Teste estrutural de programas concorrentes como uma composição de serviços na Web

Rafael Regis do Prado

SERVIÇO DE PÓS-GRADUAÇÃO DO ICMC-USP

Data de Depósito:

Assinatura: _____

Rafael Regis do Prado

Teste estrutural de programas concorrentes como uma composição de serviços na Web

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

Área de Concentração: Ciências de Computação e Matemática Computacional

Orientador: Prof. Dr. Paulo Sérgio Lopes de Souza

USP – São Carlos
Maio de 2016

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

P634t

Prado, Rafael Regis do
Teste estrutural de programas concorrentes como uma composição de serviços na Web / Rafael Regis do Prado; orientador Paulo Sérgio Lopes de Souza. - São Carlos - SP, 2016.
105 p.

Dissertação (Mestrado - Programa de Pós-Graduação em Ciências de Computação e Matemática Computacional) - Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, 2016.

1. Teste de software. 2. Programas concorrentes. 3. Composição de serviços Web. I. Souza, Paulo Sérgio Lopes de, orient. II. Título.

Rafael Regis do Prado

Structural testing of concurrent programs as a Web service composition

Master dissertation submitted to the Instituto de Ciências Matemáticas e de Computação – ICMC-USP, in partial fulfillment of the requirements for the degree of the Master Program in Computer Science and Computational Mathematics. *FINAL VERSION*

Concentration Area: Computer Science and Computational Mathematics

Advisor: Prof. Dr. Paulo Sérgio Lopes de Souza

USP – São Carlos
May 2016

AGRADECIMENTOS

Primeiramente, gostaria de agradecer aos meus pais, Maria Helena de Jesus e Ismael Dias do Prado, ao meu orientador prof. Dr. Paulo Sérgio Lopes de Souza e à profa. Dra. Simone do Rocio Senger de Souza pelo apoio, orientação, dedicação e paciência durante toda minha formação pessoal e profissional.

Agradeço também aos meus amigos Hugo, Murilo, Fabio, Renato, Jairo, Filipe, Lucas, Bruno, George, Raphael e Bianca pela valiosa amizade e companhia durante toda essa jornada e por contribuírem direta ou indiretamente para este projeto.

Agradeço aos professores João Lourenço e Rodrigo Rodrigues por me receberem no meu estágio de pesquisa na Universidade Nova de Lisboa, darem todo o suporte necessário e contribuírem para a execução deste projeto.

Agradeço aos membros do LaSDPC e LabES e aos funcionários do ICMC pelo suporte e infraestrutura que possibilitaram a execução deste projeto com qualidade.

Por fim, agradeço ao CNPq pelo apoio financeiro nos primeiros meses do mestrado e à FAPESP pelo apoio financeiro durante os meses seguintes (processos 2013/01818-7, 2013/05750-8 e 2014/15916-3).

“The mind is not a vessel to be filled,
but a fire to be kindled.”
Mestrius Plutarchos (Plutarch)

RESUMO

PRADO, R. R.. **Teste estrutural de programas concorrentes como uma composição de serviços na Web**. 2016. 105 f. Dissertação (Mestrado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação (ICMC/USP), São Carlos – SP.

O teste de programas concorrentes é essencial para assegurar a qualidade das atuais aplicações distribuídas/paralelas em desenvolvimento. Apesar de ser essencial, essa atividade de teste é dificilmente empregada adequadamente, devido a fatores como: alto custo de execução, grande lacuna entre desenvolvedores e resultados de pesquisas em testes para programas concorrentes e acesso às ferramentas de teste de programas concorrentes que automatizem/viabilizem o emprego do teste. Este projeto visa definir os parâmetros da atividade de teste estrutural de programas concorrentes que nortearão a composição de diferentes serviços na Web. Tais serviços dão suporte à atividade de teste estrutural de programas concorrentes, estabelecendo fronteiras claras em ferramentas de teste para os módulos relativos ao modelo de teste, aos critérios de teste, à linguagem de programação e aos paradigmas de sincronização. Desse modo, novas ferramentas de teste poderão ser construídas de maneira mais flexível, com menos custo de desenvolvimento e com mais eficácia. Tal abordagem traz como benefícios diretos: (1) facilitar a interação entre os setores da indústria, ensino e pesquisa que estejam interessados no desenvolvimento de programas concorrentes com qualidade; (2) diminuir os custos de instalação e manutenção de ferramentas de teste estrutural pelos desenvolvedores; (3) facilitar a incorporação da atividade de teste de programas concorrentes no ciclo de desenvolvimento das aplicações distribuídas e paralelas; (4) aumentar a abrangência do projeto TestPar, permitindo que novos usuários (desenvolvedores, professores e outros grupos de pesquisa) possam utilizar facilmente os conhecimentos gerados no projeto; e (5) realimentar o projeto TestPar com novas demandas qualificadas, estas advindas de novos programas concorrentes submetidos para teste.

Palavras-chave: Teste de software, Programas concorrentes, Composição de serviços Web.

ABSTRACT

PRADO, R. R.. **Teste estrutural de programas concorrentes como uma composição de serviços na Web**. 2016. 105 f. Dissertação (Mestrado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação (ICMC/USP), São Carlos – SP.

Testing of concurrent programs is essential to ensure the quality of today's distributed/parallel applications in development. Although it is essential that testing activity is hardly properly employed, due to factors such as high cost of implementation, big gap between developers and research results in tests for competing programs and access to competing software testing tools to automate / enable the test job. This project aims to define the parameters of structural testing activity of concurrent programs that will guide the composition of different Web services. These services support the structural testing activity of concurrent programs, establishing clear boundaries in test tools for the modules related to the test model, the test criteria, the programming and synchronization paradigms language. Thus, new test tools can be built in a more flexible way, with less development cost and more effectively. Such an approach has as direct benefits: (1) facilitate interaction between industry sectors, education and research who are interested in the development of concurrent programs with quality; (2) reduce the costs of installation and maintenance of structural testing tools for developers; (3) facilitate the incorporation of testing activity of concurrent programs in the development cycle of distributed and parallel applications; (4) increase the scope of TestPar design, allowing new users (developers, teachers and other research groups) can easily use the knowledge generated in the project; and (5) feed back into the project TestPar with new demands qualified, those arising from new concurrent programs submitted for testing.

Key-words: Software testing, Concurrent programs, Web service composition.

LISTA DE ILUSTRAÇÕES

Figura 1 – Ciclo de Vida de Processos. No diagrama, as setas indicam as transições entre os possíveis estados de um processo. 8

Figura 2 – Taxonomia de Flynn para arquiteturas computacionais. No diagrama, as setas indicam fluxo de instruções e de dados. *U.P.* indica uma unidade de processamento. A unidade de controle, implícita na figura, entrega instruções do conjunto de instrução para as unidades de processamento. 10

Figura 3 – Cenário típico da atividade de teste 22

Figura 4 – Grafo de fluxo de controle extraído do programa do Código 3 26

Figura 5 – Arquitetura da ferramenta ValiPar 34

Figura 6 – Representação do relacionamento entre os modelos na arquitetura de um *Web Service*. Baseado na especificação em (CONSORTIUM, 2004). 41

Figura 7 – Representação detalhada dos modelos associados à arquitetura de *Web Services*. Baseado na especificação em (CONSORTIUM, 2004). 42

Figura 8 – Componentes do serviço JaBUTiService (ELER *et al.*, 2009). 50

Figura 9 – Representação da Ontologia de Teste OntoTest (BARBOSA *et al.*, 2008) em UML 56

Figura 10 – Especializacão da sub-ontologia de procedimento e estratégia de teste OntoTest (BORGES; BARBOSA, 2009) em UML para o teste estrutural de programas concorrentes 56

Figura 11 – Especializacão da sub-ontologia de recurso de teste OntoTest (BARBOSA *et al.*, 2008) em UML para o teste estrutural de programas concorrentes 57

Figura 12 – Especializacão da sub-ontologia de passo de teste OntoTest (BARBOSA; NAKAGAWA; MALDONADO, 2006) em UML para o teste estrutural de programas concorrentes. 59

Figura 13 – Passo de teste baseado nas atividades da Figura 12. 60

Figura 14 – Interação entre um cliente os serviços. 70

Figura 15 – Interação entre um cliente e a ValiPar Service e entre a ValiPar Service com os serviços especializados. O cliente requisita a geração de artefatos e a ValiPar Service, atuando como um “proxy”, delega as requisições para os demais serviços. 71

Figura 16 – Arquitetura interna de cada serviço. Os serviços são descritos em termos de contratos e capacidades habilitadas (em destaque). Os modos de contrato “SI”, “SL”, “UL”, “GL” são siglas para “Sem interface”, “Somente leitura”, “*Upload* e leitura”, “Geração e leitura”. Os modos para capacidades “N”, “IS”, “IP”, “E” significam, respectivamente, “Nenhum”, “Interno sequencial”, “Interno paralelo”, “Externo”. 72
Figura 17 – Resultados obtidos para a execução dos benchmarks MC, HO, MS, NB e RW utilizando diferentes ferramentas. 77
Figura 18 – Resultados obtidos para a execução dos benchmarks PG, TR, MA e GT utilizando diferentes ferramentas. 78

LISTA DE CÓDIGOS-FONTE

Código-fonte 1 – Exemplo de utilização de semáforos 16
Código-fonte 2 – Exemplo de utilização de monitores e variáveis de condição 17
Código-fonte 3 – Exemplo de programa 26
Código-fonte 4 – Exemplo de requisição a um serviço RESTful 44
Código-fonte 5 – Exemplo de resposta de um serviço RESTful 44
Código-fonte 6 – Exemplo de requisição SOAP 46
Código-fonte 7 – Exemplo de resposta SOAP 46

LISTA DE TABELAS

Tabela 1 – *Benchmarks* utilizados no experimento. Cada programa utiliza uma variedade de primitivas de sincronização (para passagem de mensagem (PM) e memória compartilhada (MC)), possui um número variado de processos e *threads* e apresenta diferentes padrões de sincronização e comunicação. Os valores P, M, G entre parenteses indicam a classificação “pequeno”, “médio” e “grande”, respectivamente. As siglas NB, PG, GT, MC, RW, HO, MS, TR, MA significam, respectivamente, non blocking bsend, parallel gcd, gcd two slaves, micro sm, readers writers, h2o, master slave, token ring file, matrix. Por fim, as três ultimas colunas comparam o tempo de resposta entre a ferramenta *desktop* e duas versões de serviços. “desk”, “seq” e “dist” indicam, respectivamente, tempo de resposta na execução do *benchmark* com a ferramenta valipar, valipar-service-6n-0d e valipar-service-6n-8d. 75

SUMÁRIO

1 INTRODUÇÃO 1
1.1 Contextualização e Motivação 1
1.2 Objetivos 4
1.3 Organização 4

2 PROGRAMAÇÃO CONCORRENTE 7
2.1 Considerações Iniciais 7
2.2 Contextualização 7
2.3 Desenvolvimento de Programas Concorrentes 10
2.4 Paradigmas de Sincronização e Comunicação 13
2.4.1 Memória Compartilhada *14*
2.4.2 Passagem de Mensagem *18*
2.5 Considerações Finais 20

3 TESTE DE SOFTWARE 21
3.1 Considerações Iniciais 21
3.2 Contextualização 21
3.3 Fases da atividade de Teste 22
3.4 Técnicas de Teste 23
3.4.1 Teste Funcional *24*
3.4.2 Teste Estrutural *25*
3.4.3 Teste de Mutação *27*
3.5 Teste de Programas Concorrentes 28
3.5.1 Abordagens de Teste de Programas Concorrentes *29*
3.5.2 Modelo e Critérios de Teste TestPar *30*
3.6 Ferramentas de Teste de Programas Concorrentes 32
3.6.1 Ferramenta ValiPar *33*
3.7 Considerações Finais 36

4 WEB SERVICES 37
4.1 Considerações Iniciais 37
4.2 Contextualização 37
4.3 Web Services 39
4.3.1 Arquitetura de Web Services *40*

4.3.2 *Estilos Arquiteturais* *43*
4.4 *Web Services* para o Teste de Software 46
4.4.1 *Ferramenta Cloud9* *47*
4.4.2 *Ferramenta PathCrawler* *48*
4.4.3 *Ferramenta PascalMutants Service* *48*
4.4.4 *Ferramenta JABUTiService* *49*
4.5 Considerações Finais 51

5 VALIPAR SERVICE 53
5.1 Considerações Iniciais 53
5.2 Estudo de Ontologias de Teste de Software 53
5.3 Identificação da Atividade de Teste Estrutural de Programas Concorrentes 55
5.4 Definição de Capacidades de Serviços 59
5.5 Definição de Contratos de Serviços 62
5.6 Desenvolvimento dos Serviços de Teste 62
5.7 Conjunto de serviços ValiPar Service 64
5.7.1 *ValiInst Service* *65*
5.7.2 *ValiElem Service* *65*
5.7.3 *ValiExec Service* *66*
5.7.4 *ValiEval Service* *67*
5.7.5 *ValiSync Service* *67*
5.7.6 *ValiPar Service* *68*
5.8 Considerações Finais 69

6 EXPERIMENTOS E ANÁLISE DOS RESULTADOS 73
6.1 Considerações Iniciais 73
6.2 Avaliação Quantitativa dos Serviços 73
6.3 Análise Qualitativa dos Serviços 79
6.3.1 *Funcionalidade da Ferramenta* *80*
6.3.2 *Exposição de Funcionalidades* *81*
6.3.3 *Custo de Desenvolvimento de Novas Ferramentas de Teste* *83*
6.4 Considerações Finais 85

7 CONCLUSÃO 87
7.1 Considerações Iniciais 87
7.2 Caracterização da Pesquisa Realizada 87
7.3 Contribuições 88
7.4 Dificuldades e Limitações 89
7.5 Produção Científica 90

7.6 Trabalhos Futuros 91

REFERÊNCIAS 93

APÊNDICE A ESPECIFICAÇÃO DE CONTRATOS DE SERVIÇOS . 99

CAPÍTULO

1

INTRODUÇÃO

1.1 Contextualização e Motivação

A computação atual é cada vez mais pervasiva e distribuída, onde sistemas computacionais diferentes interagem para, juntos, melhorarem a nossa qualidade de vida. Avanços constantes no hardware e no desenvolvimento de software vêm gerando um ciclo virtuoso que contribui para alavancar o desenvolvimento atual da computação.

A programação concorrente é hoje amplamente utilizada para possibilitar a melhor utilização de recursos computacionais. Esse paradigma de programação é utilizado na computação de alto desempenho e em sistemas computacionais distribuídos, onde há o processamento de quantidades cada vez maiores de informações, sempre considerando o tempo de resposta requerido para o domínio de cada aplicação. De fato, hoje muitos sistemas são inerentemente concorrentes e distribuídos. Por exemplo, aplicações de comunicação por texto, voz e vídeo precisam interagir entre si e com outros serviços para realizar suas atividades. A *World Wide Web* é, também uma grande aplicação distribuída que depende da comunicação entre serviços para atender a requisições de usuários.

A ampla utilização desses paradigmas traz vários desafios para o ciclo de desenvolvimento de software. Se por um lado programas sequenciais possuem uma vasta gama de abordagens para garantir a qualidade dos mesmos, programas concorrentes ainda carecem de mais desenvolvimento. Diferentemente de programas sequenciais, programas concorrentes utilizam operações de comunicação e sincronização, as quais são responsáveis por erros que podem facilmente gerar resultados inconsistentes, difíceis de serem detectados durante os testes. Neste cenário, o teste e a depuração de aplicações concorrentes e distribuídas é um desafio não trivial.

Isso ocorre devido à necessidade de modelos de teste específicos que extraiam informa-

ções relevantes neste contexto, principalmente aquelas relacionadas à comunicação, sincronização e ao não determinismo, todas inerentes ao comportamento dinâmico de tais aplicações. Associados a esses modelos de teste, também se fazem necessários novos critérios e ferramentas que permitam conduzir a atividade de teste dentro de um custo aceitável.

O projeto TestPar desenvolve modelos, critérios e ferramentas para o teste estrutural, a fim de garantir a qualidade desse tipo de programa para apoiar a atividade de teste de programas concorrentes. Este projeto investiga o teste de programas concorrentes que utilizam recursos de memória compartilhada (SARMANHO *et al.*, 2008) e de passagem de mensagens (SOUZA *et al.*, 2008; ENDO *et al.*, 2008). Atualmente, vêm sendo investigadas também técnicas que fazem uso de ambos os paradigmas de sincronização (passagem de mensagens e memória compartilhada) (SOUZA *et al.*, 2013).

O teste de programas concorrentes possui várias questões em aberto. Uma amostra da lista dos desafios nessa área pode ser: (1) desenvolver técnicas de análise estática do código fonte combinadas à análise dinâmica, esta oriunda da execução dos processos; (2) gerar modelos e critérios de teste que capturem a semântica de programas concorrentes que interagem concomitantemente por passagem de mensagens e por memória compartilhada; (3) gerar modelos e critérios de teste que sejam ortogonais às linguagens de programação; (4) analisar a eficácia dos critérios de teste em revelar defeitos e a relação de inclusão dos mesmos; (5) desenvolver ferramentas de teste eficientes e ortogonais às linguagens de programação concorrente; (6) reduzir o custo da atividade de teste em processos concorrentes permitindo a aplicação de modelos e critérios a programas reais; (7) investigar a aplicação das técnicas funcionais, baseadas em erros e estruturais no contexto de programas concorrentes, considerando aspectos complementares, de eficiência e de custo; e (8) gerar automaticamente casos de teste.

As questões 1, 6 e 7 estão associadas à instanciação dos modelos e critérios como ferramentas. Esse aspecto é essencial para a aplicação do teste de forma parcial ou totalmente automática. De fato, a aplicação totalmente manual se torna impraticável para programas de pequeno, médio ou grande porte uma vez que é um processo demorado, repetitivo e propenso a erros.

Entretanto, mesmo a aplicação totalmente automática pode ser custosa em termos de recursos computacionais e tempo de resposta, em especial para programas distribuídos. Esse tipo de aplicação necessariamente exige múltiplas execuções para forçar a execução de sincronizações específicas. Esse requisito traz grandes impactos para as questões de recursos computacionais e tempo de resposta uma vez que programas distribuídos de maior porte com maior quantidade de processos/*threads* e de possíveis sincronizações potencialmente exigem ainda mais execuções para satisfazer critérios. Essa realidade dificulta muito o teste realizado com ferramentas *desktop* em que todo o processamento é limitado aos recursos de um computador.

Além disso, há também aspectos relacionados à manutenção e ao custo de desenvolvimento de ferramentas de teste. É desejável que, da mesma forma que ocorre com outros tipos

de ferramenta, o custo de instalação e atualização do software seja minimizado. Além dessas questões, espera-se igualmente que o as funcionalidades existentes na ferramenta possam ser re-utilizadas em outros contextos para diminuir o custo de desenvolvimento.

Todas essas questões são, de alguma forma, abordadas pelo paradigma de orientação a serviços. A disponibilização de uma funcionalidade permite que, no momento propício, a funcionalidade seja desenvolvida de forma escalável utilizando uma infra estrutura que permita a paralelização de atividades consideradas custosas com relação ao tempo de resposta. O reuso da mesma é também facilitado já que a utilização do serviço é feita por meio de uma interface independente de linguagem de programação e de aplicações específicas.

Por fim, o processo de atualização da base de usuários da ferramenta de teste se torna menos custosa, pois a atualização da funcionalidade no serviço (como a adição de novas características, melhoramento de desempenho ou correção de defeitos) impacta diretamente nas ferramentas que fazem uso de tal serviço (clientes). Assim, atualizações relacionadas à funcionalidade somente são necessárias em casos de mudança na interface (API) do serviços.

Várias ferramentas na literatura (CIORTEA *et al.*, 2010; ELER *et al.*, 2009; WILLIAMS *et al.*, 2005) aplicam a abordagem de orientação a serviços para o teste de software. São demonstradas, de um modo geral, vantagens nessa abordagem relativas à disponibilização de funcionalidades e, em vários casos, ao tempo de resposta. Não há discussão, entretanto, de questões de reusabilidade de funcionalidades. Além disso, nenhum dos serviços pesquisados realiza o teste estrutural de programas concorrentes. Atualmente a ferramenta ValiPar, desenvolvida no projeto TestPar, permite o teste estrutural de programas concorrentes de modo integrado para ambiente *desktop*.

A aplicação da abordagem de orientação a serviços pode trazer benefícios para o teste estrutural de programas concorrentes. De fato, a criação de serviços independentes focados em partes distintas da atividade de teste pode diminuir custos no desenvolvimento de novas ferramentas e a disponibilização desse conjunto de serviços na Web pode ampliar o alcance e incentivar a integração do teste de programas concorrentes no ciclo de de desenvolvimento de software.

Entretanto, uma série de questões devem ser resolvidas para se ter efetivamente um conjunto de serviços com tais benefícios. A atividade de teste estrutural de programas concorrentes segue uma visão monolítica, unindo os conceitos relativos a modelos, critérios, linguagens de programação e paradigmas de sincronização em torno de uma ferramenta de teste de software que oferece o suporte requerido a essa atividade. Isso faz com que os componentes de ferramentas de teste relacionadas sejam fortemente dependentes entre si, o que dificulta, por exemplo, a organização dos mesmos como serviço atendendo requisitos de reusabilidade.

1.2 Objetivos

O objetivo central deste projeto de pesquisa é definir uma composição de serviços na Web para o teste estrutural de programas concorrentes para apoiar a atividade de teste.

De modo específico, esse projeto objetiva a decomposição da atividade de teste estrutural de programas concorrentes em diferentes módulos independentes, definindo fronteiras entre aspectos relativos ao modelo de teste, critérios de teste, linguagens de programação e paradigmas de sincronização e comunicação. Além disso, faz parte do objetivo o desenvolvimento e disponibilização de um conjunto de serviços voltado ao teste estrutural de programas concorrentes no contexto do modelo e critérios de teste do projeto TestPar e da ferramenta ValiPar focando em aspectos de reusabilidade, custo de desenvolvimento e manutenção.

Espera-se, assim, que a abrangência do projeto TestPar seja estendida, visto que os serviços Web de teste estrutural para programas concorrentes poderão ser utilizados mais facilmente em atividades não previstas inicialmente. Um exemplo de aumento da abrangência é a possibilidade de composição de novas ferramentas para o desenvolvimento de programas concorrentes, onde os serviços disponibilizados por este projeto possam ser empregados no ciclo de desenvolvimento de programas concorrentes com propósito didático, acadêmico e também industrial.

1.3 Organização

Considerando o contexto, motivação e objetivos expostos, este projeto de pesquisa está organizado em capítulos, como detalhado a seguir. O Capítulo 2 apresenta os fundamentos da programação concorrente abordando paradigmas de sincronização, a metodologia de desenvolvimento de programas concorrentes e métricas tipicamente utilizadas na avaliação de desempenho de programas desenvolvidos.

O Capítulo 3 introduz conceitos de teste de software relacionados às fases e técnicas de teste. Neste capítulo são apresentados também os principais problemas relacionados ao desenvolvimento de programas concorrentes, as iniciativas que visam à garantia de qualidade nesse processo e os modelos e critérios do projeto TestPar, foco desse projeto. Por fim, são introduzidas as ontologias de teste atualmente desenvolvidas.

Os conceitos relativos à caracterização e ao desenvolvimento de *Web Services* são expostos em conjunto com as ferramentas de teste desenvolvidas como serviços no Capítulo 4.

Os conceitos abordados pelos capítulos 2, 3 e 4 representam a base para o desenvolvimento deste projeto.

O Capítulo 5, descreve o processo de planejamento e implementação dos serviços para o

teste estrutural de programas concorrentes. São definidas todas as atividades e artefatos presentes na atividade de teste e todo o processo de decomposição das mesmas em um conjunto de processamentos e contratos, agrupados em serviços com papéis bem definidos.

Esse conjunto de serviços é validado no Capítulo 6. São avaliados aspectos quantitativos e qualitativos da ferramenta. O tempo de resposta para a conclusão de um fluxo de execução em três versões dos serviços criados e na ferramenta *desktop* equivalente é medido e comparado quantitativamente. Aspectos de manutenção, reusabilidade e custo de desenvolvimento são analisados de modo qualitativo, relacionando-se com outras ferramentas que também utilizam o paradigma de orientação a serviços.

Por fim, o Capítulo 7 apresenta as principais conclusões deste projeto de mestrado, destacando-se as contribuições, dificuldades, limitações e produção científica derivada dos trabalhos relacionados ao projeto.

CAPÍTULO

2

PROGRAMAÇÃO CONCORRENTE

2.1 Considerações Iniciais

Neste capítulo são expostos os conceitos relacionados à programação concorrente. Inicialmente é realizada a contextualização do tema (Seção 2.2) abordando arquiteturas e sistemas computacionais e a origem da programação concorrente. Em seguida, na Seção 2.3, conceitos essenciais no desenvolvimento de programas concorrentes são explicados. Por fim, na Seção 2.4, as semânticas de comunicação e sincronização entre processos são expostas e instanciadas para os paradigmas de memória compartilhada e distribuída.

2.2 Contextualização

A execução de programas de uma forma geral está fortemente relacionada à arquitetura computacional e ao sistema operacional em que é executado. De fato, a combinação de diversos tipos de arquiteturas e de sistemas computacionais muitas vezes exige abordagens de desenvolvimento diferentes e permite a exploração dos recursos computacionais em diversos níveis quando se tem como objetivo desenvolver programas concorrentes.

Os sistemas computacionais podem ser divididos em dois tipos: sistemas mono e multiprogramados. Em sistemas monoprogramados, todos os recursos computacionais presentes estão necessariamente disponíveis para um processo (um programa em execução) até o momento em que completa sua execução (TANENBAUM, 2007). Tal característica é considerada um gargalo por subutilizar os recursos computacionais disponíveis e impedir que novos processos os utilizem mesmo quando o processo em posse dos recursos está ocioso.

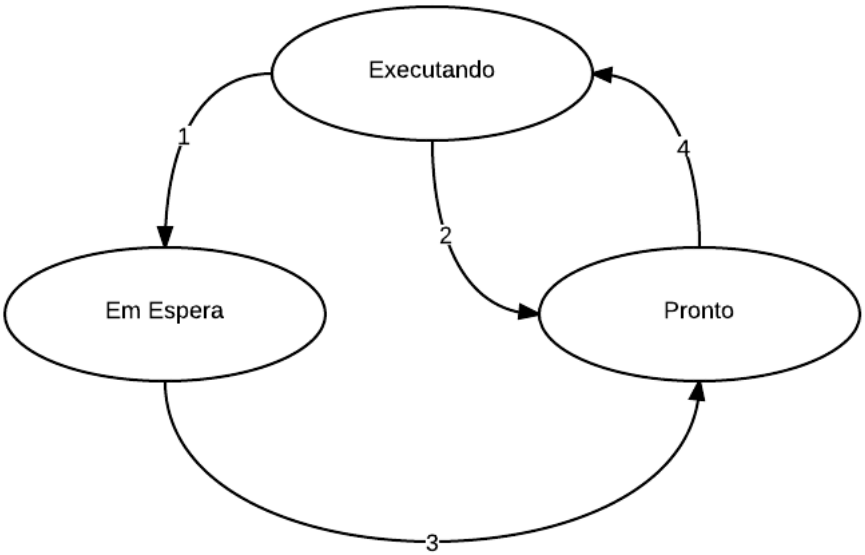


Figura 1 – Ciclo de Vida de Processos. No diagrama, as setas indicam as transições entre os possíveis estados de um processo.

Sistemas multiprogramados eliminam esse gargalo ao permitir que todos os processos progridam com o tempo. Nesse tipo de sistema, ao invés de disponibilizar o processador de forma exclusiva para um processo durante todo seu tempo de execução, o tempo de utilização desse recurso é dividido em *quanta* (porções) de tempo as quais são atribuídas a cada processo seguindo critérios predefinidos (TANENBAUM, 2007).

Nesse sentido, em um determinado instante de tempo, cada processo do sistema está em um de três estados: “em execução”, “em espera” ou “pronto” (Figura 1). Quando o sistema operacional fornece a CPU para um processo, o mesmo está “em execução”. Até o momento em que for interrompido, o mesmo pode utilizar a CPU para realizar computações.

Ao ser interrompido, o processo passa para o estado “pronto” ou “em espera”. Um processo “em execução” se torna “pronto” (transição 2 da Figura 1) caso o *quantum* atribuído a ele tenha expirado. Por outro lado, o mesmo passa para o estado “em espera” (transição 1 da Figura 1) caso tenha requisitado um recurso do sistema (memória, semáforo, etc.). Quando tal requisição for atendida, o processo “em espera” passa para o estado “pronto” (transição 3 da Figura 1). Após desalocar um processo da CPU, o sistema operacional elege um novo processo no estado “pronto” (transição 4 da Figura 1).

Esse tipo de sistema traz possibilidades de programação inexistentes em sistemas mono-programados. Pelo fato de haver mais de um processo ativo (iniciado, porém não terminado), tem-se processos concorrentes. Tal configuração viabiliza de fato a programação concorrente em que uma aplicação é dividida em um conjunto de processos que interagem entre si para o progresso de uma determinada tarefa. Essa configuração possibilita, inclusive, a execução de sistemas distribuídos, em que os processos que compõem a aplicação estão efetivamente

distribuídos fisicamente em vários sistemas e se comunicam por meio de passagem de mensagem (COULOURIS *et al.*, 2011). A união dos conceitos de programação concorrente e sistemas distribuídos é denominada computação distribuída.

Apesar de possibilitar um progresso homogêneo de processos concorrentes com o tempo, um sistema multiprogramado não permite, necessariamente, a execução paralela. Processos concorrentes são por definição, processos que iniciaram, mas não finalizaram suas execuções em um determinado instante de tempo. Processos paralelos, por outro lado, são um tipo especial de processos concorrentes, pois estão executando paralelamente no mesmo instante de tempo, em diferentes unidades de processamento (está intimamente relacionado à arquitetura) (TANENBAUM, 2007).

A arquitetura do computador desempenha um papel importante no desempenho do sistema computacional. Além de fornecer um melhor aproveitamento dos recursos, questões como o tempo de resposta do sistema estão diretamente relacionadas à arquitetura utilizada. Segundo a taxonomia de Flynn, é possível categorizar as arquiteturas computacionais de acordo com seus fluxos de dados e de instruções, os quais podem ser únicos ou múltiplos (FLYNN, 1972). Dessa forma, existem 4 grupos distintos de arquiteturas, representados na Figura 2.

Como exemplo de arquiteturas SISD (*Single Instruction Single Data*, ou fluxo único de instruções e de dados), pode-se citar a arquitetura de von Neumann. Como exemplo de arquiteturas SIMD (*Single Instruction Multiple Data*, ou fluxo único de instruções e fluxo múltiplo de dados) tem-se os processadores vetoriais. Com relação à arquitetura MISD (*Multiple Instruction Single Data*, ou fluxo múltiplo de instruções e fluxo único de dados) pode ser representada pelas arquiteturas *pipeline*, apesar de não haver unanimidade a respeito da existência de arquiteturas desse tipo.

Por fim, as arquiteturas MIMD (*Multiple Instruction Multiple Data*, ou fluxo múltiplo de instruções e de dados), foco desse projeto de mestrado, são representadas pela maioria dos computadores paralelos atuais. Nesse tipo de arquitetura, tem-se múltiplos processadores que atuam de forma independente sobre dados na memória. Dessa forma, o acesso à memória pode ser feito por memória compartilhada, em que todos os processadores possuem acesso direto à memória, ou por memória distribuída, em que cada processador possui sua própria memória e compartilha seus dados por meio de passagem de mensagem.

Junto aos benefícios relacionados ao desempenho trazido pela composição de sistemas multiprogramados e arquiteturas paralelas, uma série de novas questões são consideradas ao se implementar aplicações concorrentes. De fato, diferentemente da programação sequencial, há uma preocupação maior na interação entre os processos que compõem a aplicação, na decomposição eficiente de tarefas em sub-tarefas e em paradigmas de sincronização e de comunicação a serem utilizados (KUMAR *et al.*, 1994; QUINN, 1994). Dessa forma, cabe ao projetista identificar o grau de concorrência ideal para a aplicação sendo desenvolvida e aproveitar efetivamente a arquitetura paralela a ser empregada.

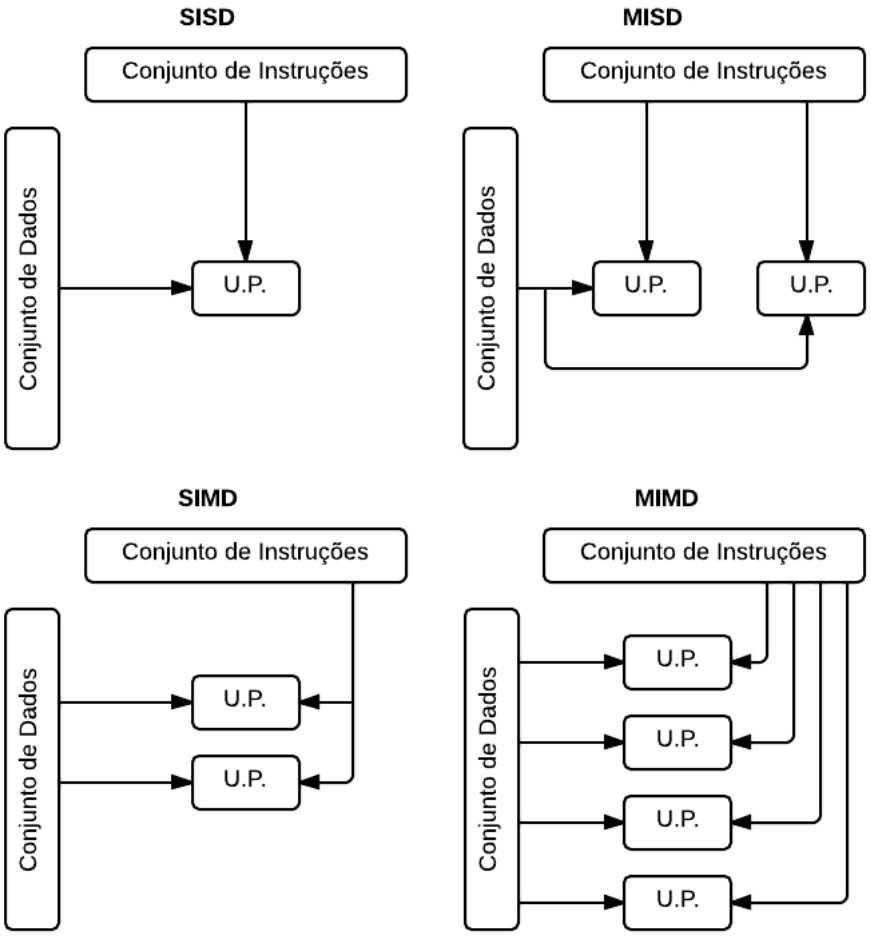


Figura 2 – Taxonomia de Flynn para arquiteturas computacionais. No diagrama, as setas indicam fluxo de instruções e de dados. *U.P.* indica uma unidade de processamento. A unidade de controle, implícita na figura, entrega instruções do conjunto de instrução para as unidades de processamento.

2.3 Desenvolvimento de Programas Concorrentes

A implementação de programas concorrentes exige, além dos requisitos estabelecidos na programação sequencial, a preocupação com uma nova dimensão: a de agendamento entre as tarefas, ou seja, a partir de que ponto uma tarefa pode ser realizada e de que modo tais tarefas devem interagir para completar uma computação.

Tal preocupação se traduz em uma série de medidas a serem tomadas durante o desenvolvimento. De modo geral, há a necessidade de se identificar porções independentes do sistema, definindo tarefas, posteriormente mapeados em processos. Além disso deve-se lidar com questões de distribuição e gerenciamento de dados entre tarefas e a sincronização dos processos para completar a tarefa (KUMAR *et al.*, 1994).

No contexto de implementação de sistemas concorrentes, define-se como tarefa uma unidade indivisível de computação que compõe uma computação de maior porte. Tais tarefas compõem um grafo de dependência em que uma tarefa só pode ser executada após o termino de outras tarefas.

Uma computação pode, em geral, ser decomposta de diversas formas, o que pode levar a diferentes grafos de dependência. Essa decomposição pode resultar inclusive em um número variado de tarefas de tamanhos diversos determinando a granularidade da decomposição. Uma decomposição em que se tem uma grande quantidade de pequenas tarefas possui uma granularidade fina. Por outro lado, uma decomposição em uma pequena quantidade de grandes tarefas possui granularidade grossa.

Define-se grau de concorrência de uma aplicação o número de tarefas executadas em paralelo em uma computação. Dessa forma, o grau máximo de concorrência é o número máximo de tarefas que podem ser executadas em paralelo. O grau médio de concorrência, analogamente, é o número médio de tarefas executadas em paralelo.

Tais medidas em geral são fortemente influenciadas pela granularidade da decomposição e pela configuração das tarefas no grafo de dependências. Uma decomposição de granularidade mais fina em geral resulta em um grau de concorrência maior. Além disso, quanto menor for o caminho crítico no grafo de dependência (caminho constituído pela maior sequência de tarefas dependentes), em geral, maior o grau de concorrência (KUMAR *et al.*, 1994).

Além da análise de dependência em um conjunto de tarefas, deve-se também analisar a interação entre as tarefas para a troca de dados e para a manutenção da consistência dos mesmos. A interação entre as tarefas pode ser representada por um grafo de interação em que as tarefas são representadas por nós e as arestas simbolizam a presença de interação entre as tarefas em questão. De modo geral o conjunto de arestas em um grafo de dependência é um sub conjunto do conjunto de arestas em um grafo de interação.

As seguintes técnicas podem ser utilizadas para se decompor uma computação em tarefas: a técnica recursiva, a baseada em dados, a exploratória e a especulativa. Dentre essas técnicas, as duas primeiras são de propósito mais geral enquanto as duas últimas possuem propósito mais específico. Além disso pode-se utilizar mais de uma técnica, resultando em soluções híbridas.

A técnica recursiva, também conhecida como uma abordagem *divide-and-conquer* é baseada na decomposição do problema em sub-problemas que podem ser resolvidos da mesma maneira. Por exemplo, uma ordenação paralela de um vetor de inteiros poderia ser recursivamente decomposta na ordenação de dois sub-vetores com a metade do tamanho.

Na técnica baseada em dados, a decomposição é feita baseada no particionamento dos dados de entrada, de saída ou ainda em dados intermediários da computação. Tal particionamento em geral induz a um paralelismo a ser explorado por um conjunto de tarefas. Como exemplo, pode-se citar a multiplicação de matrizes em que para cada tarefa é atribuído um conjunto de

linhas em colunas do dado de entrada. Essa divisão em tarefas poderia ainda ser orientada à matriz final ou ainda a um processamento intermediário.

A técnica exploratória é aplicada em contextos em que é necessária a busca pela resposta em um espaço de soluções. Dessa forma, a partir de um determinado estado, todos os próximos estados possíveis são explorados por novas tarefas de forma independente. Tal exploração ocorre até que uma tarefa identifique a solução, encerrando a computação do problema.

Por fim, a técnica especulativa pode ser utilizada em contextos em que a execução ou não de uma tarefa está condicionada ao resultado de outra. Nessa estratégia, executa-se a tarefa de forma especulativa ao mesmo tempo que é executada a tarefa a qual está condicionada. Em casos em que a tarefa não deveria ter sido executada, simplesmente descarta-se o resultado. Caso contrário, há uma economia no tempo de resposta. Essa técnica é frequentemente usada em casos em que um cálculo custoso somente é válido caso uma condição (cuja verificação também é custosa) seja atendido.

A partir de um conjunto de tarefas a serem completadas e o relacionamento (dependência e interação) entre as mesmas, ocorre o mapeamento em processos (definido como agentes de computação nesse contexto). Tal mapeamento é realizado objetivando-se maximizar o grau de concorrência e diminuir o tempo de resposta da computação. Da mesma forma em que o grau de concorrência é definido pelo modo como a computação é decomposta em tarefas, o mapeamento apropriado de tais tarefas em processos define o grau de concorrência real da computação. Tais objetivos são em geral cumpridos ao se mapear tarefas independentes em processos diferentes e tarefas com alto grau de interação em um mesmo processo.

Diversas características das tarefas definidas devem ser observadas ao se realizar o mapeamento em processos para que seja possível ganhar desempenho. Uma característica a ser observada é a natureza da geração de tarefas. Em certos contextos, o número de tarefas é fixo e previsível (geração estática) enquanto em outros contextos a quantidade de tarefas está condicionada à características dos dados de entrada (geração dinâmica).

Outro ponto a ser observado é o tamanho e a uniformidade das tarefas levadas em consideração, caso essa medida seja conhecida. Essas informações possibilitam o melhor balanceamento de tarefas nos processos, o que pode acarretar em um menor tempo de resposta da computação (KUMAR *et al.*, 1994).

Por fim, deve-se considerar também o tipo da interação entre as tarefas. Assim, pode-se ter situações em que o comportamento da interação é conhecido previamente (estático) e ou não (dinâmico). Além disso a natureza das interações pode ser regular (em que se consegue estruturar a interação de alguma forma) ou irregular.

Existem diversos modelos tipicamente utilizados na estruturação de algoritmos paralelos. Tais modelos se baseiam na escolha de técnicas de decomposição e mapeamento para minimizar aspectos como interação entre processos. Dentre os modelos estão o baseado no paralelismo de

dados, o baseado em grafos de dependência, o *work pool*, o modelo mestre-escravo e o produtor-consumidor. Tais modelos podem ser utilizados em conjunto formando modelos híbridos.

O modelo baseado no paralelismo de dados explora o particionamento de dados e execução de tarefas idênticas para cada partição. No modelo *work pool*, as tarefas são distribuídas dinamicamente para os processos em situações em que as tarefas podem ser potencialmente executadas por qualquer processo.

O modelo baseado em grafos de dependência explora o relacionamento entre as tarefas para reduzir os custos resultantes da interação entre as mesmas. O modelo mestre-escravo define processos mestre, os quais geram e delegam trabalho, e os processos escravos, que realizam o trabalho. Por fim, no produtor-consumidor, um fluxo de dados é passado por uma série de processos formando um *pipeline*. Nesse modelo, um processo produtor gera ou modifica novos dados os quais são consumidos por outros processos (denominados consumidores).

Após a decomposição da computação em tarefas e o mapeamento em processos é feita a implementação do sistema. Essa implementação deve levar em consideração aspectos como os paradigmas de sincronização e comunicação a serem utilizados assim como a natureza da interação entre os processos. Além disso, é necessário decidir por tecnologias a serem empregadas no processo de desenvolvimento. Por fim, como tais escolhas impactam diretamente no desempenho da aplicação desenvolvida, deve haver um monitoramento e análise do desempenho de forma a obter informações como possíveis gargalos a serem solucionados.

2.4 Paradigmas de Sincronização e Comunicação

No contexto de programação concorrente, dois paradigmas de comunicação e sincronização podem ser utilizados: memória compartilhada e memória distribuída. De forma geral, no paradigma de memória compartilhada, processos se comunicam utilizando um espaço de endereçamento de memória em comum e, para que isso seja feito de forma consistente, o acesso a tal espaço é realizado de forma síncrona. No paradigma de memória distribuída, por outro lado, cada processo possui seu próprio espaço de endereçamento. Para esse caso, a comunicação e a sincronização são feitas por meio de passagem de mensagem entre os processos.

Independente do paradigma de comunicação e sincronização, primitivas (operações) de sincronização e comunicação são disponibilizadas para que possa haver interação (troca de dados) entre os processos que compõem o sistema. Para que essa interação seja efetiva, tais primitivas, em geral, estabelecem uma relação de causa e efeito entre os processos participantes. Nesse sentido, operações como *SEND* (transmissão de mensagem) podem ser generalizadas como ações causadoras de um evento. Simetricamente, operações de recebimento de mensagens (*RECEIVE*) podem ser generalizados para receptoras de eventos.

O modo como essa interação é realizada depende, sobretudo, da semântica das primitivas a serem utilizadas. De modo geral, uma primitiva pode ser síncrona ou assíncrona, bloqueante ou não bloqueante, ponto-a-ponto ou coletiva. A semântica da primitiva a ser utilizada em uma interação possui grande impacto no desempenho da aplicação como um todo.

A semântica bloqueante está diretamente relacionada com a segurança dos dados utilizados na interação entre os processos. Desse modo, uma primitiva é dita bloqueante se é assegurado que, ao completar a operação, é seguro manipular a memória relacionada com a mesma. Uma primitiva não bloqueante, por outro lado, não assegura tal condição sendo necessário um teste para avaliar a possibilidade de utilização da memória envolvida na operação (KUMAR *et al.*, 1994).

Já com relação ao sincronismo, uma primitiva é considerada síncrona quando a mesma espera até que haja o pareamento com a primitiva correspondente. Uma primitiva *SEND* síncrona, nesse sentido, espera até que a primitiva *RECEIVE* no processo receptor seja executada. Uma primitiva assíncrona, em contrapartida, não espera até que o pareamento entre as primitivas ocorra.

Por fim, uma primitiva pode ter uma semântica ponto-a-ponto ou coletiva. Uma primitiva ponto-a-ponto envolve dois processos, o processo transmissor, que executou uma primitiva *SEND*, e o processo receptor, que executou a primitiva *RECEIVE*. Uma primitiva coletiva, por outro lado, pode ser classificada como um para muitos (um transmissor e vários receptores), muitos para um (vários transmissores e um receptor) ou muitos para muitos (vários transmissores e vários receptores).

Independente do paradigma de comunicação e sincronização, primitivas (operações) de sincronização e comunicação são disponibilizadas para que possa haver interação (troca de dados) entre os processos que compõem o sistema. Por exemplo, enquanto a comunicação e a sincronização são frequentemente realizadas em conjunto no paradigma de passagem de mensagem, a comunicação é feita implicitamente no paradigma de memória compartilhada por meio de variáveis compartilhadas cujo acesso é sincronizado por primitivas de sincronização. Tal relacionamento entre comunicação e sincronização resulta em primitivas específicas para cada paradigma e impacta diretamente no modo como sistemas são programados em cada paradigma.

2.4.1 Memória Compartilhada

Conceitualmente, o paradigma de memória compartilhada é baseado na existência de várias linhas de execução que possuem acesso a um espaço de endereçamento global. A comunicação, dessa forma, é realizada de forma implícita na leitura e escrita de valores nesse espaço de endereçamento. Em situações em que mais de uma linha de execução pode realizar escritas nessa memória ou em situações em que deve-se manipular variáveis (várias leituras e escritas, ou mais de uma variável) de forma atômica, deve-se estabelecer regiões de exclusão mútua (ou

regiões críticas) para garantir a consistência dos dados.

Em sistemas operacionais modernos, um processo é composto por um conjunto de recursos como descritores de arquivos, espaço de endereçamento, atributos de segurança (TANENBAUM, 2007). Além disso, cada processo pode possuir várias linhas de execução, ou *threads*. Cada *thread*, por sua vez, possui acesso aos recursos alocado para o processo em que foi criada, além de estado de execução (“em execução”, “em espera” e “pronto”) e espaço de endereçamento próprios.

Como mais de uma *thread* pode estar em execução em um determinado instante, é possível desenvolver sistemas concorrentes utilizando tal recurso. Dessa forma, o mapeamento de tarefas é orientado a *threads* e a interação é feita utilizando o espaço de endereçamento em comum entre as mesmas.

A sincronização nesse paradigma é, em geral, realizada por meio de semáforos, monitores, variáveis de condição e barreiras (TANENBAUM, 2007). A partir dessas primitivas, são derivadas construções mais complexas e de propósitos mais específicos que garantem semânticas diferentes.

O conceito de semáforos se baseia na utilização de uma variável inteira para contar o número de *wake ups* e duas operações atômicas: *DOWN* e *UP*. A operação *DOWN* verifica a quantidade de *wake ups* do semáforo, caso o valor for maior que zero, há apenas o decremento do valor. Caso contrário, a *thread* bloqueia no semáforo. A operação *UP*, por sua vez, incrementa o valor do semáforo caso o mesmo seja maior que zero ou, caso contrário, acorda uma *thread* bloqueada no semáforo.

Um semáforo é denominado *mutex* (ou *lock*) ao ser utilizado para garantir o acesso exclusivo a uma região de memória. Nesse caso, o semáforo deve possuir no máximo um *wake up*. Dessa forma, operações *DOWN* e *UP* são chamadas *LOCK* e *UNLOCK*, respectivamente.

A semântica de semáforos e *mutexes* pode ser observada no exemplo apresentado no Código 1, escrito em Java. Tal exemplo resolve o problema clássico “Produtor-Consumidor” utilizando semáforos e *mutexes* para coordenar o acesso a um *buffer* de tamanho limitado para a inserção e remoção de itens.

Os semáforos *full* (cheio) representa a quantidade de itens que podem ser retirados de *buffer* enquanto o semáforo *empty* representa a quantidade de *slots* (espaços) disponíveis em *buffer* para a inserção de elementos. A variável *lock*, por sua vez, possui a semântica de *mutex* e protege o acesso ao *buffer*.

Antes de cada acesso à região crítica, *threads* produtoras sinalizam atomicamente a utilização de um *slot* removendo um *wake up* do semáforo *empty* utilizando-se o método *acquire()* (linha 27) e, no final da operação, indicam atomicamente a inserção do item adicionando um *wake up* no semáforo *full* utilizando-se o método *release()* (linha 31). De forma análoga, *threads* consumidoras sinalizam atomicamente a requisição por um item removendo um *wake up* do semáforo *full* (linha 8) e depositando um *wake up* no semáforo *empty* (linha 12).

```
1  class Consumer extends Thread {
2    // Variaveis compartilhadas entre as threads consumidoras e produtoras.
3    private Semaphore full, empty;
4    private Lock lock;
5    private List<Item> buffer;
6    public void run() {
7      while (true) {
8        full.acquire();
9        lock.lock();
10       Item item = buffer.remove(0);
11       lock.unlock();
12       empty.release();
13       consumeItem(item);
14     }
15   }
16   // Outros metodos
17 }
18
19 class Producer extends Thread {
20   // Variaveis compartilhadas entre as threads consumidoras e produtoras.
21   private Semaphore full, empty;
22   private Lock lock;
23   private List<Item> buffer;
24   public void run() {
25     while (true) {
26       Item item = produceItem();
27       empty.acquire();
28       lock.lock();
29       buffer.add(item);
30       lock.unlock();
31       full.release();
32     }
33   }
34   // Outros metodos
35 }
```

Código-fonte 1: Exemplo de utilização de semáforos

Apenas com a utilização de semáforos não há a garantia de acesso coordenado à região crítica. Para isso, antes de manipular a variável compartilhada, é realizada a operação *lock()* (linha 9 e 28), a qual permite que apenas uma *thread* possa manipulá-la por vez. Por fim, após a manipulação da variável, a *thread* deixa a região crítica realizando a operação *unlock* (linha 11 e 30).

Essa configuração, além de coordenar o acesso ao *buffer*, mantém os limites do mesmo e evita situações de *underflow* (quando um consumidor tenta retirar um item de um *buffer* vazio) e de *overflow* (quando um produtor tenta adicionar um item em um *buffer* cheio). Até que as condições para o consumo e para a produção sejam verdadeiras, as *threads* ficam, assim, bloqueadas.

As variáveis de condição por sua vez, são primitivas síncronas atreladas a mutexes ou monitores e são baseadas em duas operações: *WAIT* e *SIGNAL*. Em geral, esse tipo de primitiva

```
1  class Monitor {
2    private Buffer buffer = null;
3    public synchronized void produce() {
4      while (buffer != null) {
5        try { wait(); } catch (Exception e) {}
6      }
7      buffer = new Buffer();
8      notifyAll();
9    }
10   public synchronized void consume() {
11     while (buffer == null) {
12       try { wait(); } catch (Exception e) {}
13     }
14     consumeBuffer(buffer);
15     buffer = null;
16     notifyAll();
17   }
18 }
```

Código-fonte 2: Exemplo de utilização de monitores e variáveis de condição

é utilizado em situações em que uma *thread* não pode prosseguir até que uma dada condição ocorra. Quando uma *thread* em uma região crítica é impedida de continuar, a operação *WAIT* é executada.

Tal operação executa um conjunto de passos atomicamente. É realizada uma operação *UNLOCK* no mutex (ou monitor) relacionado, saindo da região crítica, e tal *thread* é bloqueada até que a condição em questão seja favorável. Ao verificar tal condição, outra *thread* na região crítica pode realizar a operação *SIGNAL* e sair da região crítica. Por meio dessa operação, uma *thread* bloqueada é acordada e assume a região crítica.

Apesar de possuir uma semântica semelhante a de um semáforo, existe uma diferença fundamental entre as duas primitivas. Enquanto um semáforo é um contador, as variáveis de condição não possuem tal propriedade. Como principal consequência desse fato, em uma situação em que uma operação *SIGNAL* é executada antes de uma operação *WAIT*, não há o pareamento entre essas duas operações. Nesse caso, o sinal emitido se perde e a *thread* que executou a operação *WAIT* é bloqueada até que uma operação *SIGNAL* seja executada novamente.

Os monitores são construções da linguagens de programação que dão suporte à sincronização em um nível mais elevado. Para isso, um monitor é composto por um conjunto de procedimentos e variáveis, estas acessíveis apenas pelos procedimentos. Para que seja mantida a consistência dos dados, esses procedimentos são sincronizados, ou seja, permitem que apenas uma *thread* utilize o monitor em cada instante.

Como pode ser observado no Código 2, a linguagem java fornece a funcionalidade de monitores por meio de blocos ou métodos sincronizados (utilizando-se *synchronized*) e também fornece variáveis de condição por meio dos métodos *wait()* , *notify()* e *notifyAll()* em cada objeto.

Nesse exemplo, utiliza-se os conceitos de monitores e variáveis de condição para resolver

o problema Produtor-Consumidor com um *buffer* unitário. Apenas uma *thread* pode utilizar entrar em uma instancia de *Monitor* de cada vez uma vez que são disponibilizados apenas dois método sincronizados (com o modificador *synchronized*) *produce()* para as *threads* produtoras e *consume()* para as consumidores. Como só se pode consumir um item caso ele exista e produzir item caso haja *slot* disponível, *threads* consumidoras e produtoras esperam por essa condição na variável de condição representada pelo próprio monitor.

Dessa forma, no instante em que o método *wait()* (linha 5 e 12) é chamado, a *thread* deixa a região crítica e é bloqueada até que outra *thread* chame o método *notifyAll()* (linha 8 e 16), que acorda todas as *threads*. Ao chamar tal primitiva, é dada a oportunidade para a outra *thread* prosseguir com a computação. Tal configuração garante que as condições necessárias para a correta execução da *thread* sejam atendidas antes que a mesma manipule os dados em questão.

Por fim, as barreiras são primitivas síncronas coletivas (muitos-para-muitos). Em situações em que a aplicação concorrente é tipicamente estruturada em fases e o progresso das *threads* entre tais fases somente é permitido quando todas as *threads* envolvidas estejam prontas para prosseguir. Para garantir essa semântica, é criada uma barreira e cada *thread* realiza uma operação *WAIT* no momento em que estiver pronta para avançar para próxima fase.

2.4.2 Passagem de Mensagem

O paradigma de passagem de mensagem se baseia no mapeamento de tarefas em processos, sem o compartilhamento de memória para a comunicação. Nesse sentido a interação entre os processos ocorre por meio de mensagens. Como consequência dessa abordagem, a comunicação e a sincronização entre os processos é explícita e feita em conjunto por meio de operações de envio (*SEND*) e recebimento (*RECEIVE*) de dados. Cada uma dessas operações possui um conjunto de variações que combinam diferentes semânticas de sincronismo, bloqueio e tipo de comunicação.

No contexto de passagem de mensagem, uma operação *SEND* síncrona espera até que a operação *RECEIVE* correspondente seja executada para que haja a transferência da mensagem. Uma operação *SEND* assíncrona, por outro lado, não exige o pareamento com um *RECEIVE* para o término da execução. As operações *RECEIVE* são, em geral, assíncronas e bloqueantes.

Quanto à semântia de bloqueio, no contexto de passagem de mensagem uma operação *SEND* ou *RECEIVE* é dita bloqueante quando a mensagem está segura ao término da operação. Caso não haja essa garantia, a operação é definida como não bloqueante. Para esse último caso, operações de teste devem ser executadas para verificar o término da transferência antes que a variável envolvida seja manipulada.

A segurança dos dados pode ser feita de diversas formas variando-se, de modo geral, o desempenho resultante (KUMAR *et al.*, 1994) Uma forma de se garantir a integridade dos dados é por meio da cópia da mensagem a ser enviada (ou recebida) para um *buffer*. Com a ausência de

um *buffer*, uma operação *SEND* deve esperar até que haja uma sincronização com a operação *RECEIVE* para que a transferência dos dados seja iniciada. Por meio dessas duas possíveis estratégias de transferência, após completar a operação em questão, não há a possibilidade de manipular um dado cuja transferência esteja ainda em progresso.

O modo como os dados são transferidos também depende do suporte de hardware fornecido. Caso uma unidade DMA (*Direct Memory Access*, ou Acesso Direto à Memória) esteja presente, por exemplo, a transferência pode ocorrer de forma independente da unidade central de processamento (CPU). Caso contrário, haverá um custo adicional de processamento para a transferência. Assim, tanto o processo transmissor quanto o receptor da mensagem serão interrompidos para que a CPU realize a transferência da mensagem entre os respectivos *buffers*.

De modo geral, operações bloqueantes e síncronas, principalmente em situações em que não há *buffers* ou hardware específicos auxiliares, levam a maiores *overheads* na interação entre os processos devido ao tempo necessário para garantir a segurança e transferência dos dados. Para casos em que necessita-se diminuir tal *overhead*, operações assíncronas e não bloqueantes são preferíveis.

Ortogonal às semânticas de sincronia e bloqueio, o tipo das operações pode variar entre operações ponto-a-ponto (um para um) e operações coletivas (um para muitos, muitos para um e muitos para muitos) (KUMAR *et al.*, 1994).

Como exemplo de operação “um para um”, tem-se as operações *SEND* e *RECEIVE*, que envolvem apenas um transmissor e um receptor. A operação *BROADCAST* é considerada do tipo “um para muitos” em que uma mesma mensagem é transmitida a partir de um processo para todos os demais. De forma análoga, a operação *REDUCTION* (do tipo “muitos para um”) envolve apenas um processo receptor e vários transmissores. Por fim, a operação *BARRIER* (do tipo “muitos para muitos”) cada processo participante da interação envia uma mensagem para todos os processos e recebe uma de cada um deles, permitindo a sincronização entre os envolvidos.

Outras abstrações podem ainda ser construídas a partir das primitivas *SEND* e *RECEIVE*. As abstrações mais utilizadas são o RPC (*Remote Procedure Call*), o *RMI* (*Remote Method Invocation*) e o *Web Service* (detalhado no Capítulo 4).

A abordagem utilizada em RPC e RMI permitem a chamada de procedimentos e a manipulação de objetos (no caso da RMI) cujas implementações se encontram potencialmente em outro computador de forma transparente para o desenvolvedor. Estas implementações não serão detalhadas nesse texto, mas os detalhes sobre as semânticas disponíveis podem ser encontrados em (TANENBAUM; STEEN, 2006; COULOURIS *et al.*, 2011).

2.5 Considerações Finais

Este capítulo abordou os principais aspectos relacionados à programação concorrente. Foram discutidas as origens da programação concorrente, assim como o ambiente que propicia sua execução. Além disso foram expostos os conceitos necessários para o desenvolvimento de sistemas concorrentes os paradigmas utilizados. Por fim, as métricas comumente utilizadas para se avaliar o desempenho de sistemas concorrentes foram apresentadas.

CAPÍTULO

3

TESTE DE SOFTWARE

3.1 Considerações Iniciais

Neste capítulo serão abordados conceitos relacionados à atividade de teste. Na Seção 3.2, uma breve contextualização é apresentada, abordando conceitos gerais do teste de software. Em seguida, as diferentes fases da atividade de teste são explicadas na Seção 3.3. Na Seção 3.4, as diferentes técnicas de teste são abordadas, dando-se maior enfoque para a técnica estrutural, foco deste projeto.

Na Seção 3.5 são introduzidos conceitos relacionados ao teste de programas concorrentes em geral e, em específico, ao teste estrutural de programas concorrentes utilizando os modelos e critérios desenvolvidos no projeto TestPar. Por fim, na Seção 3.6, o atual cenário de ferramentas concorrentes é apresentado, dando-se foco para a ferramenta ValiPar, parte deste projeto.

3.2 Contextualização

Com o objetivo de garantir a qualidade de software, atividades de verificação, validação e teste de software (VV&T) são frequentemente aplicadas em todo o processo de desenvolvimento de software. Tais atividades são classificadas como estáticas, quando não exige a execução do programa sendo analisado, ou dinâmicas, as quais são baseadas na execução do programa para a verificação de seu comportamento.

A atividade de teste de software, como uma atividade de VV&T, visa minimizar a ocorrência de erros na execução do programa sendo testado. Dessa forma, tenta-se garantir a qualidade do software em questão por meio de uma análise dinâmica do mesmo, identificando

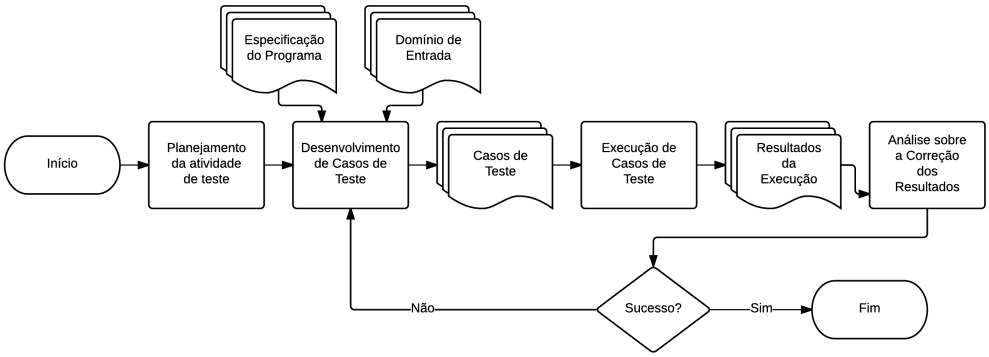


Figura 3 – Cenário típico da atividade de teste

erros (estados inconsistentes do programa) e fornecendo informações para a eliminação de defeitos (DELAMARO; MALDONADO; JINO, 2007).

Um cenário típico da atividade de teste pode ser visualizado na Figura 3.

Inicialmente é feito o planejamento em que são estabelecidos os objetivos do teste, os critérios e ferramentas a serem utilizados, por exemplo. Em seguida é feito o desenvolvimento dos casos de teste utilizando técnicas de teste previamente estabelecidas. Um caso de teste é composto por um dado de teste do domínio de entrada (conjunto de todos os possíveis valores de entrada) do programa e pelo resultado esperado. Tal desenvolvimento leva em consideração critérios de teste para que o custo da atividade de teste seja diminuído, sem que haja prejuízos quanto a sua eficácia em revelar defeitos.

O programa, na fase de execução, é submetido aos casos de teste desenvolvidos. Os resultados são então organizados e, na etapa de análise, são comparados com os resultados esperados pelo testador com base na especificação do programa. Caso o resultado seja diferente do especificado, defeitos foram revelados, o que é caracterizado como sucesso da atividade.

O modo como esse conjunto de etapas ocorre varia de acordo com as técnicas de teste e as fases da atividade de teste em questão. Quanto à técnica de teste, pode-se empregar a técnica funcional, estrutural ou baseada em modelos, às quais buscam revelar defeitos sob diferentes perspectivas. Ortogonalmente à técnica de teste, o teste pode ocorrer em três fases: teste de unidade, de integração e de sistema. Cada fase possui um propósito e é aplicado em diferentes contextos do desenvolvimento de software.

3.3 Fases da atividade de Teste

A ocorrência de erros pode aparecer em diferentes níveis. Como exemplo é possível que defeitos sejam inseridos na implementação de algoritmos específicos, na interação entre

bibliotecas do sistema ou ainda na implementação incorreta de um requisito estabelecido na especificação do software. Tais defeitos estão em níveis de abstração diferentes e devem ser abordados de modo diferenciado em diferentes etapas do desenvolvimento do projeto.

Defeitos como a implementação incorreta de um algoritmo devem ser identificados na fase de teste de unidade. Nessa fase, o software deve ser analisado nas menores unidades do programa, ou seja, funções, métodos ou ainda classes de forma isolada. Devido a essa granularidade do teste de unidade, o mesmo é em geral aplicado pelo próprio desenvolvedor da unidade durante o desenvolvimento do software, sem que o sistema esteja totalmente implementado.

Questões como a interação adequada entre bibliotecas ou módulos são analisadas na fase de teste de integração. Nesse sentido, verifica-se se a comunicação entre as partes do sistema que não levam a erros. Essa fase é frequentemente realizada pela própria equipe desenvolvedora do sistema, uma vez que exige conhecimento aprofundado das estruturas internas do sistema.

Por fim, na fase de teste do sistema, as funcionalidades especificadas na concepção do sistema são verificadas no sistema implementado. Além da verificação de aspectos funcionais como a correção e coerência do sistema, são averiguadas também questões não-funcionais como a segurança, portabilidade, performance e robustez do sistema. Essa fase é usualmente executada por equipes independentes e ocorrem no final do desenvolvimento do projeto.

3.4 Técnicas de Teste

Para se testar um programa totalmente, em teoria, é necessário que o mesmo seja submetido a um teste exaustivo de todos os dados de teste que compõem o domínio de entrada do programa. Tal abordagem, porém, é infactível uma vez que o domínio de entrada é, em geral, muito grande ou ainda infinito. Exemplos de programas cuja cobertura total é infactível estão em algoritmos de cálculo numérico, os quais devem considerar grandes intervalos de números inteiros ou reais.

Na prática são preferidas abordagens que permitam o teste de subconjuntos representativos do domínio de entrada. Para isso são determinados subdomínios de teste que consistem de dados de teste equivalentes para o programa sendo testado. Por exemplo, um programa que calcule o módulo de um número inteiro possui basicamente dois subdomínios tendo-se como base a técnica funcional: números negativos e números positivos. Nesse caso, espera-se que dois números negativos (ou positivos) sejam tratados da mesma forma pelo programa, tornando desnecessário o teste exaustivo de todos os números possíveis.

A escolha de subdomínios, no entanto, deve ser feita com base em critérios bem definidos para que haja grande confiança de que o subconjunto estabelecido represente o domínio de entrada. Um critério de teste é um predicado que deve ser satisfeito por um conjunto de casos de

teste e pode ser usado para guiar a geração dos dados de teste.

Os critérios de teste podem ser classificados em três tipos: funcionais, estruturais e baseados em defeitos. Cada critério se utiliza de informações específicas do programa ou procedimento sendo testado para derivar os subdomínios. Quando todos os requisitos do critério são atendidos pelo conjunto casos de teste definido, conclui-se que tal conjunto é adequado ao critério.

Embora não garanta a total correção do programa sendo testado (se comparado com um teste exaustivo), a utilização de um subconjunto de dados de teste estabelecido por um ou mais critérios de teste é capaz de determinar que o programa se comporta corretamente com um grau de confiança quando a atividade de teste é realizada com embasamento teórico e de modo criterioso. Além disso, diferentes técnicas de teste revelam, em geral, variedades diferentes de erros (FABBRI; VICENZI; MALDONADO, 2007) o que torna interessante a utilização complementar dos critérios de teste propostos.

3.4.1 Teste Funcional

No teste funcional, o sistema é considerado uma caixa preta, ou seja, não se analisa o código (detalhes de implementação). Ao invés disso, o mesmo se baseia na especificação do software, possíveis entradas e saídas para se gerar casos de teste.

Como exemplos de critérios funcionais, pode-se citar o Particionamento de Equivalência, Teste Funcional Sistemático e Análise do Valor Limite (FABBRI; VICENZI; MALDONADO, 2007). Na técnica Particionamento de Equivalência, o domínio de entrada é dividido em conjuntos cujos elementos (individualmente) representam toda a classe de equivalência. Tais classes determinam estados válidos e inválidos da aplicação e são distintas (sem sobreposição de valores). A partir desse particionamento são derivados casos de teste que cubram o maior número de classes válidas possível. Para cada classe inválida é criado um caso de teste exclusivo, o que evita efeitos de mascaramento, em que a soma dos efeitos de classes inválidas gera, coincidentemente, uma saída válida.

O critério Análise do Valor Limite se baseia no mesmo princípio do Particionamento de Equivalência. Nesse critério, porém, os elementos representantes das classes de equivalência não são escolhidos aleatoriamente. Ao invés disso, escolhem-se valores limitantes de cada classe, aumentando-se a probabilidade de encontrar defeitos (MYERS; SANDLER, 2004).

O Teste Funcional Sistemático combina os critérios Particionamento de Equivalência e Análise do Valor Limite. Como um diferencial desse critério, a partir do particionamento realizado, são requeridos ao menos dois casos de teste para cada classe de forma a evitar efeitos de mascaramento de falhas.

Como benefícios diretos dos critérios funcionais está a facilidade de se aplicá-las em qualquer paradigma de programação por não dependerem de construções específicas. Além disso, a técnica de teste funcional pode ser aplicada em todas as fases de teste.

Por outro lado, por basear-se apenas na especificação do programa, a eficácia da técnica funcional depende de uma boa especificação de requisitos. Além disso, não há garantias de que partes críticas do sistema sejam verificadas apropriadamente, o que torna conveniente a sua utilização em conjunto com outras técnicas de teste de modo complementar.

3.4.2 Teste Estrutural

A atividade de teste estrutural, foco deste projeto, trata o sistema como uma caixa branca, ou seja, estabelece os elementos requeridos de um software de acordo com a implementação do mesmo. Essa técnica é considerada complementar às demais técnicas, por cobrir classes distintas de defeitos e gera informações relevantes para outras atividades como a de depuração e manutenção de software (BARBOSA *et al.*, 2007).

Uma vez que tal técnica é dependente da implementação do programa sendo testado, o conhecimento de estruturas internas como desvios condicionais, laços de repetição, definições e usos de variáveis do programa é essencial para a geração de casos de teste. Tais informações são em geral extraídas do programa na forma de uma representação representação de programas chamada “Grafo de Fluxo de Controle” (CFG, *Control Flow Graph*).

No CFG, o programa é descrito por um grafo $G = (N, E, s)$ em que N representa o conjunto de nós, E o conjunto de arestas que interligam estes nós determinando o fluxo de controle do programa e s o nó de entrada. Com isso, o programa é necessariamente decomposto em blocos de comandos os quais possuem características específicas. Ao se executar o primeiro comando dentro de um bloco, todos os demais são executados sequencialmente, sem desvios condicionais para comandos dentro do bloco.

A partir do CFG são derivados caminhos de execução. Tais caminhos consistem em uma sequência finita de nós em que cada nó possui uma aresta interligando-o com o nó sucessor. Dessa forma, um caminho é considerado simples se todos os nós que o compõe são distintos (exceto possivelmente o primeiro e o último) e é considerado completo caso o primeiro nó e o último sejam o nó de entrada e o de saída do grafo G, respectivamente.

O fluxo de dados, nesse contexto, está relacionado às operações de definição e uso computacional e predicativo de variáveis. Uma definição ocorre no armazenamento de um valor em uma posição de memória. O uso computacional ocorre quando o valor de uma variável é utilizado em uma expressão aritmética ou atribuído a outra variável. O uso predicativo, por outro lado, ocorre em estruturas de decisão, ou seja, quando seu uso afeta diretamente o fluxo de controle do programa (BARBOSA *et al.*, 2007)

```
1 #include<stdio.h>
2 /* 1 */void imprime(char* palavra, int vezes) {
3 /* 1 */   int i = 0;
4 /* 2 */   while (i < vezes) {
5 /* 3 */     printf("%s\n", palavra);
6 /* 4 */     i++;
7 /* 2 */   }
8 /* 5 */}
```

Código-fonte 3: Exemplo de programa

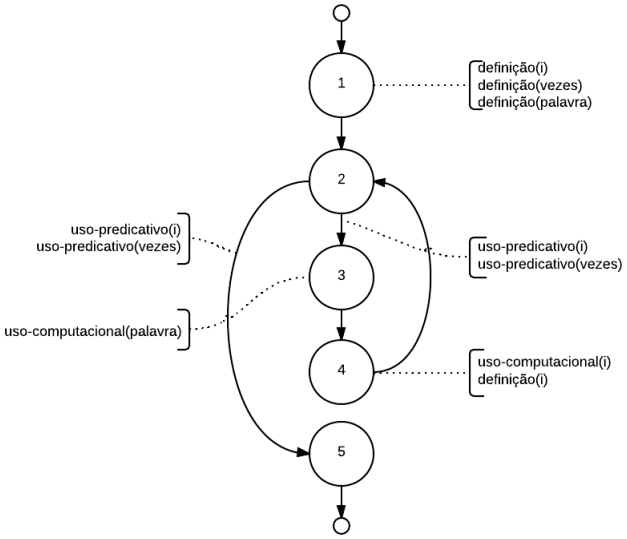


Figura 4 – Grafo de fluxo de controle extraído do programa do Código 3

O CFG da Figura 4 ilustra a extração das informações relevantes a partir do programa (desenvolvido em C) no Código 3 para a técnica de teste estrutural. Como pode ser observado, no nó 1, as variáveis *i*, *vezes* e *palavra* são definidas. Nas arestas que ligam o nó 2 com o nó 3 e o nó 4 com o nó 2, há o uso predicativo das variáveis *i* e *vezes*. No nó 3 há o uso computacional da variável *palavra*. Por fim, no nó 4, há o uso computacional seguido da definição da variável *i*.

A partir dessas informações extraídas são estabelecidos critérios de teste estrutural. Esses critérios são classificados em critérios baseados em fluxo de controle e em fluxo de dados. Os critérios de fluxo de controle utilizam apenas as informações referentes ao fluxo de controle do programa. Como exemplo desse tipo de critério tem-se o critério “Todos-Nós”, “Todas-Arestas” e “Todos-os-caminhos” que requerem a execução de todos os nós e todas as arestas e todos os caminhos possíveis do programa, respectivamente.

Analogamente, critérios de fluxo de dados se utilizam principalmente do fluxo de dados do programa para derivar seus requisitos de teste. Nesse sentido, tais critérios se baseiam nas associações entre as definições e usos de variáveis para derivar os casos de teste. Dentre os critérios existentes pode-se citar os critérios propostos por Rapps e Weyuker (RAPPS; WEYUKER,

1982). Tais critérios utilizam os conceitos de "du-caminho" e "associação definição-uso" para derivar para derivar requisitos.

Um "du-caminho" com relação a uma variável x é definido por um caminho em que há uma definição da variável x no primeiro nó, o caminho é simples e livre de definição com relação à x e há uma definição no último nó (ou última aresta, caso se trate de um uso predicativo).

Uma "associação definição-c-uso" é definida por uma tripla $\langle i, j, x \rangle$ que consiste de uma definição de uma variável x no nó i, há um caminho livre de definição entre os nós i e j e há um uso computacional da variável x no nó j. Analogamente, uma "associação definição-p-uso" é definida por uma tripla $\langle i, (j,k), x \rangle$ em que há um uso predicativo na aresta que relaciona os nós j e k.

Como exemplo de critério de fluxo de dados pode-se citar o critério "Todas-Definições", "Todos-Usos" e "Todos-Du-Caminhos". O critério "Todas-Definições" requer que todas as definições sejam cobertas por pelo menos um uso (computacional ou predicativo). O critério "Todos-Usos", por sua vez, requer que todos os pares definição e uso (predicativo ou computacional) sejam exercitados por pelo menos um caminho livre de definição. Por fim, o critério "Todos-Du-Caminhos" requer a cobertura de todas as associações definição-uso (predicativo e computacional).

Satisfazer um critério de teste estrutural nem sempre é possível devido aos elementos requeridos não executáveis (não há valores no conjunto de dados de teste capazes de cobrir um elemento requerido). Tendo em vista este cenário, cabe ao testador desenvolver conjuntos de casos de teste que exercitem de modo eficaz os elementos requeridos e identificar os não executáveis relacionados aos critérios levados em consideração.

Apesar de tal desvantagem, critérios de teste estrutural estabelecem de forma rigorosa e concreta os elementos requeridos que devem ser cobertos. Além disso os mesmos tornam a adequação do conjunto de casos de teste ao programa sendo testado facilmente mensurável, o que viabiliza inclusive a automatização da atividade.

3.4.3 Teste de Mutação

O teste de mutação é um critério de teste da técnica baseada em defeitos na qual utilizam-se defeitos comuns de implementação para derivar requisitos de teste para o teste de um determinado programa (DELAMARO *et al.*, 2007).

Nesse critério um conjunto de programas distintos, também chamados de mutantes, é gerado a partir da aplicação de operadores de mutação (como a eliminação de comandos, troca de operadores relacionais e troca de variáveis) no programa original sendo testado. Dessa forma, mostra-se a correção de um programa a partir de um conjunto de casos de teste que evidencie as

diferenças de comportamento entre o programa original e seus mutantes.

A aplicação de tal critério, nesse sentido, possui quatro fases bem definidas: geração de mutantes, execução do programa de teste, execução dos programas mutantes e análise de mutantes (DEMILLO; LIPTON; SAYWARD, 1978). Inicialmente, o conjunto de mutantes do programa sendo testado é construído. Nesse sentido, tem-se como objetivo a construção de um conjunto abrangente (que revele a maior parte dos erros) e que tenha baixa cardinalidade, ou seja, que seja um conjunto tratável durante a atividade de teste.

Em seguida, o programa original é submetido ao conjunto de casos de teste em que é verificado o comportamento do programa para um dado caso de teste que atenda as expectativas. Na fase seguinte, os programas mutantes são submetidos ao mesmo conjunto de casos de teste e, a partir dos resultados obtidos, é possível decidir a relevância do conjunto de casos de teste criado.

Se os resultados da aplicação do conjunto de casos de teste diferir entre um programa mutante e o programa original, diz-se que tal conjunto foi capaz de expor diferenças entre os dois programas e, consequentemente, o programa mutante é considerado morto e é descartado. Caso contrário, diz-se que o programa mutante continua vivo, indicando uma fraqueza no conjunto de casos de teste.

Na última fase, por meio de uma intervenção humana, é feita a análise dos mutantes vivos para decidir se a atividade de teste deve ou não continuar. Essa decisão se baseia no chamado *score* de mutação (métrica que considera a proporção de programas mutantes vivos e mortos) e na possível equivalência entre os programas mutantes vivos e o programa original. Quando o *score* é alto (poucos mutantes considerados vivos em relação ao total), pode-se considerar o conjunto de casos de teste como satisfatório.

No fim da atividade de teste utilizando tal critério, consegue-se um conjunto de casos de teste com grande capacidade em revelar defeitos. Entretanto, sua aplicação pode envolver um alto custo se considerarmos a quantidade de programas mutantes e de casos de teste a serem executados por programa mutante.

3.5 Teste de Programas Concorrentes

Como observado no Capítulo 2, o desenvolvimento de programas concorrentes adiciona uma nova dimensão ao desenvolvimento de software, inexistente na programação sequencial: a do agendamento entre tarefas. Enquanto programas sequenciais possuem comportamento determinístico, ou seja, é garantido que o resultado de uma computação a partir de uma dada entrada de dados é sempre o mesmo, tal comportamento não é presente em programas concorrentes devido às interações e compartilhamentos de dados entre processos.

Com a introdução do não-determinismo, novas categorias de erros podem ser observadas. Tais erros estão relacionados, sobretudo, ao compartilhamento inadequado de dados entre processos e a sincronizações que potencialmente levam a condições de disputa, situações de *deadlock*, em que há uma dependência circular de recursos (regiões críticas, por exemplo) em posse de processos, ou ainda *starvation* (ou inanição), em que um ou mais processos executam indefinidamente mas são incapazes de progredir por não conseguirem acessar devidamente um recurso (TANENBAUM, 2007).

Devido ao não-determinismo, diversas execuções de um mesmo programa concorrente com uma mesma entrada de dados podem ter resultados diferentes, possivelmente inconsistentes. Desta forma, deve-se observar, além das propriedades já estudadas em programas sequenciais, características próprias de aplicações concorrentes para que seja possível determinar a qualidade de softwares deste tipo (SOUZA; VERGILIO; SOUZA, 2007). Tendo em vista tal cenário, o desenvolvimento de técnicas de teste que levem em consideração as características inerentes desse tipo de programa é essencial.

O teste de programas concorrentes possui, assim, várias questões em aberto consideradas desafios, como: o desenvolvimento de técnicas de análise estática de código-fonte combinadas à análise dinâmica (execução dos processos), além da geração de modelos e critérios de teste que capturem a semântica de programas concorrentes (tanto por passagem de mensagem quanto por memória compartilhada) e que sejam ortogonais às linguagens de programação. Além disso destaca-se a necessidade de se desenvolver ferramentas eficientes, ou seja, que mantenham o custo da atividade de teste baixo, permitindo a aplicação de modelos e critérios a programas reais.

3.5.1 Abordagens de Teste de Programas Concorrentes

O teste de programas concorrentes é atualmente abordado de diversas formas. Em geral, as técnicas desenvolvidas estendem técnicas de teste já consolidadas no contexto de programas sequenciais. Nesse sentido, essas técnicas incorporam aspectos de sincronização e comunicação entre processos (ou *threads*) que compõem o programa concorrente e consideram fatores como não determinismo e condições de disputa.

O teste de mutação, por exemplo, pode ser utilizado no contexto de programas concorrentes para verificar as sincronizações e comunicações existentes nesse tipo de programa. Os operadores de mutação para programas concorrentes manipulam defeitos relacionados à semântica das primitivas utilizadas na programação e o modo como são utilizadas. Em (SEN; ABADIR, 2010) são apresentados operadores de mutação para programas escritos em C no paradigma de memória compartilhada. De modo similar, a técnica de mutação é também utilizada em (JAGANNATH *et al.*, 2010), em que é proposto o teste de mutação para programas

concorrentes baseados em *actors* (atores), derivando-se operadores específicos para esse modelo de programação.

Do ponto de vista do teste estrutural, pode-se citar as abordagens desenvolvidas em (YANG; SOUTER; POLLOCK, 1998; TAYLOR; LEVINE; KELLY, 1992; KOJIMA *et al.*, 2009) e os modelos e critérios desenvolvidos pelo projeto TestPar (apresentados na Seção 3.5.2) para o teste de programas concorrentes. De modo geral, as técnicas estruturais criam modelos e definem critérios de teste que cobrem diversos aspectos de programas concorrentes, tais como: a cobertura das possíveis sincronizações, os padrões de comunicação para ambos os paradigmas de sincronização e a interação entre o fluxo de dados e o de sincronização de tais programas.

Como exemplo, a abordagem desenvolvida em (KOJIMA *et al.*, 2009) estabelece o critério "todos caminhos concorrentes" (ACP — *all concurrent paths*) tendo como base um grafo de fluxo de módulo concorrente (CMFG — *concurrent module flow graph*). Em (YANG; SOUTER; POLLOCK, 1998) é estabelecido o critério "todos-caminhos-du" (*all-du-path*), o qual leva em consideração o fluxo de sincronização presente entre a definição e o uso de um dado compartilhado entre *threads*.

Outra técnica possível para o teste de programas concorrentes é o teste de alcançabilidade (*Reachability testing*) (LEI; CARVER, 2004; LEI; CARVER, 2006; LEI *et al.*, 2007; CARVER; LEI, 2010). Nessa abordagem, são derivadas sequências de sincronização, ou seja, combinações de pares de sincronização, a serem cobertas pela execução do programa concorrente sendo testado. Tal estratégia pode ser empregada tanto no paradigma de passagem de mensagem, quanto no de memória compartilhada e garante a cobertura de sequências de sincronização executáveis.

3.5.2 Modelo e Critérios de Teste TestPar

Os modelos e os critérios estruturais desenvolvidos pelo projeto TestPar (SOUZA *et al.*, 2005; SOUZA; VERGILIO; SOUZA, 2005; SOUZA *et al.*, 2008; SOUZA *et al.*, 2008; SOUZA *et al.*, 2011; SOUZA; SOUZA; ZALUSKA, 2012; SOUZA *et al.*, 2013) podem ser utilizados na atividade de teste estrutural de programas concorrentes. Tais modelos e critérios de teste estendem técnicas e estratégias já amplamente utilizadas no teste de softwares sequenciais para o contexto de programas concorrentes.

Para a extração dos elementos requeridos de programas concorrentes, neste modelo, são utilizados grafos de fluxo de controle paralelos (PCFG, *Parallel Control Flow Graph*) apresentados em (YANG; SOUTER; POLLOCK, 1998), adaptados em (SOUZA; VERGILIO; SOUZA, 2005) e posteriormente modificados em (SOUZA *et al.*, 2011; SOUZA *et al.*, 2013). Essa adaptação possibilita a aplicação de critérios de teste sobre programas que fazem uso do paradigma de passagem de mensagem.

Nessa representação, um programa *Prog* é representado por um conjunto de processos $Prog = \{p^0, p^1, p^2, \ldots, p^{np-1}\}$ de tamanho np fixo e cada processo *Proc* é composto por um conjunto de *threads* $Proc = \{t^0, t^1, t^2, \ldots, t^{nt-1}\}$ de tamanho nt fixo. Com isso, cada *thread* é representada por um $CFG^{p,t}$ (em que p é a identificação do processo e t a identificação da *thread*) cuja composição se assemelha ao CFG equivalente de programas sequenciais, apresentado na Seção 3.4.2. A união dos grafos de todas as *threads* formam, assim, o $PCFG$ do programa.

Como diferencial dos $CFGs$ sequenciais, é estabelecido ainda o conjunto N_{sync} composto por triplas $(n_i^{p,t}, f_g, b_u)$, em que b_u se refere à semântica da primitiva de comunicação ou sincronização que pertence a um *cluster* f_g e é executada em um nó $n_i^{p,t}$. Um *cluster* é um grupo de primitivas que podem sincronizar/comunicar entre si. A semântica, como detalhado no Capítulo 2, pode ser "Transmissor bloqueante", "Transmissor não-bloqueante", "Receptor bloqueante", "Receptor não-bloqueante", "Transmissor-receptor bloqueante" e "Transmissor-receptor não-bloqueante".

Além dos nós e das arestas, já utilizadas em programas sequenciais, as arestas de sincronização e comunicação arestas entre *threads* são representadas no modelo. Dessa forma, define-se como E_{sync} como o conjunto de arestas $(n_k^{p_i,t}, n_l^{p_j,t})$ em que $n_k^{p_i,t}$ e $n_l^{p_j,t}$ pertencem ao conjunto N_{sync} Tal aresta é do tipo inter-processo quando envolve nós em processos distintos e intra-processo quando envolve nós de um mesmo processo.

Por fim, define-se D^p como o conjunto de todos os dados alocados dinamica ou estaticamente em um processo p do programa concorrente. D^p_{shared} representa, por sua vez, o conjunto de dados alocados no processo p que estão compartilhados entre as *threads* do processo de forma que $D^p_{shared} \subseteq D^p$.

A partir desses conjuntos novos elementos são derivados. Além da definição, uso computacional e uso predicativo apresentados no Capítulo 3, três novos tipos de uso são definidos. Um uso em mensagem (m-uso) ocorre quando um dado em D^p é utilizado em um nó transmissor em N_{sync} para o envio de uma mensagem. Os usos compartilhados predicativo (s-p-uso) e computacional (s-c-uso), por sua vez, ocorrem quando o há um uso de um dado pertencente à D^p_{shared}.

De forma análoga ao que ocorre no teste sequencial, são derivadas associações a partir desses novos tipos de uso, adicionalmente às associações já existentes. Por exemplo, uma associação m-uso estabelece uma definição de um dado d em um nó $n_i^{p,t}$, um m-uso de d em um nó $n_j^{p,t}$ e há um caminho livre de definições entre $n_i^{p,t}$ e $n_j^{p,t}$.

Além disso, são derivadas associações que envolvem *threads* em um mesmo processo ou em processos distintos. Uma associação desse tipo envolve uma associação m-uso e, adicionalmente, considera a aresta de sincronização em que há a transmissão do dado entre as *threads* e o posterior uso (computacional ou predicativo) do dado na *thread* receptora. Tais associações são definidas, inclusive, para cada uma das possíveis semânticas da aresta de sincronização em que o

dado é definido.

Utilizando-se tais associações, são definidos critérios de teste que tem como objetivo exercitar as características intrínsecas de programas concorrentes que podem levar a defeitos causados por sincronizações e compartilhamentos de dados indevidos, levando a dados inconsistentes e situações de *deadlock* ou *starvation*. Além disso, leva-se em conta a característica do não determinismo presente nesse tipo de programa, exigindo que todos os caminhos possíveis sejam cobertos.

Dentre os critérios disponíveis, pode-se citar o *all-shared-uses* (ou todos usos compartilhados), o qual requer a cobertura de todas as associações s-c-uso e s-p-uso. Tal critério é específico para o fluxo de dados e tem como objetivo a revelação de defeitos baseados na comunicação por meio de memória compartilhada. Como critérios baseados em fluxo de controle e de sincronização específicos para programas concorrentes, pode-se citar o *all-nodes-s* e *all-nodes-r* que exigem, respectivamente, a cobertura de todos os nós transmissores e receptores pertencentes a N_{sync}.

3.6 Ferramentas de Teste de Programas Concorrentes

Diversas técnicas de teste de programas concorrentes são atualmente aplicadas por meio de ferramentas comerciais e acadêmicas que auxiliam no desenvolvimento de programas concorrentes. Dentre as ferramentas comerciais, pode-se citar a ferramenta ConTest, desenvolvida pela IBM e a ferramenta CHESS, desenvolvida pela Microsoft. Dentre as ferramentas acadêmicas, pode-se citar a ferramenta RichTest e a ferramenta ValiPar, que será apresentada na Seção 3.6.1

A ferramenta de teste ConTest (EDELSTEIN *et al.*, 2003), desenvolvida para a linguagem Java, utiliza heurísticas para forçar a execução das *threads* de um programa viabilizando cenários mais propensos a condições de disputa, *deadlocks* e outros tipos de erros. É possível, inclusive, re-executar (*replay*) o cenário que causou o erro. Com essa estratégia, possíveis erros ocorrem com maior frequência, evidenciando defeitos no programa sendo testado.

A ferramenta CHESS (MUSUVATHI; QADEER; BALL, 2007), para o teste de programas em C#, é baseada no teste exaustivo de programas concorrentes. O teste de programas concorrentes se inicia com a definição de cenários de concorrência interessantes, isto é, situações conhecidas que devem se comportar corretamente. O conceito de cenários de concorrência, nesse caso, é equivalente ao conceito de teste de unidade para programas sequenciais.

Após a criação dos cenários, os mesmos são executados sistematicamente diversas vezes considerando todos os possíveis *interleavings* (ordem de execução das instruções de todas as *threads* envolvidas). Como ocorre na ferramenta ConTest, essa ferramenta permite a re-execução de *interleavings* para um determinado cenário, auxiliando no processo de *debugging*. Essa

estratégia, apesar de ser custosa, é capaz de revelar vários tipos de defeitos e possui grande escalabilidade com relação ao tamanho do programa sendo testado.

Por fim, a ferramenta RichTest implementa o teste de alcançabilidade proposto em (LEI; CARVER, 2006). Para a aplicação do teste, a ferramenta permite a criação e execução de sequências de sincronização para o teste de programa em questão. Com isso, é possível identificar e reproduzir condições de disputas e forçar situações de *deadlocks*, por exemplo, acontecerem.

3.6.1 Ferramenta ValiPar

A ferramenta ValiPar tem como objetivo a aplicação do modelo e dos critérios de teste desenvolvidos pelo projeto TestPar apoiando a atividade de teste estrutural de programas concorrentes por meio da automatização dessa atividade. Versões da ferramenta foram implementadas para o teste de programas concorrentes que fazem uso do paradigma de passagem de mensagem escritos em BPEL, PVM e C/MPI (SOUZA *et al.*, 2005; SOUZA; VERGILIO; SOUZA, 2005; SOUZA *et al.*, 2008). Foi implementada também uma versão para o teste de programas que utilizem o paradigma de memória compartilhada por meio da biblioteca C/PThread (SARMANHO *et al.*, 2008).

Recentemente, houve a implementação da ferramenta com o objetivo de possibilitar o teste de programas concorrentes utilizando uma evolução do modelo e critérios de teste que levam em consideração primitivas de passagem de mensagem e memória compartilhada em um mesmo programa (SOUZA; SOUZA; ZALUSKA, 2012; SOUZA *et al.*, 2013). Tal ferramenta inicialmente suporta o teste de programas escritos em Java e, em breve, pretende-se incluir suporte às bibliotecas já estudadas (como PThread e C/MPI) de forma a validar o novo modelo no contexto das mesmas. Nessa implementação, novas técnicas foram aplicadas buscando-se suportar mais características normalmente presentes em programas em geral.

Como exemplo, pode-se citar o suporte à chamada de procedimentos. Nas versões anteriores, o suporte era dado em apenas um nível, impossibilitando que o grafo correspondente à chamada de procedimento fosse considerado na geração de elementos requeridos e, consequentemente, na cobertura dos critérios de teste.

Na versão mais recente da ferramenta, todos os procedimentos do programa concorrente são considerados na geração de grafos. Para isso, há inicialmente a construção de grafos parciais que correspondem apenas ao procedimento sendo analisado. Em uma segunda etapa, é feita a montagem do grafo final. Nesse processo, as chamadas de procedimentos são substituídas pelo grafo parcial correspondente.

Outra característica suportada na implementação mais recente é a manipulação de ponteiros e alocação dinâmica de memória. Devido à natureza da linguagem de programação Java, o suporte a tais característica se tornou imprescindível. O suporte dado possibilita que a ferramenta

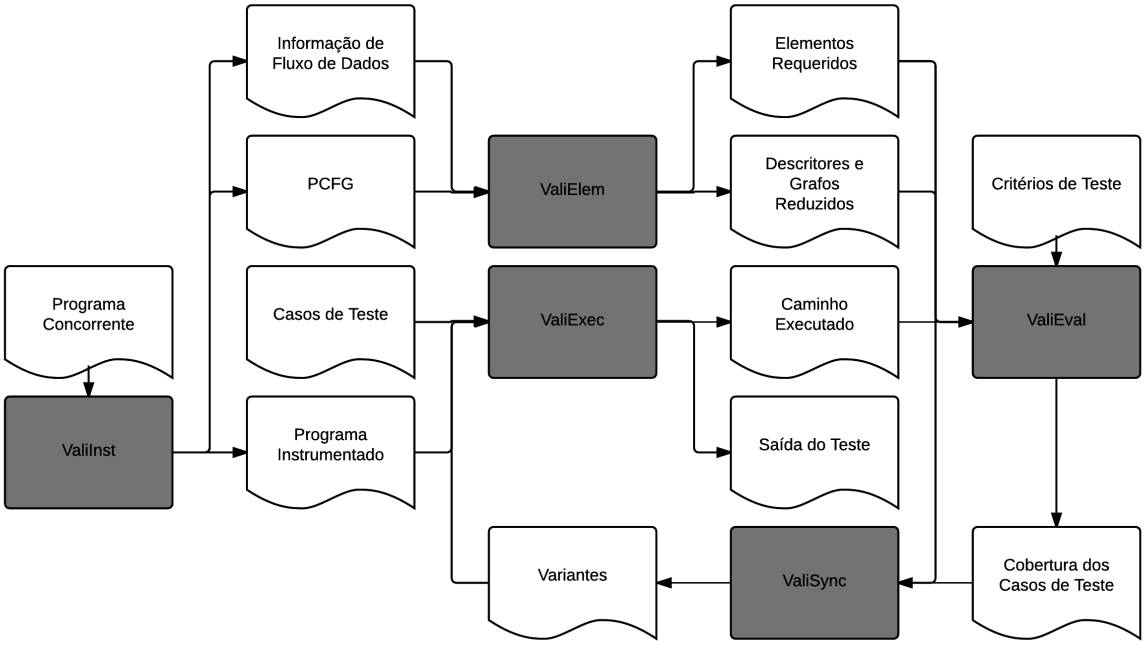


Figura 5 – Arquitetura da ferramenta ValiPar

consiga detectar o compartilhamento de dados entre *threads* (por meio de ponteiros para uma mesma região de memória) de uma forma mais completa.

Por fim, para essa implementação foi criada um formato de especificação de primitivas de sincronização e *clusters*. Com isso, possibilitou-se a separação entre o código da ferramenta e o modo como as primitivas deveriam ser interpretadas. Como consequência direta disso, viabilizou-se a inserção de novas primitivas a serem consideradas pela ferramenta sem que seja necessário o conhecimento de como isso é feito pelo código da mesma.

Apesar das diferenças existentes entre as versões da ferramenta, a arquitetura das mesmas é basicamente constante. Dessa forma, todas possuem módulos inter-relacionados responsáveis pela instrumentação do código-fonte, geração dos elementos requeridos, execução dos casos de teste e avaliação da cobertura dos critérios de teste. Estes módulos são, respectivamente: ValiInst, ValiElem, ValiExec, ValiSync e ValiEval. O relacionamento entre esses módulos pode ser visualizado na Figura 5.

Para se iniciar o processo de teste, submete-se o código-fonte do programa sendo testado para o módulo ValiInst. Tal módulo é responsável em traduzir as características e recursos de concorrência das linguagens suportadas em termos do modelo de teste em questão. Consequentemente, este é o único módulo da ferramenta que lida com as linguagens de programação, tornando todo o processo restante dependente apenas do modelo de teste. Isso trás como benefício o grande potencial de reutilização do código da ferramenta e mostra a ortogonalidade entre o modelo e critérios de teste e as linguagens de programação.

Para se obter tais benefícios, esse módulo realiza a análise do código em questão extraindo elementos como as definições e usos de variáveis (fluxo de dados), os nós e arestas (fluxo de controle) e as primitivas de comunicação e sincronização (fluxo de sincronização) na forma de um PCFG. Além disso, também é feita a instrumentação do código de forma a permitir que o programa produza *traces* (rastros) de execução e que seja possível executá-lo de modo controlado.

Em seguida, dois módulos podem ser executados de modo independente, ValiElem e ValiExec. O módulo ValiElem, tendo como base o PCFG e as informações de fluxo de dados produzidas anteriormente, gera os elementos requeridos da aplicação. Nesse processo são identificadas associações que devem ser cobertas pelos casos de teste a serem executados.

O módulo ValiExec, por sua vez, gerencia a execução dos casos de teste de forma automatizada. Durante a execução de cada caso de teste, *traces* de execução são gerados e devidamente armazenados. Tal execução pode, inclusive, ser feita de forma controlada, em que uma execução prévia é parcial ou totalmente reproduzida.

O módulo ValiEval é utilizado para se avaliar a cobertura de critérios de teste conseguida pelo conjunto de execuções. Tal módulo utiliza os elementos requeridos gerados pelo módulo ValiElem e os *traces* de execução gerados durante a execução do conjunto de casos de teste (ValiExec) para avaliar a cobertura dos mesmos com relação a determinados critérios de teste. No fim do processo é revelada a percentagem atual de cobertura com relação aos critérios escolhidos.

Por fim, o módulo ValiSync pode ser utilizado para se derivar e executar variantes de execução a partir de um conjunto de execuções de casos de teste. Nesse processo, tal módulo gera variantes baseado em *traces* de execução e as executa de forma controlada utilizando-se o módulo ValiExec. Esse processo é realizado de forma iterativa, utilizando os resultados da execução da variante para realimentar o algoritmo de geração. O módulo ValiEval é utilizado para atualizar a cobertura de critérios de teste a cada execução.

A partir desse etapa, o testador pode encerrar a atividade de teste caso a percentagem de cobertura seja satisfatória. Caso contrário, o ciclo é reiniciado a partir da criação e execução de novos casos de teste pelo módulo ValiExec avaliando-se a cobertura resultante de forma iterativa. Ao se modificar o código-fonte para corrigir possíveis defeitos detectados, por exemplo, o processo deve ser reiniciado a partir do módulo ValiInst.

Toda essa atividade de teste possui, no entanto, altos custos computacionais para sua execução. Esse alto custo surge da análise estática feita sobre o código-fonte, onde são gerados vários elementos requeridos que precisam ser cobertos durante a execução dos testes.

Diferentes pesquisas vêm sendo desenvolvidas para minimizar este custo. Em uma abordagem, tenta-se diminuir a geração de elementos requeridos não executáveis, embora não se consiga ainda gerá-los por completo (SOUZA *et al.*, 2011).

Em outra abordagem, está sendo desenvolvida uma versão distribuída da ferramenta

ValiPar. Tal versão distribuída visa melhorar tanto o tempo de resposta na execução dos testes quanto o aproveitamento dos recursos computacionais por meio da distribuição de casos de teste de programas concorrentes potencialmente de grande porte em nós de um conjunto de computadores disponível.

3.7 Considerações Finais

Este capítulo contextualizou a atividade de teste como um todo ressaltando sua importância no ciclo de desenvolvimento de software e o custo associado à sua aplicação.

Nesse sentido, foram expostas as diversas técnicas existentes e o modo como as mesmas buscam a melhoria do processo de desenvolvimento de software em diferentes fases. Além disso, foram apresentadas as diversas estratégias e ferramentas de teste de programas concorrentes existentes. Foram discutidas a existência de defeitos específicos para esse tipo de programa e as técnicas desenvolvidas para o tratamento dos mesmos. O atual cenário no desenvolvimento de ferramentas de teste de programas concorrentes foi exposto ao se apresentar as soluções comerciais e acadêmicas existentes.

Por fim, iniciativas de criação de ontologias de teste de software foram apresentadas, mostrando-se como os conceitos dessa área são classificados e como se relacionam, além da aplicação de ontologias na criação de ferramentas e arquiteturas de referência de teste de software.

CAPÍTULO

4

WEB SERVICES

4.1 Considerações Iniciais

Neste capítulo são abordados os conceitos relacionados à utilização de *Web Services* para a criação de ferramentas de teste. Inicialmente é feita uma contextualização do tema na Seção 4.2. Em seguida, na Seção 4.3, os principais conceitos, usos e as consequências da adoção de *Web Services* para a criação de sistemas são apresentados. Por fim, as possibilidades e benefícios de se utilizar *Web Services* para a atividade de teste são explicadas na Seção 4.4, apresentando as ferramentas Cloud9 e JaBUTiService como estudos de caso.

4.2 Contextualização

Como observado no Capítulo 3, a atividade de teste é tida como uma das mais custosas dentro do ciclo de desenvolvimento de software. De fato, a atividade de teste aplicada manualmente possui um custo muito elevado e é altamente propensa a erros devido ao alto volume de informações relacionadas a serem analisadas e interpretadas conforme são testados programas de porte cada vez maiores (VICENZI *et al.*, 2007).

Para que esta atividade seja viável é necessário que haja sua automação. As ferramentas de teste possuem um papel fundamental nesse sentido auxiliando na automação de todas as etapas da atividade de teste. De modo geral, ferramentas de teste permitem a análise e execução de programas, além da avaliação da cobertura de critérios de forma sistemática e com pouca, ou nenhuma, interação humana.

Como benefícios, a utilização de ferramentas permite a condução do teste de modo

efetivo e ainda incentiva a prática da atividade de teste dentro do ciclo de desenvolvimento de projetos. Por fim, a criação de ferramentas auxilia no desenvolvimento e validação de critérios de teste sendo desenvolvidos (BARBOSA *et al.*, 2007).

Apesar dos benefícios apresentados, uma série de desafios surge a partir do desenvolvimento e utilização de tais ferramentas. Além dos desafios relacionados aos modelos e critérios de teste, abordados no Capítulo 3, pode-se citar desafios operacionais, de usabilidade e de integração ao ciclo de desenvolvimento de software

Os desafios operacionais estão fortemente relacionados aos ambientes a serem suportados pela ferramenta de teste, à manutenção requerida e ao desempenho oferecido. De modo geral, espera-se que haja amplo suporte aos ambientes de desenvolvimento utilizados. Espera-se também que o processo de instalação e atualização (por exemplo) tenha complexidade baixa. Além disso é desejável que tais ferramentas sejam escaláveis com o tamanho do programa sendo testado e que possuam um tempo de resposta baixo. Tais fatores podem ser decisivos para a adoção do teste no processo de desenvolvimento.

Ambientes com grandes volumes de casos de teste a serem executados e grandes quantidades de elementos requeridos a serem cobertos levam em geral ao aumento de infraestrutura e uso de heurísticas para a seleção de casos de teste a serem executados em determinados momentos, o que leva à elevação do custo e, possivelmente, à degradação da atividade como um todo.

Problemas como incompatibilidade de uma ferramenta de teste com a linguagem de programação adotada, bibliotecas, *frameworks* e máquinas virtuais, por outro lado, atrasam e muitas vezes inviabilizam sua utilização, seja em ambientes de pesquisa, de educação ou no contexto da indústria.

Com relação aos desafios de usabilidade, ferramentas devem ser fáceis de se utilizar e, por meio de interfaces, permitir o cumprimento da atividade de teste. Tal requisito implica na incorporação da ferramenta de teste ao processo de desenvolvimento do software e que a mesma apresente interface gráfica (GUI) e programáveis (API) intuitivas e flexíveis. A atenção a requisitos como esse leva a uma atividade de teste mais dinâmica e eficiente permitindo, inclusive, que a ferramenta seja incorporada no desenvolvimento de projetos de formas não previstas em sua criação.

Por fim, com relação ao desafio de integração, é desejável que ferramentas de teste sejam facilmente integradas a outras ferramentas em geral. Atualmente, o ciclo de desenvolvimento de software envolve uma grande variedade de IDEs (*Integrated development environments*) e ferramentas de suporte que auxiliam em tarefas como o teste unitário e de integração, *debugging*, verificação de cobertura, versionamento de código, etc. Dessa forma, a capacidade integração de uma ferramenta de teste com o ambiente em que é utilizada é essencial para sua adoção, visto que, de modo geral, não é utilizada isoladamente.

A utilização de ferramentas que executam testes localmente dificultam, em geral, que tais requisitos sejam cumpridos. Projetos de software de médio e grande porte naturalmente utilizam diversas ferramentas para o apoio no ciclo de desenvolvimento e esperam que novas ferramentas adotadas se integrem da mesma forma. Adicionalmente, fatores como manutenção complexa (instalação e atualização de ferramentas) e tempos de resposta elevados podem degradar todo o desenvolvimento de um projeto.

Considerando este cenário, o conceito de *Web Services*, pode trazer benefícios para o teste de software. Dentre eles está a possibilidade de composição de ferramentas de teste, viabilizando a criação de novas funcionalidades. Além disso, a utilização de serviços torna o teste mais acessível e permite um grande aumento de produtividade no desenvolvimento dos programas devido à facilidade de manutenção e à integração de ferramentas de desenvolvimento e teste. Por fim, uma vez que o acesso às funcionalidades é centralizado, o custo de instalação, atualização e aumento de infraestrutura é muito baixo se comparado com a abordagem tradicional (*Desktop*).

4.3 Web Services

A *World Wide Web* (Web), como um grande sistema distribuído, possui hoje uma abrangência muito maior do que quando foi projetada. De fato, a Web hoje é muito mais do que uma simples aplicação distribuída baseada em documentos (TANENBAUM; STEEN, 2006). É possível observar a crescente adoção de *Web Services*.

Web Services são sistemas projetados para suportar interação entre computadores de forma interoperável por meio de uma rede (W3C, 2004). Essa interação entre serviços se torna possível a partir do desenvolvimento de interfaces de serviços, levando a interação com tais aplicações de um modo mais geral do que é possível por meio de navegadores Web (clientes de uso geral) (COULOURIS *et al.*, 2011). Essa interação é em geral realizada a partir da construção de mensagens padronizadas em uma linguagem de marcação como XML ou JSON e o envio para determinado serviço utilizando o protocolo HTTP (*Hypertext Transfer Protocol*). O serviço por sua vez, decodifica a mensagem recebida, realiza ações referentes ao especificado e, em seguida, responde ao cliente.

Tal abordagem traz inúmeras possibilidades de interação entre sistemas. Como uma consequência direta dessa forma de comunicação, requisita-se apenas que as mensagens comunicadas obedeçam a interface de serviço estabelecida, o que leva a um fraco acoplamento (baixa dependência) entre as partes. Detalhes como linguagem de programação e ambiente de execução são irrelevantes nesse sentido, criando maior interoperabilidade entre sistemas. Outra consequência é a possibilidade de integração entre sistemas e a criação de novos sistemas a partir da composição de sistemas existentes (ALONSO *et al.*, 2004).

Web Services podem ser classificados em duas categorias: os básicos e os compostos. Um *Web Service* que acessa apenas seus dados locais (ou seja, não estabelece comunicação com serviços de terceiros) para fornecer funcionalidades é classificado como básico, enquanto o que depende de outros *Web Services* para completar sua tarefa é classificado como composto (ALONSO *et al.*, 2004).

Dois padrões de composição são amplamente utilizados na criação de serviços compostos: orquestração e coreografia. Tais padrões diferem no modo como o processo é conduzido pelos serviços que integram o serviço composto e na visão que cada serviço da composição possui da operação como um todo.

A orquestração, padrão mais comum, requer um controlador central para coordenar todas as atividades do processo (JOSUTTIS, 2007). Este é o padrão em que nenhum serviço, exceto o controlador central, possui conhecimento do processo como um todo. Uma propriedade interessante derivada desse padrão é a possibilidade de se aplicar o padrão de projeto "*Composite*" em que a composição como um todo (originando um serviço composto) pode ser utilizada como um serviço básico em outras composições.

O padrão coreografia, por sua vez, exige a colaboração das diferentes partes envolvidas, cada uma responsável por uma ou mais etapas do processo. Nesse contexto, não há um controle central para a condução dos serviços e, em geral, nenhum serviço possui conhecimento de todo o processo (JOSUTTIS, 2007). Os serviços envolvidos, assim como em uma coreografia propriamente dita, possuem noções apenas dos serviços adjacentes no processo. Nesse sentido, o controle sobre o processo é passado de serviço para serviço.

Diversos fatores devem ser levados em consideração para permitir a construção de *Web Services* eficientes e interoperáveis. De modo geral, o processo de desenvolvimento de *Web Services* envolve, sobretudo, a escolhas arquiteturais apropriadas para o domínio do problema sendo resolvido.

Adicionalmente, do ponto de vista da composição de serviços, os *Web Services* devem possuir interfaces flexíveis, estáveis e padronizadas, possibilitando o reaproveitamento de recursos e tecnologias. Tal composição deve também ser consistente e robusta, ou seja, deve viabilizar a comunicação apropriada entre os serviços e prever comportamentos em casos de falhas em transações e interações em geral.

4.3.1 Arquitetura de Web Services

A implementação de um *Web Service* envolve diferentes aspectos do sistema. De fato, segundo o W3C (*Worldwide Web Consortium*) há quatro aspectos de grande importância (traduzidos em modelos arquiteturais) na arquitetura de um *Web Service*: as mensagens, os serviços, os recursos e as políticas. Tais aspectos afetam profundamente questões como a interoperabilidade

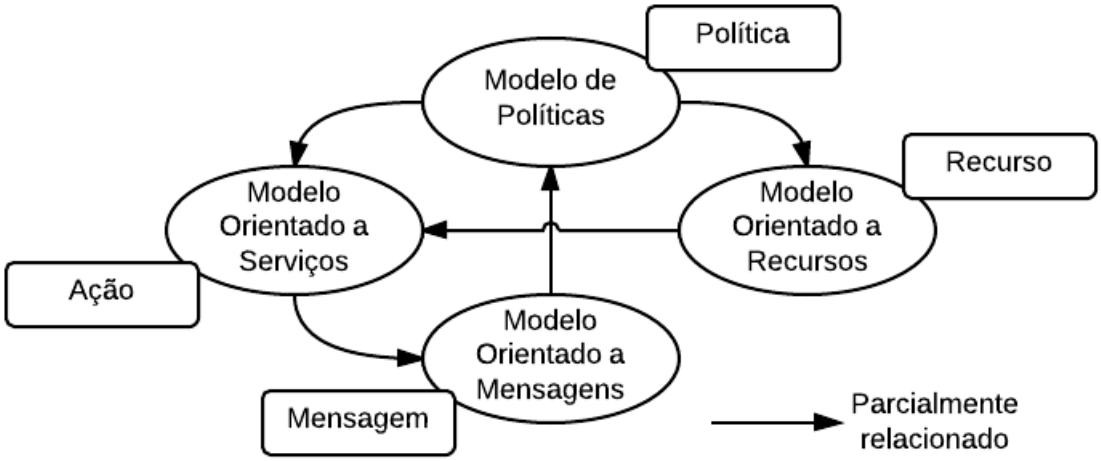


Figura 6 – Representação do relacionamento entre os modelos na arquitetura de um *Web Service*. Baseado na especificação em (CONSORTIUM, 2004).

do *Web Service* (CONSORTIUM, 2004).

A Figura 6 caracteriza o relacionamento entre os modelos. Como pode ser observado, o modelo de mensagens depende do modelo de políticas. De fato, em um sistema pode haver políticas sobre o transporte de mensagem que restringem de alguma forma o transporte de mensagem. O modelo de serviços, por sua vez, é dependente do modelo de mensagens uma vez que não há como oferecer um serviço sem um meio de comunicação (transporte de mensagens) entre as partes.

O modelo de recursos se relaciona com o de serviços principalmente pelo fato de que um *Web Service* pode ser caracterizado como um recurso dentro de um sistema. Por fim, o modelo de políticas está relacionado com o de serviços e o de recursos uma vez que restrições ou regras são aplicadas igualmente a serviços e a recursos.

O primeiro aspecto a ser considerado é o das mensagens do serviço. Como pode ser observado na Figura 7A, o modelo orientado a mensagens é focado na manipulação de mensagens. Não há preocupações a respeito do significado semântico do conteúdo da mensagem e seu relacionamento com outras mensagens. Ao invés disso, há o foco em características como seu processamento, sua estrutura e o modo como a mensagem será transportada. Assim, uma mensagem, composta por um cabeçalho e um corpo, é originada por um agente (um *Web Service*) e transportada para outro agente, o qual a consome.

O aspecto de serviços, detalhado na Figura 7B, é o mais complexo. Seu foco está principalmente em como os serviços são descritos e como ações são disponibilizadas pelo *Web Service*. Os serviços têm forte relação com as mensagens, mas possuem foco na ação causada pela mensagem ao invés da mensagem em si. O propósito central desse modelo é, assim, descrever o

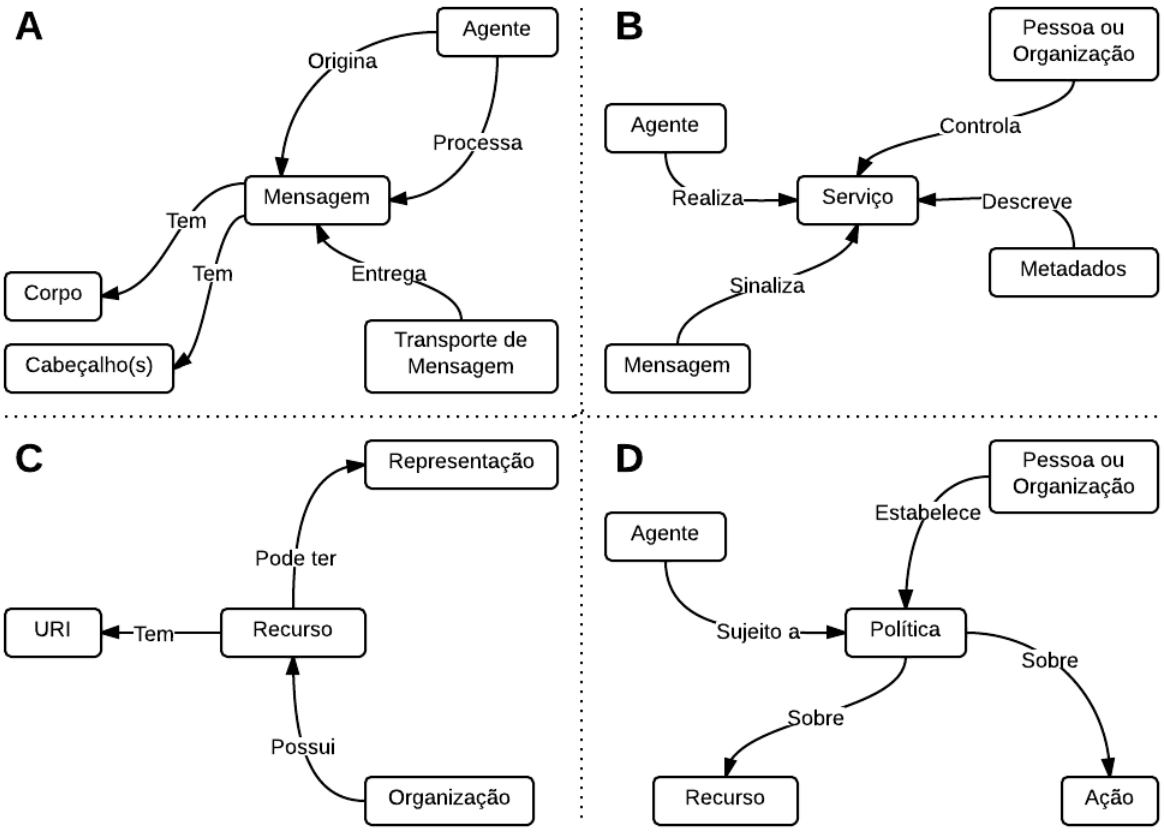


Figura 7 – Representação detalhada dos modelos associados à arquitetura de *Web Services*. Baseado na especificação em (CONSORTIUM, 2004).

relacionamento entre os agentes e o serviços fornecidos. Dessa forma, um serviço é realizado por um agente, controlado por uma pessoa ou organização e descrito por metadados. Além disso, uma mensagem sinaliza a necessidade de se realizar o serviço, ou seja, dispara uma ação.

O aspecto de recursos, por sua vez, representados na Figura 7C, possui foco na manipulação de recursos. Nesse sentido, há uma preocupação quanto à propriedade e às políticas associadas a cada recurso. Há também o foco na representação e na endereçabilidade, com a atribuição de URIs (*Unified Resource Identifier*), de recursos para que o mesmo possa ser identificado e utilizado. Um recurso é, ainda, propriedade de alguma organização.

Por fim, deve-se também desenvolver políticas que governem o *Web Service*, especificado na Figura 7D. O foco, nesse sentido, está nos comportamentos a serem restringidos ou permitidos com relação ao acesso a recursos e ações pelos agentes. Tais políticas são estabelecidas pelo dono do serviço para implementar questões de segurança, de qualidade de serviços e de gerenciamento relativas a recursos e ações dentro do sistema.

Tais aspectos estão, mesmo que em diferentes níveis, presentes em qualquer *Web Service*. De fato, esses aspectos são explicitamente tratados ao se escolher as tecnologias e estilos

arquiteturais para compor um serviço. Ao se escolher o protocolo de comunicação, por exemplo, são decididos aspectos de mensagem da arquitetura do serviço. Da mesma forma, ao se escolher entre os diferentes estilos arquiteturais existentes (detalhados na Seção 4.3.2), decide-se a forma como os recursos serão disponibilizados e como serão as políticas e as interfaces de acesso aos mesmos.

4.3.2 Estilos Arquiteturais

O planejamento de uma arquitetura de *Web Service* e as possibilidades de composição de serviços são fortemente influenciados pelo estilo arquitetural adotado. De fato, estilos arquiteturais diferentes potencialmente exigem escolhas diferenciadas quanto ao transporte de mensagens, por exemplo. Além disso, os estilos arquiteturais expõe serviços de acordo com diferentes perspectivas e com diferentes propósitos. A adoção de estilos arquiteturais, sobretudo, está intimamente ligada ao domínio do problema sendo resolvido.

A construção de *Web Services* pode, basicamente, seguir dois estilos arquiteturais bem definidos: o estilo RPC (*Remote Procedure Call*) e o estilo orientado a recursos RESTful (RICHARDSON; RUBY, 2007). A partir desses dois estilos, pode-se ainda ter serviços que possuam e se beneficiam de características de ambos. Esses serviços são categorizados como sendo híbrido REST-RPC.

Um *Web Service* é considerado RESTful quando o mesmo segue os critérios estabelecidos no estilo arquitetural REST (*Representational State Transfer*) (FIELDING, 2000). Tal estilo arquitetural, por sua vez, consiste de uma série de restrições quanto ao modo como os recursos são expostos. De uma forma mais abrangente, *Web Services* RESTful se comportam como o restante da Web: orientado à manipulação de recursos (ou documentos) utilizando-se o protocolo HTTP.

A utilização do protocolo HTTP, apesar de ser amplamente utilizado, não é um requisito do estilo arquitetural em questão. Por meio do protocolo HTTP, quatro principais métodos, com semânticas diferentes, são utilizados para se especificar a ação sobre o recurso: *GET*, *POST*, *PUT*, *DELETE*. A informação no nível da aplicação sendo desenvolvida, por sua vez, é especificada na URI endereçada ao serviço, no cabeçalho e no corpo do pacote HTTP sendo transmitido.

O método *GET* possui a semântica de recuperar informações, analogamente *DELETE* possui o propósito de remover o recurso do serviço. O método *PUT*, por sua vez, serve para adicionar ou modificar um recurso identificado (caso o recurso identificado não exista, o mesmo é adicionado). Por fim, o método *POST* tem a semântica de criação de recurso subordinado a outro recurso (por exemplo, adicionar um item a um carrinho de compras identificado).

Um exemplo de interação com um serviço RESTful pode ser observado no exemplo de requisição dos itens de um carrinho de compra em um serviço de uma loja. Inicialmente, como

```
1 GET /v1/carrinhos/1 HTTP/1.1
2 Host: api.loja.com.br
3 Accept: application/xml
```

Código-fonte 4: Exemplo de requisição a um serviço RESTful

```
1 HTTP/1.1 200 OK
2 Content-Type: application/xml; charset=UTF-8
3 Content-Length: 131
4 Connection: close
5 <carrinho> <item>
6     <id> 1 </id> <nome> Item 1 </nome>
7     <preco> 10.00 </preco> <quantidade> 100 </quantidade>
8 </item> </carrinho>
```

Código-fonte 5: Exemplo de resposta de um serviço RESTful

pode ser observado no Código 4, são requisitadas informações sobre os itens de um carrinho de compras em um serviço de uma loja *online* utilizando-se o método *GET*. As informações da aplicação (a identificação do carrinho) são especificadas na URI da mensagem. Como resposta (Código 5) é entregue um documento XML contendo informações sobre os itens no carrinho de compras, além de metadados como o status da requisição (200 OK) e o tipo de documento (XML).

Esta abordagem de interfaces genéricas e mínimas contribui para o desenvolvimento de serviços fracamente acoplados, reduzindo a dependência e necessidade de se ter operações específicas com nomes fixos. Como consequência direta, os dados (ou recursos) se tornam mais importantes que as operações e interfaces em si (COULOURIS *et al.*, 2011). Além disso, a referenciação de recursos pela sua URI propicia a exploração do serviço de forma similar a que ocorre com a Web tradicional.

Como exemplo de *Web Services* RESTful, pode-se citar o *Amazon's Simple Storage Service (S3)* o qual possibilita o armazenamento de arquivos utilizando-se os métodos disponíveis no protocolo HTTP para sua manipulação. Além disso, *Web Services* que possibilitem apenas a leitura de dados e que não usem SOAP, websites estáticos e aplicações Web "somente leitura" como os motores de busca são, em geral, caracterizados como serviços RESTful (RICHARDSON; RUBY, 2007).

Web Services que possuem o estilo arquitetural RPC seguem a dinâmica original da abstração RPC. Nesse sentido, esse estilo é basicamente orientado a ações, ou seja, a interpretação da mensagem transferida entre o cliente e o servidor resulta na chamada de procedimentos.

A interface neste estilo arquitetural é baseada em envelope de dados. Assim, um serviço aceita envelopes de seus clientes e, como resposta, retorna novos envelopes. Toda a informação necessária para a requisição do serviço está contida no envelope de dados, ou seja, tanto as informações a respeito da ação (ou procedimento) a ser executada quanto a respeito dos parâ-

metros e outras informações compõem o corpo do envelope a ser transmitido. Este envelope é necessariamente processável pelo computador o qual, com base nos metadados apresentados, transporta a mensagem e realiza a ação apropriada.

Os protocolos HTTP e o SOAP (*Simple Object Access Protocol*) (CONSORTIUM, 2007) são usualmente utilizados para a transmissão de mensagens. No caso do SOAP, tais dados são transmitidos dentro de um envelope SOAP especificado na linguagem de marcação XML que, por sua vez, é transportado utilizando-se protocolos na camada de aplicação ou de transporte como o HTTP, SMTP, TCP e UDP. No protocolo HTTP, em específico, esse transporte é realizado por meio do método *POST* mesmo que o envelope não tenha como objetivo a criação de um recurso, propósito original no método. No contexto de serviços RPC, o método POST é utilizado por permitir o transporte de dados arbitrários no corpo do pacote HTTP.

Essa organização imposta pelo protocolo SOAP possibilita uma grande flexibilidade na criação da interface de comunicação. De fato, diferentemente dos *Web Services* RESTful, que possuem basicamente quatro métodos para interagir com recursos, os *Web Services* RPC definem vocabulários específicos para cada aplicação do mesmo modo como se definem procedimentos diferentes para aplicações distintas.

Além disso, o transporte de mensagens utilizando-se apenas o envelope SOAP (especificadas em XML) tornam o modo como a mensagem é transmitida transparente para o desenvolvedor, ou seja, viabiliza a utilização de protocolos para o transporte de forma independente da informação sendo transmitida.

Como desvantagem, no entanto, há uma grande sobrecarga na geração e no transporte de mensagens originada na geração e decodificação dos envelopes, além do tamanho do envelope resultante. Isso muitas vezes reduz a eficiência da comunicação entre as partes, principalmente quando se trata de serviços menos complexos.

Um exemplo de utilização do protocolo SOAP, similar aos exemplos apresentados nos Códigos 4 e 5, para a comunicação com um serviço com estilo arquitetural RPC pode ser observado na requisição (Códigos 6) e resposta (Código 7). A requisição pelas informações (Código 6) é feita por meio de um envelope na linha 6 (*tag soap:Envelope*) que contém o método (*GetCarrinho*) e seus parâmetros (*IDUsuario*) na linha 8 e a resposta (Código 7) é entregue também por meio de um envelope SOAP contendo os dados requisitados nas linhas 7 a 10.

O protocolo SOAP em si não especifica a semântica e o formato das mensagens sendo transportadas de acordo com o requisitado pelo serviço. De fato, este é um protocolo encarregado apenas da transferência e acesso a informações estruturadas, entre clientes e serviços. A descrição das funcionalidades, dessa forma, é usualmente feita por meio da linguagem de descrição de *Web Services* chamada WSDL (*Web Services Description Language*).

Tal formato possibilita a construção de especificações em XML processáveis pelo computador. Dessa forma, um programa cliente pode determinar as operações possíveis para um

```
POST /carrinhos HTTP/1.1
Host: api.loja.com.br
Content-Type: application/soap+xml; charset=utf-8
Content-Length: 123
<?xml version="1.0"?>
<soap:Envelope soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding"
    >
  <soap:Body>
    <GetCarrinho> <IDUsuario>1</IDUsuario> </GetCarrinho>
  </soap:Body>
</soap:Envelope>
```

Código-fonte 6: Exemplo de requisição SOAP

```
HTTP/1.1 200 OK
Content-Type: application/soap+xml; charset=utf-8
Content-Length: 123
<?xml version="1.0"?>
<soap:Envelope soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding"
    >
  <soap:Body>
    <carrinho> <item>
      <id> 1 </id> <nome> Item 1 </nome>
      <preco> 10.00 </preco> <quantidade> 100 </quantidade>
    <item> </carrinho>
  </soap:Body>
</soap:Envelope>
```

Código-fonte 7: Exemplo de resposta SOAP

determinado serviço e se comunicar efetivamente com o mesmo utilizando um protocolo adequando, como o SOAP, a partir da especificação do mesmo.

4.4 *Web Services* para o Teste de Software

Pesquisas têm sido realizadas no sentido de possibilitar que a atividade de teste se torne mais prática e eficaz por meio de sua disponibilização como serviço na Web. Como exemplo de ferramenta neste contexto pode-se citar a ferramenta *Cloud9* (CIORTEA *et al.*, 2010) (detalhada na Seção 4.4.1) e JaBUTiService (ELER *et al.*, 2009) (detalhada na Seção 4.4.4), as quais possibilitam a atividade de teste sobre códigos sequenciais utilizando modelos desenvolvidos pelas respectivas instituições de pesquisa como serviços na Web.

Tais exemplos de ferramentas demonstram a viabilidade de se utilizar *Web Services* no desenvolvimento de ferramentas de teste, apesar de dificuldades técnicas encontradas. Os benefícios dessa abordagem são muitas vezes imprescindíveis para que programas de médio e grande porte sejam testados com a eficiência esperada e sejam amplamente acessíveis para testadores.

Os *Web Services* analisados, embora atendam as expectativas ao se utilizar os modelos propostos, não abordam especificamente programas concorrentes, o que caracteriza a falta de ferramentas que analisem a qualidade de programas desse tipo, usufruindo das vantagens proporcionadas pela abordagem de *Web Services*. Além disso, nota-se ainda a carência de pesquisas acerca da integração entre ferramentas, ponto importante para o reaproveitamento de recursos e composição de ferramentas para se criar novas funcionalidades.

De um modo geral, o serviço JaBUTiService caracteriza interfaces de serviços que possibilitam a interação remota com a ferramenta. No entanto, a interação com outras ferramentas de forma genérica é limitada à interface imposta e não contempla partes importantes da atividade. Além disso são notados problemas de desempenho que prejudicam a atividade como um todo. O serviço Cloud9, apesar de dar grande enfoque no tempo de resposta do teste e na utilização apropriada de recursos, não dá ênfase na interfaces de serviço, o que potencialmente dificulta sua inclusão no ciclo de desenvolvimento de projetos.

4.4.1 Ferramenta Cloud9

O serviço *Cloud9* (CIORTEA *et al.*, 2010) aplica a técnica de execução simbólica de programas na atividade de teste. Nesta técnica, ao invés de executar o programa com as entradas de dados da forma tradicional, o *engine* (motor) de execução da ferramenta executa o programa com entradas simbólicas, as quais assumem diferentes valores para explorar os diferentes desvios condicionais do programa. Essa atribuição de valores ocorre em tempo de execução e gera um grande número de réplicas de programas (uma para explorar a condição verdadeira e uma para explorar a condição em um laço de repetição, por exemplo).

Essa abordagem possui grande potencial para a automação de teste e potencial em revelar defeitos ao explorar todas os valores necessários para a cobertura de nós e arestas em um programa. Porém, como condição direta, uma grande quantidade de recursos (como memória e processamento) deve ser garantido para que a mesma seja factível. De fato, em um computador *Desktop*, tal atividade seria dificultada devido aos limites de recursos e, para programas de médio e grande porte, a mesma se torna infactível devido à quantidade exponencial de réplicas geradas (CIORTEA *et al.*, 2010).

Para possibilitar a ampla utilização da técnica de forma geral, tal ferramenta foi implementada como um *web service* a ser executada em ambiente de *cluster*. A arquitetura desenvolvida para a ferramenta é fortemente voltada para o balanceamento de carga. Uma vez que uma quantidade crescente de réplicas devem ser executadas, deve haver uma forma eficiente de distribuição das mesmas entre os nós de um *cluster* assim como formas de coletar os resultados.

Resultados (CIORTEA *et al.*, 2010) demonstram ganhos notáveis de desempenho ao se comparar tal ferramenta em execução em uma infraestrutura de computação em nuvem com a

ferramenta equivalente utilizando um único nó como recurso computacional (chamada Klee). O *benchmark* foi baseado, assim, no tempo obtido pela ferramenta Klee dividido pelo tempo obtido pelo Cloud9 na execução de 32 programas que incluem, dentre outros tipos, ferramentas comumente utilizadas em sistemas Unix. O *speedup* resultante foi, em média, de 47, ou seja, a ferramenta Klee demora, em média, 47 vezes mais para cumprir a mesma tarefa.

4.4.2 Ferramenta PathCrawler

O serviço PathCrawler é uma ferramenta para geração caso de teste baseada em execução simbólica dinâmica para garantir cobertura para critérios de teste estrutural para programas em C (WILLIAMS *et al.*, 2005).

A interação com este serviço consiste no envio de uma requisição que inclui artefatos como um conjunto de arquivos de código-fonte em C e parâmetros como pré-condições e estratégias de geração de caso de teste. Como resposta, o usuário recebe estatísticas e informação sobre a cobertura da sessão de teste, além de um conjunto de casos de teste e detalhes da exploração de caminhos.

Os autores da ferramenta (WILLIAMS *et al.*, 2005) descrevem experiências e preocupações relacionadas com desempenho, modelo de negócios, segurança na execução, confidencialidade e a utilização da ferramenta para o ensino. Segundo os autores, um dos maiores benefícios com relação a desempenho está na possibilidade de se paralelizar cada sessão de teste utilizando uma infraestrutura de computação em nuvem.

Com relação a execução e segurança, os autores apontam a necessidade de isolamento da execução do programa utilizando estratégias como utilização de maquinas virtuais, restrição de recursos para mitigar possíveis ataques como saturação de memória, multipla criação de threads e ataques vindo pela rede. São também destacadas analises que possibilitam rejeição de programas com código possivelmente malicioso.

A questão da confidencialidade é destacada como um dos principais fatores para se ter a adoção da indústria. Para alcançar tal público, há a necessidade de se utilizar criptografia, anonimação e obfuscação para garantir a segurança dos artefatos de teste do cliente, que são frequentemente confidenciais.

Por fim, a questão de modelo de negócios (*pricing*) é discutida em termos de cobrança pela utilização de recursos computacionais (tempo de CPU, por exemplo). Os autores também destacam a possibilidade de se cobrar por teste realizado.

4.4.3 Ferramenta PascalMutants Service

O serviço PascalMutants Service (OLIVEIRA, 2011) foi desenvolvida a partir da fer-

ramenta PascalMutants (OLIVEIRA, 2010) como validação da RefTEST-SOA (OLIVEIRA; NAKAGAWA, 2011), uma arquitetura de referencia para ferramentas de teste utilizando arquitetura orientada a serviços (SOA). Tal ferramenta realiza o teste baseado em mutantes de programas implementados em Pascal (WAKERLY, 1979).

PascalMutants Service é composta por quatro serviços (em conjunto com um serviço orquestrador) que tratam de aspectos específicos do teste baseado em mutantes. Foram instanciados serviços de critério de teste, artefatos de teste, requisito de teste e caso de teste. Tais serviços foram implementados utilizando guiados por uma arquitetura SOA visando reúso.

O serviço *Mutation Test Pascal Artifact* (MTPA) gerencia artefatos relacionados a análise de mutantes a partir do do código-fonte sendo testado. O serviço *Mutation Test Pascal Requirements* (MTPR) possui o papel de gerar mutantes, compilá-los e executar casos de teste. O serviço *Mutation Test Criteria* (MTC) possui a funcionalidade de verificar o número de mutantes vivos, mortos e equivalentes. Por fim, o serviço *Test Case Management* gerencia o conjunto de casos de teste. Todos esses serviços são orquestrados pelo serviço PascalMutants Service que guia a atividade de teste, atribuindo tarefas aos serviços específicos.

Cada serviço implementado possui contratos com uma granularidade que permite o gerenciamento de projetos e consulta de artefatos de teste. Segundo os autores, os serviços resultantes possuem grande capacidade de reusabilidade em contextos de teste baseados em mutantes em outras linguagens. Certos componentes como o serviço de caso de teste podem também ser utilizados em outros tipos de teste por ser um serviço independente de técnica de teste.

4.4.4 Ferramenta JABUTiService

A ferramenta JaBUTiService (ELER *et al.*, 2009) foi desenvolvida tendo-se como base a ferramenta *JaBUTi* (Java *Bytecode Understanding and Testing*) (VINCENZI *et al.*, 2005), uma ferramenta *Desktop* que possibilita o teste estrutural de unidade e de integração de programas em Java. Tal ferramenta foi desenvolvida como prova de conceito a respeito da viabilidade de se aplicar a atividade de teste utilizando-se ferramentas de teste como serviço.

Na construção da ferramenta JaBUTiService foi realizada uma análise detalhada da interação entre o usuário e a ferramenta JaBUTi em si, a fim de derivar as principais operações que devem existir para que não haja prejuízos na forma como o teste é feito. Além disso, foi feita a identificação e separação do "núcleo" da ferramenta *Desktop*(descartando aspectos relacionados com interface gráfica) para que a mesma pudesse ser utilizado pelo serviço.

Todas as operações possíveis para a realização do teste por meio da ferramenta *Desktop* gráfica foram disponibilizadas com sucesso no serviço como uma interface de comunicação (API), o que possibilita a integração entre serviços e sua utilização em diferentes atividades de

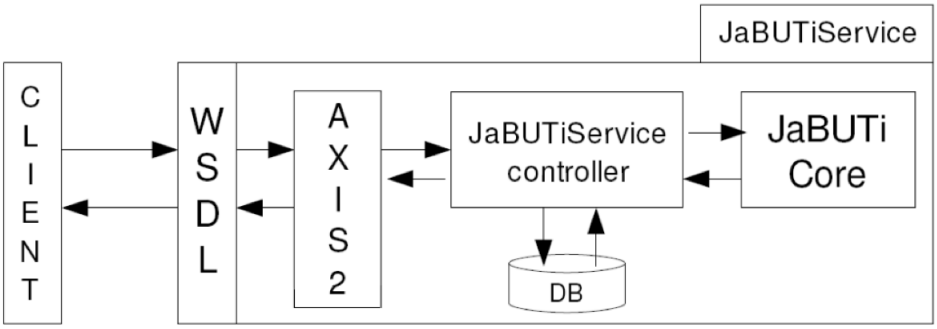


Figura 8 – Componentes do serviço JaBUTiService (ELER *et al.*, 2009).

teste.

O serviço resultante é composto por 4 componentes apresentados na Figura 8: o *engine* (ou motor) de *parsing* e construção das mensagens XML para a comunicação entre cliente e servidor (AXIS2), o controlador JaBUTiService (*JaBUTiService controller*), o qual implementa as operações disponíveis, a base de dados (DB) para armazenamento de códigos e casos de teste e o núcleo da ferramenta JaBUTi (*JaBUTi Core*) em si, que provê meios de instrumentar e analisar cobertura de código.

Como resultado dessa implementação, tem-se um serviço Web *stateful* (em que há o armazenamento do estado do cliente no serviço entre requisições) que disponibiliza 16 operações de baixo nível que possibilitam a construção de operações de alto nível. Dentre as operações, podem-se destacar: *addTestCase*, *getGraph* e *createProject*. Elas possuem equivalente na versão *Desktop* e possibilitam o gerenciamento de projeto com maior grau de liberdade.

Por ser um *Web Service* de natureza *stateful* é também fornecida a sequência de passos que deve ser rigorosamente seguida para completar a atividade. Essa sequência é disponibilizada na forma de uma máquina de estados. Não é permitido por exemplo que seja chamada uma operação *getGraph* sem que tenha sido invocada a operação *createProject*, por exemplo.

Devido ao fato de, muitas vezes o software em questão depender de bancos de dados específicos ou de um processo de compilação diferente, a execução dos casos de teste ocorre localmente. Dessa forma, entre as operações disponível estão a *getInstrumentedProject* que disponibiliza o programa instrumentado para o teste e *sendTraceFile* que permite o envio dos arquivos gerados pela execução local. Tal ponto, apesar de ser compreensivo nesse contexto, pode inviabilizar a integração com outros serviços uma vez que, intuitivamente, espera-se que todo o processo ocorra remotamente.

Para a validação desse serviço, explicada em (ELER *et al.*, 2009), escolheu-se como programas de teste certos componentes da biblioteca Apache Commons BeanUtils (64 classes e 478 métodos) e planejaram-se 243 casos de teste. A partir desse cenário foi criado um *script* que utiliza a interface do serviço e o mesmo foi executado, coletando-se no fim a cobertura do teste.

Como resultado foi obtido um tempo de resposta do teste maior à versão equivalente *Desktop*. Essa perda de desempenho pode ser atribuída ao processamento de XML e latência da rede, fatores intrínsecos dessa abordagem. Além disso, consideram-se como possíveis fatores a questão da segurança e autenticação no processo de teste por esse meio.

4.5 Considerações Finais

Este capítulo abordou conceitos fundamentais para a utilização, desenvolvimento e implementação de *Web Services* em três níveis. Inicialmente, a utilização de *Web Services* no contexto atual foi exposta. Foram apresentados, também, aspectos arquiteturais internos de *Web Services* e as possibilidades de interação entre serviços trazidas pelos diferentes estilos arquiteturais. Em seguida, a composição de serviços para a implementação de novas funcionalidades foi descrita tendo-se em vista os padrões de composição e os principais aspectos relacionados com a sua modelagem.

Com relação à aplicação de *Web Services* no contexto do teste foram apresentadas abordagens de criação de ferramentas de teste como serviços na Web. De acordo com as pesquisas realizadas, trabalhos relativos à construção de ferramentas de teste de programas concorrentes como serviços são escassos e, de modo geral, não é dado foco a questões como reuso e composição no desenvolvimento de serviços de teste, questões essas abordadas neste projeto.

CAPÍTULO

5

VALIPAR SERVICE

5.1 Considerações Iniciais

Este capítulo descreve o processo de desenvolvimento de serviços para o teste estrutural de programas concorrentes. Inicialmente foi feito um estudo sobre as ontologias de teste (Seção 5.2). A ontologia selecionada foi utilizada para a identificação da atividade de teste estrutural de programas concorrentes (Seção 5.3). A partir dos elementos identificados foi feita a definição de capacidades e contratos de serviços (Seções 5.4 e 5.5). Os serviços foram implementados seguindo uma estratégia que promove a flexibilidade (Seção 5.6) resultando em um conjunto de serviços para o teste em questão (Seção 5.7).

5.2 Estudo de Ontologias de Teste de Software

O desenvolvimento de software exige, de modo geral, o conhecimento do domínio da aplicação para que cumpra com os requisitos levantados. A falta de abrangência e estrutura no desenvolvimento do software pode levar à incompatibilidade com outros programas e à dificuldade de integrá-lo nos devidos contextos.

Ontologias podem ser utilizadas como ferramentas para identificar o domínio da aplicação de modo estruturado, mitigando problemas de abrangência e integração. Uma ontologia é definida como uma especificação formal de conceitos compartilhados em um determinado domínio, incluindo o vocabulário e afirmações lógicas a respeito do relacionamento dos mesmos (USCHOLD; GRUNINGER, 1996; GRUBER, 1995). Por meio de uma ontologia pode-se estabelecer tanto um vocabulário para comunicar e representar o conhecimento quanto relacionar

conceitos de uma forma sistemática e padronizada (SOUZA; FALBO; VIJAYKUMAR, 2013).

O teste de software, no entanto, possui um domínio grande e complexo, tornando o estabelecimento de ontologias neste contexto um grande desafio (SOUZA; FALBO; VIJAYKUMAR, 2013). Dentre os principais problemas existentes na criação de ontologias para o teste está a falta de uniformidade no vocabulário utilizado pela comunidade de teste, isto é, autores frequentemente associam um mesmo termo a conceitos distintos.

Como consequência, muitas das ontologias desenvolvidas possuem cobertura limitada, ou seja, envolvem apenas partes da atividade. Segundo SOUZA; FALBO; VIJAYKUMAR, as ontologias TaaS (YU *et al.*, 2009), STOWS (*Software Testing Ontology for Web Services*) (HUO; ZHU; GREENWOOD, 2003) e OntoTest (BARBOSA; NAKAGAWA; MALDONADO, 2006; BARBOSA *et al.*, 2008) são as que possuem maior cobertura do domínio de teste.

A ontologia STOWS (HUO; ZHU; GREENWOOD, 2003) é utilizada em um ambiente de software baseado em agentes para o teste de aplicações Web. Essa ontologia separa os conceitos de teste entre conceitos básicos e conceitos compostos. Os conceitos básicos são divididos em 5 tipos: testador, contexto (como fases de teste), método empregado, artefatos e ambiente. Enquanto isso, os conceitos compostos são definidos com base nos conceitos básicos. A capacidade de um testador, por exemplo é composto pelas atividades que ele pode executar, pelo contexto, método e ambiente em que executará a atividade, além dos artefatos necessários.

A ontologia TaaS (YU *et al.*, 2009) foi desenvolvida para possibilitar composição automática de serviços de teste no contexto de um *framework* para a criação de plataformas de "teste como um serviço" (TaaS - *Test as a Service*). Nessa ontologia, são definidos conceitos principais como a tarefa de teste e a capacidade de teste. Uma tarefa é composta por restrições e recursos e possui uma atividade, composta pela execução, análise e planejamento. A capacidade, por sua vez, é composta pelo programa testado, qualidade de serviço e por serviços de teste.

A ontologia OntoTest (detalhada na Seção 5.3) descreve a atividade de teste utilizando um vocabulário baseado no padrão ISO/IEC 12207 (BARBOSA; NAKAGAWA; MALDONADO, 2006; BARBOSA *et al.*, 2008). A ontologia é modularizada em dois níveis: a ontologia principal, que define a atividade de teste de modo geral, e as sub-ontologias, que detalham cada termo definido. Os conceitos definidos pela OntoTest foram empregados no processo de definição da arquitetura de referência para ferramentas de teste RefTEST (*Reference Architecture for Software Testing Tools*) (NAKAGAWA *et al.*, 2007; NAKAGAWA; BARBOSA; MALDONADO, 2009) e no desenvolvimento de uma arquitetura de referência para ferramentas de teste baseadas em SOA (RefTEST-SOA) (OLIVEIRA; NAKAGAWA, 2011). No contexto de SOA, a arquitetura de referência foi avaliada utilizando serviços de teste baseado em mutantes.

Apesar da RefTEST-SOA tratar de ferramentas de teste orientadas a serviço, não existe sobreposição com este projeto de mestrado. A RefTEST-SOA define as camadas da arquitetura orientada a serviços para serviços de teste (como camada de processo de negócio, intermediação,

apresentação, aplicação e persistência) e os serviços primários e ortogonais de modo generalizado. Em contrapartida, este trabalho define os serviços primários para o teste estrutural de programas concorrentes e quais devem ser os contratos dos mesmos, considerando uma atividade de teste efetiva.

O desenvolvimento de serviços para o teste estrutural de programas concorrentes adotou a ontologia OntoTest para a definição de um vocabulário e do relacionamento entre conceitos. Os critérios de seleção foram grau de cobertura oferecido e padronização do vocabulário utilizado. De fato, a OntoTest aborda os elementos presentes no teste estrutural de programas concorrentes de forma mais completa e a insere em um contexto mais amplo da atividade de teste de software. Além disso, o fato de utilizar uma terminologia padronizada facilita a comunicação de conceitos e a adoção da ferramenta.

5.3 Identificação da Atividade de Teste Estrutural de Programas Concorrentes

A identificação da atividade de teste estrutural de programas concorrentes ocorreu a partir da especialização da OntoTest. Inicialmente foram selecionadas as partes da ontologia que se relacionam diretamente com ferramentas de teste. Este procedimento visou o mapeamento direto com a ontologia para aumentar a compatibilidade dos serviços implementados com o ciclo de teste de software. Os elementos da atividade de teste estrutural de programas concorrentes foram, assim, mapeados utilizando-se o vocabulário e relacionamentos estabelecidos previamente pela OntoTest.

A ontologia principal da OntoTest (Figura 9) define os conceitos de processo, fase, artefato, etapas, recursos, procedimentos e estratégias de teste. Um processo de teste é definido de acordo com o paradigma de desenvolvimento e modelos de ciclo de vida do teste e é composto por uma sequência de passos de teste (*Testing steps*) e pelas dependências entre eles. Um passo de teste é composto por atividades de teste (primárias, organizacionais e de suporte), interage com outros passos de teste e auxilia na condução do procedimento ou estratégia de teste. Esse conceito pode depender de artefatos de teste, produzir artefatos e também pode utilizar recursos de teste.

Procedimentos e estratégias de teste são adequados a processos de teste, utilizam artefatos, podem ser suportados por um recurso de teste. Os procedimentos e estratégias variam de acordo com os artefatos em teste e dependem do modo como a fase de teste, abordagem e técnicas de teste são combinados. Os conceitos de recurso, artefato, passo e estratégia, e procedimento de teste estão detalhados em (BARBOSA; NAKAGAWA; MALDONADO, 2006; BARBOSA *et al.*, 2008; BORGES; BARBOSA, 2009).

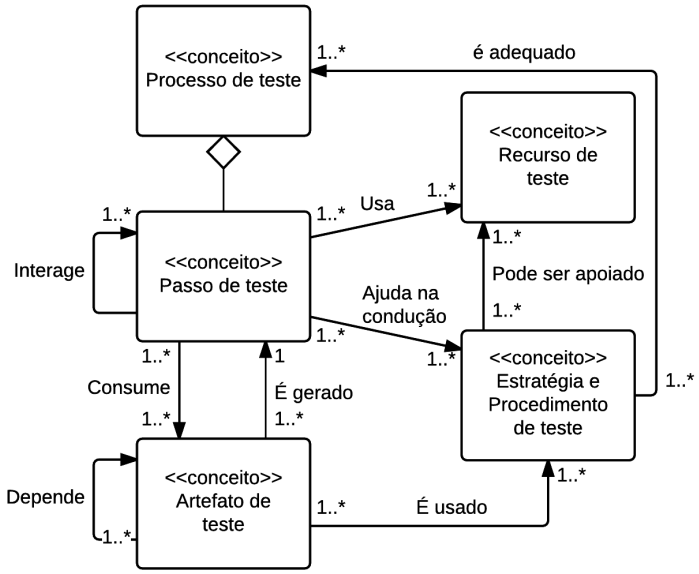


Figura 9 – Representação da Ontologia de Teste OntoTest (BARBOSA *et al.*, 2008) em UML

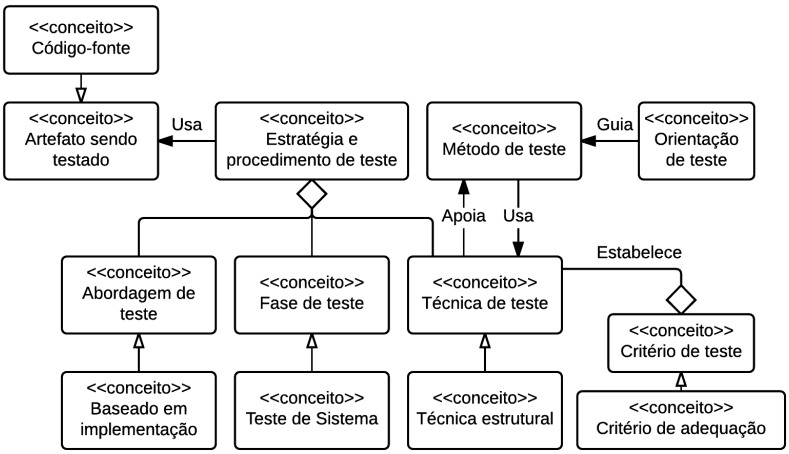


Figura 10 – Especializacão da sub-ontologia de procedimento e estratégia de teste OntoTest (BORGES; BARBOSA, 2009) em UML para o teste estrutural de programas concorrentes

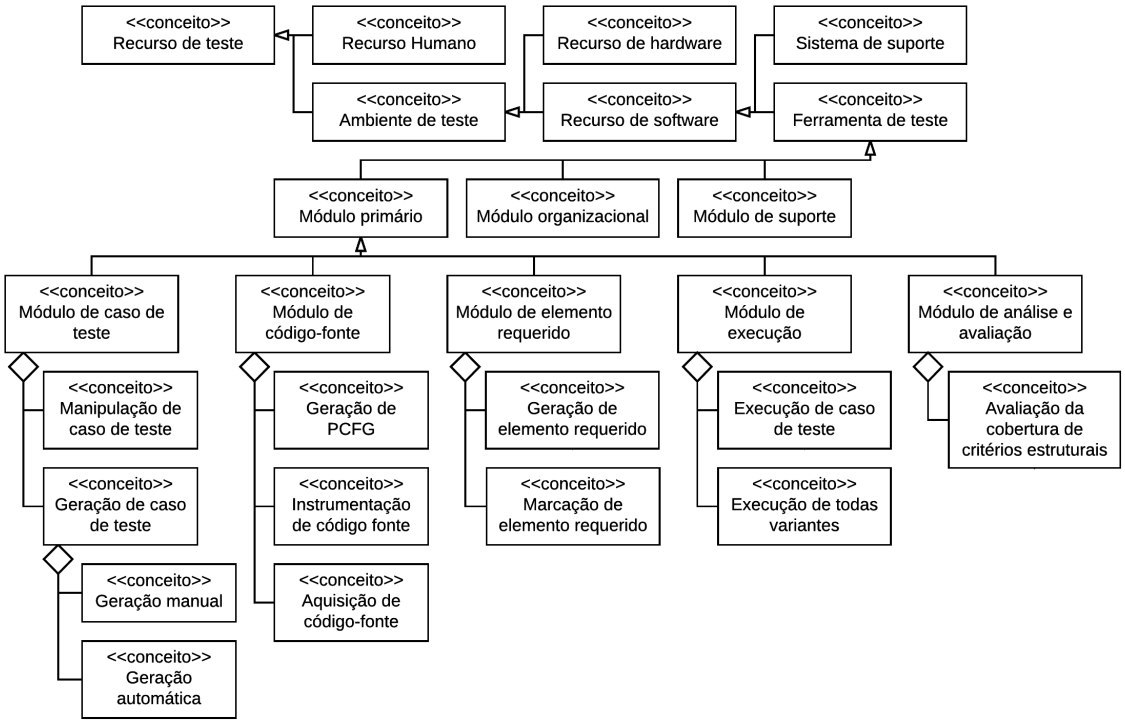


Figura 11 – Especializacão da sub-ontologia de recurso de teste OntoTest (BARBOSA *et al.*, 2008) em UML para o teste estrutural de programas concorrentes

A sub-ontologia de procedimento e estratégia de teste (BORGES; BARBOSA, 2009) define esses conceitos como uma composição de abordagens de teste (baseadas em requisito, especificação, *design* ou implementação), fases de teste (unidade, integração, sistema ou regressão) e técnicas de teste (estrutural, baseada em erros, funcional, baseada em estados, aleatória e *ad-hoc*). Uma técnica de teste estabelece um critério de teste (seleção ou adequação), apoia e usa uma orientação de teste (*testing guidance*) que pode ser um padrão ou uma diretriz. Por fim, procedimentos e estratégias de teste utilizam artefatos sendo testados, que podem ser requisitos de sistema, especificações, *design* ou código-fonte. O teste estrutural de programas concorrentes está definido nesse contexto como uma abordagem baseada na implementação que utiliza a técnica de teste estrutural, estabelecendo critérios de adequação. O artefato utilizado nesse procedimento é o código-fonte do programa sendo testado. O relacionamento entre esses conceitos pode ser observado na Figura 10.

Os recursos de teste são categorizados na sub-ontologia de mesmo nome (BARBOSA *et al.*, 2008) e consideram, por exemplo, recursos humanos e de ambientes de teste. Ambientes de teste podem ser um recurso de hardware ou de software. Recursos de software são divididos em sistemas de suporte e ferramentas de teste, compostos por módulos primários, organizacionais (como módulos de treinamento e de infraestrutura de teste) e módulos de suporte (módulos de documentação e de auditoria).

O teste estrutural de programas concorrentes abordado neste projeto concentra-se em ferramentas de teste primárias que podem ser observadas na Figura 11. Dentre as ferramentas necessárias para o teste estrutural de programas concorrentes estão os módulos de caso de teste, código-fonte em teste, elemento requerido e execução. Um módulo de caso de teste pode ser de geração automática ou manual e também de manipulação de casos de teste. Um módulo de código-fonte em teste, por sua vez, engloba módulos de instrumentação de código-fonte, módulos de geração de PCFG e módulos de aquisição de código-fonte em teste. Os módulos de elementos requeridos são módulos de geração e de marcação de elementos requeridos. Os módulos de execução de teste incluem módulos de execução de um caso de teste e módulos de execução de todas as variantes de execução. Por fim, o módulo de análise e avaliação de teste é especializado para a avaliação da cobertura de critérios de teste estrutural de programas concorrentes.

A sub-ontologia de passo de teste (BARBOSA; NAKAGAWA; MALDONADO, 2006) classifica um passo de teste como de planejamento de teste, desenvolvimento de casos de teste, execução e análise de testes. Além disso, a mesma é composta por um conjunto de atividades que podem ser primárias (como a geração de casos e requisitos de teste), organizacionais (treinamento e infraestrutura de teste) e atividades de suporte (verificação, validação e gerenciamento de configurações).

O teste estrutural de programas concorrentes investigado neste projeto se concentra em passos de execução do teste em si. Esses passos são compostos principalmente por atividades primárias. As demais atividades e passos descritos na sub-ontologia ainda são pertinentes, porém não são específicos para o contexto de programas concorrentes ou da técnica de teste estrutural.

Passos relacionados com o *design* de casos de teste incluem a geração de casos de teste manual ou automática. Embora seja um passo relevante, ele não é abordado diretamente neste projeto. No entanto, a composição de serviços resultante prevê uma interface que pode ser utilizada para especificar os casos de teste produzidos. Com isso, um futuro serviço de geração de casos de teste pode ser integrado à composição - espera-se - mais facilmente.

Como pode ser observado na Figura 12, o teste estrutural de programas concorrentes possui atividades de manipulação de artefatos, geração de elementos requeridos, execução de casos de teste, análise e avaliação de cobertura. A manipulação de artefatos inclui a instrumentação de código-fonte e geração de PCFG, geração de casos de teste, geração de variantes de execução a partir de um caso de teste executado. A geração de elementos requeridos produz os elementos requeridos a serem marcados e avaliados. A análise e avaliação de cobertura utilizam critérios de teste estrutural que consideram características de programas concorrentes em conjunto com os elementos requeridos gerados. Por fim, a execução de casos de teste é uma atividade subdividida em execução determinística e não-determinística. A execução não-determinística envolve uma ou mais execuções de um caso de teste de forma não controlada, enquanto a execução determinística implica na geração de variantes e execução das mesmas de forma controlada, recursivamente.

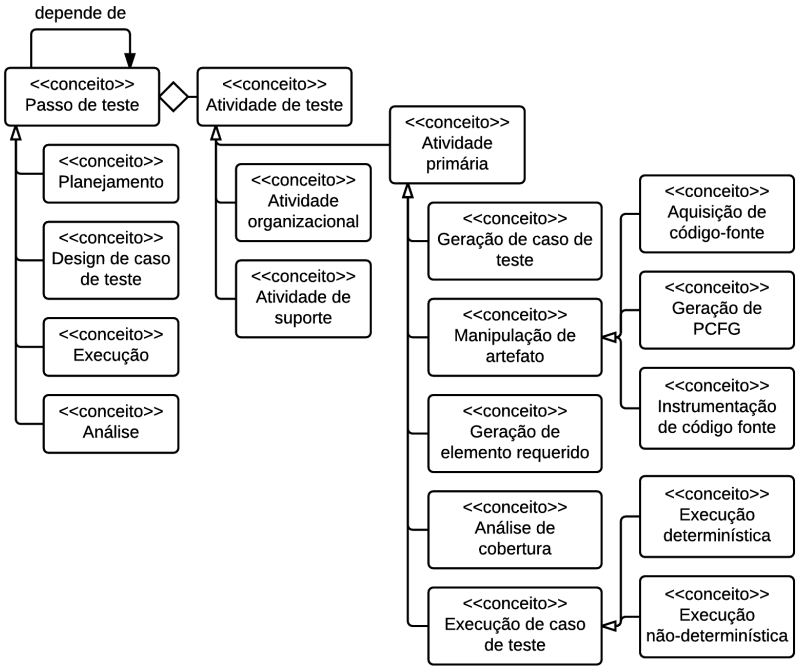


Figura 12 – Especializacão da sub-ontologia de passo de teste OntoTest (BARBOSA; NAKAGAWA; MALDONADO, 2006) em UML para o teste estrutural de programas concorrentes.

Todas atividades apresentadas, essenciais para o teste estrutural de programas concorrentes, podem ser utilizadas para a criação de passos de teste diversos. Um possível passo de teste está representado no diagrama da Figura 13. Nessa configuração, o teste se inicia com a aquisição do código-fonte. Em seguida, é realizada a instrumentação e a geração do PCFG, o qual é utilizado para a geração de elementos requeridos. A partir desse ponto, o testador pode criar casos de teste e executá-los de forma determinística ou não-determinística. O resultado da execução é então analisado. Caso a execução tenha revelado algum defeito ou a cobertura tenha sido atingida, o passo de teste termina. Caso contrário, novos casos de teste são criados e executados, repetindo o processo descrito. O desenvolvimento de ferramentas deve levar em conta essas atividades e o modo como podem ser executadas.

5.4 Definição de Capacidades de Serviços

O mapeamento do teste estrutural de programas concorrentes para serviços resultou em um conjunto de 8 (oito) capacidades de serviço que devem ser implementadas. Cada capacidade de serviço envolve um processamento relativo às atividades de teste especificadas.

A primeira a ser implementada é a **aquisição de código-fonte**. O código-fonte é transferido para o serviço e armazenado pelo mesmo junto a detalhes como: qual a linguagem utilizada.

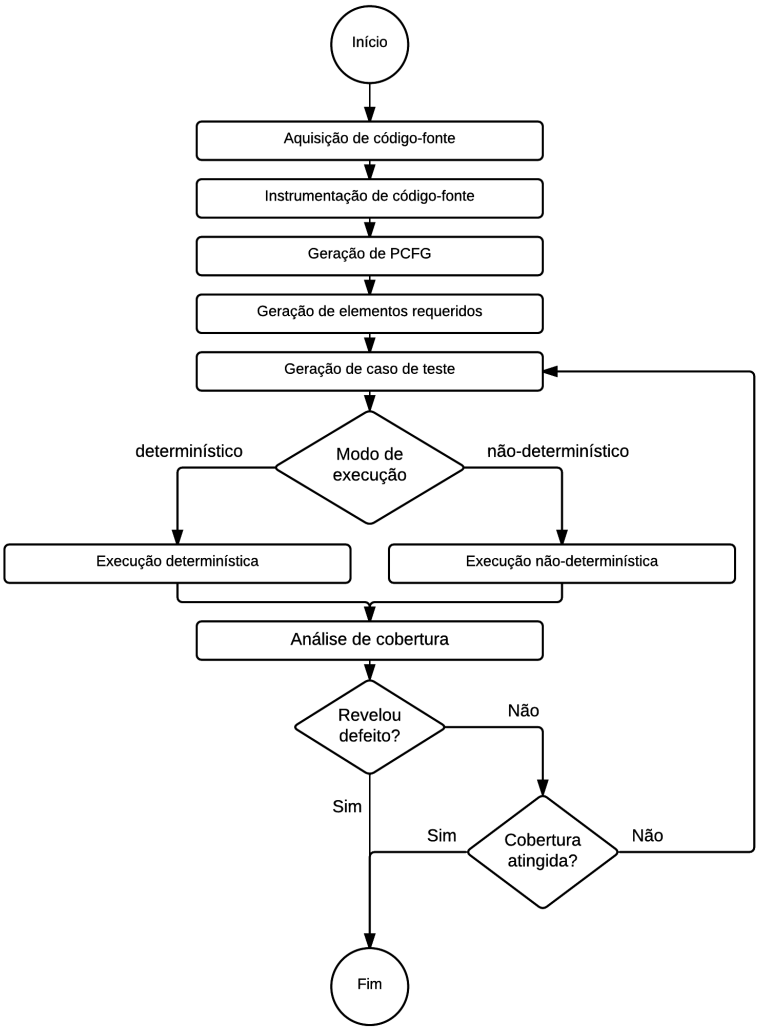


Figura 13 – Passo de teste baseado nas atividades da Figura 12.

Isso é importante para os processamentos posteriores baseados em código-fonte.

Em seguida é necessária a **instrumentação**. Essa capacidade tem como objetivo preparar o código-fonte para a produção de rastros de execução (reportando eventos dos fluxos do programa) e execução controlada. Para tanto, o código-fonte e a definição dos processos e *threads* que compõem o programa concorrente são fornecidos como parâmetros. Como resultado, é retornado o código-fonte instrumentado.

Outra capacidade é a **geração de PCFG**. Essa depende dos mesmos parâmetros de entrada da capacidade de instrumentação. No entanto, ela apenas analisa o código-fonte e descreve o programa concorrente em termos de elementos de modelo. O resultado final é um PCFG que descreve o fluxo de dados, controle e instrumentação.

A instrumentação deve ser compatível com a geração de PCFG para a realização da atividade de teste estrutural. De fato, os elementos de modelo de teste presentes no PCFG devem

ser os mesmos reportados durante a execução do programa concorrente para que seja possível verificar a cobertura de critérios de teste (PRADO *et al.*, 2015). Não há como realizar tal análise de cobertura caso a análise e a representação das informações sejam diferentes para essas duas capacidades.

A **geração de elementos requeridos** permite a análise de elementos do modelo para a criação, obviamente, de elementos requeridos baseados nos critérios de teste. O testador deve fornecer os elementos de modelo de teste do programa concorrente em questão e quais elementos requeridos devem ser produzidos. A partir desses parâmetros, são retornados os elementos requeridos requisitados relativos ao programa concorrente.

A **avaliação de critérios de teste** recebe como parâmetro os critérios que devem ser considerados, e os elementos requeridos em questão. A avaliação de cobertura é feita como resultado de cada submissão de execuções de casos de teste, a qual retorna a cobertura resultante com base nos critérios selecionados.

O conjunto de serviços deve permitir também a **execução (controlada e não-controlada) de casos de teste**. O testador deve fornecer o código-fonte instrumentado, o caso de teste e o modo da execução. No modo controlado, deve ser fornecido ainda, os arquivos de rastro que devem ser reproduzido. O serviço deve então executar o programa instrumentado com a entrada especificada e conferir, no final da execução, se o resultado é correto. A execução do programa, no entanto, está sujeita a parâmetros como limite de tempo, o qual estabelece que a execução deve ser abortada caso o limite seja excedido.

O conjunto de serviços também deve prover a **geração de variantes**. O testador especifica a execução "base" e têm como resultado as variantes relacionadas a ela. Opcionalmente, pode ser fornecida a cobertura atual de critérios de teste para guiar a geração.

Por fim, deve-se fornecer a **execução determinística**. Essa capacidade tem como objetivo analisar um conjunto de execuções, derivar variantes e executar todas para potencialmente aumentar a cobertura de critérios para um conjunto de casos de teste. São fornecidos parâmetros como as execuções "base" e quantos níveis devem ser explorados. No final desse processo, são retornadas todas as execuções de caso de teste realizadas.

A execução determinística é dependente da execução de casos de teste, geração de variantes e avaliação de cobertura de critérios. Durante o processamento realizado, a execução determinística executa o caso de teste, avalia a cobertura de critérios da execução resultante e deriva novas variantes tendo-se como base os arquivos de rastro e a cobertura atual de critérios de teste.

As capacidades investigadas podem ser agrupadas em serviços de acordo com a afinidade das tarefas em questão e de acordo com os requisitos da ferramenta de teste. Por exemplo, as capacidades de instrumentação e de geração de PCFG devem ser compatíveis e, portanto, podem ser oferecidas em conjunto em um mesmo serviço. Pode-se pensar, também, no agrupamento

de capacidade de avaliação de critérios de teste e geração de elementos requeridos, já que estão relacionadas com o mesmo tipo de informação (elementos requeridos).

5.5 Definição de Contratos de Serviços

Os serviços de teste estrutural de programas concorrentes devem possuir contratos (ou interfaces) que promovam os princípios do paradigma de orientação a serviços (apresentados no Capítulo 4). A definição de contratos afeta princípios como "baixo acoplamento", em que os contratos dos serviços não devem ser acoplados à implementação dos mesmos. Além disso deve haver "padronização de contratos", ou seja, todos os serviços dentro de um mesmo repositório devem estar em um mesmo formato. Os contratos de serviço também afetam, mesmo que indiretamente, outros princípios como o de reusabilidade e abstração.

Independente do estilo arquitetural de serviço utilizado, os contratos de serviços devem ser abrangentes e possibilitar o uso dos serviços em contextos diversos. Esta implementação do teste de programas concorrentes como serviço utilizou o estilo arquitetural REST (explicado na Seção 4.3.2). Assim, as capacidades de serviços e os artefatos de teste identificados foram traduzidos em termos de processamentos e recursos (e seus relacionamentos), respectivamente.

Por exemplo, foram modelados nos serviços os contratos para os recursos de caso de teste e execução de caso de teste. A criação de casos de teste envolve a especificação dos argumentos de linha de comando, da entrada padrão de dados e a saída esperada para cada processo do programa concorrente. A criação de execuções, através do processamento da capacidade de serviço "execução (controlada e não-controlada) de casos de teste", envolve a especificação de parâmetros como o tempo limite de execução, que encerra a execução, caso seu tempo de execução exceda o permitido.

Os contratos de serviço (descritos com no Apêndice A) impõem um protocolo a ser seguido para a execução de todas as atividades. Na prática, isso significa que há uma dependência implícita entre os processamentos. Por exemplo, a geração de PCFG não pode ocorrer antes do armazenamento do código-fonte. Essas dependências se traduzem em pré- e pós-condições e a especificação dos possíveis erros, caso as condições não sejam atendidas.

5.6 Desenvolvimento dos Serviços de Teste

O conjunto de serviços ValiPar Service foi criado utilizando-se a linguagem Java utilizando o *framework* Web Spring. Java foi escolhida como linguagem uma vez que a base de código da ferramenta ValiPar *desktop* está implementada nesta linguagem. O *framework* Spring

foi adotado para expor as funcionalidades devido ao seu nível de maturidade, à sua alta adoção e à diversidade de recursos disponibilizados pelo mesmo. Além disso, os recursos necessários para implementação dos serviços (como descrito a seguir) são completamente disponibilizados pelo *framework*. Os serviços são executados utilizando o servidor Web Apache Tomcat (VUKOTIC; GOODWILL, 2011), que implementa todas as tecnologias necessárias para servir clientes utilizando o protocolo HTTP.

Os artefatos de teste são armazenados em um banco de dados PostgreSQL (GROUP, 2016). Pode-se, no entanto, trocar de banco de dados com relativa facilidade devido à interface de acesso banco de dados uniforme disponibilizada pelo ambiente Java. Tais artefatos são servidos utilizando o formato JSON uma vez que é o formato comumente utilizado para serviços REST. Além disso, esse formato resulta em mensagens menores que o equivalente em XML, isto é, é um formato mais compacto.

O desenvolvimento dos serviços ocorreu em duas etapas. Inicialmente foi criado um serviço que realiza todas as funcionalidades do teste estrutural e um cliente que interage com o serviço pelos contratos estabelecidos. Essa etapa teve como objetivo definir por completo toda a interface do serviço que orquestra toda a atividade de teste. Em seguida, as funcionalidades desse serviço foram decompostas em serviços especializados para que seja possível disponibilizar funcionalidades, para fins de reuso e de escalabilidade, de forma descentralizada.

Os serviços foram implementados em uma única base de código. Isso se deve ao relacionamento existente entre os recursos. Por exemplo, o recurso “PCFG” está presente no processo de “geração de PCFG” e no processo de “geração de elementos requeridos”. A unificação da base de código facilitou o desenvolvimento e promoveu aspectos de reuso e modularidade de código, sem duplicação de código relacionada aos recursos e aos processamentos definidos. Uma arquitetura em duas camadas foi então desenvolvida para permitir a criação dos serviços. Estas camadas são chamadas de contratos de serviços e capacidades de serviços. Estas camadas dividem as responsabilidades e permitem o desenvolvimento de serviços variados. A camada de contrato de serviços é responsável por disponibilizar as interfaces oferecidas para o cliente. Cada recurso implementado pode ser acessado de quatro modos diferentes: “sem-interação (SI)”, “somente-leitura (SL)”, “*upload*-e-leitura (UL)”, “geração-e-leitura (GL)”. O modo “SI” é usado quando o serviço em questão não produz nem consome o recurso, o modo “SL” ocorre quando o serviço apenas produz o recurso, enquanto os modos “UL” e “GL” são utilizados quando o serviço apenas consome e gera o recurso, respectivamente.

A camada de capacidades de serviços, por sua vez, define quais os processamentos habilitados para o serviço. Cada operação pode ser interna ou externa. Operações internas podem ainda ser sequenciais (“IS”) ou paralelas (“IP”). Uma operação interna sequencial utiliza apenas os recursos computacionais locais enquanto operações internas paralelas utilizam uma arquitetura distribuída para executar uma tarefa. Operações externas (“E”), por sua vez, delegam a computação para um serviço especializado.

Essa organização é flexível e permite a criação de serviços para necessidades específicas. Os modos de operações internas permitem que o serviço seja testado utilizando uma infraestrutura simples usando o modo sequencial e permite que o serviço seja escalável usando o modo paralelo, que requer uma infraestrutura mais complexa com várias máquinas. Além disso, essa organização permite a criação de experimentos para comparação de modos de execução. Por exemplo, um novo modo de operação interna pode ser criado e avaliado em relação aos já existentes. Ela também permite a criação de serviços convenientes, ou mais eficientes, para determinados contextos. Um grupo de pesquisa que queira instalar os serviços em sua própria infraestrutura, mas que necessitará apenas da geração de elementos requeridos e avaliação de critérios de teste pode configurar um único serviço com ambas capacidades e recursos relacionados.

Por fim, essa arquitetura permite a criação de composições diversas. Pode-se derivar novos serviços para funcionalidades que utilizem as capacidades implementadas em qualquer ordem e de diferentes modos, reusando serviços especializados ou executando sequencialmente, por exemplo.

Os serviços resultantes aderem a uma mesma interface uniforme. Isto é, os recursos são representados utilizando um mesmo formato reconhecido por todos os serviços do conjunto. Por exemplo, o PCFG é representado da mesma forma em um serviço que gere PCFG e em um serviço que gere elementos requeridos. Isso facilita a adoção dos serviços e a criação de novas ferramentas e clientes uma vez que elimina processos intermediários de conversão de mensagem para comunicação entre serviços.

A partir dessa organização, a definição de um serviço de teste é feita de forma estática por meio de arquivos de configuração. Dessa forma, o desenvolvedor de serviços de teste deve especificar quais as interfaces e capacidades devem ser habilitadas e como devem ser habilitadas. Isso facilita a adoção dos serviços por outros grupos uma vez que evita o envolvimento desnecessário do testador ou desenvolvedor com o código-fonte dos serviços.

5.7 Conjunto de serviços ValiPar Service

O serviço ValiPar Service (detalhado na Seção 3.6.1) e mais 5 (cinco) serviços especializados foram criados a partir da arquitetura especificada. O ValiPar Service atua como um *proxy*, ou seja, recebe requisições, interpreta e gerencia outros serviços especializados para atender as mesmas.

Cada serviço especializado é responsável por pelo menos uma das capacidades implementadas e habilita os contratos necessários para fornecimento de informações e acesso aos recursos produzidos. O conjunto de serviços ValiPar Service conta com os serviços ValiInst (Seção 5.7.1), ValiElem (Seção 5.7.2), ValiExec (Seção 5.7.3), ValiEval (Seção 5.7.4), e ValiSync

(Seção 5.7.5).

5.7.1 ValiInst Service

A ValiInst Service é descrito na Figura 16a. Esse serviço é responsável pela instrumentação e geração do PCFG. Tais capacidades foram agrupadas por estarem fortemente relacionadas, ou seja, ambas dependem de uma mesma análise para gerarem artefatos compatíveis. Assim, são habilitados contratos para "armazenamento-e-leitura (UL)" de código-fonte e para "geração-e-leitura (GL)" de código-fonte instrumentado e PCFG. O armazenamento de código-fonte, instrumentação e geração de PCFG são habilitados como internas e sequenciais para permitir a transferência do programa concorrente para o sistema e a derivação de novos recursos.

O cliente pode interagir com o serviço como descrito na Figura 14a. Inicialmente o código-fonte é transferido para o serviço como um arquivo compactado (formato ZIP) contendo todos os arquivos do programa concorrente. Como este serviço suporta a análise de programas em Java, os arquivos compactados são arquivos ".class" (*bytecode*). O cliente deve, então, criar uma sessão de teste para adicionar os recursos relacionados ao programa em questão.

Em seguida o PCFG e programa instrumentado são criados a partir dos parâmetros de especificação de sincronizações e arquivos ignorados. O parâmetro de especificação descreve quais operações devem ser consideradas primitivas de sincronização e permite que novas operações sejam levadas em consideração na análise. O parâmetro de arquivos ignorados especifica quais arquivos fazem parte do programa, mas não devem ser analisados. Isso é necessário para casos em que o programa utiliza estruturas da linguagem não suportadas pela análise, por exemplo. Após a submissão da requisição, há o processamento e a disponibilização dos recursos para uso futuro.

Por meio dessa interação, o serviço expõe as mesmas funcionalidades implementadas pela ValiPar *desktop*. Futuras versões da ferramenta *desktop* e do serviço podem envolver parâmetros como nível de detecção de fluxos do programa e tipos de análise. Por isso essa interface deverá ser estendida para aumentar a flexibilidade do serviço.

5.7.2 ValiElem Service

A ValiElem Service é descrita na Figura 16b. Esse serviço tem como única responsabilidade gerar elementos requeridos. Para isso, a camada de contratos possui contratos habilitados para o "armazenamento-e-leitura" de PCFG ("UL") e "geração-e-leitura" de elementos requeridos ("GL"). O único processamento ativado na camada de capacidades é a geração de elementos requeridos, configurado como uma operação interna sequencial ("IS"). Como ilustrado na Figura 14b, a interação com esse serviço tem 4 (quatro) operações.

Inicialmente, cria-se uma sessão de teste. Em seguida, associa-se o PCFG do programa concorrente à sessão utilizando a operação "*Upload* de PCFG". O PCFG enviado para o serviço deve obedecer o formato pré-estabelecido, equivalente ao PCFG retornado pela ValiInst Service. A partir desse ponto é possível requisitar a geração de elementos requeridos utilizando a operação "Gerar elementos requeridos". Essa operação têm como parâmetro o campo "types" que permite definir quais os tipos de elementos requeridos que devem ser gerados. Caso nenhum tipo seja especificado, o serviço gera todos os tipos de elementos. Os tipos de elementos requeridos suportados pela ferramenta podem ser conhecidos pela operação "Consultar tipos de elementos requeridos".

Os contratos e capacidades estabelecidas para esse serviço expõem todas as funcionalidades já existentes para a ferramenta *desktop*. Futuras versões das ferramentas podem conter novos tipos de elementos requeridos. No contexto do serviço desenvolvido, isso implica na presença de um novo tipo na consulta de tipos de elementos requeridos. Assim, o cliente deve incluir explicitamente esse tipo na geração de elementos requeridos para obter os respectivos elementos. Essa dinâmica incremental ajuda na manutenção de compatibilidade com clientes.

5.7.3 ValiExec Service

A ValiElem Service é descrita na Figura 16c. Esse serviço é responsável por todos aspectos de execução de casos de teste. Para isso seus contratos permitem a geração ("GL") de casos de teste e de execuções, além do *upload* ("UL") de programas instrumentados e variantes. A única capacidade habilitada é a de execução de casos de teste, que ocorre de modo interno e paralelo, utilizando uma arquitetura para distribuição de tarefas.

Como ilustrado na Figura 14c, a interação com esse serviço se inicia com a criação de uma sessão de teste, especificando-se quais são as *threads* e processos do programa concorrente. Nessa sessão, o cliente deve, inicialmente, submeter o código-fonte instrumentado a ser executado ("Armazenar código-fonte instrumentado"). Em seguida o cliente pode executar a operação "Criar caso de teste" para registrar novos casos de teste e, em seguida, a operação "Executar casos de teste" para requisitar que os mesmos sejam executados.

Opcionalmente, o cliente pode adquirir o resultado da execução (arquivos gerados, entrada e saída, arquivos de rastros) para derivar variantes baseadas na mesma. Tais variantes geradas podem, então, ser submetidas ao serviço utilizando a operação "Executar variante", que a relaciona com a execução inicial e a executa de modo controlado.

Os contratos resultantes permitem o mesmo conjunto de operações já realizado utilizando a ferramenta *desktop*. A organização da ferramenta como serviço, no entanto, permite maior escalabilidade na criação de execuções, uma vez que pode disponibilizar uma arquitetura distribuída para paralelizar tarefas e distribuí-las em diversas máquinas. Isso é particularmente

importante para a execução de casos de teste, uma vez que é a atividade que mais consome recursos computacionais para ser completada.

5.7.4 ValiEval Service

A ValiEval Service (Figura 16d) é responsável pela avaliação de critérios de teste. A mesma possui contratos para o *upload* e leitura (UL) de elementos requeridos, geração e leitura (GL) de casos de teste, *upload* e leitura de execuções, além da geração e leitura de cobertura de teste. Na camada de capacidades, apenas a avaliação critérios é habilitada como interna paralela.

A interação com esse serviço é feita por meio de 4 operações, como pode ser observado na Figura 14d. Inicialmente, deve-se criar uma sessão de teste para incluir artefatos relacionados. Em seguida é feito o armazenamento de elementos requeridos (“Armazenar elementos requeridos”). Casos de teste são criados iterativamente para armazenar execuções (“Armazenar execução”). Nesse processo, os rastros de execução armazenados são avaliados e os elementos requeridos cobertos são marcados como tal. O cliente pode solicitar, a qualquer momento, a cobertura de critérios de teste com a operação “Avaliar cobertura”. Nessa operação, totaliza-se o número de elementos cobertos em comparação com o número total de elementos requeridos e retorna-se uma taxa de cobertura.

Do mesmo modo como a ValiExec Service, a ValiEval Service pode dar suporte à execução de tarefas em uma arquitetura distribuída. Isso é necessário para possibilitar um sistema mais escalável, já que a avaliação de critérios é uma operação frequentemente utilizada durante o teste estrutural. De fato, todas as execuções são submetidas à avaliação e, em casos típicos, milhares de execuções de caso de teste são considerados.

5.7.5 ValiSync Service

A ValiSync Service (Figura 16e) permite a execução de todas as variantes baseadas em um conjunto de casos de teste. Internamente, o serviço executa casos de teste e, iterativamente, gera novas variantes baseado numa fila de execuções, que é alimentada com a execução de variantes geradas. Esse processo pode utilizar a cobertura de critérios de teste para podar variantes que exercitem elementos requeridos já cobertos.

Para fornecer essa funcionalidade, esse serviço possui contratos para manipulação de casos de teste, execução, variantes e cobertura. São oferecidos contratos para geração e leitura (GL) de casos de teste e de variantes, contratos para leitura (SL) de execuções e para armazenamento e leitura (UL) de cobertura.

O serviço também é composto pelas capacidades de “Geração de variantes”, “Execução

determinística", "Execução de casos de teste" e "Avaliação de critérios". Enquanto as duas primeiras capacidades são executadas internamente, as duas últimas são executadas externamente (E) por serviços especializados (ValiExec Service e ValiEval Service, respectivamente). A geração de variantes é feita de modo paralelo (IP) e a execução determinística de modo sequencial (IS).

Como ilustrado na Figura 14e, a utilização do serviço se inicia pela criação de uma sessão de teste. Em seguida, o cliente deve armazenar o código-fonte instrumentado e o endereço para a cobertura de critérios de teste relacionada. Então, o cliente cria casos de teste e, por fim, solicita a execução determinística de todos os casos de teste. O serviço então realiza a execução e retorna uma lista com todas as variantes e execuções correspondentes.

5.7.6 ValiPar Service

A ValiPar Service oferece ao cliente todas as funcionalidades descritas anteriormente. Isso se deve à estratégia adotada de se ter um serviço que atua como "*proxy*" do conjunto de serviços, orquestrando os serviços para completar as tarefas. A composição é então transparente para o cliente, que enxerga a ValiPar Service como um único serviço monolítico.

Os contratos e capacidades desse serviço podem ser visualizadas na Figura 16f. Todos os contratos estão habilitados como geração e leitura (GL), com exceção do código-fonte, que deve ser armazenado (UL). Todas as capacidades também são habilitadas, mas realizam o processamento de modo externo (E).

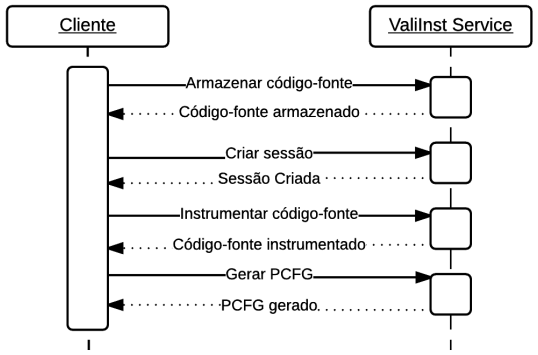
A Figura 15 descreve a interação entre um cliente e a ValiPar Service, e a interação da ValiPar Service com o restante dos serviços especializados. O armazenamento de código-fonte é redirecionado para a ValiInst Service. A criação de uma sessão de teste na ValiPar Service resulta na criação de uma sessão de teste em todos os demais serviços.

A partir desse ponto o cliente pode requisitar a instrumentação, geração de PCFG, geração de elementos requeridos, avaliação de cobertura, criação de casos de teste e execução dos mesmos. Para essas operações, a ValiPar Service repassa a requisição para os serviços especializados. Por exemplo, após a geração de PCFG na ValiInst Service, o recurso é submetido para a ValiElem Service para futura geração de elementos requeridos.

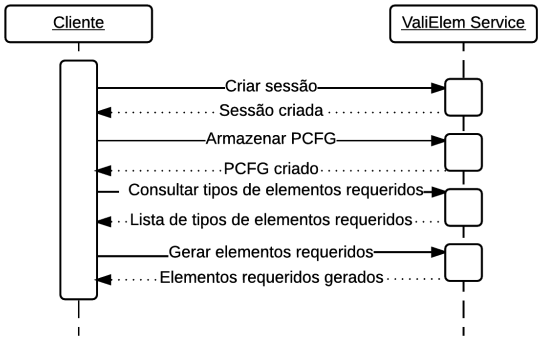
Essa composição evidencia os benefícios da organização de ferramentas como serviços. Os contratos padronizados evitam transformações intermediárias das mensagens. Além disso, os serviços são reusáveis e permitem futuras extensões. Por fim, a implementação dos serviços permite a sua instalação em ambientes simples e complexos, facilitando a adoção dos serviços por outros grupos de pesquisa, por exemplo. Aspectos qualitativos e quantitativos relativos aos serviços e à composição são detalhados no Capítulo 6

5.8 Considerações Finais

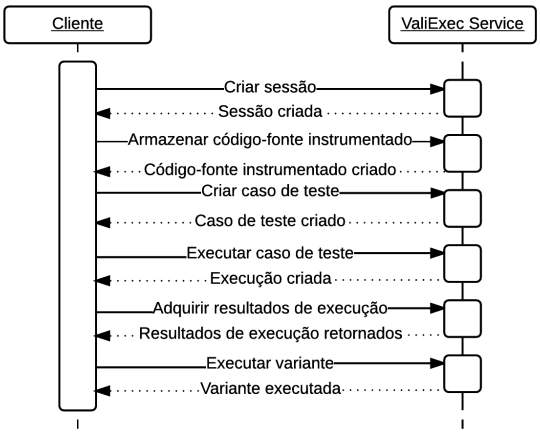
Esse capítulo apresentou os principais aspectos do processo de desenvolvimento do conjunto de serviços para o teste estrutural de programas concorrentes. Uma ontologia serviu de base para a criação de serviços mais abrangentes. Também foi criada uma arquitetura que enfatiza aspectos de reuso, uniformidade de contratos e criação de serviços para contextos diversos. O ValiPar Service é responsável por interpretar requisições e delegar tarefas aos serviços especializados. Assim, o conjunto de serviços resultantes permite a execução de todas as atividades de teste e os serviços, individualmente, oferecem funções flexíveis para composições não previstas inicialmente.



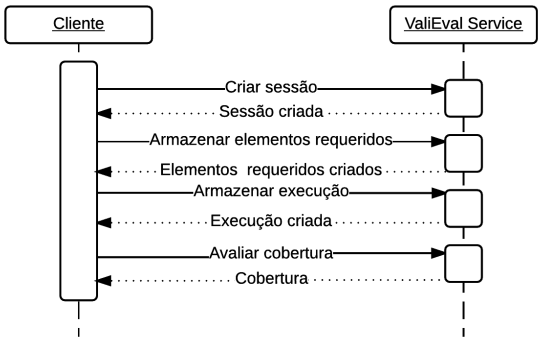
(a) Interação com ValiInst Service. O código-fonte é armazenado, uma sessão de teste é criada e, em seguida, o PCFG e código-fonte instrumentados são gerados.



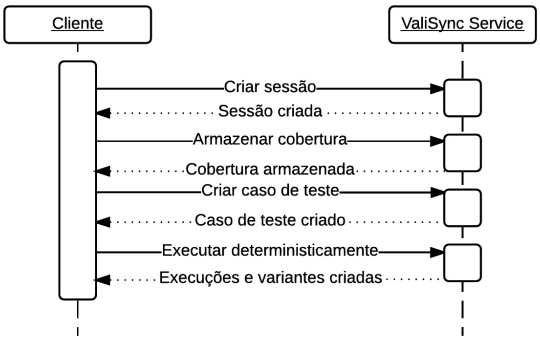
(b) Interação com ValiElem Service. O PCFG é armazenado e é feita a requisição para geração de elementos requeridos.



(c) Interação com ValiExec Service. O código-fonte instrumentado é armazenado e, de forma iterativa, casos de teste são criados e executados. Variantes também podem ser armazenadas e executadas no serviço.



(d) Interação com ValiEval Service. Os elementos requeridos são armazenados e, de modo incremental, execuções são armazenadas e avaliadas no serviço. O cliente pode consultar a cobertura dos critérios selecionados.



(e) Interação com ValiSync Service. O código-fonte instrumentado é armazenado. De forma iterativa, casos de teste são criados e executados de modo determinístico

Figura 14 – Interação entre um cliente os serviços.

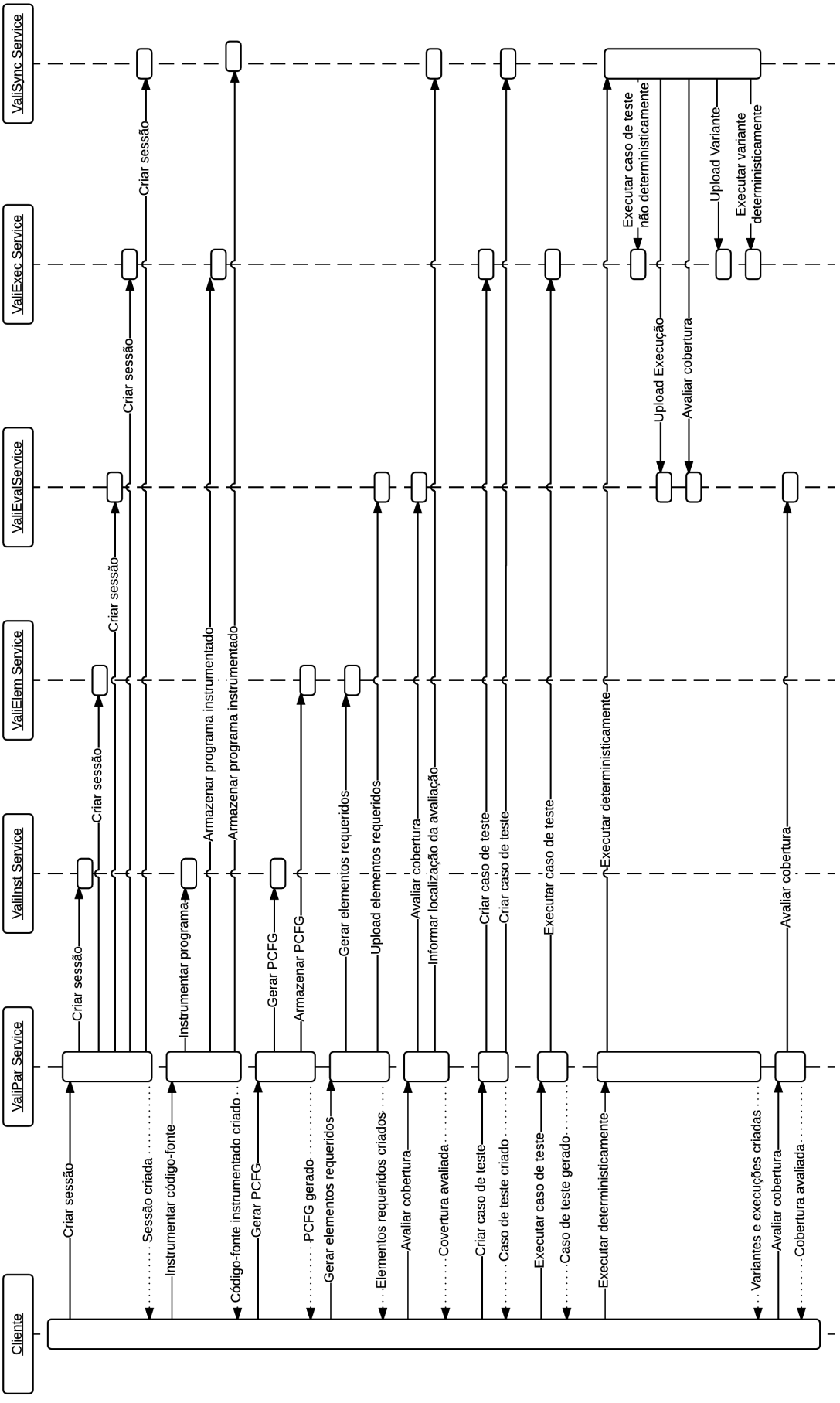


Figura 15 – Interação entre um cliente e a ValiPar Service e entre a ValiPar Service com os serviços especializados. O cliente requisita a geração de artefatos e a ValiPar Service, atuando como um "proxy", delega as requisições para os demais serviços.

Serviço			
Contrato		Capacidade	
Modo	Recurso	Modo	Tipo
UL	Código-fonte	IS	Armazenamento de código-fonte
GL	PCFG	IS	Geração de PCFG
GL	Instrumentado	IS	Instrumentação de código-fonte
SI	Elemento requerido	N	Geração de elemento requerido
SI	Caso de teste	N	Execução de caso de teste
SI	Execução	N	Geração de variante
SI	Variante	N	Execução determinística
SI	Cobertura	N	Avaliação de critérios

(a) ValiInst Service

Serviço			
Contrato		Capacidade	
Modo	Recurso	Modo	Tipo
SI	Código-fonte	N	Armazenamento de código-fonte
UL	PCFG	N	Geração de PCFG
SI	Instrumentado	N	Instrumentação de código-fonte
GL	Elemento requerido	IS	Geração de elemento requerido
SI	Caso de teste	N	Execução de caso de teste
SI	Execução	N	Geração de variante
SI	Variante	N	Execução determinística
SI	Cobertura	N	Avaliação de critérios

(b) ValiElem Service

Serviço			
Contrato		Capacidade	
Modo	Recurso	Modo	Tipo
SI	Código-fonte	N	Armazenamento de código-fonte
SI	PCFG	N	Geração de PCFG
UL	Instrumentado	N	Instrumentação de código-fonte
SI	Elemento requerido	N	Geração de elemento requerido
GL	Caso de teste	IP	Execução de caso de teste
GL	Execução	N	Geração de variante
UL	Variante	N	Execução determinística
SI	Cobertura	N	Avaliação de critérios

(c) Valiexec Service

Serviço			
Contrato		Capacidade	
Modo	Recurso	Modo	Tipo
SI	Código-fonte	N	Armazenamento de código-fonte
SI	PCFG	N	Geração de PCFG
SI	Instrumentado	N	Instrumentação de código-fonte
UL	Elemento requerido	N	Geração de elemento requerido
UL	Caso de teste	N	Execução de caso de teste
UL	Execução	N	Geração de variante
SI	Variante	N	Execução determinística
GL	Cobertura	IP	Avaliação de critérios

(d) ValiEval Service

Serviço			
Contrato		Capacidade	
Modo	Recurso	Modo	Tipo
SI	Código-fonte	N	Armazenamento de código-fonte
SI	PCFG	N	Geração de PCFG
UL	Instrumentado	N	Instrumentação de código-fonte
SI	Elemento requerido	N	Geração de elemento requerido
GL	Caso de teste	IP	Execução de caso de teste
SL	Execução	IS	Geração de variante
GL	Variante	IP	Execução determinística
UL	Cobertura	N	Avaliação de critérios

(e) ValiSync Service

Serviço			
Contrato		Capacidade	
Modo	Recurso	Modo	Tipo
UL	Código-fonte	E	Armazenamento de código-fonte
GL	PCFG	E	Geração de PCFG
GL	Instrumentado	E	Instrumentação de código-fonte
GL	Elemento requerido	E	Geração de elemento requerido
GL	Caso de teste	E	Execução de caso de teste
GL	Execução	E	Geração de variante
GL	Variante	E	Execução determinística
GL	Cobertura	E	Avaliação de critérios

(f) ValiPar Service

Figura 16 – Arquitetura interna de cada serviço. Os serviços são descritos em termos de contratos e capacidades habilitadas (em destaque). Os modos de contrato “SI”, “SL”, “UL”, “GL” são siglas para “Sem interface”, “Somente leitura”, “*Upload* e leitura”, “Geração e leitura”. Os modos para capacidades “N”, “IS”, “IP”, “E” significam, respectivamente, “Nenhum”, “Interno sequencial”, “Interno paralelo”, “Externo”.

CAPÍTULO

6

EXPERIMENTOS E ANÁLISE DOS RESULTADOS

6.1 Considerações Iniciais

Este capítulo apresenta os resultados da avaliação realizada sobre o conjunto de serviços desenvolvidos, e uma discussão crítica sobre os mesmos. Tal avaliação foi realizada em duas formas: uma quantitativa e outra qualitativa. A Seção 6.2 descreve a avaliação quantitativa com foco no tempo de resposta para a execução da atividade de teste estrutural de programas concorrentes. São apresentados os objetivos, o tipo de experimento, os resultados obtidos e as conclusões obtidas a partir dos mesmos. A Seção 6.3 apresenta a análise qualitativa dos serviços, onde as qualidades e limitações da ferramenta foram descritas e relacionadas com o cenário atual de ferramentas de teste como serviço.

6.2 Avaliação Quantitativa dos Serviços

A avaliação quantitativa verificou o impacto no desempenho de uma ferramenta de teste estrutural de programas concorrentes quando esta é organizada como um conjunto de serviços. A métrica que quantifica o desempenho neste caso é o tempo de resposta do teste realizado. Essa análise é importante uma vez que esse tipo de ferramenta apresenta novas características como latência de rede e serialização de mensagens, que podem comprometer seu desempenho quando leva-se em consideração a ferramenta *desktop* equivalente.

O experimento consistiu na execução de todas as atividades de teste utilizando um con-

junto de *benchmarks*. Foram consideradas, além da ferramenta ValiPar Service, a ferramenta de teste estrutural de programas concorrentes ValiPar para ambiente *desktop* (não foram encontradas na literatura outras ferramentas de teste estrutural de programas concorrentes como serviços na Web). As ferramentas foram testadas utilizando *benchmarks*, extraindo-se o tempo de resposta da execução de todas as atividades de teste. Cada experimento foi replicado de forma a se obter a média dos tempos de resposta com um intervalo de confiança de 95%.

Foram utilizados um conjunto de nove *benchmarks* concorrentes variados apresentados em (DOURADO, 2015), descritos na Tabela 1. Cada *benchmark* consiste de um programa concorrente junto a um conjunto de casos de teste desenvolvido para contemplar a cobertura do critério de teste estrutural todos-os-nós.

Tais programas concorrentes possuem uma grande diversidade de semânticas e características variadas. Eles são, dessa forma, representativos em termos de recursos para o desenvolvimento de programas concorrentes e distribuídos, visto que exercitam diferentes padrões (simples e complexos) de comunicação e sincronização, os quais são considerados pontos chave para aplicações concorrentes.

O conjunto de *benchmark* contém programas que utilizam passagem de mensagem e memória compartilhada, programas com tamanho (linhas de código) que varia de 43 até 246 LOC, com número de processos que varia de 1 até 5 e com número de *threads* que varia de 3 até 13. Além disso, os programas possuem número variado de sincronizações que variam de 4 a 96, requerendo a execução de até 606 variantes para a cobertura de todas as arestas de sincronização existentes.

Os *benchmarks* NB, PG e GT utilizam o paradigma de passagem de mensagem. O NB usa primitivas de comunicação não-bloqueantes para troca de mensagens entre 3 processos. O PG calcula MDC (máximo divisor comum) entre 3 números utilizando 4 processos e primitivas de comunicação bloqueantes. O GT também calcula o MDC, porém utiliza 3 processos com um padrão de comunicação distinto, com primitivas de comunicação bloqueantes, dentro de estruturas de repetição.

Os *benchmarks* MC, HO e RW, por sua vez, utilizam o paradigma de memória compartilhada. O MC exercita o compartilhamento de memória entre duas *threads* sincronizadas por um semáforo binário. O HO simula a formação de moléculas de água utilizando semáforos para sincronização. O RW implementa o problema dos leitores e escritores utilizando semáforos binários.

Por fim, os *benchmarks* MS, TR e MA fazem uso de ambos paradigmas. O MS implementa o problema do mestre e escravo em que o processo mestre faz uma requisição que é atendida pelos dois escravos utilizando primitivas de comunicação bloqueantes e semáforos binários para sincronização de acesso. O TR simula uma topologia de anel (*token ring*) utilizando primitivas de comunicação bloqueantes e não-bloqueantes, semáforos binários e eventos de

Tabela 1 – *Benchmarks* utilizados no experimento. Cada programa utiliza uma variedade de primitivas de sincronização (para passagem de mensagem (PM) e memória compartilhada (MC)), possui um número variado de processos e *threads* e apresenta diferentes padrões de sincronização e comunicação. Os valores P, M, G entre parenteses indicam a classificação "pequeno", "médio" e "grande", respectivamente. As siglas NB, PG, GT, MC, RW, HO, MS, TR, MA significam, respectivamente, non blocking bsend, parallel gcd, gcd two slaves, micro sm, readers writers, h2o, master slave, token ring file, matrix. Por fim, as três ultimas colunas comparam o tempo de resposta entre a ferramenta *desktop* e duas versões de serviços. "desk", "seq" e "dist" indicam, respectivamente, tempo de resposta na execução do *benchmark* com a ferramenta valipar, valipar-service-6n-0d e valipar-service-6n-8d.

Benchmark	**Paradigma**	**LOC**	**Processos**	***Threads***	***Syncs***	**Variantes**	**dist < desk**	**dist < seq**	**seq < desk**
NB	PM	71 (P)	3 (M)	3 (P)	6 (P)	12 (P)	Não	Não	Não
MC	MC	43 (P)	1 (P)	3 (P)	20 (P)	4 (P)	Não	Não	Não
RW	MC	145 (M)	1 (P)	4 (P)	52 (M)	606 (G)	Não	Não	Não
HO	MC	113 (M)	1 (P)	4 (P)	27 (P)	6 (P)	Não	Não	Não
MS	PM/MC	102 (P)	2 (P)	3 (P)	4 (P)	0 (P)	Não	Não	Não
PG	PM	180 (G)	4 (G)	4 (P)	24 (P)	54 (P)	Sim	Sim	Não
GT	PM	201 (G)	3 (M)	3 (P)	12 (P)	58 (P)	Sim	Sim	Não
TR	PM/MC	246 (G)	3 (M)	12 (G)	96 (G)	12 (P)	Sim	Sim	Não
MA	PM/MC	214 (G)	5 (G)	13 (G)	68 (G)	34 (M)	Sim	Sim	Não

comunicação. O MA implementa uma multiplicação de matriz usando uma configuração mestre e escravo utilizando características semelhantes ao TR.

Cada *benchmark* foi executado em quatro configurações diferentes de ferramentas de teste totalizando 36 experimentos distintos. Foi utilizado um fluxo de execução típico do teste de programas concorrentes. Inicialmente o programa é importado pela ferramenta de teste. Em seguida esse é submetido ao processo de instrumentação de código-fonte e geração de PCFG. Em seguida são gerados os elementos requeridos. É inserido, assim, todo o conjunto de casos de teste para os programas em questão. A partir desse ponto, é requisitada a execução determinística dos casos de teste em que os mesmos são executados de forma controlada e submetidos para avaliação de cobertura de critérios. Essa última atividade ocorre iterativamente com a geração intermediária de novas variantes de execução. A Figura 13 ilustra o passo de teste descrito graficamente.

Esse fluxo de execução foi instanciado para quatro versões de ferramenta de teste estrutural de programas concorrentes. Foram utilizadas as ferramentas ValiPar *Desktop* (chamada "valipar") como base de comparação e três versões da ferramenta ValiPar *Service*. A primeira versão, chamada "valipar-service-1n-0d" consiste nos seis serviços instalados em uma mesma máquina sem utilização de arquitetura distribuída. A segunda, "valipar-service-6n-0d", é composta pelos seis serviços distribuídos em seis diferentes máquinas, mas cada um dos serviços utilizando apenas recursos locais (sem nós da arquitetura distribuída).

A última versão, "valipar-service-6n-8d" (serviços em 6 nós utilizando uma arquitetura distribuída com 8 nós) , também distribui os seis serviços em seis máquinas, porém utiliza ainda a arquitetura distribuída para distribuição de tarefas (execução paralela) dos serviços de execução (ValiExec *Service*), de avaliação (ValiEval Service) e de geração de variantes (ValiSync Service). Os demais serviços não foram utilizados de forma paralela por serem utilizados apenas no início

do teste, oferecendo pouco *overhead* para a atividade como um todo.

A execução de cada *benchmark* em cada configuração definida foi realizada em um *cluster* computacional do Laboratório de Sistemas Distribuídos e Programação Concorrente (LaSDPC). Foram utilizadas quatro máquinas físicas, cada uma com 2 máquinas virtuais. As máquinas físicas contam com um processador Intel(R) Core(TM) i7-4790 3.60GHz com 32 GB de RAM. As máquinas virtuais contam com processadores virtuais de quatro núcleos e são gerenciadas pelo virtualizador KVM. Em termos de software, as máquinas virtuais possuem o sistema operacional Ubuntu 14.04.1 LTS e as ferramentas (escritas na linguagem de programação Java) rodam na máquina virtual (JVM) OpenJDK Java 1.8.

Os resultados obtidos podem ser observados nas Figuras 17 e 18 e na Tabela 1. De um modo geral, é possível observar que o tempo de resposta das ferramentas valipar-service-1n-0d e valipar-service-6n-0d são sempre superiores ao da ferramenta valipar. Embora ofereçam as mesmas funcionalidades, as ferramentas instanciadas como serviço requerem um processamento adicional relacionado, principalmente à latência de rede, à serialização de mensagens e ao tratamento de protocolos intermediários para a comunicação entre o cliente e serviços.

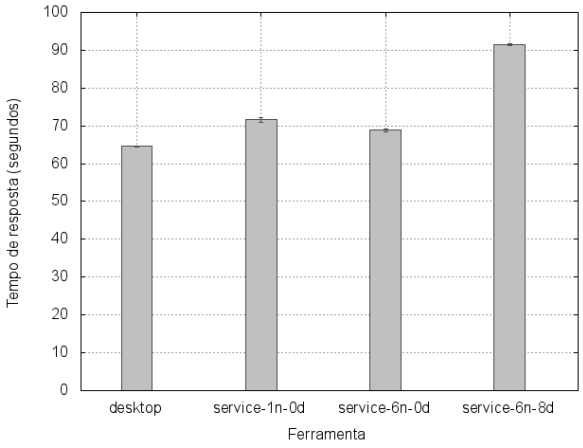
Embora não haja latência de rede para a ferramenta valipar-service-1n-0d, os demais fatores ainda estão presentes. Além disso, os seis serviços estão competindo pelos mesmos recursos computacionais (ou seja, memória e processamento). Pode-se concluir, nesse cenário, que os serviços, utilizando a mesma quantidade de recursos fornecida para a valipar e sem a presença da latência de rede, ofereceu um tempo de resposta maior para os experimentos realizados.

Quando se compara o tempo de resposta da ferramenta valipar com a valipar-service-6n-0d, percebe-se ainda um *overhead*. No entanto, em quase todos os casos, o mesmo é menor se comparado com a ferramenta valipar-service-1n-0d. Isso se deve à distribuição dos serviços em máquinas distintas, o que evita a competição por recursos computacionais.

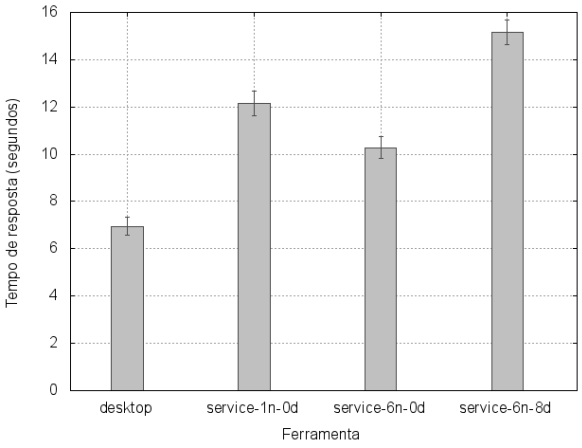
Os resultados obtidos para a ferramenta valipar-service-6n-8d podem ser divididos em duas classes distintas. A primeira classe é representada pelos *benchmarks* PG, TR, MA e GT. Como pode ser observado na Tabela 1 e na Figura 18 houve um ganho de desempenho (diminuição no tempo de resposta) nesses casos se comparado com as demais ferramentas.

A segunda classe, por outro lado, evidencia uma perda de desempenho da valipar-service-6n-8d, quando comparada com valipar e valipar-service-6n-0d. Como pode ser observado na Tabela 1, fazem parte dessa classe os *benchmarks* MC, HO, MS, NB e RW (Figura 17).

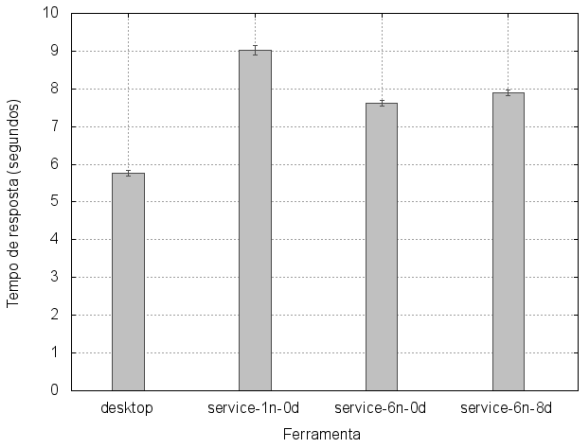
A partir de uma análise nas características dessas duas classes observadas, pode-se notar uma influência de “LOC”, “Processos” e “Variantes”. *Benchmarks* classificados como grandes ou médios nesses fatores estão com maior frequência no padrão em que o tempo de resposta da versão valipar-service-6n-8d é menor que o da versão valipar. Simetricamente, *benchmarks* classificados como pequenos em todas as características pertencem com maior frequência ao



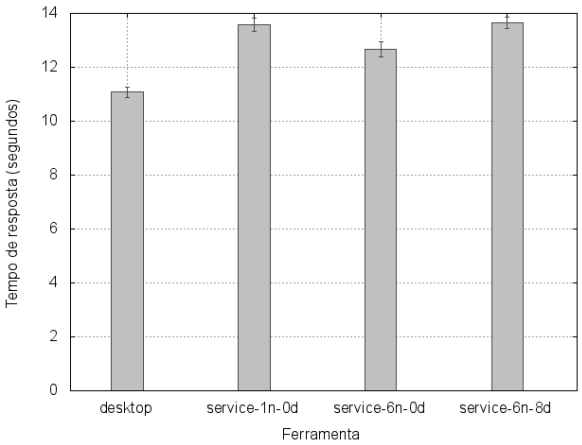
(a) Resultados para o *benchmark* MC



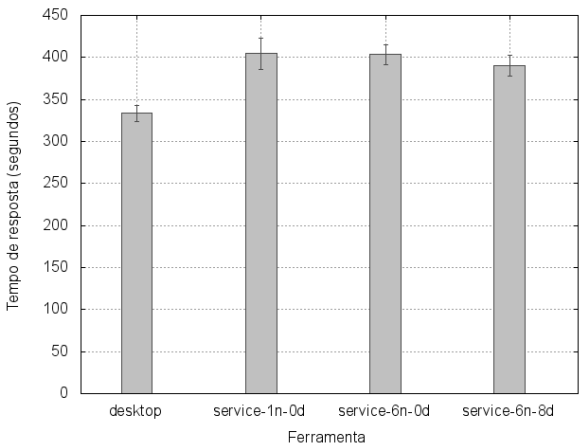
(b) Resultados para o *benchmark* HO



(c) Resultados para o *benchmark* MS



(d) Resultados para o *benchmark* NB

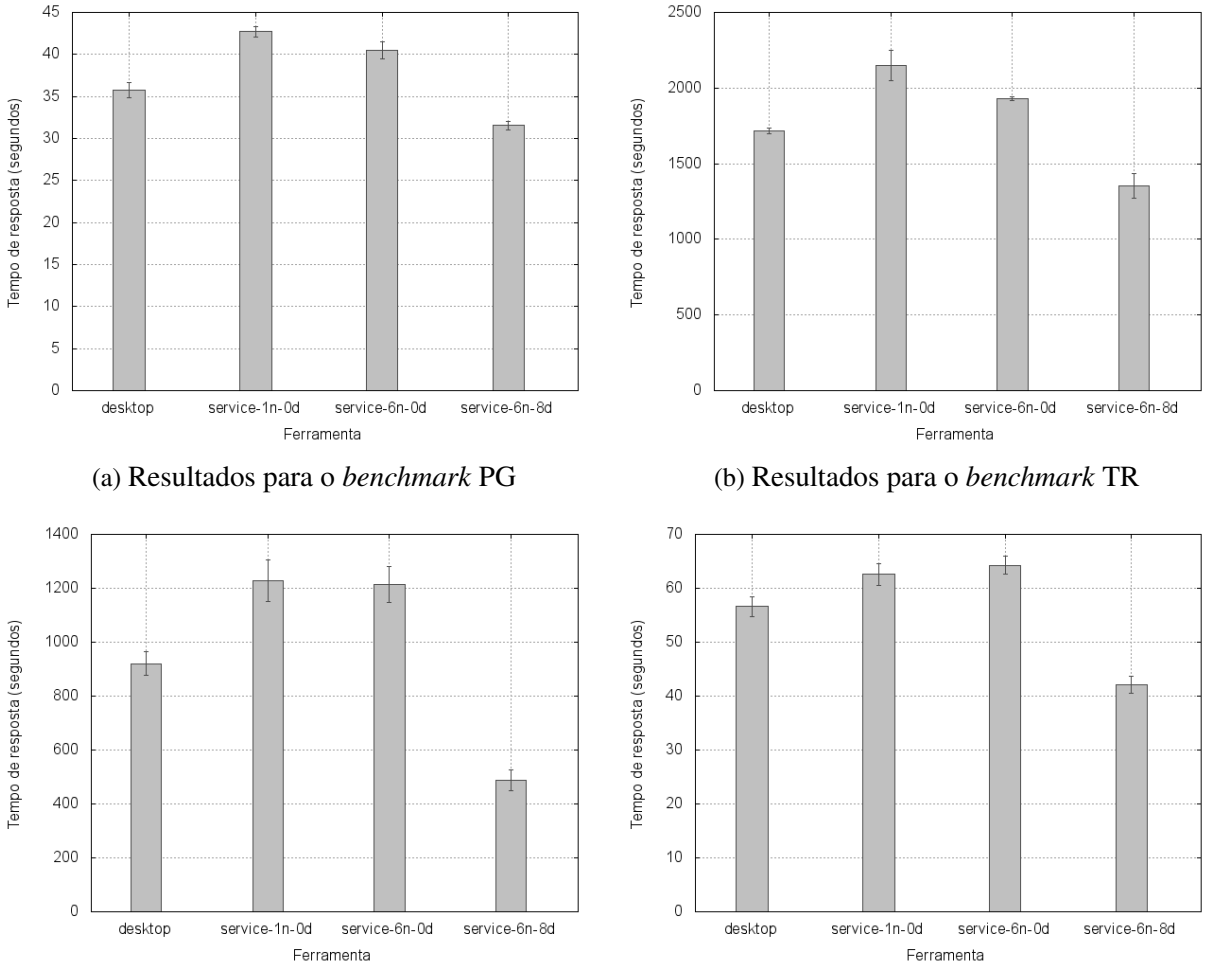


(e) Resultados para o *benchmark* RW

Figura 17 – Resultados obtidos para a execução dos benchmarks MC, HO, MS, NB e RW utilizando diferentes ferramentas.

outro padrão.

A característica “Variantes” tem influência direta no tempo de resposta analisado. De fato, esse número indica quantas execuções (potencialmente independentes) devem ser feitas. Quanto



(a) Resultados para o *benchmark* PG

(b) Resultados para o *benchmark* TR

(c) Resultados para o *benchmark* MA

(d) Resultados para o *benchmark* GT

Figura 18 – Resultados obtidos para a execução dos benchmarks PG, TR, MA e GT utilizando diferentes ferramentas.

maior a independência entre as execuções de variantes, maior o potencial de paralelização das mesmas, o que se beneficia da arquitetura distribuída adotada pela versão valipar-service-6n-8d. As demais características indicam o tamanho e complexidade dos programas, destacando-se as interações possíveis entre *threads* e processos. Quanto maior o valor para essas características, em geral, maior o tempo de resposta na execução do programa.

Além disso, deve-se considerar que o tempo de resposta total da execução de cada *benchmark* utilizando-se cada ferramenta é composto pelo tempo de resposta da execução do programa e, para o caso dos serviços, pelo *overhead* da organização da ferramenta como serviço, isto é, o tempo de serialização e de-serialização de mensagens, latência de rede, distribuição de tarefas e tratamento de protocolos Web. Quanto maior o tempo de resposta da execução de *benchmarks*, maior a participação desse componente no tempo total, o que diminui a influência do *overhead* na atividade como um todo. Isso também contribui para a diminuição da diferença de tempo de resposta entre as ferramentas analisadas.

Tendo-se esse cenário em vista, constatou-se que a utilização da arquitetura distribuída

traz vantagens em casos em que a diferença no tempo de resposta supera o tempo do *overhead*. Por exemplo, caso a paralelização reflita em 10 segundos a menos no tempo de resposta, mas o *overhead* da distribuição aumente o tempo de resposta em 20 segundos, não houve benefício na utilização dessa configuração.

Os resultados obtidos evidenciam o impacto do paradigma de orientação a serviços na implementação de ferramentas de teste estrutural de programas concorrentes. Como observado, a melhoria no tempo de resposta está condicionada à natureza do programa sendo testado e ao modo como os serviços são instalados.

A utilização dos serviços com a arquitetura distribuída para o teste de programas que executam com um maior tempo de resposta e programas que exijam um maior número de variantes a serem executadas oferece um desempenho comparável ou até superior para muitos casos. Independente do cenário, o uso da orientação a serviços traz benefícios para o custo de desenvolvimento de um modo geral, como discutido na Seção 6.3.

A partir dos resultados foi possível observar possíveis pontos que podem ser melhorados de forma a reduzir o tempo de resposta na execução dos serviços de teste. Um dos principais gargalos da composição de serviços está na transferência de dados pela rede entre serviços. Embora não seja possível eliminar esse gargalo por completo, a transferência pode ser mais eficiente. Pode-se utilizar técnicas de compactação para se transferir um volume menor de dados contendo a mesma informação (cabe ressaltar aqui que se substituiria o tempo de transmissão pelo tempo de processamento para a compactação e descompactação).

Além disso, pode-se implementar mecanismos que permitam ao cliente determinar quais informações sobre os artefatos de teste a serem transmitidas pelo serviço, o que potencialmente reduziria a transmissão desnecessária de dados. Essa otimização inclui a paginação de listas de artefatos por exemplo, isto é, o envio de seções de uma coleção para o cliente.

A utilização de tais técnicas é ortogonal aos contratos implementados, ou seja, não afetam conceitualmente o planejamento realizado para o conjunto de serviços desenvolvido. Essas técnicas são aprimoramentos relacionados com a tecnologia utilizada para implementação dos serviços que são aplicadas conforme o sistema amadurece e tais gargalos são percebidos. Essas e outras otimizações podem ser aplicadas em trabalhos futuros.

6.3 Análise Qualitativa dos Serviços

A análise qualitativa dos serviços teve como objetivo a verificação do impacto do desenvolvimento dos serviços em questões como a funcionalidade da ferramenta de teste (Seção 6.3.1), exposição das funcionalidades (Seção 6.3.2) e o custo de desenvolvimento de novas ferramentas (Seção 6.3.3).

Tais aspectos foram analisados de modo qualitativo, apresentando as estratégias adotadas na implementação da ValiPar Service e de outras ferramentas de teste encontradas na literatura para abordá-los. Não houve, no entanto, comparação direta entre as ferramentas de teste, uma vez que tratam de técnicas de teste e fases de teste distintas, possuem focos diversos, e nenhuma das ferramentas encontradas tratam do teste estrutural de programas concorrentes.

6.3.1 Funcionalidade da Ferramenta

A questão da funcionalidade de um serviço de teste pode ser relacionada com sua abrangência, ou seja, a capacidade da ferramenta executar todas as atividades primárias do teste proposto. Serviços mais abrangentes disponibilizam conjuntos mais completos de funcionalidades da atividade de teste para execução remota. Esse aspecto pode ter impacto em questões de manutenabilidade e escalabilidade da atividade de teste. A ValiPar Service abrange todas as atividades requeridas da atividade de teste estrutural de programas concorrentes, de acordo com a ontologia utilizada (OntoTest).

Como explicado no capítulo 5, o desenvolvimento dessa ferramenta foi baseado em um estudo das atividades presentes no teste estrutural de programas concorrentes que resultou na extensão de uma ontologia e na definição dos processamentos essenciais para o teste. Tal ontologia, representada na Figura 12, determina que um passo de teste é composto por uma série de atividades de teste que, por sua vez, podem ser primárias, organizacionais e de suporte. Dentre as atividades primárias estão a geração de casos de teste, manipulação de artefatos, geração de elementos requeridos, análise de cobertura e execução de casos de teste.

Como evidenciado no experimento da Seção 6.2, todos os processamentos (que instanciam as atividades de teste) são executados de modo totalmente remoto por meio de contratos que disponibilizam todas as funcionalidades. De fato, o serviço ValiInst cobre as atividades de manipulação de artefatos (programa sendo testado, programa instrumentado e grafo de fluxo de controle paralelo). O serviço ValiElem, por sua vez, gera os elementos requeridos enquanto o serviço ValiEval faz análise de cobertura de critérios.

A geração e execução (determinística e não-determinística) de casos de teste são atividades cobertas pelo serviço ValiExec e ValiSync. Tais serviços implementados na ferramenta cobrem todos os requisitos para o teste estrutural de programas concorrentes, dentro do projeto TestPar. No entanto, os requisitos para a execução desse passo como um processamento remoto pode variar de acordo com a natureza do programa sendo testado.

Uma extensão possível a esse serviço seria o suporte para a execução de programas concorrentes com acesso a sistemas de gerenciamento de bancos de dados (SGBDs) específicos como PostgreSQL e MySQL (GROUP, 2016; TEAM, 2016b). A ValiPar Service atualmente apenas permite que o programa testado utilize bancos de dados embutidos (*embedded*) como o

H2 para Java (TEAM, 2016a), os quais não utilizam sistemas externos.

A execução remota de casos de teste também traz questões de segurança a serem resolvidas. Segurança no teste de software envolve a restrição de recursos computacionais para o usuário para mitigar possíveis ataques. Esse não foi o foco deste trabalho de mestrado, mas deverá ser abordado em trabalhos futuros. Atualmente o acesso à ferramenta é restrito para usuários confiáveis para endereçar essas questões.

Outras ferramentas de teste de software como serviço também abordam a questão de funcionalidade em diversos níveis. Todas as ferramentas estudadas realizam todo o processamento estático, como instrumentação e análise de cobertura, de modo remoto (o que varia de acordo com o tipo de teste).

A questão da execução remota de casos de teste não é abordada pela ferramenta JaBUTi-Service devido a questões de configuração de ambiente para programas que exigem bancos de dados e bibliotecas externas, o que pode se tornar um processo complexo. Para essa ferramenta, a execução de casos de teste deve ser realizada localmente pelo cliente (ELER *et al.*, 2009). Todos os demais processamentos podem ocorrer remotamente.

Os serviços Cloud9, PathCrawler e PascalMutants Service permitem também a execução remota de todas as atividades. A execução remota de casos de teste é abordada com estratégias de segurança. São utilizadas estratégias como restrição do uso de rede, de sistema de arquivos, execução em máquinas virtuais isoladas e restrição no número de processos criados.

6.3.2 Exposição de Funcionalidades

O modo como as funcionalidades são expostas é também um ponto essencial para possibilitar a adoção da ferramenta de teste dentro do ciclo de desenvolvimento de software. Em termos práticos, isso se traduz em dois aspectos relacionados com os contratos dos serviços: a capacidade de descoberta (*discoverability*) do serviço e o nível de abstração disponibilizado pelo mesmo.

É desejável que serviços implementados sejam suplementados com meta dados comunicativos, pelos quais os serviços podem ser efetivamente descobertos e interpretados (ERL, 2005). Além disso, os contratos resultantes devem apenas conter informações essenciais e as informações sobre o serviço devem ser limitadas ao que é publicado pelos contratos.

A ValiPar Service aborda a questão da abstração através da definição de contratos que expõem apenas informações do domínio da aplicação, ou seja, os contratos descrevem artefatos de teste e parâmetros para execução de processamentos relacionados ao mesmo. O sistema como um todo é visto como uma caixa preta, sem que detalhes internos, como *flags* ou outras condições específicas, sejam expostos para o usuário. Além disso, não há contratos não documentados que

possam ser utilizados por clientes.

Isso permite mudanças e aprimoramentos internos em como cada processamento é realizado pelo serviço sem alterações no cliente. Dessa forma, uma possível alteração no cliente só é necessária quando há alterações no domínio da aplicação ou no contrato. Essa estratégia endereça diretamente questões de manutenabilidade, reduzindo consideravelmente o custo de atualização da base de clientes devido à modificação da ferramenta para adição de novas funcionalidades e correção de defeitos.

A interface dos serviços JaBUTiService e PascalMutants Service, detalhadas em (ELER *et al.*, 2009; OLIVEIRA, 2011), também não apresentam exposição de detalhes de implementação e especifica somente os artefatos a serem gerados ou guardados, e as estratégias empregadas nas operações disponíveis. Os serviços Cloud9 e PathCrawler não detalham seus contratos nos artigos publicados.

A capacidade de descoberta de serviços é abordada na ValiPar Service por meio do conceito de *hypermedia* do estilo arquitetural REST. Esse conceito está presente na Web como um todo e consiste na disponibilização de informações e o relacionamento das mesmas por meio de *links*.

A ValiPar Service disponibiliza todos os artefatos de teste como informações e indica seus relacionamentos com outros artefatos como URLs. Dessa forma, o cliente precisa apenas saber qual a URL inicial do serviço para explorar novos recursos através das URLs disponibilizadas.

Como exemplo, o cliente pode inicialmente acessar a sessão de teste criada (*valipar.icmc.usp.com/sessions/1*). Como resultado, são disponibilizadas URLs para o PCFG (*valipar.icmc.usp.com/pcfgs/1*), para os elementos requeridos (`valipar.icmc.usp.com/required_elements/1`) e para os casos de teste (`valipar.icmc.usp.com/testcases/1`, por exemplo). Ao requisitar os casos de teste por essa URL, o cliente recebe como resposta, além de informações como entrada e saída esperada, um conjunto de URLs para as execuções de caso de teste (`valipar.icmc.usp.com/executions/1`, por exemplo).

Essa rede de conteúdos e links entre os mesmos elimina a necessidade de se implementar no cliente a lógica de construção de URLs para acessar recursos. O próprio serviço se responsabiliza por guiar o cliente pelos artefatos que oferece. Essa estratégia também traz consequências diretas de manutenabilidade para a ferramenta de teste.

O serviço pode assumir que o cliente irá se guiar pelas URLs oferecidas e, assim, possui certa independência para modificar suas interfaces. Por exemplo, o serviço pode modificar o padrão da URL de acesso a casos de teste de (`valipar.icmc.usp.com/testcases/<ID>` para (`valipar.icmc.usp.com/sessions/<ID>/testcases/<ID>` de forma transparente para o cliente, que apenas sabe como identificar tal URL na mensagem.

Os demais serviços pesquisados oferecem metadados utilizando outra estratégia: a descrição das funcionalidades do serviço no formato WSDL (*Web Service Description Language*).

Esse formato descreve a interface dos serviços incluindo quais os tipos de dados de entrada e de saída. Essa estratégia permite o estabelecimento de um contrato entre cliente e serviço e a interpretação das funcionalidades do mesmo para a criação de clientes.

6.3.3 Custo de Desenvolvimento de Novas Ferramentas de Teste

O modo como os serviços são implementados para diminuir o custo do desenvolvimento de software também é um ponto importante a ser avaliado. Essa questão envolve principalmente a reusabilidade. Desse modo, serviços devem conter e expressar a lógica que pode ser usada em diversos contextos (ERL, 2005). Além disso os serviços devem possibilitar a criação de diferentes composições, independente de tamanhos e complexidades. Esses são aspectos importantes para o teste de software uma vez que diversos processamentos são comuns a vários tipos de teste ou ainda são ortogonais à atividade de teste como um todo.

A ValiPar Service aborda este aspecto por meio da divisão dos processamentos da atividade de teste em serviços especializados com contratos flexíveis. Como explicado no capítulo 5, a divisão dos serviços levou em consideração o acoplamento existente entre os mesmos. Os seis serviços resultantes permitem que um cliente realize todo fluxo de execução típico do teste estrutural de programas concorrentes por meio do serviço ValiPar Service ou ainda crie seu próprio fluxo, utilizando um ou mais serviços especializados.

A questão da reusabilidade foi verificada por meio de um experimento. Foram desenvolvidos casos de teste automatizados que se comportam como clientes dos serviços desenvolvidos. Cada caso de teste exercita uma operação de modo individual e independente das demais (o conjunto total de operações exercitadas pode ser observado na Seção 5.4). Dessa forma, para cada caso, são fornecidas as entradas necessárias para a execução da operação e o resultado da mesma é verificado em seguida.

Foi possível verificar a independência fornecida pelos contratos e pela divisão das funcionalidades em serviços distintos. De fato, cada operação desenvolvida necessita apenas que os artefatos requeridos sejam devidamente adicionados ao serviço para que o processamento possa ocorrer. Os serviços não possuem noção da procedência dos artefatos de entrada e não exigem que os mesmos sejam gerados por determinada ferramenta ou serviço.

Como consequência direta, os contratos dos serviços possuem um alto grau de reuso e possibilitam o desenvolvimento de fluxos de execução para finalidades diversas não previstas durante o desenvolvimento dos serviços. Além do fluxo de execução que engloba todas as operações desenvolvidas, é possível ainda utilizar subconjuntos de operações juntamente a outras ferramentas para contextos específicos.

Um caso de uso válido envolve o teste de um programa concorrente de grande porte que necessite de um ambiente complexo para execução, não disponível no serviço ValiExec.

Para esse caso, pode-se desenvolver uma ferramenta que utilize todos os serviços, exceto o de execução. Dessa forma, todo o processamento estático é realizado remotamente. A ferramenta, nesse sentido, faria o *download* dos artefatos necessários à execução, executaria localmente e submeteria os arquivos de rastro para análise de cobertura remota.

Outro exemplo se refere à geração de elementos de modelo de teste. Ferramentas de teste estrutural de programas sequenciais e concorrentes podem utilizar o serviço ValiInst para analisar o programa sendo testado e adaptar o grafo de fluxo de controle e informações sobre o fluxo de dados e de sincronização para suas necessidades. As operações fornecidas pelo serviço ValiInst são totalmente desacopladas de outros processos como geração de elementos requeridos e execução.

O serviço de geração de elementos requeridos, por sua vez, pode derivar elementos requeridos com base em qualquer PCFG, desde que esteja no formato estabelecido pelo projeto TestPar. Nesse sentido, caso outra ferramenta necessite dessa funcionalidade, é necessário apenas que ela utilize esse formato ou faça conversão de formatos para interagir com o serviço ValiElem apropriadamente. Como resultado do processamento, o cliente recebe um conjunto de elementos requeridos suportados pelo serviço. Esses dados podem assim integrar outros tipos de elementos requeridos não previstos pelo serviço ValiElem, gerados por outras ferramentas de modo complementar.

Como último exemplo, pode-se considerar a avaliação de critérios de teste feita pelo serviço ValiEval. Uma ferramenta que gere arquivos de rastro em conformidade com o formato estabelecido pelo projeto TestPar e faça uso de um ou mais critérios suportados pelo serviço ValiEval pode utilizar o mesmo de modo independente de outros processos oferecidos pelo conjunto de serviços.

Todos esses casos de uso dos serviços apresentam o potencial de redução de custo de desenvolvimento e manutenção e aumento na qualidade de ferramentas de teste de software. Vários processos são recorrentes em vários cenários e muitos deles não são triviais de serem implementados, o que torna o desenvolvimento de novas ferramentas de teste custoso e repetitivo. A estratégia de disponibilização da ferramenta ValiPar em um conjunto de serviços independentes traz efetivamente os benefícios da orientação a serviços para o teste estrutural de programas concorrentes e permite a extensão de ferramentas inicialmente não consideradas.

Outros serviços pesquisados abordam com a questão do reuso em diversos níveis. Os contratos da JaBUTiService expressam uma lógica em um mesmo nível semântico dos contratos da ValiPar Service, ou seja, expõem e operam artefatos com interfaces similares. As interfaces divergem no estilo arquitetural utilizado e na extensão feita pela ValiPar para programas concorrentes. Apesar das similaridades dos contratos, existem diferenças no modo como as mesmas são usadas.

Diferentemente da ValiPar Service, a JaBUTiService permite a execução de apenas um

fluxo definido. Nesse sentido, o serviço impõe necessariamente um fluxo de execução que é iniciado com a geração de artefatos estáticos e é finalizado com a avaliação de critérios de teste. Os processamentos intermediários não são disponibilizados de modo independente, o que impede a aplicação dos casos de uso discutidos acima.

O serviço PascalMutants-Service trata da questão do reuso de modo similar à ValiPar Service. Esse serviço atribui tarefas especificas dentro da atividade de teste utilizando mutantes a outros serviços realizando uma composição. Tais serviços também podem ser utilizados de modo individual em outros contextos relacionados ao teste de mutantes de programas em Pascal.

Tal conjunto de serviço, apesar de tratar de uma técnica de teste diferente e ter como alvo programas sequencias em Pascal, possui uma organização similar a da ValiPar Service. Isso se deve principalmente pelo fato de esse conjunto ser desenvolvido como validação (como descrito na Seção 4.4.3) em uma arquitetura de referência que, por sua vez, é baseada na ontologia OntoTest, também utilizada neste projeto.

Os serviços Cloud9 e PathCrawler são projetados como grandes serviços com todas as funcionalidades. Eles possuem contratos que restringem as possibilidades do cliente para apenas uma operação. Para esses casos, todos os parâmetros para a execução de todas as etapas do teste são especificados e submetidos em apenas um requisição inicial. Portanto, tais sistemas não foram projetados visando a reutilização de subprodutos intermediários das atividades realizadas, o que afeta o desenvolvimento de novas ferramentas ou a adaptação do teste para casos específicos.

6.4 Considerações Finais

Este capítulo contém os principais aspectos qualitativos e quantitativos relacionados ao desenvolvimento da ferramenta ValiPar Service. Do ponto de vista quantitativo foi possível verificar o impacto de fatores como latência de rede e serialização de dados no tempo de resposta da atividade de teste estrutural de programas concorrentes. Desse modo, a atividade de teste é altamente impactada pelo modo como os serviços são disponibilizados e pela natureza do programa sendo testado.

Nota-se, no entanto, que o acréscimo no tempo de resposta é em geral aceitável e não inviabiliza a atividade como um todo. Considerando a versão paralela da ValiPar Service e a ferramenta *desktop*, houve um acréscimo de até 7 segundos no tempo de resposta para 3 experimentos com tempo de resposta inferior a 20 segundos. 2 experimentos com tempos de resposta superiores a 50 segundos apresentam um acréscimo de 15 a 25% no tempo de resposta. Os demais experimentos realizados apresentam diminuição no tempo de resposta.

Do ponto de vista qualitativo foi possível notar os benefícios trazidos pelas decisões de projeto apresentadas no Capítulo 5. Os requisitos centrados em aspectos de reuso, funcionalidade e manutenção foram atingidos e a ferramenta resultante apresenta contribuições significativas

para o teste estrutural de programas concorrentes e para o paradigma de teste como serviço na Web. Os nossos resultados mostraram que os serviços possuem um alto grau de independência e flexibilidade, permitindo o reuso de componentes para fins diferentes e a construção de fluxos de trabalhos não previstos inicialmente.

Foi possível notar também as limitações da ferramenta desenvolvida e de outras relacionadas em diversos aspectos na área de ferramentas como serviços na Web. O tópico de segurança para o teste de software representa uma vasta área de pesquisa. A questão da configuração do ambiente para execução remota também representa um desafio, uma vez que esse aspecto varia de programa para programa e é mais complexo para programas de grande porte.

Além das questões de desempenho, reusabilidade, manutenção e funcionalidade (endereçados neste projeto), segurança e configuração de ambiente são altamente desejáveis e são fatores decisivos para adoção de serviços, principalmente no contexto da indústria. Esses tópicos precisam ser endereçados em trabalhos futuros.

CAPÍTULO

7

CONCLUSÃO

7.1 Considerações Iniciais

Este capítulo apresenta as conclusões acerca deste trabalho. A Seção 7.2 caracteriza a pesquisa realizada. A Seção 7.3 destaca as principais contribuições desta pesquisa. A Seção 7.5 destaca toda a produção científica resultante deste trabalho durante o período do projeto. Por fim, a Seção 7.6 lista possíveis melhorias e trabalhos futuros.

7.2 Caracterização da Pesquisa Realizada

O teste de software é considerado uma atividade de grande importância dentro do ciclo de desenvolvimento de software. O teste como área de pesquisa exige o desenvolvimento de modelos, critérios e ferramentas de teste para atingir o objetivo de garantir qualidade ao processo de desenvolvimento de software. O teste estrutural de programas concorrentes, em especial o desenvolvido pelo projeto TestPar, hoje conta com um conjunto de ferramentas *desktop* que possibilitam a automatização do processo de extração de elementos requeridos, produção de rastros de execução e avaliação de critérios.

Um dos desafios do teste de software em geral é possibilitar a automatização da atividade de teste a um custo operacional aceitável. O custo, neste caso, está relacionado com questões de manutenção, desempenho e reusabilidade. Estas questões são diretamente abordadas pelo paradigma de orientação a serviços. Nesse contexto, as funcionalidades fornecidas pela ferramenta são disponibilizadas como serviços que centralizam o acesso às mesmas. Dessa forma, usuários devem requisitar operações aos serviços, que atenderá a requisição de forma remota.

Essa abordagem endereça tais desafios uma vez que facilita atualização de ferramentas e a disponibilização de funcionalidades desenvolvidas para fins diversos. Além disso, há ainda a possibilidade de diminuição no tempo de resposta da atividade por meio da distribuição de tarefas para um conjunto de nós, por exemplo.

Este projeto de pesquisa visou apoiar a criação de ferramentas para o teste estrutural de programas concorrentes utilizando-se a orientação a serviços para solucionar os desafios citados. Dessa forma, tal atividade de teste foi organizada em um conjunto de seis serviços independentes que tratam aspectos específicos da atividade de forma coordenada para realizar a execução totalmente remota do teste. Tais serviços possuem contratos e capacidades bem definidos, expondo as dependências existentes entre os serviços e o modo como artefatos de teste são utilizados. Os serviços foram implementados e disponibilizados na forma da ferramenta ValiPar Service.

7.3 Contribuições

Como principal contribuição deste projeto, destaca-se a definição de um conjunto de serviços e seus contratos para o teste estrutural de programas concorrentes. O processo de definição levou em consideração uma ontologia de teste de software. Nesse sentido, os processamentos e artefatos do teste estrutural de programas concorrentes foram modelados de acordo com o vocabulário e relacionamentos existentes na ontologia já validada.

Todo esse estudo resultou em um conjunto de definições que podem, inclusive, servir como base para o desenvolvimento de novas ferramentas de teste e de ferramentas complementares à composição existente. Isso auxilia na popularização dessa abordagem de desenvolvimento que se beneficia do paradigma de orientação a serviços para aumentar a qualidade do desenvolvimento de ferramentas e diminuição de custos de uma forma geral.

O conjunto de serviços e seus contratos foram implementados com o propósito de validação e disponibilização. Tais serviços utilizam as funcionalidades já existentes na ferramenta *desktop* contextualizando-as para o ambiente de execução remota. Os contratos foram implementados de forma a seguir o estilo arquitetural REST, trazendo benefícios como a possibilidade futura de se interagir com os serviços por meio de navegadores Web.

O foco da definição e implementação foi em fatores como reusabilidade, manutenabilidade e funcionalidade. Além disso houve uma preocupação quanto ao impacto do *overhead* trazido pela implementação de serviços no tempo de resposta da atividade. Tais fatores foram analisados de forma qualitativa e quantitativa utilizando-se como base os contratos definidos e a implementação do conjunto de serviços.

Pela análise quantitativa foi possível verificar que o tempo de resposta é afetado pela

implementação dos serviços devido à latência de rede e ao processo de serialização de dados para interação entre os serviços. Esse impacto, no entanto, não inviabiliza o teste estrutural de programas concorrentes e pode ser mitigado utilizando estratégias de distribuição (ou paralelização) de tarefas, como foi demonstrado na análise.

De forma qualitativa, o conjunto de serviços ValiPar Service foi analisado de acordo com o foco definido no cenário atual de teste de software como serviço. Apesar de não poder haver uma comparação direta entre os serviços analisados, foi possível verificar que os serviços que compõem a ValiPar Service atingem seus objetivos e trazem colaborações para essa área uma vez que executam toda a atividade de teste de forma remota e permitem reutilizar funcionalidades em contextos não inicialmente previstos por meio de contratos flexíveis.

Os serviços resultantes estão atualmente disponíveis para testadores do projeto TestPar, com perspectivas de disponibilização para outros grupos de pesquisa, para indústria e ensino. Dessa forma os serviços (e consequentemente os critérios e modelos de teste estrutural de programas concorrentes desenvolvidos pelo projeto TestPar) poderão ser integrados ao ciclo de desenvolvimento de software de forma total ou parcial, diminuindo o custo de desenvolvimento e aumentando a adoção e divulgação do teste de programas concorrentes.

7.4 Dificuldades e Limitações

As dificuldades e limitações deste trabalho estão principalmente relacionadas à validação dos serviços de teste. De fato dois fatores dificultam uma validação mais ampla e efetiva da ferramenta. Primeiramente, não se tem conhecimento de outras ferramentas de teste estrutural de programas concorrentes como serviço que possibilitem a comparação direta do tempo de resposta, do conjunto de funcionalidades e decisões de projeto tomadas. As ferramentas encontradas, apesar de endereçarem o teste de software como serviço, implementam tipos diferentes de teste que geram demandas diferentes para a definição de serviços e contratos, o que dificulta a comparação direta.

Além disso, não foi possível desenvolver métodos para se validar os fatores que fazem parte do foco deste projeto. De fato, é difícil estabelecer meios concretos para avaliar a funcionalidade, reusabilidade e manutenabilidade de modo quantitativo, através de experimentos sistemáticos. Por esse motivo, neste trabalho foi analisado se a implementação resultante atinge de forma satisfatória os requisitos estabelecidos, apontando as suas contribuições, dado o contexto atual do teste como serviço.

Houve dificuldades também na fase de mapeamento sistemático da área de teste como serviço. O termo "*Test as a Service*" ou TaaS é atualmente utilizado em dois contextos próximos, mas distintos. Além do contexto deste trabalho, ou seja, do de ferramentas de teste desenvolvidas como serviços, esse termo também é utilizado no contexto de teste de serviços Web e teste de ambientes de computação em nuvem. Isso dificultou o processo de busca e resultou em diversos

falsos positivos nos resultados.

Por fim, a ferramenta resultante ainda não suporta a execução de programas que utilizem ambientes de execução complexos e também não possui mecanismos de segurança específicos para a atividade de teste. Tais aspectos não foram foco deste trabalho uma vez que são tópicos extensos o suficiente para serem tratados individualmente em projetos de pesquisa específicos. Ha porém a necessidade de se atender a tais questões para que a ferramenta seja amplamente adotada por outros projetos de pesquisa e indústria.

7.5 Produção Científica

Durante o período deste projeto de mestrado, os seguintes artigos foram submetidos e aceitos:

1. Souza, P. S. L. ; Souza, S.R.S. ; Rocha, M.G. ; **PRADO, R.R.** ; Batista, R. N. . Data flow testing in concurrent programs with message passing and shared memory paradigms. In: International Conference on Computational Science (ICCS2013), 2013, Barcelona. Procedia Computer Science. Amsterdam: Elsevier, 2013. v. 18. p. 149-158. - Qualis A1

   Esse artigo apresenta uma extensão para o modelo e critérios de teste do projeto TestPar para possibilitar o teste de programas concorrentes que fazem uso de ambos os paradigmas de comunicação e sincronização.

   Apesar de não estar posicionada no centro deste projeto de mestrado, essa publicação foi fundamental para o desenvolvimento das ferramentas de teste estrutural de programas concorrentes que sustentam os serviços criados. Houve a participação tanto na definição dos elementos da extensão apresentada quanto no processo de validação, em que foi construído um protótipo para a geração de elementos do modelo.

2. **PRADO, R.R.** ; Souza, P. S. L. ; Dourado, G. G. M. ; Souza, S.R.S. ; Estrella, J.C. ; Bruschi, S.M. ; Lourenço, J.M.S . Extracting Static and Dynamic Structural Information from Java Concurrent Programs for Coverage Testing. In: XLI Conference Latino Americana de Informática, 2015, Arequipa. Proceedings of CLEI2015. Arequipa: CLEI, 2015. v. 1. p. 66-73. - Qualis B4

   Esse artigo apresenta técnicas de extração de informações estáticas e dinâmicas de programas concorrentes em Java para a geração de elementos requeridos e de rastros de execução para o teste de cobertura. Esse trabalho resultou no modulo ValiInst para Java, tendo impacto direto no desenvolvimento da ferramenta *desktop* e dos serviços.

3. Dourado, G. G. M. ; Souza, P. S. L. ; **PRADO, R.R.** ; Batista, R. N. ; Souza, S.R.S. ; Estrella, J.C. ; Bruschi, S.M. ; Lourenço, J.M.S.;A Suite of Java Message-Passing Bench-

marks to Support the Validation of Testing Models, Criteria and Tools. In: International Conference on Computational Science (ICCS2016), 2016, San Diego.

Esse artigo apresenta *benchmarks* de passagem de mensagem com e sem defeitos para o teste de programas concorrentes. Apesar de não estar diretamente relacionado com este projeto de mestrado, o desenvolvimento e validação desses *benchmarks* auxiliou e teve grande impacto no amadurecimento das ferramentas de teste do projeto ValiPar, incluindo os serviços que compõem a ValiPar Service. Os *benchmarks* utilizados na validação deste trabalho são também apresentados nesse artigo.

4. **PRADO, R.R.** ; Souza, P. S. L. ; Souza, S.R.S. ; Dourado, G. G. M. ; Batista, R. N. ; ValiPar Service: Structural Testing of Concurrent Programs as a Web Service Composition. In: 13th International Conference on Information Technology : New Generations (ITNG2016), 2016, Las Vegas, Nevada. Springer, 2016. Qualis B1

   Por fim, esse artigo apresenta o processo de definição, desenvolvimento e validação dos serviços que compõem a ValiPar Service, como descrito de forma detalhada nos Capítulos 5 e 6. Dessa forma, o escopo desse artigo está diretamente relacionado aos trabalhos desenvolvidos neste projeto.

7.6 Trabalhos Futuros

Este projeto de pesquisa abre possibilidades para o desenvolvimento de novas pesquisas na área de teste de software e desenvolvimento de software como um todo. Uma lista de possíveis contribuições futuras pode ser encontrada abaixo:

- **Extensão do serviço ValiInst para novas linguagens:** O serviço ValiInst atualmente fornece o processo de instrumentação e geração de PCFG para programas em Java. O suporte a linguagens como C utilizando bibliotecas como Pthreads, MPI e PVM amplia a abrangência dos serviços e pode aumentar a adoção da mesma. Esse trabalho envolve possivelmente a extensão de contratos para possibilitar a descrição de programas que usam mais de uma linguagem de programação, por exemplo.

- **Extensão do serviço ValiExec para novos ambientes de execução:** Atualmente o serviço ValiExec estabelece estaticamente o modo como o programa será executado. Essa estratégia não cobre programas de grande porte que necessitam de ambientes complexos para execução. Dentre os aspectos a serem considerados estão o uso de bancos de dados e bibliotecas específicas. Este trabalho envolve a especialização de contratos para descrever ambientes de execução, além do desenvolvimento de mecanismos para preparação de ambiente e execução.

- **Aprimoramento das interações realizadas entre serviços e clientes:** A validação do conjunto de serviços feita neste trabalho identificou potenciais pontos de melhorias relacionados à latência de rede e à serialização de mensagens (descritos no Capítulo 6), os quais podem reduzir o consumo de recursos e o tempo de resposta da atividade de teste como um todo.

- **Implementação de medidas de segurança:** Não foi o foco deste projeto de pesquisa fornecer mecanismos para garantir uma execução do teste segura. Esse tópico é extenso e envolve questões de criptografia, confidencialidade, autenticidade e níveis de acesso a artefatos de teste. O cumprimento dessa tarefa tem potencial para ampliar a adoção dessa ferramenta de teste, principalmente pela indústria.

- **Desenvolvimento de interfaces gráficas:** Os serviços são, atualmente acessíveis por meio de um cliente de linha de comando, como ocorre na versão *desktop*. Embora cubra a demanda atual, o desenvolvimento de um cliente acessível por meio de um navegador Web ampliaria o acesso de testadores à ferramenta, uma vez que não haveria a necessidade de instalação da mesma. Além disso, tal interface gráfica deixaria a atividade de teste mais dinâmica. O processo de desenvolvimento deve levar em consideração conceitos de interação humano-computador (*Human-Computer interaction* (HCI)).

- **Ensino do teste estrutural de programas concorrentes:** Outro desdobramento possível é a utilização dos serviços para a criação de objetos de aprendizagem para o ensino de programação concorrente e de teste de software. Há a possibilidade de se realizar estudos acerca dos enganos mais comuns cometidos por programadores, assim como se testar a efetividade de novos modelos e critérios. Essa atividade pode também aprimorar o desenvolvimento dos serviços e de outras ferramentas uma vez que a utilização em massa da ferramenta pode expor defeitos e limitações. Por fim, pode haver contribuições na forma de novos programas com e sem defeitos para integrar os *benchmarks* já desenvolvidos pelo projeto TestPar.

REFERÊNCIAS

ALONSO, G.; CASATI, F.; KUNO, H.; MACHIRAJU, V. **Web Services: Concepts, Architectures and Applications**. [S.l.]: Springer, 2004. (Data-Centric Systems and Applications). ISBN 9783540440086. Citado 2 vezes nas páginas 39 e 40.

BARBOSA, E. F.; CHAIM, M. L.; VICENZI, A. M. R.; DELAMARO, M. E.; JINO, M.; MALDONADO, J. C. Introdução ao teste de software. In: . [S.l.]: CAMPUS, 2007. cap. Teste Estrutural, p. 47–76. ISBN 9788535226348. Citado 2 vezes nas páginas 25 e 38.

BARBOSA, E. F.; NAKAGAWA, E. Y.; MALDONADO, J. C. Towards the establishment of an ontology of software testing. In: ZHANG, K.; SPANOUDAKIS, G.; VISAGGIO, G. (Ed.). **SEKE**. [S.l.: s.n.], 2006. p. 522–525. ISBN 1-891706-18-7. Citado 5 vezes nas páginas 13, 54, 55, 58 e 59.

BARBOSA, E. F.; NAKAGAWA, E. Y.; RIEKSTIN, A. C.; MALDONADO, J. C. Ontology-based development of testing related tools. In: **SEKE**. [S.l.]: Knowledge Systems Institute Graduate School, 2008. p. 697–702. ISBN 1-891706-22-5. Citado 5 vezes nas páginas 13, 54, 55, 56 e 57.

BORGES, V. A.; BARBOSA, E. F. Using ontologies for modeling educational content. In: **Proc. 7th International Workshop on Ontologies and Semantic Web for E-Learning**. [S.l.: s.n.], 2009. Citado 4 vezes nas páginas 13, 55, 56 e 57.

CARVER, R.; LEI, J. A stateful approach to testing monitors in multithreaded programs. In: **High-Assurance Systems Engineering (HASE), 2010 IEEE 12th International Symposium on**. [S.l.: s.n.], 2010. p. 54–63. ISSN 1530-2059. Citado na página 30.

CIORTEA, L.; ZAMFIR, C.; BUCUR, S.; CHIPOUNOV, V.; CANDEA, G. Cloud9: a software testing service. **SIGOPS Oper. Syst. Rev.**, ACM, New York, NY, USA, v. 43, n. 4, p. 5–10, jan. 2010. ISSN 0163-5980. Citado 3 vezes nas páginas 3, 46 e 47.

CONSORTIUM, W. **SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)**. 2007. Disponível em: <http://www.w3.org/TR/soap12-part1/>. Citado na página 45.

CONSORTIUM, W. W. W. **Web Services Architecture**. 2004. Disponível em: <http://www.w3.org/TR/ws-arch/>. Citado 3 vezes nas páginas 13, 41 e 42.

COULOURIS, G. F.; DOLLIMORE, J.; KINDBERG, T.; BLAIR, G. **Distributed Systems: Concepts and Design**. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2011. ISBN 0-13-214301-1. Citado 4 vezes nas páginas 9, 19, 39 e 44.

DELAMARO, M. E.; BARBOSA, E. F.; VICENZI, A. M. R.; MALDONADO, J. C. Introdução ao teste de software. In: . [S.l.]: CAMPUS, 2007. cap. Teste de Mutação, p. 77–118. ISBN 9788535226348. Citado na página 27.

DELAMARO, M. E.; MALDONADO, J. C.; JINO, M. In: **Introdução ao teste de software**. [S.l.]: CAMPUS, 2007. cap. Conceitos Básicos, p. 1–8. ISBN 9788535226348. Citado na página 22.

DEMILLO, R. A.; LIPTON, R. J.; SAYWARD, F. G. Hints on test data selection: Help for the practicing programmer. **Computer**, IEEE, n. 4, p. 34–41, 1978. Citado na página 28.

DOURADO, G. G. M. **Contribuindo para a avaliação do teste de programas concorrentes: uma abordagem usando benchmarks**. [S.l.]: Universidade de São Paulo, 2015. Citado na página 74.

EDELSTEIN, O.; FARCHI, E.; GOLDIN, E.; NIR, Y.; RATSABY, G.; UR, S. Framework for testing multi-threaded java programs. **Concurrency and Computation: Practice and Experience**, v. 15, n. 3-5, p. 485–499, 2003. Citado na página 32.

ELER, M.; ENDO, A.; MASIERO, P.; DELAMARO, M.; MALDONADO, J.; VINCENZI, A.; CHAIM, M.; BEDER, D. Jabutiservice: A web service for structural testing of java programs. In: **Software Engineering Workshop (SEW), 2009 33rd Annual IEEE**. [S.l.: s.n.], 2009. p. 69–76. ISSN 1550-6215. Citado 7 vezes nas páginas 13, 3, 46, 49, 50, 81 e 82.

ENDO, A.; SIMAO, A. da; SOUZA, S.; SOUZA, P. Web services composition testing: A strategy based on structural testing of parallel programs. In: **Practice and Research Techniques, 2008. TAIC PART '08. Testing: Academic Industrial Conference**. [S.l.: s.n.], 2008. p. 3–12. Citado na página 2.

ERL, T. **Service-Oriented Architecture: Concepts, Technology, and Design**. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2005. ISBN 0131858580. Citado 2 vezes nas páginas 81 e 83.

FABBRI, S. C. P.; VICENZI, A. M. R.; MALDONADO, J. C. Introdução ao teste de software. In: **Introdução ao teste de software**. [S.l.]: CAMPUS, 2007. cap. Teste Funcional, p. 27–46. ISBN 9788535226348. Citado na página 24.

FIELDING, R. T. **Architectural Styles and the Design of Network-based Software Architectures**. Tese (Doutorado) — University of California, Irvine, 2000. AAI9980887. Citado na página 43.

FLYNN, M. J. Some computer organizations and their effectiveness. **IEEE Trans. Comput.**, IEEE Computer Society, Washington, DC, USA, v. 21, n. 9, p. 948–960, set. 1972. ISSN 0018-9340. Citado na página 9.

GROUP, T. P. G. D. **PostgreSQL 9.5.0 Documentation**. 2016. Disponível em: <http://www.postgresql.org/docs/9.5/interactive/index.html>. Citado 2 vezes nas páginas 63 e 80.

GRUBER, T. R. Toward principles for the design of ontologies used for knowledge sharing. **Int. J. Hum.-Comput. Stud.**, Academic Press, Inc., Duluth, MN, USA, v. 43, n. 5-6, p. 907–928, dez. 1995. ISSN 1071-5819. Disponível em: <http://dx.doi.org/10.1006/ijhc.1995.1081>. Citado na página 53.

HUO, Q.; ZHU, H.; GREENWOOD, S. A multi-agent software engineering environment for testing web-based applications. In: **Computer Software and Applications Conference, 2003. COMPSAC 2003. Proceedings. 27th Annual International**. [S.l.: s.n.], 2003. p. 210–215. ISSN 0730-3157. Citado na página 54.

JAGANNATH, V.; GLIGORIC, M.; LAUTERBURG, S.; MARINOV, D.; AGHA, G. Mutation operators for actor systems. In: **Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on**. [S.l.: s.n.], 2010. p. 157–162. Citado na página 29.

JOSUTTIS, N. **Soa in Practice**. First. [S.l.]: O'Reilly, 2007. ISBN 9780596529550. Citado na página 40.

KOJIMA, H.; KAKUDA, Y.; TAKAHASHI, J.; OHTA, T. A model for concurrent states and its coverage criteria. In: **Autonomous Decentralized Systems, 2009. ISADS '09. International Symposium on**. [S.l.: s.n.], 2009. p. 1–6. Citado na página 30.

KUMAR, V.; GRAMA, A.; GUPTA, A.; KARYPIS, G. **Introduction to Parallel Computing: Design and Analysis of Algorithms**. Redwood City, CA, USA: Benjamin-Cummings Publishing Co., Inc., 1994. ISBN 0-8053-3170-0. Citado 7 vezes nas páginas 9, 10, 11, 12, 14, 18 e 19.

LEI, Y.; CARVER, R. Reachability testing of semaphore-based programs. In: **Computer Software and Applications Conference, 2004. COMPSAC 2004. Proceedings of the 28th Annual International**. [S.l.: s.n.], 2004. p. 312–317 vol.1. ISSN 0730-3157. Citado na página 30.

LEI, Y.; CARVER, R. H. Reachability testing of concurrent programs. **IEEE Trans. Softw. Eng.**, IEEE Press, Piscataway, NJ, USA, v. 32, n. 6, p. 382–403, jun. 2006. ISSN 0098-5589. Disponível em: <http://dx.doi.org/10.1109/TSE.2006.56>. Citado 2 vezes nas páginas 30 e 33.

LEI, Y.; CARVER, R. H.; KACKER, R.; KUNG, D. A combinatorial testing strategy for concurrent programs. **Softw. Test. Verif. Reliab.**, John Wiley & Sons Ltd., Chichester, UK, v. 17, n. 4, p. 207–225, dez. 2007. ISSN 0960-0833. Citado na página 30.

MUSUVATHI, M.; QADEER, S.; BALL, T. CHESS: A Systematic Testing Tool for Concurrent Software. 2007. Citado na página 32.

MYERS, G. J.; SANDLER, C. **The Art of Software Testing**. [S.l.]: John Wiley & Sons, 2004. ISBN 0471469122. Citado na página 24.

NAKAGAWA, E.; BARBOSA, E.; MALDONADO, J. Exploring ontologies to support the establishment of reference architectures: An example on software testing. In: **Software Architecture, 2009 European Conference on Software Architecture. WICSA/ECSA 2009. Joint Working IEEE/IFIP Conference on**. [S.l.: s.n.], 2009. p. 249–252. Citado na página 54.

NAKAGAWA, E.; SIMãO, A. da S.; FERRARI, F. C.; MALDONADO, J. C. Towards a reference architecture for software testing tools. In: **SEKE**. [S.l.]: Knowledge Systems Institute Graduate School, 2007. p. 157–162. ISBN 1-891706-20-9. Citado na página 54.

OLIVEIRA, L.; NAKAGAWA, E. A service-oriented reference architecture for software testing tools. In: CRNKOVIC, I.; GRUHN, V.; BOOK, M. (Ed.). **Software Architecture**. Springer Berlin Heidelberg, 2011, (Lecture Notes in Computer Science, v. 6903). p. 405–421. ISBN 978-3-642-23797-3. Disponível em: <http://dx.doi.org/10.1007/978-3-642-23798-0_42>. Citado na página 54.

OLIVEIRA, L. B. R. **Estabelecimento de uma arquitetura de referência orientada a serviços para ferramentas de teste de software**. Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos, 2011. Disponível em: <http://www.teses.usp.br/teses/disponiveis/55/55134/tde-30032011-131052/>. Citado 2 vezes nas páginas 48 e 82.

OLIVEIRA, L. B. R.; NAKAGAWA, E. Y. A service-oriented reference architecture for software testing tools. In: **Proceedings of the 5th European Conference on Software Architecture**. Berlin, Heidelberg: Springer-Verlag, 2011. (ECSA'11), p. 405–421. ISBN 978-3-642-23797-3. Disponível em: <http://dl.acm.org/citation.cfm?id=2041790.2041842>. Citado na página 49.

OLIVEIRA, L. B. R. d. **Pascal Mutants**. 2010. Disponível em: <http://ccsl.icmc.usp.br/pt-br/content/pascal-mutants>. Citado na página 49.

PRADO, R. R.; SOUZA, P. S. L. L. de; DOURADO, G. G. M.; SOUZA, S. R. S.; ESTRELLA, J. C.; BRUSCHI, S. M.; LOURENCO, J. Extracting static and dynamic structural information from java concurrent programs for coverage testing. In: CANCELA, H.; CUADROS-VARGAS, A.; CUADROS-VARGAS, E. (Ed.). **2015 XLI Latin American Computing Conference (CLEI)**. Arequipa-Peru: CLEI, 2015. p. 667–674. ISBN 978-1-4673-9143-6. Disponível em: <http://clei.org/clei2015/144484>. Citado na página 61.

QUINN, M. J. **Parallel Computing: Theory and Practice**. New York, NY: McGraw–Hill, Inc., 1994. Citado na página 9.

RAPPS, S.; WEYUKER, E. J. Data flow analysis techniques for test data selection. In: OHNO, Y.; BASILI, V. R.; ENOMOTO, H.; KOBAYASHI, K.; YEH, R. T. (Ed.). **ICSE**. [S.l.]: IEEE Computer Society, 1982. p. 272–278. Citado na página 27.

RICHARDSON, L.; RUBY, S. **Restful Web Services**. First. [S.l.]: O'Reilly, 2007. ISBN 9780596529260. Citado 2 vezes nas páginas 43 e 44.

SARMANHO, F.; SOUZA, P.; SOUZA, S.; SIMãO, A. Structural testing for semaphore-based multithread programs. In: BUBAK, M.; ALBADA, G.; DONGARRA, J.; SLOOT, P. (Ed.). **Computational Science – ICCS 2008**. [S.l.]: Springer Berlin Heidelberg, 2008, (Lecture Notes in Computer Science, v. 5101). p. 337–346. ISBN 978-3-540-69383-3. Citado 2 vezes nas páginas 2 e 33.

SEN, A.; ABADIR, M. Coverage metrics for verification of concurrent systemc designs using mutation testing. In: **High Level Design Validation and Test Workshop (HLDVT), 2010 IEEE International**. [S.l.: s.n.], 2010. p. 75–81. ISSN 1552-6674. Citado na página 29.

SOUZA, E. F.; FALBO, R. A.; VIJAYKUMAR, N. L. Ontologies in software testing: A systematic literature review. In: BAX, M. P.; ALMEIDA, M. B.; WASSERMANN, R. (Ed.). **ONTOBRAS**. [S.l.]: CEUR-WS.org, 2013. (CEUR Workshop Proceedings, v. 1041), p. 71–82. Citado na página 54.

SOUZA, P.; SAWABE, E.; SIMAO, A.; VERGILIO, S.; SOUZA, S. Valipvm - a graphical tool for structural testing of pvm programs. In: **Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface**. Berlin, Heidelberg: Springer-Verlag, 2008. p. 257–264. ISBN 978-3-540-87474-4. Citado 2 vezes nas páginas 30 e 33.

SOUZA, P. S.; SOUZA, S. S.; ROCHA, M. G.; PRADO, R. R.; BATISTA, R. N. Data flow testing in concurrent programs with message passing and shared memory paradigms. **Procedia Computer Science**, v. 18, n. 0, p. 149 – 158, 2013. ISSN 1877-0509. 2013 International Conference on Computational Science. Citado 3 vezes nas páginas 2, 30 e 33.

SOUZA, S.; SOUZA, P.; MACHADO, M.; CAMILLO, M.; SIMAO, A.; ZALUSKA, E. Using coverage and reachability testing to improve concurrent program testing quality. In: **23rd International Conference on Software Engineering and Knowledge Engineering**. [S.l.: s.n.], 2011. v. 1, n. 1, p. 207–212. Event Dates: July 7-9, 2011. Citado 2 vezes nas páginas 30 e 35.

SOUZA, S. do Rocio Senger de; VERGILIO, S. R.; SOUZA, P. S. L. Introdução ao teste de software. In: . [S.l.]: CAMPUS, 2007. cap. Teste de Programas Concorrentes, p. 231–250. ISBN 9788535226348. Citado na página 29.

SOUZA, S. R. S.; SOUZA, P. S. L.; ZALUSKA, E. Structural testing for message-passing concurrent programs: an extended test model. **Concurr. Comput. : Pract. Exper.**, 2012. Citado 2 vezes nas páginas 30 e 33.

SOUZA, S. R. S.; VERGILIO, S. R.; SOUZA, P. S. L. Coverage testing criteria for message-passing parallel programs. In: **LATW2005**. Salvador, Ba: 6th IEEE Latin-American Test Workshop, 2005. p. 161–166. Citado 2 vezes nas páginas 30 e 33.

SOUZA, S. R. S.; VERGILIO, S. R.; SOUZA, P. S. L.; SIMAO, A. S.; GONCALVES, T. B.; LIMA, A. M.; HAUSEN, A. C. Valipar: A testing tool for message-passing parallel programs. In: **SEKE'05**. [S.l.: s.n.], 2005. p. 386–391. Citado 2 vezes nas páginas 30 e 33.

SOUZA, S. R. S.; VERGILIO, S. R.; SOUZA, P. S. L.; SIMAO, A. S.; HAUSEN, A. C. Structural testing criteria for message-passing parallel programs. **Concurrency and Computation: Practice and Experience**, John Wiley & Sons, Ltd., v. 20, n. 16, p. 1893–1916, 2008. ISSN 1532-0634. Citado 2 vezes nas páginas 2 e 30.

TANENBAUM, A. S. **Modern Operating Systems**. 3rd. ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2007. ISBN 9780136006633. Citado 5 vezes nas páginas 7, 8, 9, 15 e 29.

TANENBAUM, A. S.; STEEN, M. v. **Distributed Systems: Principles and Paradigms (2Nd Edition)**. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2006. ISBN 0132392275. Citado 2 vezes nas páginas 19 e 39.

TAYLOR, R.; LEVINE, D.; KELLY, C. Structural testing of concurrent programs. **Software Engineering, IEEE Transactions on**, v. 18, n. 3, p. 206–215, 1992. ISSN 0098-5589. Citado na página 30.

TEAM, H. D. **H2 Database Engine**. 2016. Disponível em: <http://www.h2database.com/html/main.html>. Citado na página 81.

TEAM, M. D. **MySQL 5.7 Reference Manual**. 2016. Disponível em: <https://dev.mysql.com/doc/refman/5.7/en/>. Citado na página 80.

USCHOLD, M.; GRUNINGER, M. Ontologies: Principles, methods and applications. **KNOWLEDGE ENGINEERING REVIEW**, v. 11, p. 93–136, 1996. Citado na página 53.

VICENZI, A. M. R.; DOMINGUES, A. L. dos S.; DELAMARO, M. E.; MALDONADO, J. C. Introdução ao teste de software. In: . [S.l.]: CAMPUS, 2007. cap. Teste Orientado a Objetos e de Componentes, p. 119–174. ISBN 9788535226348. Citado na página 37.

VINCENZI, A. M. R.; MALDONADO, J. C.; WONG, W. E.; DELAMARO, M. E. Coverage testing of java programs and components. **Science of Computer Programming**, Elsevier, Amsterdam, Netherlands, v. 56, n. 1-2, p. 211–230, abr. 2005. ISSN 0167-6423. Citado na página 49.

VUKOTIC, A.; GOODWILL, J. **Apache Tomcat 7**. 1st. ed. Berkely, CA, USA: Apress, 2011. ISBN 1430237236, 9781430237235. Citado na página 63.

W3C. **Web Services Glossary**. 2004. Disponível em: <http://www.w3.org/TR/ws-gloss/>. Citado na página 39.

WAKERLY, J. The programming language pascal. **Microprocessors and Microsystems**, v. 3, n. 9, p. 405 – 412, 1979. ISSN 0141-9331. Disponível em: <http://www.sciencedirect.com/science/article/pii/0141933179901121>. Citado na página 49.

WILLIAMS, N.; MARRE, B.; MOUY, P.; ROGER, M. Pathcrawler: Automatic generation of path tests by combining static and dynamic analysis. In: **Proceedings of the 5th European Conference on Dependable Computing**. Berlin, Heidelberg: Springer-Verlag, 2005. (EDCC'05), p. 281–292. ISBN 3-540-25723-3, 978-3-540-25723-3. Disponível em: <http://dx.doi.org/10.1007/11408901_21>. Citado 2 vezes nas páginas 3 e 48.

YANG, C.-S. D.; SOUTER, A. L.; POLLOCK, L. L. All-du-path coverage for parallel programs. **SIGSOFT Softw. Eng. Notes**, ACM, New York, NY, USA, v. 23, n. 2, p. 153–162, mar. 1998. ISSN 0163-5948. Citado na página 30.

YU, L.; ZHANG, L.; XIANG, H.; SU, Y.; ZHAO, W.; ZHU, J. A framework of testing as a service. In: **Management and Service Science, 2009. MASS '09. International Conference on**. [S.l.: s.n.], 2009. p. 1–4. Citado na página 54.

APÊNDICE

A

ESPECIFICAÇÃO DE CONTRATOS DE SERVIÇOS

O conjunto de serviços ValiPar é composto por seis serviços que são operados de modo coordenado para realizar o teste estrutural de programas concorrentes. Esse conjunto de serviços possui um total de 18 contratos relacionados com o armazenamento, geração e atualização de artefatos de teste.

Cada artefato de teste é gerado por um serviço e, em geral, transferido para serviços dependentes. Atualmente o conjunto de serviços reconhece os seguintes artefatos: Programa, Programa instrumentado, PCFG, Caso de teste, Execução, Variante de execução, Avaliação de cobertura e Elemento requerido.

A listagem abaixo relaciona todos os 18 contratos presentes no conjunto de serviço relacionados à geração e armazenamento de artefatos de teste. A descrição conta com o comando a ser requisitado ao serviço, os parâmetros de entrada que devem ser fornecidos, as pré-condições e pós-condições requeridas e respeitadas e o processamento realizado pelo contrato. Por fim, são identificados os serviços da composição que disponibilizam tal contrato.

Além das operações expostas, existem também contratos de aquisição de artefatos de teste. Tais contratos apenas retornam o artefato baseado na identificação do mesmo fornecida como parâmetro de entrada, sem realizar outro processamento específico. Por esse motivo, tais operações não são descritas neste Apêndice.

Os contratos seguem uma arquitetura REST e cada mensagem é estruturada como uma requisição HTTP. Dessa forma, uma mensagem é composta por um verbo HTTP (“GET”, “POST”, “PUT”) e uma URL que representa unicamente um recurso ou uma coleção de recursos. Adicionalmente, uma requisição pode conter um corpo com informações adicionais sobre a requisição ou sobre o recurso.

Os serviços ValiPar representam textualmente seus recursos no formato HAL+JSON.

Assim, os recursos são especificados no formato JSON e seguem o padrão HAL que, além de informações sobre o recurso em si, também descreve *links* (URLS) para recursos relacionados, permitindo que o cliente navegue pelos recursos e tenha o controle de toda a execução.

Muitos recursos não podem ser descritos de modo textual, como por exemplo o programa a ser testado, o programa instrumentado e resultados de execução do programa por conterem informações em formato binário e grande quantidade de arquivos. Esses recursos são dessa forma representados como arquivos no formato ZIP contendo todas as informações necessárias. Por fim, a representação gráfica de PCFGs, no serviço ValiInst, é disponibilizada no formato PNG.

1. **Armazenar programa concorrente**
   - **Comando:** `POST /programs`
   - **Parâmetros de entrada:** Arquivo no formato ZIP contendo programa a ser testado em formato *bytecode*.
   - **Pré-condições:** Arquivo do programa em formato válido.
   - **Processamento:** Armazenamento do programa.
   - **Pós-condições:** Programa disponível para operações.
   - **Serviços:** ValiPar, ValiInst.
2. **Gerar sessão de teste**
   - **Comando:** `POST /programs/<id>/sessions`
   - **Parâmetros de entrada:** Identificação (ID) do programa alvo do teste; Descrição dos processos que compõem o programa sendo testado;
   - **Pré-condições:** Existência de programa.
   - **Processamento:** Preparação do ambiente para o teste.
   - **Pós-condições:** Sessão disponível para operações.
   - **Serviços:** ValiPar, ValiInst, ValiElem, ValiEval, ValiExec, ValiSync.
3. **Gerar caso de teste**
   - **Comando:** `POST /sessions/<id>/testcases`
   - **Parâmetros de entrada:** Identificação (ID) da sessão de teste; Especificação de caso de teste (entrada e saída esperada para cada processo).
   - **Pré-condições:** Existência de uma sessão de teste; Processos descritos na especificação são devidamente descritos na especificação do conjunto de *threads* e processos. Todos os processos possuem argumentos, entrada e saída esperada.
   - **Processamento:** Criação do caso de teste.

- **Pós-condições:** Caso de teste disponível para operações.
- **Serviços:** ValiPar, ValiExec, ValiSync, ValiEval.

4. **Gerar *Parallel Control Flow Graph* (PCFG)**

   - **Comando:** `POST /sessions/<id>/pcfg`
   - **Parâmetros de entrada:** Identificação (ID) da sessão de teste; Especificação do conjunto de processos e *threads*.
   - **Pré-condições:** Existência da sessão de teste, de programa; O programa deve utilizar estruturas de controle, de dados e de sincronização suportadas pelo analisador estático que extrai os dados para o modelo; As funções especificadas na especificação do conjunto de processos e *threads* devem estar presentes; Especificação em conformidade com o padrão proposto.
   - **Processamento:** Análise estática do código fonte; Geração de dados para o modelo de teste (PCFG).
   - **Pós-condições:** PCFG disponível para operações.
   - **Serviços:** ValiPar, ValiInst.

5. **Armazenar *Parallel Control Flow Graph* (PCFG)**

   - **Comando:** `POST /sessions/<id>/pcfg`
   - **Parâmetros de entrada:** Identificação (ID) da sessão de teste; Especificação dos fluxos do programa concorrente em JSON.
   - **Pré-condições:** Sessão existente; JSON no formato correto; PCFG inexistente na sessão especificada.
   - **Processamento:** Armazenamento de PCFG.
   - **Pós-condições:** Programa instrumentado disponível para operações.
   - **Serviços:** ValiElem.

6. **Gerar programa instrumentado**

   - **Comando:** `POST /testcases/<id>/executions`
   - **Parâmetros de entrada:** Identificação (ID) da sessão de teste; Especificação do conjunto de processos e *threads*; Listagem de nomes de rotinas “main()” e identificação de processo e de *thread*; Especificação do conjunto de primitivas de sincronização e semânticas (opcional).
   - **Pré-condições:** Existência da sessão de teste e de programa; O programa deve utilizar estruturas de controle, de dados e de sincronização suportadas pelo analisador

estático que extrai os dados para o modelo; As funções especificadas na especificação do conjunto de processos e *threads* devem estar presentes; Especificação em conformidade com o padrão proposto.

- **Processamento:** Análise estática do programa; Geração de programa instrumentado.
- **Pós-condições:** Programa instrumentado disponível para operações.
- **Serviços:** ValiPar, ValiInst.

7. **Armazenar programa instrumentado**

   - **Comando:** `POST /sessions/<id>/instrumented_program`
   - **Parâmetros de entrada:** Identificação (ID) da sessão de teste; Arquivo no formato ZIP contendo programa instrumentado em formato *bytecode*.
   - **Pré-condições:** Existência sessão de teste; Arquivo do programa instrumentado em formato válido; Programa instrumentado inexistente na sessão.
   - **Processamento:** Armazenamento do programa instrumentado.
   - **Pós-condições:** Programa instrumentado disponível para operações.
   - **Serviços:** ValiExec, ValiSync.

8. **Gerar elementos requeridos**

   - **Comando:** `POST /sessions/<id>/required_elements_sets`
   - **Parâmetros de entrada:** Identificação (ID) da sessão de teste; Especificação dos elementos requeridos em JSON.
   - **Pré-condições:** Existência de sessão e de PCFG; Conjunto de elementos requeridos inexistente na sessão.
   - **Processamento:** Geração de elementos requeridos.
   - **Pós-condições:** Conjuntos de elementos requeridos criados e disponíveis de acordo com critérios de teste e com o modelo de teste.
   - **Serviços:** ValiPar, ValiElem.

9. **Armazenar conjunto de elementos requeridos**

   - **Comando:** `POST /sessions/<id>/required_elements_sets`
   - **Parâmetros de entrada:** Identificação (ID) da sessão de teste; Especificação dos elementos requeridos em JSON.
   - **Pré-condições:** Existência de sessão de teste; Conjunto de elementos requeridos inexistente na sessão.
   - **Processamento:** Armazenamento de conjunto de elementos requeridos.

- **Pós-condições:** Conjunto de elementos requeridos disponíveis para operações.
- **Serviços:** ValiEval.

10. **Atualizar elemento requerido**

    - **Comando:** `PUT /required_elements/<id>/<tipo>/<eid>`
    - **Parâmetros de entrada:** Identificação (ID) do conjunto de elementos requeridos; Tipo de elemento requerido; Identificação (EID) do elemento requerido dentro do conjunto.
    - **Pré-condições:** Existência de conjunto de elementos requeridos, tipo e identificação de elemento requerido; Elemento requerido em questão é "não coberto" e estado desejado é "não executável".
    - **Processamento:** Marcação de elemento requerido como não executável.
    - **Pós-condições:** Elemento requerido atualizado para novo estado.
    - **Serviços:** ValiEval.

11. **Executar caso de teste no modo determinístico**

    - **Comando:** `POST /executions/<id>/deterministic_executions`
    - **Parâmetros de entrada:** Identificação (ID) de execução de caso de teste a ser reexecutado deterministicamente.
    - **Pré-condições:** Existência de execução de caso de teste e de programa instrumentado.
    - **Processamento:** Execução do caso de teste utilizando o programa instrumentado e uma sequência de sincronização de forma controlada.
    - **Pós-condições:** Execução com rastros e saídas disponível para operações.
    - **Serviços:** ValiPar, ValiExec.

12. **Executar variante de execução no modo determinístico**

    - **Comando:** `POST /variants/<id>/execution`
    - **Parâmetros de entrada:** Identificação (ID) de variante a ser executada de modo determinístico.
    - **Pré-condições:** Existência de variante de execução e de programa instrumentado.
    - **Processamento:** Execução determinística do caso de teste utilizando o programa instrumentado e a descrição de variante.
    - **Pós-condições:** Execução com rastros e saídas disponível para operações.
    - **Serviços:** ValiExec.

13. **Executar caso de teste no modo não-determinístico**

    - **Comando:** `POST /testcases/<id>/executions`
    - **Parâmetros de entrada:** Identificação (ID) do caso de teste a ser executado de modo não-determinístico.
    - **Pré-condições:** Existência de caso de teste e de programa instrumentado.
    - **Processamento:** Execução do caso de teste utilizando o programa instrumentado.
    - **Pós-condições:** Execução com rastros e saídas disponível para operações.
    - **Serviços:** ValiPar, ValiExec.

14. **Executar todas variantes para conjunto de casos de teste no modo determinístico**

    - **Comando:** `POST /sessions/<id>/all_variants_execution`
    - **Parâmetros de entrada:** Identificação (ID) da sessão de teste.
    - **Pré-condições:** Existência de sessão de teste, de programa instrumentado, de casos de teste e de avaliação de critérios.
    - **Processamento:** Derivação de sequências de sincronização; Execução do caso de teste utilizando o programa instrumentado e uma sequência de sincronização de forma controlada; Avaliação de cobertura para eliminação de sequências de sincronização já cobertas.
    - **Pós-condições:** Execução de todas as sequências de sincronização para o caso de teste. Arquivos de rastro de cada execução. Comparação da saída do programa com a saída esperada de cada execução.
    - **Serviços:** ValiPar, ValiSync.

15. **Armazenar variante de execução de caso de teste**

    - **Comando:** `POST /executions/<id>/variants`
    - **Parâmetros de entrada:** Identificação (ID) da execução de caso de teste; Especificação de variante de execução.
    - **Pré-condições:** Existência de execução.
    - **Processamento:** Armazenamento de variante.
    - **Pós-condições:** Variante de execução disponível para operações.
    - **Serviços:** ValiExec.

16. **Armazenar execução de caso de teste**

    - **Comando:** `POST /testcases/<id>/executions`

- **Parâmetros de entrada:** Identificação do caso de teste (ID); Arquivos de rastro, entradas e saídas de execução em um arquivo no formato ZIP.
- **Pré-condições:** Existência de caso de teste; Arquivo no formato correto.
- **Processamento:** Armazenamento execução.
- **Pós-condições:** Execução disponível para operações.
- **Serviços:** ValiPar, ValiEval, ValiExec.

17. **Avaliar a cobertura uma execução de um caso de teste**

    - **Comando:** `POST /sessions/<id>/evaluation`
    - **Parâmetros de entrada:** Identificação (ID) da sessão de teste; Nome dos critérios a serem analisados durante o teste.
    - **Pré-condições:** Existência de sessão de teste e elementos requeridos.
    - **Processamento:** Marcação de elementos requeridos cobertos pela execução de casos de teste; Disponibilização da porcentagem de cobertura de cada critério de teste.
    - **Pós-condições:** Elementos requeridos marcados como cobertos. Porcentagem de elementos requeridos cobertos disponibilizada.
    - **Serviços:** ValiPar, ValiEval.

18. **Armazenar referência para avaliação de cobertura de teste**

    - **Comando:** `POST /sessions/<id>/evaluation`
    - **Parâmetros de entrada:** Identificação (ID) da sessão de teste; Critérios suportados; URL para a avaliação de cobertura.
    - **Pré-condições:** Existência de sessão de teste.
    - **Processamento:** Armazenamento de URL para avaliação de cobertura.
    - **Pós-condições:** URL disponível para consulta de cobertura.
    - **Serviços:** ValiSync.