n_{38}^{0,0}, \text{Mpid}), (n_{44}^{0,1}, n_{46}^{0,1}, X), (n_{44}^{0,1}, n_{46}^{0,1}, Y), (n_{44}^{0,1}, n_{50}^{0,1}, X), (n_{44}^{0,1}, n_{50}^{0,1}, Y), (n_{44}^{0,1}, n_{54}^{0,1}, \text{Mpid}), (n_{44}^{0,1}, n_{55}^{0,1}, X), (n_{44}^{0,1}, n_{55}^{0,1}, Y), (n_{37}^{1,0}, n_{38}^{1,0}, X), (n_{37}^{1,0}, n_{38}^{1,0}, Y), \dots, (n_{44}^{1,1}, n_{54}^{1,1}, \text{Mpid}), (n_{44}^{1,1}, n_{55}^{1,1}, X), (n_{37}^{2,0}, n_{38}^{2,0}, X), (n_{37}^{2,0}, n_{38}^{2,0}, Y), \dots, (n_{44}^{2,1}, n_{54}^{2,1}, \text{Mpid}), (n_{44}^{2,1}, n_{55}^{2,1}, X)$
all-p-uses	$(n_{18}^{m,2}, (n_{22}^{m,2}, n_{23}^{m,2}), X), (n_{20}^{m,0}, (n_{22}^{m,0}, n_{23}^{m,0}), Y), (n_{44}^{0,1}, (n_{44}^{0,1}, n_{45}^{0,1}) X), (n_{44}^{0,1}, (n_{44}^{0,1}, n_{45}^{0,1}) Y), (n_{44}^{0,1}, (n_{49}^{0,1}, n_{50}^{0,1}) X), (n_{44}^{0,1}, (n_{49}^{0,1}, n_{50}^{0,1}) Y), (n_{44}^{0,1}, (n_{53}^{0,1}, n_{54}^{0,1}) X), (n_{44}^{0,1}, (n_{53}^{0,1}, n_{54}^{0,1}) Y), (n_{44}^{1,1}, (n_{44}^{1,1}, n_{45}^{1,1}) X), (n_{44}^{1,1}, (n_{44}^{1,1}, n_{45}^{1,1}) Y), (n_{44}^{1,1}, (n_{49}^{1,1}, n_{50}^{1,1}) X), (n_{44}^{1,1}, (n_{49}^{1,1}, n_{50}^{1,1}) Y), (n_{44}^{1,1}, (n_{53}^{1,1}, n_{54}^{1,1}) X), (n_{44}^{1,1}, (n_{53}^{1,1}, n_{54}^{1,1}) Y), (n_{44}^{2,1}, (n_{44}^{2,1}, n_{45}^{2,1}) X), (n_{44}^{2,1}, (n_{44}^{2,1}, n_{45}^{2,1}) Y), (n_{44}^{2,1}, (n_{49}^{2,1}, n_{50}^{2,1}) X), (n_{44}^{2,1}, (n_{49}^{2,1}, n_{50}^{2,1}) Y), (n_{44}^{2,1}, (n_{53}^{2,1}, n_{54}^{2,1}) X), (n_{44}^{2,1}, (n_{53}^{2,1}, n_{54}^{2,1}) Y)$
all-m-uses	$(n_1^{m,0}, (n_6^{m,0}, n_{36}^{0,0}), \text{NumX}), (n_1^{m,0}, (n_6^{m,0}, n_{36}^{0,0}), \text{NumY}), (n_5^{m,0}, (n_6^{m,0}, n_{36}^{0,0}), \text{Pidm}), (n_1^{m,0}, (n_7^{m,0}, n_{36}^{1,0}), \text{NumY}), (n_1^{m,0}, (n_7^{m,0}, n_{36}^{1,0}), \text{NumZ}), (n_5^{m,0}, (n_7^{m,0}, n_{36}^{1,0}), \text{Pidm}), (n_{18}^{m,2}, (n_{24}^{m,2}, n_{36}^{2,0}), X), (n_{20}^{m,2}, (n_{24}^{m,2}, n_{36}^{2,0}), Y), (n_2^{m,0}, (n_{24}^{m,2}, n_{36}^{2,0}), \text{pidm}), (n_{44}^{0,1}, n_{54}^{0,1}, n_{12}^{m,1}, X), (n_{44}^{0,1}, n_{54}^{0,1}, n_{19}^{m,2}, X), (n_{44}^{0,1}, n_{54}^{0,1}, n_{31}^{m,3}, X), (n_{44}^{1,1}, n_{54}^{1,1}, n_{12}^{m,1}, X), (n_{44}^{1,1}, n_{54}^{1,1}, n_{19}^{m,2}, X), (n_{44}^{1,1}, n_{54}^{1,1}, n_{31}^{m,3}, X), (n_{44}^{2,1}, n_{54}^{2,1}, n_{12}^{m,1}, X), (n_{44}^{2,1}, n_{54}^{2,1}, n_{19}^{m,2}, X), (n_{44}^{2,1}, n_{54}^{2,1}, n_{31}^{m,3}, X)$
all-f-uses	$(n_{13}^{m,1}, (n_{15}^{m,1}, n_8^{m,2}), X), (n_{37}^{0,0}, (n_{39}^{0,0}, n_{44}^{0,1}), X), (n_{37}^{0,0}, (n_{39}^{0,0}, n_{44}^{0,1}), Y), (n_{37}^{0,0}, (n_{39}^{0,0}, n_{44}^{0,1}), \text{Mpid}), (n_{37}^{1,0}, (n_{39}^{1,0}, n_{44}^{1,1}), X), (n_{37}^{1,0}, (n_{39}^{1,0}, n_{44}^{1,1}), Y), (n_{37}^{1,0}, (n_{39}^{1,0}, n_{44}^{1,1}), \text{Mpid}), (n_{37}^{2,0}, (n_{39}^{2,0}, n_{44}^{2,1}), X), (n_{37}^{2,0}, (n_{39}^{2,0}, n_{44}^{2,1}), Y), (n_{37}^{2,0}, (n_{39}^{2,0}, n_{44}^{2,1}), \text{Mpid})$
all-r-uses	$(n_{44}^{0,1}, (n_{47}^{0,1}, n_{44}^{0,1}) X), (n_{44}^{0,1}, (n_{47}^{0,1}, n_{44}^{0,1}) Y), (n_{44}^{0,1}, (n_{47}^{0,1}, n_{44}^{0,1}) X), (n_{44}^{0,1}, (n_{47}^{0,1}, n_{44}^{0,1}) Y), (n_{44}^{0,1}, (n_{51}^{0,1}, n_{44}^{0,1}) X), (n_{44}^{0,1}, (n_{51}^{0,1}, n_{44}^{0,1}) Y), (n_{44}^{0,1}, (n_{51}^{0,1}, n_{44}^{0,1}) X), (n_{44}^{0,1}, (n_{51}^{0,1}, n_{44}^{0,1}) Y), (n_{44}^{1,1}, (n_{47}^{1,1}, n_{44}^{1,1}) X), (n_{44}^{1,1}, (n_{47}^{1,1}, n_{44}^{1,1}) Y), (n_{44}^{1,1}, (n_{47}^{1,1}, n_{44}^{1,1}) X), (n_{44}^{1,1}, (n_{47}^{1,1}, n_{44}^{1,1}) Y), (n_{44}^{1,1}, (n_{51}^{1,1}, n_{44}^{1,1}) X), (n_{44}^{1,1}, (n_{51}^{1,1}, n_{44}^{1,1}) Y), (n_{44}^{1,1}, (n_{51}^{1,1}, n_{44}^{1,1}) X), (n_{44}^{1,1}, (n_{51}^{1,1}, n_{44}^{1,1}) Y), (n_{44}^{2,1}, (n_{47}^{2,1}, n_{44}^{2,1}) X), (n_{44}^{2,1}, (n_{47}^{2,1}, n_{44}^{2,1}) Y), (n_{44}^{2,1}, (n_{47}^{2,1}, n_{44}^{2,1}) X), (n_{44}^{2,1}, (n_{47}^{2,1}, n_{44}^{2,1}) Y), (n_{44}^{2,1}, (n_{51}^{2,1}, n_{44}^{2,1}) X), (n_{44}^{2,1}, (n_{51}^{2,1}, n_{44}^{2,1}) Y), (n_{44}^{2,1}, (n_{51}^{2,1}, n_{44}^{2,1}) X), (n_{44}^{2,1}, (n_{51}^{2,1}, n_{44}^{2,1}) Y)$

Fonte: Elaborada pelo autor.

Essas falhas são reveladas pela aplicação de critérios tradicionais (todas-arestas, todos-nós, etc) e testando cada GFC separadamente. Executando os dados de teste $\{\text{NumX} = 1, \text{NumY} = 2, \text{NumZ} = 1\}$, o nó **47^{p1}** é coberto e a primeira falha é

revelada. Considerando a segunda falha, os dados de teste {NumX = 2, NumY = 3, NumZ = 2} executa um caminho que cobre a aresta **(45^{p1}, 46^{p1})** e revela a falha. Para ambas as entradas, o programa executa o *loop* do nó **45^{p1}** para sempre, e uma falha é produzida. Estas situações ilustram a importância de investigar a aplicação de critérios para testes sequenciais em software paralelo.

Em Krawczyk e Wiszniewski, (1994) são apresentados dois tipos de falhas relacionadas a programas paralelos: observabilidade e falhas de bloqueio. A falha de observabilidade é um tipo especial de falha de domínio, relacionada a falhas de sincronização. Estas falhas podem ser observadas ou não durante a execução de um mesmo dado de teste; a observação depende do ambiente paralelo e do tempo de execução (não-determinismo). Falhas de bloqueio ocorrem quando o programa paralelo não termina sua execução, permanecendo bloqueado, esperando para sempre. Para ilustrar essa falha, considere novamente a execução do programa GCD com os dados de teste {NumX = 2, NumY = 3, NumZ = 2}. A saída esperada é (7) e os pontos de sincronização esperados entre os pares **send** e **receive** são **(6^{pm}, 36^{p0})**, **(7^{pm}, 36^{p1})**, **(54^{p0}, 12^{pm})** ou **(54^{p0}, 19^{pm})**, **(54^{p1}, 19^{pm})** ou **(54^{p1}, 12^{pm})**, **(24^{pm}, 36^{p2})**, **(54^{p2}, 31^{pm})**. É importante salientar que os nós **6^{pm}** e **7^{pm}** têm primitivas de recebimento não determinísticas.

Sem perda de generalidade, considere que os pontos de sincronização alcançados são **(54^{p0}, 12^{pm})** e **(54^{p1}, 19^{pm})**. Suponha que no nó **19^{pm}** a instrução *receive* foi erroneamente alterada para *receive* com *timeout*, uma primitiva que desbloqueia o *receive* depois de um tempo. Neste caso, a mensagem enviada pelo processo **p¹** pode não ser alcançada pelo *receive* no nó **19^{pm}**, antes da execução deste nó. Esta é uma falha de sincronização. Assim, no nó **22^{pm}** a condição será falsa e o processo será terminado. Essa falha pode ser revelada pelo critério de teste **todos-nós (N_{recv})**. **ValiErlang** (seção 5) fornece suporte neste caso, permitindo a análise do *trace* de execução. Ao analisar o *trace* de execução, o testador pode observar que um ponto de sincronização errado foi atingido.

Esta falha está relacionada ao não determinismo e a ocorrência da sincronização ilustrada não é garantida. Por exemplo, se o processo **p¹** for rápido o suficiente para ser executado, a mensagem enviada alcança o nó **19^{pm}** e a falha não será observada. Não obstante, as sincronizações ilustradas anteriormente são mais prováveis, considerando a ordem de criação dos processos.

Um tipo especial de erro de bloqueio é o *deadlock* (TANENBAUM, 2010), um

problema clássico em programas paralelos. Idealmente, ele deve ser detectado antes da execução do programa paralelo. Não é o foco dos critérios de teste propostos neste trabalho; no entanto, as informações extraídas dos programas paralelos durante a aplicação dos critérios de cobertura podem ser utilizadas para detectar estaticamente situações de *deadlock*.

Neste contexto, considere a criação dos processos p^0 e p^1 (nós 3^{pm} e 4^{pm}) e o envio de mensagens para os referidos processos nos nós 6^{pm} e 7^{pm} .

Em função da intercalação na execução de cada um dos processos (*interleaving*) envolvidos, as mensagens enviadas pelo processo p^m podem não ser alcançadas pelos *receives* nos nós 36^0 e 36^1 , respectivamente, dos processos p^0 e p^1 . Como consequência, os processos p^0 e p^1 ficarão bloqueados pela primitiva *receive* e, depois o processo p^m também vai bloquear pois não receberá as mensagens dos processos p^0 e p^1 ; isso vai gerar uma situação de *deadlock* na aplicação concorrente.

Esta situação é definida como um erro de existência em Clocksin e Mellish, (2003). Esta falha devido ao erro de existência é relacionada ao *interleaving* e foi abordada em Gotovos, Christakis e Sagonas (2011). Ela pode ser revelada pelo critério **todos-nós-sync** (N_{sync}), pelo critério **todos-nós-inter-funções** se antes ocorreu uma chamada de função ou pelo critério **todos-nós-recursivos** se antes ocorreu uma chamada recursiva. **ValiErlang** (seção 5) fornece suporte neste caso, por meio da análise do *trace* de execução, o testador pode observar a ordem errada de execução dos processos pela chamada de função ou chamada recursiva sem, seus respectivos, retornos.

Com relação às chamadas recursivas (empilhamento), **ValiErlang** pode guiar o testador por meio de dados de teste para detectar possíveis falhas relativas a alocação de memória (stack overflow). Apesar de poderem ser considerados tradicionais, tais erros ocorrem por falta de testes que indiquem tal situação. **ValiErlang** pode revelar esta falha por meio dos critérios **todos-nós-recursivos**. Para ilustrar, considere novamente a execução do programa GCD com os dados de teste {NumX = 1, NumY = 2, NumZ = 1}, neste caso é improvável ocorrer a falha. Entretanto, tal falha é provável que ocorra em função de um grande número de chamadas recursivas, por exemplo, com os dados de teste {NumX = 1, NumY = 10000000, NumZ = 1}.

4.7 Considerações finais

Este capítulo especificou um conjunto de critérios de teste baseado em um modelo de teste que representa as principais características dos programas ***Erlang***. Em seguida foi apresentada a aplicação dos critérios de teste em um exemplo de programa ***Erlang***. O exemplo detalhou todas as características que foram propostas pelo modelo de teste e também mostrou os elementos requeridos para os critérios de teste.

Na sequência, em uma seção sobre a eficácia dos critérios propostos para revelar falhas, foram mostrados alguns tipos de falhas que poderiam estar presentes no programa GCD, bem como direcionando os critérios que contribuem para revelar esses tipos de falhas.

O próximo capítulo apresenta a ferramenta ***ValiErlang*** que apoia à validação de programas ***Erlang***. A ***ValiErlang*** foi dividida em quatro estágios e todas as informações sobre sua arquitetura de construção e funcionamento serão descritas de forma detalhada estágio por estágio.

5 VALIERLANG

5.1 Considerações Iniciais

Este capítulo disserta os detalhes da ferramenta de teste **ValiErlang**, destacando sua arquitetura composta por cinco módulos. O primeiro detalhamento é do módulo **Parse**, que abstrai as informações do código fonte com base no modelo de teste descrito na seção 4.2. Na sequência são especificados os outros quatro módulos da **ValiErlang**, são eles: **Inst**, **Elem**, **Exec** e **Eval**. A apresentação de cada um destes módulos é realizada de forma sequencial devido à dependência das informações que são geradas pelo módulo anterior. O primeiro módulo executado é o módulo **Inst**, que utiliza as informações do módulo **Parse**, em seguida, o módulo **Elem** utiliza as informações geradas pelo módulo **Inst** e, assim sucessivamente para os módulos **Exec** e **Eval**.

O módulo **Parse** foi desenvolvido em **Erlang**, os demais módulos foram desenvolvidos em **Java**. Para executar a **ValiErlang**, é necessário fazer a instalação das máquinas virtuais² **Erlang** e **Java**. Isso permite o uso da **ValiErlang** em sistemas operacionais que tem suporte para as respectivas máquina virtuais, por exemplo, **Windows**, **MAC/OS**, **Linux**, **Solaris** e **BSD**.

Todos os módulos são abordados com explicações sobre seu funcionamento e com exemplos dos arquivos texto que são gerados por cada um dos módulos. A abordagem foi realizada desta forma para um melhor entendimento da **ValiErlang**, ou seja, todos os arquivos gerados pelos módulos são ilustrados.

Este capítulo não aborda nenhum tipo de análise de dados em relação à ferramenta **ValiErlang**, apenas demonstra o seu funcionamento por meio de um programa exemplo.

O capítulo mostra ao leitor as etapas da atividade de teste por meio da

² Disponíveis em: https://www.java.com/pr_BR/download e <https://www.erlang.org/downloads>

ValiErlang. Com tal detalhamento, é esperado que o leitor entenda todas as funcionalidades e também as informações que são geradas para cada módulo. As informações apresentadas neste capítulo são importantes para mostrar a aplicação dos critérios de teste estrutural, que foram descritos na seção 4.3. Além disso, é por meio da **ValiErlang** que foram realizados os estudos experimentais para analisar a eficácia dos critérios propostos.

5.2 Uma Visão Geral da ValiErlang

A **ValiErlang** foi implementada para apoiar a efetiva aplicação dos critérios de teste definidos na seção anterior. A **ValiErlang** trabalha com o conceito de sessões de teste, que podem ser configuradas para testar um determinado programa paralelo e permite parar a atividade de teste e retomá-la mais tarde. Basicamente, a ferramenta fornece funcionalidades para criar sessões de teste, salvar e executar dados de teste e gerar em um arquivo as informações de cobertura dos critérios de teste. A **ValiErlang** não realiza a geração automática de casos de teste e também não avalia se os resultados estão de acordo com os casos de teste. O testador é responsável em gerar os casos de teste manualmente e também fazer o papel de oráculo. A arquitetura da ferramenta é mostrada na Figura 5.1.

Figura 5.1 - Arquitetura completa da ferramenta **ValiErlang**.

Fonte: Elaborada pelo autor.

O objetivo de **ValiErlang** é validar programas sequenciais e/ou concorrentes escritos em **Erlang** por meio de critérios de testes estruturais. A ferramenta foi

construída em duas etapas. O primeiro é o módulo **Parse**, que abstrai as informações do código-fonte, e o segundo foi dividido em quatro módulos: módulo **Inst** que realiza toda a análise estática do programa; módulo **Elem** lista os elementos requeridos; módulo **Eval** realiza a avaliação de casos de teste (resultados de cobertura); e módulo **Exec** envolve a execução do programa e a geração dos caminhos executados. Os detalhes da arquitetura serão exemplificados durante as próximas seções e os exemplos serão baseados no programa GCD mostrado na Figura 4.2.

Os diferentes módulos que compõem a **ValiErlang** serão descritos nas próximas seções.

5.3 Módulo Parse

No primeiro módulo é utilizada uma biblioteca da distribuição **Erlang**/OTP para transformar programas **Erlang**, em termos estruturados **Erlang**. Esta representação é conhecida como *Abstract Syntax Tree* (AST) **Erlang** (FRANCO, 2010). A partir da AST **Erlang** é possível realizar uma análise sintática do código **Erlang** e extrair informações de nós e arestas que o modelo de teste necessita.

Todos os módulos se comunicam através de arquivos que são gravados a cada etapa concluída, o módulo **Inst** é dividido em três estágios. O funcionamento de cada módulo e a interação entre eles são especificados nas próximas seções deste capítulo.

O módulo **Parse**, usa a biblioteca Syntax Tools (2016) da distribuição **Erlang**/OTP. O módulo `parse_file` [epp] abstrai uma AST a partir de um código-fonte **Erlang** (SMERL, 2016). Com o código transformado em AST, é possível manipular esta AST e extrair informações sobre o fluxo de controle e fluxo de dados. A Figura 5.2 mostra a correspondência de um pequeno programa **Erlang** e sua representação em AST.

Como pode ser observado na Figura 5.2, uma AST é composta por listas de formulários que são *tuplas* que representam as construções de alto nível, como declarações de atributos e funções de uma forma bem estruturada. Por exemplo, `{attribute, 1, module, hello}` é uma *tupla* que representa a declaração do módulo na linha 2 do código fonte: `-module(hello) .`

Figura 5.2- Exemplo de um código fonte a sua correspondência em AST.

Source code
<pre>/*1*/ -module(hello) . /*2*/ -compile(export_all) . /*3*/ start() -> /*4*/ io:format('Hello World!') .</pre>
AST
<pre>[[{attribute,1,file,[[hello.erl],1}}, {attribute,1,module,hello}, {attribute,2,compile,export_all}, {function,3,start,0,[[{clause,3,[],[]}, [{call,4,{remote,4,{atom,4,io},{atom,4,format}},[{atom,4,'Hello World!'}]]]}]],{eof,4}]</pre>

Fonte: Elaborada pelo autor.

Depois que o código fonte foi transformado em AST, o módulo **Parse** extrai informações sobre fluxo de controle, fluxo de dados de acordo com o que foi especificado pelo modelo de teste e grava em arquivo texto. A Figura 5.3 mostra um trecho do arquivo texto que é gravado pelo módulo **Parse** baseado no programa da Figura 4.2.

Figura 5.3- Exemplo de um trecho do arquivo gravado pelo módulo Parse.

```
program,gcd2,0,0,0,0
function,1,start,3,0,0
start,3,1,nStart,0,0
start,3,99,def,'NumX',nStart
start,3,99,def,'NumY',nStart
start,3,99,def,'NumZ',nStart
start,4,99,def,pidm,nReg_Global
start,4,1,nReg_Global,0,0
start,5,99,def,pid0,nDef_Global_Var
start,5,1,nSpawn,gcd2,wait
start,6,99,def,pid1,nDef_Global_Var
start,6,1,nSpawn,gcd2,wait
```

Fonte: Elaborada pelo autor.

A Figura 5.3 contém informações do código fonte, como, funções, fluxo de dados e fluxo de controle. O arquivo texto grava por linha cada ocorrência de tais informações, e cada linha (que representa os nós, por exemplo) é composta por seis colunas, e de acordo com a informação extraída pode ou não preencher todas as colunas.

Na segunda linha são utilizadas apenas quatro colunas: a primeira coluna é

um identificador para início de uma função; a segunda é um número sequencial que identifica a função; na sequência é o nome da função; e a quarta coluna indica a linha no código fonte.

A terceira linha se refere a informações da função **start** na primeira coluna e o número da linha na segunda coluna. A terceira coluna é um número sequencial que controla blocos de instruções, como por exemplo, instruções que pertencem a uma estrutura de decisão. O nó **nStart**, definido na quarta coluna, não utiliza as outras duas colunas.

A quarta linha mostra um exemplo de fluxo de dados. Na terceira coluna identifica que ocorreu uma definição de variável, cujo nome é armazenado na quarta coluna e esta definição ocorreu no nó (**nStart**) armazenado na quinta coluna.

5.4 Módulo Inst

O módulo **Inst** é responsável pela extração das informações de fluxo e pela instrumentação do programa com as instruções que farão o registro dos caminhos executados, e foi dividido em três estágios, conforme ilustrado pela Figura 5.4. Esse módulo recebe como entrada o arquivo texto com as informações que foram geradas e gravadas pelo módulo **Parse**.

Figura 5.4- Arquitetura parcial da **ValiErlang** com o módulo Inst.

Fonte: Elaborada pelo autor.

Na primeira fase as informações geradas pelo módulo **Parse** são otimizadas e estruturadas. A otimização é realizada nas informações relacionadas aos nós, pois o módulo **Parse** gera um nó para cada instrução do programa original. Por exemplo,

três nós em sequência, que definem três variáveis, podem ser representados por apenas um nó no CFG.

Após a otimização de nós, os dados são estruturados e separados para atender aos requisitos dos próximos estágios do módulo **Inst** e também para os demais módulos da **ValiErlang**. Para isso, são gerados quatro novos arquivos com informações relacionadas aos fluxos de controle e de dados, funções, processos e ao CFG. Este conjunto de informações é gerado para cada processo.

Para um melhor entendimento, a Tabela 5.1 contém uma parte dos dados dos quatro arquivos que são gerados pelo módulo **Inst** neste estágio. No arquivo com dados dos processos existem quatro colunas, a primeira coluna é um identificador único para cada processo; a segunda é o nome que foi dado ao processo; na sequência é o nome do programa e a função que o processo inicia sua execução. Esses dados são gravados de duas formas, na primeira forma os dados são obtidos de forma estática, ou seja, processos que são executados pela linha de comando. Na segunda, os dados são obtidos de forma dinâmica. Para isso é realizada uma análise do código fonte para identificar a instrução **spawn**, responsável pela criação de processos em programas **Erlang**. Esse arquivo é utilizado para a geração do PCFG, no terceiro estágio do módulo **Inst**.

Tabela 5.1- Informações geradas no primeiro estágio do módulo Inst.

Arquivos	Sequência	Informações geradas em arquivos
Processos	1	1,pidm,gcd2,start
	2	2,pid0,gcd2,wait
	3	3,pid1,gcd2,wait
	4	4,P2,gcd2,wait
Funções	5	gcd2,start,waitx
	6	gcd2,waitx,waity
	7	gcd2,waity,fim
	8	gcd2,wait,calc
Fluxo de dados e fluxo de controle	9	1,start,3,1,nStart,0,0
	10	2,start,4,1,nReg_Global,0,0
	11	3,start,5,1,nSpawn,gcd2,wait
	12	4,start,3,def,'NumX',nStart,0
	13	5,start,3,def,'NumY',nStart,0
CFG	14	1,1,nStart,2,nReg_Global,edgeIntra,start
	15	2,2,nReg_Global,3,nSpawn,edgeIntra,start

Fonte: Elaborada pelo autor.

As informações sobre funções são armazenadas em três colunas, a primeira coluna identifica o programa, na segunda é o nome da função e na terceira é a função executada na sequência. Esse arquivo é utilizado para a geração do código

fonte instrumentado, e também para identificar a sequências de definições e usos de variáveis que foi detalhada pelo modelo de teste e funções pertencentes ao processo *p*.

As informações sobre fluxo de dados e fluxo de controle são armazenadas em sete colunas, e de acordo com a informação extraída pode preencher todas as colunas. Na sequência 10 da Tabela 5.1, a primeira coluna é um identificador único; a segunda coluna indica a função e o número da linha na terceira coluna. A quarta coluna é um número sequencial que identifica os blocos de instruções do programa. O nó *nReg_Global* é armazenado na quinta coluna, este tipo de nó não utiliza as outras duas colunas. Esses dados são utilizados para geração do programa instrumentado e para gerar os elementos requeridos.

As informações sobre o CFG são armazenadas em sete colunas, e são armazenadas por um conjunto de dois nós que são interligados por uma aresta. A primeira coluna é um identificador único do CFG; da segunda até a quinta coluna armazena o identificador e o nome dos nós, que são interligados por uma aresta armazenada na sexta coluna. A última coluna identifica a função que o primeiro nó está vinculado. Essas informações são utilizadas para gerar o PCFG.

Para ilustrar, a Figura 5.5 apresenta o CFG (de apenas um processo) do programa GCD mostrado na Figura 4.2.

O CFG foi gerado com informações sobre nós e arestas pela **ValiErlang**. A **ValiErlang** realiza a análise sintática e semântica do programa, de acordo com a gramática da linguagem **Erlang**, extraindo as informações necessárias para instrumentação.

A ferramenta gera um arquivo de texto que é importado pelo software **GraphViz**. Na segunda fase o programa fonte original é processado pelo módulo **Inst** junto com as informações relacionadas aos fluxos de controle e das funções para gerar o programa instrumentado.

O programa instrumentado é obtido através da inserção de instruções de *checkpoint* no programa a ser testado. Essas instruções não alteram a semântica do programa. Eles apenas escrevem as informações necessárias em um arquivo de rastro, registrando os nós, informações sobre o fluxo de dados e o identificador de processo nos comandos *send* e *receive*. O programa instrumentado produzirá os caminhos executados em cada processo, bem como a sequência de sincronização produzida dentro de um caso de teste. O programa instrumentado (parcial) é

Figura 5.6 : Programa GCD instrumentado

```

1  -module(gcd2).
2  -export([start/3, wait/0]).
3  start(NumX, NumY, NumZ) ->
4      file:write_file("gcd2-trace.txt",io_lib:fwrite("1,start,nStart,0,0~n",[]),[append]),
5      file:write_file("gcd2-trace.txt",io_lib:fwrite("4,start,def,'NumX',nStart,0~n",[]),[append]),
6      file:write_file("gcd2-trace.txt",io_lib:fwrite("5,start,def,'NumY',nStart,0~n",[]),[append]),
7      file:write_file("gcd2-trace.txt",io_lib:fwrite("6,start,def,'NumZ',nStart,0~n",[]),[append]),
8      global:register_name(pidm,self()),
9      file:write_file("gcd2-trace.txt",io_lib:fwrite("2,start,nReg_Global,0,0~n",[]),[append]),
10     file:write_file("gcd2-trace.txt",io_lib:fwrite("7,start,def,pidm,nStart,0~n",[]),[append]),
11     global:register_name(pid0,spawn(gcd2,wait,[])),
12     file:write_file("gcd2-trace.txt",io_lib:fwrite("3,start,nSpawn,gcd2,wait~n",[]),[append]),
13     file:write_file("gcd2-trace.txt",io_lib:fwrite("8,start,def,pid0,nStart,0~n",[]),[append]),

```

Fonte: Elaborada pelo autor.

Para geração do PCFG, o módulo **Inst** obtém os dados da primeira linha do arquivo que contém as informações sobre os processos. De acordo com esses dados, mapeia todas as funções que fazem parte deste processo e gera um CFG para o referido processo, esse mapeamento é possível com as informações contidas nos arquivos de dados com as funções.

O PCFG é composto por todos os CFGs dos processos que fazem parte da aplicação concorrente, e então as arestas interprocessos são inseridas a partir das primitivas *send* e *receive*. Uma parte do PCFG é mostrado pela Figura 5.7. A aresta interprocesso interliga a primitiva *send* da linha 10 (função *start*) do programa GCD (Figura 4.2) do processo principal (pidm), com a primitiva *receive* da linha 38 (função *wait*) de um dos processos secundários (pid0).

5.5 Módulo Elem

O módulo **Elem** é responsável pela geração dos elementos requeridos dos critérios de teste de cobertura definidos na seção 4.2. Esse módulo recebe como entrada o arquivo texto com as informações sobre os processos, funções, fluxo de dados, fluxo de controle e o PCFG que foram geradas e gravadas pelo módulo **Inst**, respectivamente, nos estágios 1 e 3.

De acordo com as informações recebidas, o módulo **Elem** gera um novo arquivo com informações dos elementos requeridos pelos critérios de teste, que foram especificados pelo modelo de teste na seção 4.1.

Para ilustrar, a Figura 5.8 contém uma parte do arquivo que é gerado módulo **Elem** considerando o programa GCD. Os elementos requeridos mostrados pela Figura 5.8 correspondem apenas ao critério **todos-nós-sync**.

Figura 5.7- PCFG parcial do programa GCD.

Fonte: Elaborada pelo autor.

No arquivo são gravados dados separados por cinco colunas: a primeira coluna corresponde ao identificador único de cada nó; a segunda é o nome que foi dado ao nó; a terceira indica o tipo de primitiva de comunicação especificada pelo conjunto **B** do modelo de teste (*send* ou *receive*); na sequência encontra-se o nome dado ao processo e um número que indica se o critério foi coberto ou não pelos casos de teste. A informação da última coluna será alterada pelo módulo **Eval**, caso algum caso de teste atenda o respectivo elemento requerido. A primeira linha do arquivo indica que no nó 6 existe uma primitiva **send** para o processo pid0.

Figura 5.8- Arquivo com os elementos requeridos correspondentes ao critério todos-nós-sync.

```
6,nSync,BS,pid0,0
7,nSync,BS,pid1,0
12,nSync,BR,RECEIVE,0
19,nSync,BR,RECEIVE,0
24,nSync,BS,P2,0
31,nSync,BR,RECEIVE,0
36,nSync,BR,RECEIVE,0
54,nSync,BS,Mpid,0
36,nSync,BR,RECEIVE,0
54,nSync,BS,Mpid,0
36,nSync,BR,RECEIVE,0
54,nSync,BS,Mpid,0
```

Fonte: Elaborada pelo autor.

5.6 Módulo Exec

Este módulo é responsável pela execução do programa instrumentado com os dados de teste, sua arquitetura é mostrada pela Figura 5.1. Tais informações são determinadas pelo testador no início do processo de teste.

O módulo **Exec** armazena os casos de teste, os parâmetros de execução e os respectivos rastros de execução. O rastro de execução contém os caminhos executados por um ou vários processos, em caso de uma aplicação concorrente. O rastro será usado pela **ValiErlang** para determinar os elementos que foram cobertos.

A execução de uma aplicação pode ser realizada de várias formas. A forma mais simples é indicar apenas o parâmetro **-p** e o nome do programa a ser testado, por exemplo:

```
valierl_exec -p program.erl
```

No exemplo mostrado anteriormente é considerado que a função padrão é a **start()** e será executada pela VM **Erlang**. Em casos que não é utilizada a função padrão **start()**, faz-se necessário indicar o nome da função a ser executada com o parâmetro **-f**, por exemplo:

```
valierl_exec -p program.erl -f function
```

Caso a execução do programa necessite de dados de entrada, é preciso informar o parâmetro **-a**, por exemplo:

valierl_exec -p program.erl -f function -a arg1 -a arg2 -a argn

Após a execução, o testador pode visualizar as saídas e o rastro de execução para determinar se a saída obtida é a mesma esperada. Se não for, uma falha foi identificada e pode ser corrigida antes de continuar o teste.

A Figura 5.9 contém o arquivo parcial do rastro de execução dos processos do programa GCD. Os dados de teste utilizados foram: {NumX = 1, NumY = 2, NumZ = 1}. Para este teste, o processo P2 não executa porque o resultado final já foi produzido pelos processos pid0 e pid1.

De acordo com a execução realizada, os caminhos executados foram:

$\pi \text{ pidm} = \{1, 2, 3, 4, 5, 6, 7, 8, 11, 12, 13, 14, 15, 18, 19, 20, 21, 22, 27, 28, 29, 16, 17, 9, 10\}$.

$\pi \text{ pid0} = \{35, 36, 37, 38, 39, 44, 45, 46, 47, 44, 45, 49, 53, 54, 55, 56, 48, 56, 40, 43\}$.

$\pi \text{ pid1} = \{35, 36, 37, 38, 39, 44, 45, 49, 50, 51, 44, 45, 49, 53, 54, 55, 56, 52, 56, 40, 43\}$.

Figura 5.9- Rastro parcial do arquivo gerado pelo módulo Exec da **ValiErlang**.

Processo Pidm	Processo pid0	Processo pid1
1, start, nStart, 0, 0	35, wait, nStart, 0, 0	35, wait, nStart, 0, 0
57, start, def, NumX, nStart	36, wait, nSync, receive, 0	36, wait, nSync, receive, 0
58, start, def, NumY, nStart	37, wait, nClauseRecv, 0, 0	37, wait, nClauseRecv, 0, 0
59, start, def, NumZ, nStart	93, wait, def, X, nSync	93, wait, def, X, nSync
2, start, nReg_Global, 0, 0	94, wait, def, Y, nSync	94, wait, def, Y, nSync
60, start, def, pidm, nDef_Var_Global	95, wait, def, Mpid, nSync	95, wait, def, Mpid, nSync
3, start, nSpawn, gcd2pid0, wait	38, wait, nIo, 0, 0	38, wait, nIo, 0, 0
61, start, def, pid0, nDef_Var_Global	96, wait, c-use, X, nExpression	96, wait, c-use, X, nExpression
4, start, nSpawn, gcd2pid1, wait	97, wait, c-use, Y, nExpression	97, wait, c-use, Y, nExpression
62, start, def, pid1, nDef_Var_Global	98, wait, c-use, Mpid, nExpression	98, wait, c-use, Mpid, nExpression
5, start, nDef_Global_Var, 0, 0	39, wait, nCall, 0, calc	39, wait, nCall, 0, calc
63, start, def, Pidm, nDef_Var	44, calc, nStart, 0, 0	49, calc, nGuard, 0, 0
64, start, c-use, pidm, Uso_Atcm_Global	45, calc, nGuard, 0, 0	114, calc, def, X, nStart
65, start, def, Pid0, nDef_Var	103, calc, def, X, nStart	115, calc, def, Y, nStart
66, start, c-use, pid0, Uso_Atcm_Global	104, calc, def, Y, nStart	116, calc, def, Mpid, nStart
67, start, def, Pid1, nDef_Var	105, calc, def, Mpid, nStart	117, calc, p-use, X, Guard
68, start, c-use, pid1, Uso_Atcm_Global	106, calc, p-use, X, Guard	118, calc, p-use, Y, Guard
6, start, nSync, send, 0	107, calc, p-use, Y, Guard	50, calc, nIo, 0, 0
69, start, m-use, NumX, nSync	46, calc, nIo, 0, 0	119, calc, c-use, X, nExpression
70, start, m-use, NumY, nSync	108, calc, c-use, X, nExpression	120, calc, c-use, Y, nExpression
71, start, m-use, Pidm, nSync	109, calc, c-use, Y, nExpression	51, calc, nCallRec, 0, calc
72, start, c-use, Pid0, nExpression	47, calc, nCallRec, 0, calc	53, calc, nGuard, 0, 0
7, start, nSync, send, 0	44, calc, nStart, 0, 0	125, calc, def, X, nStart
73, start, m-use, NumY, nSync	45, calc, nGuard, 0, 0	126, calc, def, Y, nStart
74, start, m-use, NumZ, nSync	103, calc, def, X, nStart	127, calc, def, Mpid, nStart
75, start, m-use, Pidm, nSync	104, calc, def, Y, nStart	128, calc, p-use, X, Guard
76, start, c-use, Pid1, nExpression	105, calc, def, Mpid, nStart	129, calc, p-use, Y, Guard
8, start, nCall, 0, waitx	106, calc, p-use, X, Guard	54, calc, nSync, send, 0
11, waitx, nStart, 0, 0		

Fonte: Elaborada pelo autor.

Note que o caminho executado que foi registrado no rastro não ilustra o caminho completo do CFG, isso ocorreu devido ao uso de **guards** no programa GCD. A **ValiErlang** leva isso em conta e analisa os **guards** (condições) da função.

Se apenas o segundo **guard** foi executado, subentende-se que o primeiro não executou, pois a condição era falsa. Neste caso, a ferramenta considera o caminho completo para efetuar a avaliação de cobertura no módulo **Eval**.

Pode-se notar a execução do nó 44 (**nStart**) duas vezes no rastro do processo **pid0**. Isso devido a uma chamada recursiva por meio do nó 47 (**nCallRec**) que ocorreu durante a execução. A recursão é um aspecto dinâmico (pois depende dos dados de entrada) e a cobertura desse empilhamento de chamadas recursivas é considerada pela **ValiErlang**, especificamente pelo módulo **Eval**.

5.7 Módulo Eval

A avaliação da cobertura dos casos de teste é responsabilidade do módulo **Eval**. Esse módulo recebe como entrada os elementos requeridos que foram gerados pelo módulo **Elem**, os caminhos executados gerados pelo módulo **Exec** e os critérios de teste que se pretende avaliar. O módulo **Eval** compara os elementos requeridos com os caminhos executados, e de acordo com o critério que deseja avaliar, identifica se ele foi coberto ou não.

O módulo **Eval** gera um arquivo de saída idêntico ao arquivo gerado pelo módulo **Elem**. A diferença é que na última coluna a informação é alterada para “covered” caso o critério foi coberto, ou “not” se não foi coberto.

Um exemplo é mostrado no lado esquerdo da Figura 5.10, baseado nas informações dos elementos requeridos da Figura 5.8 e nos caminhos executados pelos processos. Os critérios cobertos são identificados pela identificação “covered” na última coluna de cada linha.

O arquivo com os resultados sumarizados é mostrado do lado direito da Figura 5.10. As informações dos elementos cobertos e inviáveis correspondem aos critérios de todos os processos. Os elementos inviáveis são indicados no arquivo para uma melhor análise do testador.

Em relação às chamadas recursivas, a cobertura dos elementos relacionados aos nós **nCalrec** e **nRetrec** do processo **pid1** é mostrado na figura 5.11. A quinta coluna mostra o número de vezes que os nós foram executados. Para que ocorra o funcionamento correto da recursão, este número deve ser igual para os dois nós.

A Figura 5.12 mostra a estrutura dos diretórios que são criados pela **ValiErlang** durante o processo de teste de uma aplicação. A **ValiErlang** cria um

diretório com o nome do programa **Erlang** a ser testado e dentro dele mais quatro diretórios para armazenar os arquivos dos módulos **Inst**, **Elem**, **Exec** e **Eval**.

Figura 5.10- Informações sobre a cobertura do critério todos-nós-sync e o arquivo com o sumário dos resultados.

Process Pidm	-----
6,nSync,BS,pid0,COVERED	Coverage result - nsync.txt
7,nSync,BS,pid1,COVERED	file,process,line
12,nSync,BR,RECEIVE,COVERED	
19,nSync,BR,RECEIVE,COVERED	Coverage total = 12
24,nSync,BS,P2,COVERED	Infeasible total = 0
31,nSync,BR,RECEIVE,COVERED	-----
Process P2	Coverage result - nrecv.txt
36,nSync,BR,RECEIVE,COVERED	file,process,line
54,nSync,BS,Mpid,COVERED	nrecv.txt,1,2
Process pid0	nrecv.txt,2,2
36,nSync,BR,RECEIVE,COVERED	nrecv.txt,3,2
54,nSync,BS,Mpid,COVERED	-----
Process pid1	Coverage total = 6
36,nSync,BR,RECEIVE,COVERED	Infeasible total = 3
54,nSync,BS,Mpid,COVERED	-----

Fonte: Elaborada pelo autor.

Figura 5.11- Informações sobre a cobertura das chamadas recursivas.

Process pid1	
ncallrec	nretrec
47,nCallRec,calc,calc,0,COVERAGE	48,nRecRet,calc,calc,0,COVERAGE
47,nCallRec,calc,calc,1,COVERAGE	48,nRecRet,calc,calc,1,COVERAGE
51,nCallRec,calc,calc,0,COVERAGE	52,nRecRet,calc,calc,0,COVERAGE
51,nCallRec,calc,calc,1,COVERAGE	52,nRecRet,calc,calc,1,COVERAGE
51,nCallRec,calc,calc,2,COVERAGE	52,nRecRet,calc,calc,2,COVERAGE

Fonte: Elaborada pelo autor.

Os diretórios dos módulos criam um diretório para cada processo (em caso de uma aplicação concorrente) e grava os arquivos separados para cada tipo de nós, arestas, definições e usos de variáveis. O módulo **Exec** cria um diretório para cada um dos testes realizados.

5.8 Considerações finais

Este capítulo descreveu a **ValiErlang**, uma ferramenta de teste estrutural para programas **Erlang**, detalhando seus cinco módulos. Cada módulo foi apresentado com exemplos e figuras para demonstrar sua funcionalidade e

integração com os demais módulos.

Figura 5.12- Estrutura de diretórios criada pela **ValiErlang**.

Nome	Data de modificaç...	Tipo	Tamanho
cfg.dot	13/01/2017 12:11	Arquivo DOT	1 KB
cfg.txt	13/01/2017 12:11	Documento de Te...	2 KB
countlines.bat	26/01/2017 14:40	Arquivo em Lotes ...	3 KB
countlines-onlydata.bat	26/01/2017 16:42	Arquivo em Lotes ...	2 KB
cuse.txt	26/01/2017 18:50	Documento de Te...	1 KB
cuseassoc.txt	16/01/2017 14:58	Documento de Te...	1 KB
def.txt	26/01/2017 18:50	Documento de Te...	1 KB
ecall.txt	26/01/2017 18:50	Documento de Te...	1 KB
eintra.txt	26/01/2017 18:50	Documento de Te...	1 KB
elink.txt	26/01/2017 18:50	Documento de Te...	0 KB
ereturn.txt	26/01/2017 18:50	Documento de Te...	1 KB
estack.txt	26/01/2017 18:50	Documento de Te...	1 KB
eunlink.txt	26/01/2017 18:50	Documento de Te...	0 KB
eunstack.txt	26/01/2017 18:50	Documento de Te...	1 KB
fuse.txt	26/01/2017 18:50	Documento de Te...	1 KB
fuseassoc.txt	16/01/2017 14:58	Documento de Te...	1 KB
muse.txt	26/01/2017 18:50	Documento de Te...	1 KB
museassoc.txt	17/01/2017 01:21	Documento de Te...	1 KB
ncall.txt	26/01/2017 18:50	Documento de Te...	1 KB
ncallrec.txt	26/01/2017 18:50	Documento de Te...	1 KB
ncase.txt	26/01/2017 18:50	Documento de Te...	0 KB
nend.txt	26/01/2017 18:50	Documento de Te...	1 KB
nquard.txt	26/01/2017 18:50	Documento de Te...	1 KB

Fonte: Elaborada pelo autor.

A **ValiErlang** tem como objetivo cumprir todas as etapas do teste estrutural para programas **Erlang**. Para isso todas as características propostas pelo modelo de teste do capítulo anterior foram utilizadas. Os resultados mostrados neste capítulo ilustram o funcionamento da ferramenta, a qual implementa o que foi proposto para o modelo e os critérios de teste estrutural para programas **Erlang**.

O próximo capítulo apresenta os resultados da aplicação dos critérios de teste para programas **Erlang** maiores e reais. Os resultados foram obtidos pelo uso da **ValiErlang** com a aplicação do teste estrutural em seis programas implementados em **Erlang** com características distintas.

6 ESTUDOS EXPERIMENTAIS

6.1 Considerações iniciais

Este capítulo descreve os resultados obtidos com a aplicação dos critérios de testes em seis programas **Erlang**, considerando o uso do modelo de teste e da ferramenta de teste **ValiErlang** propostos. Os programas **Erlang** possuem as características necessárias para avaliar as propostas deste projeto de pesquisa, em especial por apresentarem principalmente estas características: estruturas condicionais, *guards*, *pattern matching*, recursão, comunicação entre funções, comunicação entre processos e *links* entre processos.

Na sequência é apresentada uma discussão a respeito dos resultados obtidos, uma análise da eficácia dos critérios, em relação aos tipos de erros existentes, e uma breve descrição das principais características das ferramentas de teste identificadas no mapeamento sistemático e outros trabalhos. Para finalizar, essas ferramentas de teste são comparadas com a **ValiErlang** em relação aos aspectos da linguagem **Erlang** e da metodologia de teste de software. Esta comparação tem como objetivo caracterizar a **ValiErlang** no contexto de teste para programas **Erlang**.

6.2 Aplicação dos critérios de testes

Os resultados da aplicação dos critérios de testes para programas **Erlang** são apresentados nesta seção. Para a realização desse estudo foram utilizados seis programas **Erlang** que fornecem as características necessárias para validar os critérios de teste que foram propostos a partir do modelo de teste.

Os programas implementados em **Erlang** que foram utilizados na realização deste experimento são:

gcd - calcula o máximo divisor comum de três números (exemplo utilizado na Figura 4.2). Este programa é composto por até quatro processos e pode explorar a maioria das características que foram estabelecidas pelo modelo de teste, chamadas de funções, recursão, primitivas *send* e *receive*, cláusulas *receive*, *guards* e *pattern matching*;

omega - simula o funcionamento de uma rede ômega (Tanenbaum, 2007), que é uma rede de comutação multiestágio. Este programa possui treze processos e o objetivo para sua utilização é explorar as arestas de sincronização entre as primitivas *send* e *receive*. O perfil da aplicação também foi um fator para sua escolha, aplicações para roteamento e comutação de pacotes foi uma das inspirações para a concepção da linguagem **Erlang**.

crossbar - simula o funcionamento de uma rede crossbar (Tanenbaum, 2007), que é uma rede de comutação utilizada há muito tempo em centrais de comutação telefônicas para conectar linhas de entrada a um conjunto de linhas de saída. Este programa possui treze processos e o objetivo para sua utilização são os mesmos que foram descritos pelo programa **omega**.

mon_pingpong – realiza a troca de mensagens entre processos. Este programa possui quatro processos, sendo dois processos que monitoram dois processos (*ping* e *pong*), os quais trocam mensagens por meio das primitivas *send* e *receive*. Este programa possui *link* entre os processos e essa característica permite a construção de sistemas tolerantes a falhas (esse foi o principal objetivo pela escolha deste programa).

Os programas **Erlang** a seguir foram utilizados na realização deste experimento, entretanto, foram desenvolvidos por terceiros. Tais programas são importantes para testar a **ValiErlang** com estilos diferentes de programação e não se limitar apenas ao teste com programas próprios.

messenger³ – é um programa que permite um processo cliente conectar ao processo servidor e enviar mensagens simples para outro processo cliente conectado. Este programa simula uma aplicação de chat e o teste foi realizado com dois processos clientes e o processo servidor. O objetivo é explorar as mesmas características do programa **gcd**, a diferença é o uso de estruturas **case** ao invés das cláusulas **if** e **guards**.

³ http://erlang.org/documentation/doc-5.6/doc/getting_started/part_frame.html

eldap – o programa **eldap** (Törnkvist, 2014) fornece uma *Application Programming Interface* (API) cliente para acesso a um servidor *Lightweight Directory Access Protocol* (LDAP). Este programa possui dois processos, sendo um processo servidor que recebe as requisições de um processo cliente. Este é um software utilizado na indústria e possui um grande número de informações (em relação aos demais programas já descritos) de nós, arestas, definições e usos de variáveis, esse foi o principal objetivo pela escolha deste programa.

Para cada programa, um conjunto adequado de testes foi gerado manualmente. A exceção foi o programa **eldap**, no qual o conjunto de teste foi gerado a partir do manual de uso do software. Por consequência disso, o percentual de cobertura dos elementos cobertos pelos casos de teste foi de aproximadamente 73%. A opção por usar um percentual mais baixo de cobertura ilustra um comportamento possível (e usual) da atividade de teste, no qual os testes são conduzidos até se atingir uma relação custo x benefício esperada no projeto de desenvolvimento do software. Os programas foram executados com os conjuntos de teste usando a ferramenta **ValiErlang** e observando a cobertura obtida. Foram adicionados novos casos de teste até que todos os elementos necessários exequíveis tenham sido executados. Nesta fase, os elementos não executáveis foram identificados manualmente. O custo foi avaliado considerando o tamanho dos conjuntos de teste adequados. Além disso, foram utilizadas informações sobre a quantidade de elementos não executáveis necessários para calcular o custo de cada critério de teste.

A Tabela 6.1 apresenta o total de elementos requeridos, número de elementos requeridos cobertos e não executáveis para cada critério. A quantidade adequada de casos de teste que realmente contribuiu para cobrir elementos na ordem executada é apresentada na Tabela 6.2.

Na verdade, algum esforço é necessário para identificar elementos não executáveis. Uma boa estratégia é analisar os elementos necessários para decidir a não executabilidade apenas quando a adição de novos casos de teste não contribui para melhorar a cobertura. Neste caso, caminhos são identificados para cobrir os elementos restantes e, se possível, casos de teste específicos são gerados. Outra estratégia é usar padrões não executáveis para a classificação dos caminhos. Padrões não executáveis são estruturas compostas de sequência de nós com condições inconsistentes (Vergilio; Maldonado; Jino, 2006).

Tabela 6.1- Números de elementos cobertos e não executáveis por estudo de caso.

Critérios/Programas	Total de elementos requeridos / elementos requeridos cobertos / elementos requeridos não executáveis					
	gcd	omega	crossbar	mon_pingpong	messenger	eldap
processos	4/-/-	13/-/-	13/-/-	4/-/-	3/-/-	2/-/-
todos-nós	100/120/6	147/99/48	153/114/39	41/131/54	103/143/54	651/459/192
todas-arestas	139/150/15	225/149/76	220/190/30	53/143/54	169/141/28	1304/927/377
todos-nós-call	6/6/0	0/0/0	0/0/0	0/0/0	10/10/0	139/101/38
todos-nós-return	6/6/0	0/0/0	0/0/0	0/0/0	10/9/1	139/101/38
todos-nós-callrec	6/19/0	8/8/0	12/12/0	2/74/0	5/5/0	23/15/8
todos-nós-retrec	6/19/0	8/0/8	12/0/12	2/20/54	5/2/3	23/6/17
todos-nós-start	10/10/0	13/13/0	13/13/0	4/4/0	11/11/0	102/75/27
todos-nós-end	10/10/0	13/5/8	13/1/12	4/4/0	11/9/1	102/66/36
todos-nós-intra	21/18/3	13/13/0	16/16/0	11/11/0	14/14/0	3/3/0
todos-nós-case	0/0/0	0/0/0	0/0/0	0/0/0	6/6/0	74/56/18
todos-nós-guard	9/9/0	0/0/0	0/0/0	0/0/0	0/0/0	33/25/8
todos-nós-if	2/2/0	36/20/16	33/24/9	2/2/0	0/0/0	0/0/0
todos-nós-recv	9/6/3	8/8/0	12/12/0	7/7/0	13/13/0	11/9/2
todos-nós-spawn	3/3/0	4/4/0	0/0/0	0/0/0	0/0/0	0/0/0
todos-nós-link	0/0/0	0/0/0	0/0/0	2/2/0	0/0/0	0/0/0
todos-nós-unlink	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0
todos-nós-sync	12/12/0	44/28/16	42/36/6	7/7/0	18/18/0	2/2/0
todas-arestas-intra	100/91/9	169/117/52	166/154/12	41/41/0	105/95/10	978/702/276
todas-arestas-call	6/6/0	0/0/0	0/0/0	0/0/0	10/10/0	139/101/38
todas-arestas-return	6/6/0	0/0/0	0/0/0	0/0/0	10/9/1	139/101/38
todas-arestas-stack	6/19/0	8/8/0	12/12/0	2/74/0	5/5/0	23/15/8
todas-arestas-unstack	6/19/0	8/0/8	12/0/12	2/20/54	5/2/3	23/6/17
todas-arestas-link	0/0/0	0/0/0	0/0/0	2/2/0	0/0/0	0/0/0
todas-arestas-unlink	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0
todas-arestas-sync	12/6/6	36/20/16	30/24/6	6/6/0	34/20/14	2/2/0
todas-arestas-spawn	3/3/0	4/4/0	0/0/0	0/0/0	0/0/0	0/0/0
todas-defs	53/50/3	98/98/0	97/97/0	26/26/0	45/45/0	426/350/76
todos-c-usos	40/40/0	93/77/16	58/52/6	23/23/0	26/26/0	134/106/28
todos-p-usos	20/20/0	104/56/48	51/39/12	2/2/0	12/12/0	200/142/58
todos-m-usos	12/12/0	124/76/48	117/93/24	4/4/0	34/34/0	2/2/0
todos-f-usos	10/10/0	0/0/0	0/0/0	0/0/0	16/16/0	286/205/81
todos-r-usos	24/76/0	0/0/0	18/18/0	2/74/0	5/5/0	75/61/14
todas-assoc-c-usos	40/40/0	93/77/16	58/52/6	19/19/0	25/25/0	134/106/28
todas-assoc-p-usos	20/20/0	104/56/48	51/39/12	2/2/0	12/12/0	200/142/58
todas-assoc-m-usos	11/11/0	124/76/48	117/93/24	3/3/0	24/24/0	2/2/0
todas-assoc-f-usos	10/10/0	0/0/0	0/0/0	0/0/0	16/16/0	286/205/81
todas-assoc-r-usos	24/76/0	0/0/0	18/18/0	2/74/0	5/5/0	75/61/14
todas-assoc-p-link	0/0/0	0/0/0	0/0/0	2/2/0	0/0/0	0/0/0

Fonte: Elaborada pelo autor.

O uso de padrões é um mecanismo importante para identificar a não executabilidade em programas tradicionais. Se um caminho contém esses padrões, será não executável. A fim de reduzir o problema de caminhos não executáveis, pretende-se implementar na **ValiErlang** um mecanismo para descartar

automaticamente caminhos não executáveis de acordo com um padrão fornecido pelo testador.

Tabela 6.2 - Quantidade de casos de teste utilizados por estudo de caso.

Programas	Tamanho efetivo de conjunto de casos de teste
gcd	3
omega	16
crossbar	9
mon_pingpong	3
messenger	12
eldap	12

Fonte: Elaborada pelo autor.

De acordo com os resultados do experimento, alguns elementos requeridos não executáveis estão relacionados com os critérios **todas-arestas-sync**, **todos-m-usos** e **todas-assoc-m-usos**. Essa situação ocorre porque foi adotada uma estratégia conservadora gerando todas as arestas de sincronização possíveis, mesmo quando a comunicação pode não ser possível na prática.

Os elementos requeridos não executáveis do critério **todos-nós-recursivos** estão relacionados com o perfil das aplicações que foram utilizadas. Neste caso, existe um processo que executa uma primitiva *receive* e fica aguardando dados de outro processo. Quando a mensagem é recebida, o processo executado continua e, no final, faz uma chamada recursiva (*loop*) para aguardar uma nova mensagem. Um exemplo pode ser observado pelo código da Figura 6.1.

Como pode ser observado na parte superior da Figura 6.1, o processo **server**, por meio da primitiva *receive*, espera por requisições dos processos clientes. De acordo com o padrão da requisição, ele toma uma ação. Após cada ação realizada, o processo **server** executa uma recursão. Desta forma, ocorre apenas o empilhamento de chamadas recursivas, diferente do desempilhamento que nunca vai ocorrer. Esta situação pode ser vista na parte de baixo da Figura 6.1, onde o nó 17 foi coberto pelo dado de teste, entretanto, o nó 18 não foi coberto pelo dado de teste. Esta situação acaba afetando outros critérios, por exemplo, **todos-nós-end**, **todos-nós-intra** e **todas-arestas-unstack**.

Átomo é um tipo de dado em **Erlang** que não precisa de uma definição e pode-se fazer qualquer tipo de uso, por exemplo, **m-uso**. Neste contexto, não existe

associação relacionada com atom, por exemplo, associação **m-uso**. Por exemplo, esta situação ocorreu em alguns programas e pode ser observado o número menor de elementos para o critério **todas-assoc-m-usos** em relação a **todos-m-usos**. Em **Erlang** é possível encerrar a execução de um processo com a função `exit()`. Esta função foi utilizada no programa **messenger** e por consequência, um elemento dos critérios **todos-nós-return** e **todos-nós-end** não foi coberto.

Figura 6.1 - Função `server` do programa **messenger**. E trace de execução de um dado de teste para os critérios **todos-nós-callrec** e **todos-nós-retrec**.

<pre>server(User_List) -> receive {From, logon, Name} -> New_User_List = server_logon(From, Name, User_List), server(New_User_List); {From, logoff} -> New_User_List = server_logoff(From, User_List), server(New_User_List); {From, message_to, To, Message} -> server_transfer(From, To, Message, User_List), io:format("list is now: ~p~n", [User_List]), server(User_List) end.</pre>	
6,nCallRec,server,server,0,NOT	7,nRetRec,server,server,0,NOT
11,nCallRec,server,server,0,NOT	12,nRetRec,server,server,0,NOT
17,nCallRec,server,server,0,COVERAGE	18,nRetRec,server,server,0,NOT

Fonte: Elaborada pelo autor.

Um ponto importante é ainda sobre a recursão. Pode ser observado na Tabela 6.1 que o número de elementos cobertos para os critérios **todos-nós-callrec**, **todos-r-usos** e **todas-assoc-r-usos** é maior que o número total de elementos. Esta situação ocorre devido ao aspecto dinâmico da recursão, que depende do fluxo de dados para determinar o número de recursões. A Figura 6.2 ilustra esta situação.

Como pode ser observado na Figura 6.2, do lado esquerdo mostra a cobertura do programa **GCD** com os dados de teste {NumX = 1, NumY = 2, NumZ = 1}. No lado direito a cobertura com os dados de teste {NumX = 68, NumY = 120, NumZ = 90}. As linhas 5 e 6 (lado direito) mostram que ocorreram duas vezes o **m-uso** de X (identificado pelo número sequencial da quinta coluna), diferente do lado esquerdo que ocorreu apenas uma vez (linha 5).

Figura 6.2 - Exemplo de cobertura para o critério *todos-m-usos*.

1	calc,110,47,X,0,NOT	1	calc,110,47,X,0,COVERAGE
2	calc,111,47,Y,0,NOT	2	calc,111,47,Y,0,COVERAGE
3	calc,112,47,X,0,NOT	3	calc,112,47,X,0,COVERAGE
4	calc,113,47,Mpid,0,NOT	4	calc,113,47,Mpid,0,COVERAGE
5	calc,121,51,X,0,COVERAGE	5	calc,121,51,X,0,COVERAGE
6	calc,122,51,Y,0,COVERAGE	6	calc,121,51,X,1,COVERAGE
7	calc,123,51,Y,0,COVERAGE	7	calc,122,51,Y,0,COVERAGE
8	calc,124,51,Mpid,0,COVERAGE	8	calc,122,51,Y,1,COVERAGE
		9	calc,123,51,Y,0,COVERAGE
		10	calc,123,51,Y,1,COVERAGE
		11	calc,124,51,Mpid,0,COVERAGE
		12	calc,124,51,Mpid,1,COVERAGE

Fonte: Elaborada pelo autor.

6.3 Análise de falhas típicas pela Valierlang

Esta seção apresenta uma relação entre as falhas típicas encontradas em programas *Erlang* (e que foram descritas na seção 3.5) e a metodologia de teste que foi adotada neste trabalho. Desta forma, pretende-se demonstrar que tais erros podem ser detectados pela *ValiErlang* por meio dos critérios de teste que foram propostos.

Um exemplo de erro de concorrência e erro de existência foi descrito na seção 3.5.1 e será novamente abordado. A Figura 6.3, apresenta um trecho do programa **GCD**.

Figura 6.3 - Exemplo de erro relacionado com condição de disputa do programa **GCD**.

```

1 -module(gcd2).
2 -export([start/3, wait/0]).
3 start(NumX, NumY, NumZ) ->
4     global:register_name(pidm,self()),
5     global:register_name(pid0, spawn(gcd2,wait, [])),
6     global:register_name(pid1, spawn(gcd2,wait, [])),
7     Pidm = global:whereis_name(pidm),
8     Pid0 = global:whereis_name(pid0),
9     Pid1 = global:whereis_name(pid1),
10    Pid0 ! {NumX, NumY, Pidm},
11    Pid1 ! {NumY, NumZ, Pidm},
12    waitx().
13 waitx() ->
14     receive
15     {X} ->
16         io:format("Receive X de Slave! ~n"),
17         waity(X)
18     end.

```

Fonte: Elaborada pelo autor.

De acordo com a Figura 6.3, nas linhas 5 e 6 são criados dois processos que foram nomeados de **pid0** e **pid1**. Na sequência (linhas 10 e 11) é enviada uma mensagem para cada um dos respectivos processos criados. Neste caso, um erro pode ser revelado se ocorrer a execução do envio das mensagens nas linhas 10 e 11 antes da efetiva criação dos processos. Isto está relacionado com o *interleaving*, devido à condição de disputa dos processos pelo uso do processador.

O problema é que a mensagem foi enviada para um processo que não existe ainda, isso caracteriza um erro de existência, e a mensagem não será entregue. Por consequência, o processo que está bloqueado pela primitiva *receive* (linha 14) não receberá uma mensagem de volta do processo criado (por exemplo, **pid0**), que estará esperando pela mensagem que não foi entregue. Esta situação gera um erro de *deadlock*, pois um processo está esperando uma ação do outro processo.

De acordo com a análise do *trace* de execução e a cobertura dos elementos obtida, o testador pode detectar os erros descritos anteriormente por meio dos critérios **todos-nós-sync** e **todos-nós-recv**.

Um exemplo de erro de domínio pode ser ilustrado no programa GCD por meio da inserção manual de dois defeitos no programa. A Figura 6.4 mostra duas versões de um trecho do programa GCD, do lado esquerdo o código com os defeitos inseridos (linhas 48 e 51). No lado direito é apresentado o programa original.

Figura 6.4 - Exemplo de erro de domínio do programa **GCD**.

<pre> 45 calc(X, Y, Mpid) when X < Y -> 46 io:format("x(~w) < y(~w)~n ", [X, Y]), 47 calc (X, Y - X, Mpid); 48 calc(X, Y, Mpid) when X < Y -> 49 io:format("x(~w) > y(~w)~n ", [X, Y]), 50 calc (X - Y, Y, Mpid); 51 calc(X, Y, Mpid) => 52 Mpid ! {X}, 53 io:format("x(~w) == y(~w)~n ", [X, Y]). </pre>	<pre> 45 calc(X, Y, Mpid) when X < Y -> 46 io:format("x(~w) < y(~w)~n ", [X, Y]), 47 calc (X, Y - X, Mpid); 48 calc(X, Y, Mpid) when X > Y -> 49 io:format("x(~w) > y(~w)~n ", [X, Y]), 50 calc (X - Y, Y, Mpid); 51 calc(X, Y, Mpid) when X == Y -> 52 Mpid ! {X}, 53 io:format("x(~w) == y(~w)~n ", [X, Y]). </pre>
--	--

Fonte: Elaborada pelo autor.

O primeiro defeito (linha 48) está relacionado a cláusulas inacessíveis, pois a cláusula (*guard*) em questão ficará idêntica à cláusula (*guard*) da linha 45 e, conseqüentemente, nunca será executada. O segundo defeito omite a cláusula (*guard*) que faria de igualdade dos números.

A identificação do erro no programa com defeitos inseridos não ocorre para o caso de teste $\{(NumX = 2, NumY = 4, NumZ = 6), 2\}$, entretanto, o erro é identificado

pelo caso de teste $\{\{\text{NumX} = 6, \text{NumY} = 4, \text{NumZ} = 6\}, 2\}$, pois o resultado (6) gerado é diferente do esperado. O testador pode identificar os erros com a execução do programa na **ValiErlang**, por exemplo, a cláusula inacessível pode ser detectada pelo critério **todos-nós-guard**.

Outro exemplo relacionado à cláusula inacessível está presente no programa GCD, e pode ser observado nos resultados apresentados da Tabela 6.1. Para o critério **todos-nós-recv**, três elementos não executáveis foram detectados, estes elementos estão relacionados com a segunda cláusula *receive* dos processos p^0 , p^1 e p^2 (linha 42 da Figura 4.2). Esta cláusula dos processos citados nunca executa, pois para todos os casos de teste a primeira cláusula *receive* sempre vai executar. Pela análise dos elementos não executáveis que foram gerados pela **ValiErlang**, o testador pode identificar este erro.

As falhas relacionadas ao *deadlock* de processos e também falha na rede (para aplicações que consideram ambientes distribuídos) podem ser detectadas pelo critério **todos-p-link**. Caso a aplicação considera o uso de tolerância a falhas, o processo que está monitorando recebe uma mensagem do processo monitorado diferente de 'EXIT' que seria o término normal. A Figura 6.5 mostra um exemplo de mensagens recebidas.

Figura 6.5 - Exemplo de recebimento de mensagem pelo processo monitor.

```
(Node4@alexandre)3> mon_pingpong:start_pong().
Pong received helloworld
Pong received helloworld
Pong received helloworld
Pong received helloworld
Pong received helloworld
Pong terminou
Processo pong dead(<7077.54.0>). Reason: normal
ok
(Node4@alexandre)4> mon_pingpong:start_pong().
Pong received helloworld
Pong received helloworld
Pong received helloworld
Pong received helloworld
Pong received helloworld
Pong received helloworld
Processo pong dead(<7077.55.0>). Reason: noconnection
ok
(Node4@alexandre)5> █
```

Fonte: Elaborada pelo autor.

Como pode ser observado na Figura 6.5, a primeira execução (comando 3) mostra que o processo monitor (`start_pong`) recebeu a mensagem 'EXIT' do processo monitorado. Entretanto, na segunda execução (comando 4) a mensagem recebida foi diferente ('noconnection'). Como ação pode ser necessária a criação de um novo processo para substituir o processo finalizado de forma anormal. A **ValiErlang** por meio de análise dos elementos não executáveis, por exemplo, mostra ao testador que o elemento requerido para o critério **todos-p-link** não foi coberto.

No erro descrito anteriormente, os processos envolvidos trocam mensagens e utilizam chamadas recursivas. Neste caso, o erro pode ser detectado também pelo critério **todos-nós-recursivos**. A Figura 6.6 mostra o resultado da cobertura do critério **todos-nós-recursivos**.

Figura 6.6 - Exemplo de cobertura para os critérios todos-nós-recursivos.

30, nCallRec, pong, pong, 0, COVERAGE	31, nRecRet, pong, pong, 0, NOT
30, nCallRec, pong, pong, 1, COVERAGE	
30, nCallRec, pong, pong, 2, COVERAGE	
30, nCallRec, pong, pong, 3, COVERAGE	
30, nCallRec, pong, pong, 4, COVERAGE	
30, nCallRec, pong, pong, 5, COVERAGE	
30, nCallRec, pong, pong, 6, COVERAGE	
30, nCallRec, pong, pong, 7, COVERAGE	

Fonte: Elaborada pelo autor.

Como pode ser observado pela Figura 6.6, os processos foram finalizados de uma forma anormal e, por isso, não realizaram o desempilhamento das chamadas recursivas. A diferença na cobertura dos nós **nCallRec** e **nRetRec** sinaliza que o funcionamento da função envolvida nas chamadas recursivas não foi realizado da forma correta. E isso é possível de identificar pelos resultados que são gerados pela **ValiErlang**.

6.4 Trabalhos relacionados

Esta seção descreve os trabalhos relacionados que apresentem alguma técnica de teste e utilizem ferramenta de teste para validar a técnica.

Tóth e Bozó (2011) utiliza a ferramenta **RefactorErl** para realizar a análise estática, e para extrair partes do programa que podem ser paralelizados.

D'Oswaldo, Kochems e Ong (2012) apresentam a ferramenta **Soter**, que realiza análise estática para verificar as propriedades que foram especificadas pelo usuário. **Soter** extrai um modelo abstrato a partir de um conjunto de propriedades e depois verifica se as propriedades estão satisfeitas. Silva, Tamarit e Tomás (2012) utilizam a ferramenta **Slicerl** para realizar análise estática e extrair partes do código que representam as funções de um programa **Erlang**. Li e Thompson (2009 e 2011) descrevem técnicas para detecção e eliminação de códigos semelhantes em programas **Erlang**. Tais técnicas estão integradas na ferramenta **Wrangler**, utilizada para refatoração de programas de **Erlang**.

Arts, Castro e Hughes (2008) apresentam uma metodologia para testar tipos de dados de acordo com as definições dos usuários usando **Quviq QuickCheck**, uma ferramenta para testes baseados em modelos e propriedades.

Sagonas, Silva e Tamarit (2013) usam uma técnica que realiza análise estática para detectar erros de tipo e código não executável com auxílio da ferramenta **Dialyzer**. Gotovos, Christakis e Sagonas (2011) apresentam a ferramenta **Concuerror** cujo objetivo é auxiliar no processo de *Test Driven Development* (TDD).

A seguir é feita uma breve descrição das ferramentas que foram utilizadas pelos trabalhos descritos anteriormente.

Dialyzer é uma ferramenta de análise estática que identifica discrepâncias de software, como erros de tipos definidos, código que se tornou inacessível devido a algum erro de programação, erros de relacionados à *deadlocks* e passagem de mensagens (Dialyzer, 2016).

Quviq QuickCheck (Quviq AB, 2008), **PropEr** (Proper, 2011) e **Soter** (D'Oswaldo; Kochems; Ong, 2012) são ferramentas de teste baseadas em propriedades. As ferramentas necessitam que o usuário defina as propriedades que o programa deve satisfazer usando uma notação especial. A partir das propriedades, a ferramenta gera casos de teste aleatoriamente para verificar erros nos programas.

As ferramentas **Wrangler** (Li; Thompson, 2009) e **RefactorErl** (HORVÁTH et al., 2008) têm como objetivo refatorar um programa original, seja para otimizar ou para organizar, mantendo suas funcionalidades.

A ferramenta **Concuerror** (Gotovos; Christakis; Sagonas, 2011) auxilia o processo de TDD. A ferramenta utiliza conjuntos de dados de testes para detectar erros relacionados à concorrência, tais como, *deadlocks* e condições de disputa em

programas Erlang.

A ferramenta **Cover** (2016) é parte integrante da biblioteca OTP/Erlang. **Cover** aplica o teste de cobertura contabilizando quantas vezes cada linha do código fonte é executada, porém, não aplica análise de fluxo de dados.

CutEr (Giantsios; Papaspyrou; Sagonas, 2015) é uma ferramenta de teste concólcico para um subconjunto da linguagem **Erlang**, que suporta *pattern matching*, listas, recursões e *higher-order functions*. **Smother** (Taylor; Derrick, 2015) é uma ferramenta que analisa a cobertura MC/DC para programas **Erlang**.

No geral, as ferramentas (**Dialyzer** e **Cover**) não aplicam corretamente as técnicas de teste e não exploram o processo de teste, tais como, cobertura de código baseado em critérios. Outras ferramentas (**Wrangler**, **RefactorErl** e **SliceErl**) não tiveram como foco testar a qualidade dos programas e sim, objetivos mais específicos como, refatoração para identificar códigos semelhantes, teste baseado em propriedades (**QuickCheck**, **PropEr** e **Soter**). A Tabela 6.3 mostra uma comparação de diversas características entre as ferramentas apresentadas nesta seção.

Tabela 6.3 - Comparativo entre as ferramentas de teste avaliadas.

Ferramentas	Fluxo de Dados	Fluxo de Controle	Concorrência	GFC	GFCP	Depuração	Refatoração TDD	Cobertura Critérios
Concuerror	√	√	√					√
Cover		√				√		√
CutEr	√	√						√
Dialyzer	√	√	√			√		√
QuickCheck	√	√					√	√
PropEr		√					√	√
RefactorErl		√					√	
SliceErl	√	√		√			√	
Smother	√	√						√
Soter		√					√	√
Wrangler		√					√	
ValiErlang	√	√	√	√	√	√		√

Fonte: Elaborada pelo autor.

A Tabela 6.4 especifica os critérios de cobertura que são utilizados nas ferramentas cujo propósito é identificar erros nos programas **Erlang**. Nesta tabela são consideradas apenas as ferramentas que utilizam algum tipo de critério para a identificação dos erros.

Tabela 6.4 - Descrição dos critérios que são utilizados pelas ferramentas de teste.

Ferramentas	Definição e Usos	Nós e Arestas	Concorrência	MC/DC	Tolerância a Falhas	Recursão	Interfunção
Concuerror		√	√				
Cover		√	√				√
CutEr		√				√	
Dialyzer		√	√				√
QuickCheck		√	√				
PropEr		√	√				
Smoother				√			
Soter		√	√				
ValiErlang	√	√	√		√	√	√

Fonte: Elaborada pelo autor.

A Tabela 6.4 mostra que a **ValiErlang** investiga em detalhes várias características de programas **Erlang** e, conseqüentemente, consegue identificar todos os tipos de erros descritos na seção 3.5. Ela ainda identifica erros relacionados às aplicações tolerantes a falhas. Este é um fator que destaca a **ValiErlang** das outras ferramentas, por exemplo, a **Concurren** que restringe apenas a investigação de erros relacionados à concorrência. Como exemplo disso, pode-se verificar a capacidade da **ValiErlang** de detectar falhas relativas a cláusulas não executáveis por meio dos nós N_{recv} . Esta falha ocorreu com o programa GCD, sendo que a segunda cláusula *receive* dos processos *slaves* não executa, pois em todos os casos de teste a primeira cláusula *receive* sempre executou.

6.5 Considerações Finais

Este capítulo apresentou os resultados e a análise da aplicação dos critérios de teste na **Valierlang** e também discutiu a eficácia dos critérios e da ferramenta em relação à identificação dos tipos de erros. Por fim, foi realizada uma comparação que abordou vários aspectos entre as ferramentas de teste que foram identificadas no mapeamento sistemático.

A partir dos resultados obtidos é possível ressaltar que a aplicação dos critérios é pertinente e comprovou-se a eficácia dos mesmos em revelar defeitos. Da mesma forma, a **ValiErlang** conseguiu realizar toda a atividade de teste que foi proposta pelos critérios de teste. Outro aspecto importante é a gama de critérios que a **ValiErlang** oferece, são 15 tipos de nós, 9 tipos de arestas e 5 tipos de usos de

variáveis que permitem a cobertura das principais características que foram descritas na seção 4.3.

O capítulo seguinte apresenta as conclusões finais do trabalho, as contribuições para a linha de pesquisa em que este trabalho está inserido e os trabalhos futuros vislumbrados para a continuidade da pesquisa.

7 CONCLUSÕES

7.1 Considerações finais do projeto

Neste trabalho de doutorado foi abordado o teste de software considerando a técnica de teste estrutural para linguagem funcional **Erlang**. O intuito do trabalho é contribuir para melhorar a qualidade de tais programas, por meio de novos critérios de teste. A aplicação dos critérios propostos foi realizada com o auxílio de um modelo de teste estrutural e de uma ferramenta de teste, denominada **ValiErlang**.

Na literatura há trabalhos e ferramentas que fornecem metodologias para aplicação de teste em programas **Erlang**. Contudo, não foram encontradas alternativas para a aplicação de teste de software em programas **Erlang** para avaliar a cobertura por meio de critérios que consideram ao mesmo tempo e em um único ambiente a seguintes características essenciais desses programas: comunicação entre funções, comunicação entre processos, sincronização, concorrência, recursividade e tolerância a falhas. Esta última característica não foi abordada em nenhum dos trabalhos apurados pelo mapeamento sistemático. Neste contexto, constatou-se que as abordagens e ferramentas pesquisadas não conseguem atender aos objetivos propostos neste projeto. Este cenário foi motivado no início do trabalho e reforçou a necessidade de conduzir uma pesquisa de testes para a linguagem **Erlang**.

O objetivo geral desta tese foi definir de critérios de teste estruturais para aplicações desenvolvidas na linguagem funcional **Erlang**, considerando aspectos sequenciais e distribuídos. Com base nos estudos experimentais, foi possível demonstrar a aplicação do conjunto de critérios em relação à cobertura de código e sua eficácia para detectar erros.

Foram delineados quatro objetivos específicos como parte da atividade de teste desenvolvida neste trabalho. O primeiro objetivo abordou o estado da arte do

teste de software aplicado em linguagens funcionais. A realização do mapeamento sistemático que identificou os trabalhos relacionados com teste de software para linguagens funcionais e também a identificação de falhas típicas encontradas em programas dão por concluído este objetivo.

O segundo objetivo específico foi cumprido com a especificação do modelo de teste que captura as principais características dos programas **Erlang**. Este modelo foi usado para a especificação dos critérios de teste. O modelo especificou 15 tipos diferentes de nós, 9 tipos diferentes de arestas e 5 tipos diferentes de usos. Desta forma foi possível caracterizar os 15 novos critérios propostos que englobam as principais características de **Erlang** que foram propostas.

O terceiro objetivo específico foi desenvolver uma ferramenta de teste para apoiar a aplicação dos critérios de teste definidos. Baseado no mapeamento sistemático, foi identificado que não existia uma ferramenta para atender esta necessidade, optando-se, portanto, por implementar uma ferramenta de teste. A implementação da **ValiErlang** cumpriu este objetivo específico e foi possível dar suporte aos critérios propostos.

A validação dos critérios ocorreu por meio da **ValiErlang**. Os resultados obtidos comprovam que é possível a aplicação dos critérios e comprovou-se a eficácia dos mesmos em revelar defeitos. Para isso foram utilizados seis programas **Erlang** com diferentes características.

Todas as atividades de teste necessárias foram cumpridas pela **ValiErlang** e os resultados foram obtidos a partir de teste de programas que representaram as características necessárias para os testes. Ressalta-se que dois programas utilizados foram desenvolvidos por terceiros, demonstrando que a ferramenta pode ser utilizada por outros programas com as características necessárias para o teste.

A Tabela 6.4 mostra que a **ValiErlang**, valida uma gama de critérios de teste, que consideram várias características de programas **Erlang**, e consequentemente, consegue identificar todos os tipos de erros descritos na seção 3.4 e complementa com os erros para aplicações tolerantes a falhas.

Neste contexto, pode-se considerar como válida a tese de que é possível aplicar o teste estrutural para aplicações concorrentes e sequenciais desenvolvidas em **Erlang**, considerando fluxo de dados, fluxo de controle e fluxo de comunicação e sincronização.

7.2 Contribuições

O projeto de pesquisa referente a esta tese gerou contribuições para as áreas de pesquisa em que este trabalho se insere. Esta seção descreve as contribuições resultantes.

A principal contribuição foi a especificação de critérios de teste estruturais para aplicação da atividade de teste em **Erlang**. Os critérios consideram as principais características da linguagem **Erlang**, tais como: comunicação entre processos, comunicação entre funções, funções recursivas, sincronização, concorrência e primitivas que permitem o monitoramento de processos. Os critérios propostos permitem a aplicação do teste estrutural para verificação de códigos em aplicações desenvolvidas na linguagem funcional **Erlang**, considerando aspectos sequenciais e concorrentes. Também, fornecem uma medida de cobertura que pode ser usada para auxiliar o testador na escolha de casos de teste com a capacidade de revelar defeitos, e também, avaliar se um programa foi testado o suficiente, em função das necessidades da aplicação e dos recursos disponíveis para o teste.

Um modelo de teste que captura as informações sobre o fluxo de controle, fluxo de dados e fluxo de comunicação de programas **Erlang** também foi proposto. Em resumo, para atender às características da linguagem **Erlang**, o modelo define 15 tipos de nós, 9 tipos de arestas e 5 tipos de usos de variáveis. Todas essas informações recuperadas do código fonte e armazenadas no modelo de teste são usadas como suporte à aplicação dos critérios de teste.

A **ValiErlang** é uma ferramenta de teste construída para validar os critérios de teste propostos. A aplicação prática de um critério de teste só é possível se uma ferramenta estiver disponível. **ValiErlang** oferece duas funcionalidades básicas para satisfazer a demanda por ferramentas de suporte à aplicação de critérios de teste em programas **Erlang** e avaliar os critérios propostos: a execução e a avaliação dos dados de teste.

O mapeamento sistemático contribuiu para identificar as contribuições científicas relacionadas ao projeto, além de fornecer uma visão geral para o teste de linguagens funcionais e conhecimento para o desenvolvimento deste projeto. Esta também foi uma contribuição importante para a falta de evidências em relação a critérios e ferramentas de teste capazes de serem aplicados de forma abrangente à atividade de teste em programas **Erlang**.

Outra contribuição desta tese é o experimento conduzido, no qual, seis programas foram escolhidos para aplicar o conjunto de critérios de teste utilizando o apoio computacional da **ValiErlang**. O principal objetivo deste experimento foi avaliar, após as etapas de teste, a eficácia dos critérios em revelar defeitos e também o funcionamento dos cinco módulos da **ValiErlang** em relação ao processo de teste.

A **ValiErlang** possui um conjunto extenso de critérios de teste, o que propicia ao testador de software uma variedade de análises em relação à cobertura do código a ser testado. Caso o objetivo do teste seja verificar apenas a comunicação entre funções, o testador pode analisar apenas os critérios relacionados à comunicação entre funções; neste caso, usando os critérios **todos-nós-inter-funções**, **todas-arestas-inter-funções** e **todos-f-usos**. Se o objetivo é testar apenas a recursão de uma função, o testador pode concentrar a análise apenas nos critérios **todos-nós-recursivos**, **todas-arestas-recursivas** e **todos-r-usos**. Para analisar as sincronizações e a comunicação entre processos, o testador pode limitar a análise aos critérios **todos-nós-sync**, **todas-arestas-sync** e **todos-m-usos**. Para verificar apenas o monitoramento entre os processos, o testador pode se limitar apenas aos critérios **todos-nós-link**, **todas-arestas-link** e **todos-p-link**.

Com base em todo trabalho realizado foi possível perceber que a aplicação dos critérios e a eficácia em detectar defeitos é possível de ser atingida, entretanto, ainda há vários desafios a serem superados. O processo de instrumentação mostrou-se como o grande desafio, devido à diversidade de formas que a linguagem proporciona para realizar a codificação. Essa diversidade gera padrões diferentes de estrutura quando da obtenção da AST. Tais padrões precisam ser mapeados para que o módulo de **Parse** capture as informações propostas pelo modelo. Outro desafio encontrado também é a inexistência de uma função principal para iniciar a execução de uma aplicação. Esta não uniformidade torna o processo de instrumentação e a geração do rastro de execução (*trace*) um desafio. Porém, acredita-se que as contribuições destacadas nesta tese contribuem para o avanço do ideal que se espera para o processo de teste de software para a linguagem **Erlang**. Em especial, a abordagem aqui apresentada contribui com as perspectivas do testador de software e o apoio computacional por meio da **Valierlang** auxilia o testador de software a realizar as etapas da atividade de teste estrutural, e o mais importante: guiar o testador na escolha de casos que teste com a capacidade de

revelar defeitos.

7.3 Limitações e trabalhos futuros

Este trabalho foi idealizado, desde o início, para deixar um legado a partir de suas contribuições relacionadas à atividade de teste com software para o suporte à automatização de desta atividade. Atualmente existem limitações em relação à atividade de teste abordada e do software de apoio. Todavia, a experiência na implementação que o projeto de pesquisa ofereceu, muitas ideias, críticas e sugestões foram sendo coletadas neste período e, logo, são vislumbradas algumas questões como trabalhos futuros.

Licenciamento de software livre e publicação da **ValiErlang**, que é o software de apoio para a aplicação da atividade de teste, em um repositório de projetos de software (por exemplo, GitHub).

O modelo de teste atual captura informações de um subconjunto da linguagem **Erlang**. Existem algumas características que não são consideradas, por exemplo, *higher-order functions*, funções anônimas (*fun*s) e registros. Essas características também são importantes para o contexto da **Erlang** e tornaria a abordagem mais completa, desta forma, e sugere-se como trabalhos futuros a incorporação das respectivas características.

Para atender a incorporação de novas características da linguagem serão necessários novos critérios de teste. Pretende-se também incluir o critério MD/DC que foi abordado em Taylor e Derrick (2015), este critério tem como objetivo verificar a cobertura individual de uma condição em caso de cláusula com múltiplas condições.

A identificação dos caminhos inviáveis é realizada pelo testador, isto demanda de esforço para a realização do processo. Para a redução deste esforço, sugere-se a incorporação de padrões para identificar a inviabilidade em programas tradicionais. Neste contexto, pretende implementar na **ValiErlang** um mecanismo para descartar automaticamente caminhos inviáveis de acordo com padrões que são fornecidos pelo testador.

Os casos de teste são gerados de forma manual e randomicamente, este processo não é tão eficiente e depende do nível de conhecimento do testador para definir a quantidade de casos de teste. Como trabalhos futuros propõem-se a

implementação de um módulo para geração automática de casos de teste. Isso aumenta a eficiência na descoberta de defeitos, consequentemente, reduzindo o tempo e o custo da atividade de teste.

As aplicações telefônicas e de roteamento de pacotes, por exemplo, manipulam e encaminham mensagens. Para detectar a sequência de definições e usos das variáveis neste tipo de aplicação, pretende-se implementar um mecanismo para capturar todo o fluxo de informação a partir do processo de origem até o processo destino, ou seja, verificar as mensagens durante seu caminho fim-a-fim.

Outra proposta de extensão da **Valierlang** é trocar a forma da gravação das informações relacionadas à análise estática. Atualmente a gravação é realizada em arquivos textos, e a gravação em banco de dados permite a gravação com mais eficiência de um volume maior de informações. Esta mudança permite a aplicação do teste para programas maiores, além de melhorar a recuperação das informações gravadas.

Pretende-se criar um mecanismo na **ValiErlang** para auxiliar o testador na análise dos resultados. Atualmente a **ValiErlang** gera dados que mostram os elementos requeridos que foram cobertos e não cobertos, por critério de teste. De acordo com esses dados gerados, e baseados nas informações sobre os tipos de falhas, o objetivo é que esse mecanismo ajude o testador a inferir sobre eventuais defeitos no código fonte. O cruzamento de informações sobre a cobertura de diferentes critérios em **Erlang** pode ser usada nesse sentido.

7.4 Publicações Resultantes

Por meio da abordagem realizada, critérios de teste e o desenvolvimento da ferramenta de apoio **ValiErlang**, foram escritos e publicados artigos cujas informações de publicação são mencionadas a seguir:

- Oliveira, A. P., Souza, P. S. L., Souza, S. R. S., Estrella, J. C., Bruschi, S. M. A Systematic Mapping about Testing of Functional Programs. In: 13th International Conference on Software Engineering Research and Practice (SERP). Las Vegas, July 27-30, 2015.
- Oliveira, A. P.; Souza, P. S. L.; Souza, S. R. S.; Estrella, J. C.; Bruschi, S. M.

Teste Estrutural Aplicado à Linguagem Funcional Erlang In: Proceedings of the 9th Brazilian Workshop on Systematic and Automated Software Testing (SAST), p. 41-50, 2015. Belo Horizonte/MG - Brazil, Brazilian Computer Society.

- Oliveira, A. P.; Souza, P. S. L.; Souza, S. R. S. ValiErlang: A Structural Testing Tool for Erlang Programs. In: Proceedings of the 1st Brazilian Symposium on Systematic and Automated Software Testing (SAST), 2016. Maringá/PR – Brazil, September 19 - 20. New York: ACM Press, Article 13, 10 pages. DOI: <http://dx.doi.org/10.1145/2993288.2993300>.

Submissão de artigo para periódico:

- Oliveira, A. P.; Souza, P. S. L.; Souza, S. R. S. Structural testing criteria for Erlang functional language. In Journal of Functional Programming. 2017. Cambridge University Press.

REFERÊNCIAS

- Aho, A. V.; Lam, M. S.; Sethi, R.; Ullman, J. D. **Compilers. Principles, Techniques, and Tools**. 2. ed. Addison-Wesley, 1986.
- Ammann, P.; Offutt, J. **Introduction to software testing**. Cambridge University Press 2008.
- Armstrong, J.; Däcker, B; Viriding, R.; Williams, M. **Implementing a functional language for highly parallel real time applications**. Eighth International Conference on Software Engineering for Telecommunication Systems and Services, 1992.
- Armstrong, J. **The Development of Erlang**. Proceedings of the ACM SIGPLAN International Conference on Functional Programming. 1996.
- Armstrong, J.; Viriding, R.; Williams, M.; Wikstrom, C. **Concurrent Programming in Erlang**. Prentice Hall Europe, Herfordshire, Great Britain, second edition. 1996.
- Armstrong, J. Concurrency Oriented Programming in Erlang. Invited talk, FFG. 2003.
- Armstrong, J. **A history of Erlang**. Proceedings of the third ACM SIGPLAN conference on History of programming languages, 2007a.
- Armstrong, J. **Programming Erlang: Software for a Concurrent World**. Pragmatic Bookshelf. 2007b.
- Arts, T.; Castro, L.M.; Hughes, J. **Testing Erlang Data Types with Quviq QuickCheck**. In: Proceedings of the ACM SIGPLAN Workshop on Erlang, ACM Press , 2008.
- Arts, T.; Thompson, S. **From test cases to FSMs: augmented test-driven development and property inference**. In Proceedings of the 9th ACM SIGPLAN workshop on Erlang (Erlang '10). ACM, New York, USA. 2010.
- Balakrishnan, A. and Anand, N. (2009). **Development of an automated testing software for real time systems**. In *Industrial and Information Systems (ICIIS), 2009 International Conference on*, pages 193 - 198.
- Bozó, I.; Tóth, M. **Selecting Erlang test cases using impact analysis**. In Proceedings of Symposium on Computer Languages, Implementations and Tools, Kassandra, Halkidiki, Greece, September 2011.
- Cesarini, F. and Thompson, S. **Erlang Programming - A Concurrent Approach to Software Development**. O'Reilly Media, 2009.

- Christakis, M., Gotovos, A., Sagonas, K. Systematic testing for detecting concurrency errors in **Erlang** programs. In: Proc. 6th International Conference on Software Testing, Verification and Validation, IEEE Computer Society, 2013, 154-163 p.
- Christakis, M.; Sagonas, K. **Detection of asynchronous message passing errors using static analysis**. In: Proceedings of the 13th international conference on Practical aspects of declarative languages, PADL'11, Berlin, Heidelberg: Springer-Verlag, 2011.
- Clocksin, W., Mellish, C.S.: Programming in PROLOG. Springer, New York, 2003.
- Copeland, L. **A practitioner's guide to software test design**. Artech House Publishers, 2004.
- Cover. Cover Tools Users's Guide. http://erlang.org/doc/apps/tools/users_guide.html. 2016.
- Cronqvist, M. **Troubleshooting a large Erlang system**. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Erlang*, pages 11–15, New York, NY, USA, 2004. ACM.
- Culler D., Gupta A., Singh J. P. Parallel Computer Architecture: A Hardware/Software Approach (1st ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. 1997.
- Delamaro, M. E.; Maldonado, J. C.; Jino, M. **Introdução ao teste de software**, cap. 1. Rio de Janeiro: Elsier, p. 1-7, 2007.
- D'Oswaldo, E.; Kochems, J.; Ong, L. **Soter: an automatic safety verifier for erlang**. In Proceedings of the 2nd edition on Programming systems, languages and applications based on actors, agents, and decentralized control abstractions. ACM. New York, NY, USA. 2012.
- Dialyzer. Dialyzer User's Guide. Disponível em: <http://erlang.org/doc/apps/dialyzer/>. 2016.
- Do, Q. H.; Ajakaiye, T. D. **Fault Injection Technique for Evaluating Erlang Test Suites**. Technical Report/Department of Applied Information Technology. University of Gothenburg, 2009.
- Endo, A.T.; Simão, A.S.; Souza, S. R. S.; Souza, P. S. L. **Web services composition testing: a strategy based on structural testing of parallel programs**. Testing: Academic Industrial Conference - Practice and Research Techniques. (TAIC-PART 2008), Windsor, UK. p. 3–12. 2008.
- Ericsson AB. **Erlang Reference Manual User's Guide**, 2016. Disponível em: http://erlang.doc/doc/reference_manual/users_guide.html.

- Erlang Bugs. **Official Erlang Bugs Report. 2016.** Disponível em: <http://www.erlang.org/pipermail/erlang-bugs/>. 2016.
- Franco, C. T. Desarrollo de una herramienta de refactorización en **Erlang** integrada en un IDE. Universidad Politécnica de Valencia. 2010.
- Giantsios, A.; Papaspyrou, N.; Sagonas, K. **Concolic testing for functional languages**. In *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming (PPDP '15)*. ACM, New York, NY, USA, 137-148. 2015.
- Gotovos, A.; Christakis, M.; Sagonas, K. **Test-driven development of concurrent programs using concuererror**. In *Proceedings of the 10th ACM SIGPLAN workshop on Erlang (Erlang '11)*. ACM, New York, USA, 2011.
- Guo, Q.; Derrick, J.; Walkinshaw, N. **Applying Testability Transformations to Achieve Structural Coverage of Erlang Programs**. In *Proceedings of the 21st IFIP WG 6.1 International Conference on Testing of Software and Communication Systems and 9th International FATES Workshop (TESTCOM '09/FATES '09)*. Springer-Verlag, Berlin, Heidelberg. 2009.
- Hausen, A. C., Vergílio, S. R., Souza, S., Souza, P. and Simão, A. (2007). **A tool for structural testing of MPI programs**. In *8th IEEE Latin-American Test Workshop (LATW 2007)*. Cuzco, Peru, p. 1–6. 2007.
- Harrold, M. J.; Rothermel, G. Performing data flow testing on classes, *Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering*, p.154-163, December 06-09, 1994, New Orleans, Louisiana, USA.
- Harrold, M. J.; Soffa, M. L. Efficient computation of interprocedural definition-use chains, *ACM Transactions on Programming Languages and Systems (TOPLAS)*, v.16 n.2, p.175-204, March 1994.
- Horváth, Z.; Lövei, L.; Kozsik, T.; Kitlei, R.; Víg, A.; Nagy, T.; Tóth, M.; AND Király, R. Building a refactoring tool for Erlang. In *Workshop on Advanced Software Development Tools and Techniques, WASDETT 2008*, Paphos, Cyprus, Jul 2008.
- Howden WE. Reliability of the path analysis testing strategy. *IEEE Transactions on Software Engineering* 1976;2:208–215.
- Javed, M; Ahmad, B.; Abbas, Z.; Nawaz, A.; Abid, M. A.; Ullah, I. Decreasing defect rate of test cases by designing and analysis for recursive modules of a program structure: Improvement in test cases. *IJCSIS*, Vol. 10 No. 8. August, 2012.
- Kitchenham, B.; Charters, S. **Guidelines for performing systematic literature reviews in software engineering**. Technical Report. EBSE 2007-001, Keele University and Durham University Joint Report, 2007.
- Krawczyk H, Wiszniewski B. Classification of software defects in parallel

- programs. Technical Report 2, Faculty of Electronics, Technical University of Gdansk, Poland, 1994.
- Le, D.; Alipour, M. A.; Gopinath, R.; Groce, A. MuCheck: an extensible tool for mutation testing of haskell programs. In Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA 2014). ACM, New York, NY, USA, p. 429-432. 2014.
- Li, H.; Thompson, S. Testing Erlang Refactorings with QuickCheck. 19th International Symposium on Implementation and Application of Functional Languages, IFL 2007, LNCS. 2007.
- Li, H; Thompson, S. **Clone detection and removal for Erlang/OTP within a refactoring environment**. In Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation (PEPM '09). ACM, New York, USA. 2009.
- Li, H; Thompson, S. **Incremental clone detection and elimination for erlang programs**. In Proceedings of the 14th international conference on Fundamental approaches to software engineering: part of the joint European conferences on theory and practice of software (FASE'11/ETAPS'11). Springer-Verlag, Berlin, Heidelberg, 2011.
- Lindahl, T.; Sagonas, K. **Detecting Software Defects in Telecom Applications Through Lightweight Static Analysis: A War Story**. In: Chin, Programming Languages and Systems: Proceedings of the Second Asian Symposium (APLAS'04). LNCS, vol. 3302, pp. 91—106. Springer, Heidelberg. 2004.
- Louden, Kenneth C. **Programming Languages: Principles and Practice**. PWS-KENT Publishing Company, 1993.
- Maldonado, J. C. **Crterios Potenciais Usos: Uma Contribuio ao Teste Estrutural de Software**. Tese de Doutorado, DCA-FEE-UNICAMP, Campinas, 1991.
- McCarthy, John; Abrahams, Paul W.; Edwards, Daniel J.; Hart, Timothy P.; Levin, Michael I. **Lisp 1.5 Programmer's Manual**. Cambridge, Massachusetts: The MIT Press, 1962.
- Myers, G. J. **The Art of Software Testing**. 2 ed. John Wiley & Sons, 2004.
- Nagy, T., Vg, A.N. Erlang testing and tools survey. Proceedings of the 7th ACM SIGPLAN workshop on ERLANG, September 27-27, 2008, Victoria, BC, Canada.
- Pawe, T. W. **Typing for Reliable Distributed Systems - Recent Advances**. IEEE Workshop on Dependable Software – Tools and Methods, Yokohama, Japan. 2005.
- Pezz, M; Young, M. **Teste e Anlise de Software - Processos, Princpios e Tcnicas**. Porto Alegre: Bookman. 2008.

- Pressman, R. S. **Engenharia de Software**. 5 ed. Rio de Janeiro: Mc Graw Hill, 2002.
- Pressman, R. S. **Engenharia de Software: Uma Abordagem Profissional**. 7. Ed. Porto Alegre: AMGH, 2011.
- Proper. Property-based testing for Erlang. Disponível em: <http://proper.softlab.ntua.gr>. 2011.
- Quviq AB. QuickCheck for Erlang Users. Stockholm: Quviq. 2008.
- Rapps, S.; Weyuker, E. J. **Selecting software test data using data flow information**. IEEE Transactions on Software Engineering, v. 11. n. 4. p. 367-375. 1985.
- Reps, T.; Horwitz, S.; Sagiv, M. Precise Interprocedural Dataflow Analysis via Graph Reachability. University of Wisconsin. POPL '95/95 San Francisco CA USA, 1995. ACM O-89791-892-1/95/0001.
- Sagonas, K. F; Silva, J.; Tamarit, S. **Precise explanation of success typing errors**. In Proceedings of the Workshop on Partial evaluation and program manipulation. PEPM. ACM SIGPLAN, 2013.
- Sarmanho, F. S., Souza, P. S. L., Souza, S. R. S., and Simão, A. S. **Structural testing for semaphore-based multithread programs**. In ICCS2008 - International Conference on Computational Science, Lecture Notes in Computer Science. Krakow. Springer-Verlag. v. 5101. p. 337-346. 2008.
- Sarmanho, F. S. **Teste de Programas Concorrentes com Memória Compartilhada**. Dissertação de Mestrado. Instituto de Ciências Matemáticas e de Computação. USP. São Carlos, 2009.
- Sebesta, R. W. **Concepts of Programming Languages**. 10th ed. Addison-Wesley, 2012.
- Silva J.; Tamarit, S; Tomás, C. **System dependence graphs in sequential erlang**. In Proceedings of the 15th international conference on Fundamental Approaches to Software Engineering (FASE'12). Springer-Verlag, Berlin, 2012.
- Syntax Tools. User's Guide. http://Erlang.org/doc/apps/syntax_tools/users_guide.html. 2016.
- Smerl: Simple Metaprogramming for **Erlang**, Module smerl, <http://erlyweb.org/doc/smerl.html>. 2016.
- Souza, S. R. S.; Vergilio, S. R.; Souza, P. S. L. **Introdução ao teste de software**, cap. 9. Rio de Janeiro: Elsevier, p. 231–249, 2007.

- Souza, P. S. L.; Sawabe, E.; Simão, A. S.; Vergilio, S. R.; Souza, S. R. S. **ValiPVM—a graphical tool for structural testing of PVM programs**. In Recent Advances in Parallel Virtual Machine and Message Passing Interface, Vol. 5205, Lastovetsky A, Kechadi T, Dongarra J (eds), Lecture Notes in Computer Science. Springer: Berlin, p. 257–264. 2008a.
- Souza, S. R. S.; Vergilio, S. R.; Souza, P. S. L.; Simão, A. S.; Hausen, A. C. **Structural testing criteria for message-passing parallel programs**. Concurrency and Computation: Practice and Experience, v. 20, n. 16, p. 1893–1916, 2008b.
- Tanenbaum A. S. **Sistemas Operacionais Modernos**. 3. ed. São Paulo: Pearson Prentice Hall, 2010.
- Tanenbaum, A. S. **Organização Estruturada de Computadores**. 5 ed. Pearson Prentice Hall: São Paulo, 2007.
- Tasharofi, S.; Pradel, M.; Lin, Y. and Johnson, R. Bita: Coverage-guided, automatic testing of actor programs. In 2013 IEEE/ACM 28th International Conference on Automated Software Engineering (ASE), 2013.
- Tate, Bruce A. **Seven Languages in Seven Weeks: A Pragmatic Guide to Learning Programming Languages**. Pragmatic Bookshelf, 2010.
- Taylor, R.; Derrick, J. **Smother: an MC/DC analysis tool for Erlang**. In Proceedings of the 14th ACM SIGPLAN Workshop on *Erlang (Erlang 2015)*. 2015. ACM, New York, NY, USA, 13-18.
- Tóth, M.; Bozó, I. **Static Analysis of Complex Software Systems Implemented in Erlang**. 4th Summer - Central European Functional Programming School. CEFPS. 2011.
- Tóth, M.; Bozó, I. **Building Dependency Graphs for Slicing Erlang Programs**. In 7th Conference of PhD Students in Computer Science (Paper submitted to Periodica Polytechnica), Szeged, Hungary, June 2012.
- Tóth, M.; Bozó, I.; Horváth, Z.; Lövei, L.; Tejfel, M.; Kozsik, T. **Impact Analysis of Erlang Programs Using Behaviour Dependency Graphs**. Proceedings of the 3th Conference on Central European Functional Programming School, p.372-390, Komarno, Slovakia, May 25-30, 2010.
- Törnkvist, T. **Eldap User's Guide**. Ericsson AB. 2014.
- Varejão, Flávio M. **Linguagens de Programação: Conceitos e Técnicas**. Rio de Janeiro: Campus. 2004.
- Vergilio SR, Maldonado JC, Jino M. Infeasible paths in the context of data flow based testing criteria: Identification, classification and prediction. Journal of the

Brazilian Computer Society 2006.

Vidal, G. **Towards Symbolic Execution in Erlang** (short paper). In Proc. of the 9th International Andrei Ershov Memorial Conference on Perspectives of Systems Informatics (PSI'14), pages 351–360. Springer LNCS 8974, 2014.

Watt, David A. **Programming Languages: Concepts and Paradigms**. Prentice Hall International Series in Computer Science, 1990.

Weyuker, E. **The cost of data flow testing: an empirical study**. Software Engineering, IEEE Transactions on, v. 16, n. 2, p. 121 –128, 1990.

Widera, M. **Data flow considerations for source code directed testing of functional programs**. In H.-W. Loidl, editor, Draft Proceedings of the Fifth Symposium on Trends in Functional Programming, Nov. 2004.

Widera, M. **Concurrent Erlang flow graphs**. In Proceedings of the Erlang/OTP User Conference 2005, Stockholm, 2005a.

Widera, M. **Data flow coverage for testing Erlang programs**. In Marko van Eekelen, editor, Proceedings of the Sixth Symposium on Trends in Functional Programming (TFP'05), September 2005b.

Widera, M. **Why Testing Matters in Functional Programming**. 7th Symposium on Trends in Functional Programming, University of Nottingham, TFP, 2006.