Pesquisa de teste de software: conquistas, desafios, sonhos

Antônia Bertolino

Antonia Bertolino (http://www.isti.cnr.it/People/A.Bertolino) é uma Diretor de Pesquisa do Conselho Nacional de Pesquisa Italiano do ISTI em Pisa, onde lidera o Laboratório de Engenharia de Software. Ela também coordena o laboratório Pisatel, patrocinado pela Ericsson Lab Itália. Seus interesses de pesquisa são em arquitetura baseada em componentes e metodologias de teste orientadas a serviços, bem como métodos para análise de propriedades não funcionais. Ela é Editora Associada do and of e Diário de Sistemas e Programas anteriormente, Revista Empírica de Engenharia de Software , servido para o. Ela Transações IEEE em Engenharia de software Presidente do programa para o conjunto a ser realizado em Conferência ESEC/FSE Dubrovnik, Croácia, em setembro de 2007, e é membro regular da os Comitês de Programa de conferências internacionais, incluindo ACM ISSTA, Joint ESEC-FSE, ACM/IEEE ICSE, IFIP TestCom. Ela tem (co)autor de mais de 80 artigos em revistas internacionais e conferências.

Futuro da Engenharia de Software (FOSE'07)
0-7695-2829-5/07 $ 20,00 © 2007 IEEE

Pesquisa de teste de software: conquistas, desafios, sonhos

Antônia Bertolino
Istituto di Scienza e Tecnologie dell'Informazione “A. Faedo”
Consiglio Nazionale delle Ricerche
56124 Pisa, Itália
antonia.bertolino@isti.cnr.it

Abstrato

A engenharia de software abrange várias disciplinas destinadas a prevenir e remediar avarias e a garantir um comportamento adequado. Testes, o assunto deste artigo, é uma abordagem de validação generalizada na indústria, mas é ainda em grande parte ad hoc, caro e imprevisivelmente eficaz. De fato, teste de software é um termo amplo que abrange uma variedade de atividades ao longo do ciclo de desenvolvimento e além, voltados para diferentes objetivos. Assim, a pesquisa de teste de software enfrenta uma série de desafios. Um roteiro consistente de os desafios mais relevantes a serem enfrentados são aqui propostos. Nela, o ponto de partida é constituído por algumas conquistas importantes do passado, enquanto o destino consiste em quatro objetivos identificados para os quais a pesquisa tende, em última análise, mas que permanecem tão inalcançáveis quanto os sonhos. As rotas de as conquistas dos sonhos são pavimentadas pelos grandes desafios de pesquisa, que são discutidos no artigo juntamente com trabalhos interessantes em andamento.

1. Introdução

O teste é uma atividade essencial na engenharia de software. Em termos mais simples, equivale a observar a execução de um sistema de software para validar se ele se comporta conforme pretendido e identificar potenciais avarias. O teste é amplamente utilizado na indústria para garantia de qualidade: de fato, por examinando diretamente o software em execução, ele fornece uma feedback realista do seu comportamento e, como tal, continua a ser o complemento inevitável a outras técnicas de análise.

Além da aparente simplicidade de verificar um uma amostra de corridas, no entanto, o teste abrange uma variedade de atividades, técnicas e atores, e apresenta muitos desafios complexos. De fato, com a complexidade, abrangência e criticidade do software crescendo incessantemente, garantir que ele seja oferecido de acordo com os níveis desejados de qualidade e dependência torna-se mais crucial e cada vez mais difícil e caro. Estudos anteriores estimaram que os testes podem somar cinquenta por cento, ou até mais, dos custos de desenvolvimento [3], e uma pesquisa detalhada recente nos Estados Unidos [63] quantifica os altos impactos econômicos de uma infraestrutura de teste de software inadequada.

Da mesma forma, surgem novos desafios de pesquisa, como como, por exemplo, como conciliar a derivação baseada em modelos de casos de teste com sistemas modernos de evolução dinâmica, ou como selecionar e usar efetivamente os dados de tempo de execução coletados do uso real após a implantação. Esses recém-emergidos desafios vão aumentar problemas abertos de longa data, como como qualificar e avaliar a eficácia dos testes critérios, ou como minimizar a quantidade de reteste após o software é modificado.

Ao longo dos anos, o tema tem despertado crescente interesse de pesquisadores, como testemunham os muitos eventos especializados e workshops, bem como pelo crescente percentual de artigos de teste em conferências de engenharia de software; por exemplo na 28ª Conferência Internacional de Engenharia de Software (ICSE 2006) quatro das doze sessões do trilha de pesquisa focada em “Teste e Análise”.

Este artigo organiza as muitas pesquisas pendentes desafios para teste de software em um roteiro consistente. Os destinos identificados são um conjunto de quatro objetivos finais e inatingíveis chamados “sonhos”. Aspirando a esses sonhos, pesquisadores estão enfrentando vários desafios, que são aqui vistas como facetas viáveis interessantes do problema insolúvel maior. A imagem resultante é proposta para a comunidade de pesquisadores de testes de software como um trabalho em andamento tecido a ser adaptado e ampliado.

Na Seção 2 discutimos a natureza multifacetada do software testar e identificar um conjunto de seis perguntas subjacentes a qualquer teste abordagem. Na Seção 3, apresentamos a estrutura de o roteiro proposto. Resumimos alguns mais maduros áreas de investigação, que constituem o ponto de partida para a nossa viagem no roteiro, na Seção 4. Depois, na Seção 5, que é a parte principal do artigo, apresentamos vários desafios de pesquisa pendentes e os sonhos para os quais Eles tendem. Breves observações finais na Seção 6 fecham o papel.

Futuro da Engenharia de Software (FOSE'07)
0-7695-2829-5/07 $ 20,00 © 2007 IEEE

2. As muitas faces do teste de software

Teste de software é um termo amplo que abrange uma ampla espectro de atividades diferentes, desde o teste de uma pequena pedaço de código pelo desenvolvedor (testes unitários), à validação pelo cliente de um grande sistema de informação (aceitação testes), para o monitoramento em tempo de execução de uma rede centrada aplicação orientada a serviços. Nas várias etapas, o teste casos poderiam ser planejados visando objetivos diferentes, como como expor desvios dos requisitos do usuário, ou avaliar a conformidade com uma especificação padrão, ou avaliar a robustez a condições de carga estressantes ou a entradas, ou medir determinados atributos, como desempenho ou usabilidade, ou estimar a confiabilidade operacional, e assim sobre. Além disso, a atividade de teste pode ser realizada de acordo com um procedimento formal controlado, exigindo planejamento e documentação rigorosos, ou melhor, informalmente e ad hoc (teste exploratório).

Como consequência dessa variedade de objetivos e escopo, um multiplicidade de significados para o termo “teste de software” surge, o que gerou muitos desafios peculiares de pesquisa. Para organizar esta última em uma visão unificadora, no restante desta seção, tentamos uma classificação de problemas comum aos muitos significados de teste de software. o primeiro conceito a capturar seria qual é o denominador comum, se existir, entre todos os testes diferentes possíveis "rostos". Propomos que tal denominador comum possa ser a visão muito abstrata que, dado um pedaço de software (qualquer que seja a sua tipologia, tamanho e domínio) testando sempre consiste em *observar uma amostra de execuções,* e dar uma veredicto sobre eles.

Partindo desta visão muito geral, podemos então concretizar diferentes instâncias, distinguindo os aspectos específicos que podem caracterizar a amostra de observações:

PORQUÊ: por que é que fazemos as observações? este questão diz respeito ao *objetivo do teste,* por exemplo: estamos procurando por falhas? ou precisamos decidir se o produto pode ser liberado? ou melhor, precisamos avaliar a usabilidade da interface do usuário?

COMO: qual amostra observamos e como Escolha? Este é o problema da *seleção de testes,* que pode ser feito ad hoc, de forma aleatória ou sistemática, aplicando-se alguma técnica algorítmica ou estatística. tem inspirado muita pesquisa, o que é compreensível não só porque é intelectualmente atraente, mas também porque como o teste casos são selecionados - o critério de teste - influencia muito o teste eficácia.

QUANTO: qual o tamanho de uma amostra? Dupla à questão de como escolhemos as observações da amostra (seleção de teste), é a de quantas delas fazemos *(adequação do teste,* ou regra de parada). Análise de cobertura ou confiabilidade medidas constituem duas abordagens “clássicas” para responder tal pergunta.

O QUE: o que é que executamos? Dado o sistema (possivelmente composto) em teste, podemos observar sua execução tomando-o como um todo, ou focando apenas em um parte dele, que pode ser mais ou menos grande (teste unitário, teste de componente/subsistema, teste de integração), mais ou menos definido: este aspecto dá origem aos vários *níveis de teste,* e o andaime necessário para permitir a execução de teste de uma peça de um sistema maior.

ONDE: onde realizamos a observação? Estritamente relacionado ao que executamos, está a questão se isso é feito em casa, em um ambiente simulado ou no contexto final de destino. Esta pergunta pressupõe a maior relevância quando se trata de testes de sistemas.

QUANDO: quando é no ciclo de vida do produto que realizamos as observações? O argumento convencional é que o mais cedo, o mais conveniente, pois o custo da remoção de falhas aumenta à medida que o ciclo de vida avança. Mas, algumas observações, em particular as que dependem da envolvente contexto, nem sempre pode ser antecipado em laboratório, e não podemos realizar qualquer observação significativa até que o sistema está implantado e em operação.

Essas perguntas fornecem um esquema de caracterização muito simples e intuitivo das atividades de teste de software, que pode ajudar na organização do roteiro para futuros desafios de pesquisa.

3. Roteiro de pesquisa de teste de software

Um roteiro fornece direções para chegar a um destino desejado a partir do ponto vermelho “você está aqui”. O roteiro de pesquisa de teste de software de software está organizado da seguinte forma:

- o ponto vermelho "você está aqui" consiste no mais notável *conquistas* de pesquisas anteriores (mas observe que algumas das esses esforços ainda estão em andamento);
- o destino desejado é representado na forma de um conjunto de (quatro) *sonhos:* usamos este termo para significar que estes são metas assintóticas no final de quatro rotas identificadas para o progresso da pesquisa. Eles são inalcançáveis por definição e seu valor fica exatamente em atuar como os pólos de atração para pesquisas úteis e previdentes;
- no meio estão os *desafios* enfrentados pelos atuais e pesquisas de teste futuras, em estágio mais ou menos maduro, e com mais ou menos chances de sucesso. Esses desafios constituem as direções a serem seguidas no jornada em direção aos sonhos, e como tal eles são os parte central e mais importante do roteiro.

O roteiro está ilustrado na Figura 1. Nela, temos situou as direções de pesquisa emergentes e em andamento no centro, com temas mais maduros -as conquistas- em sua

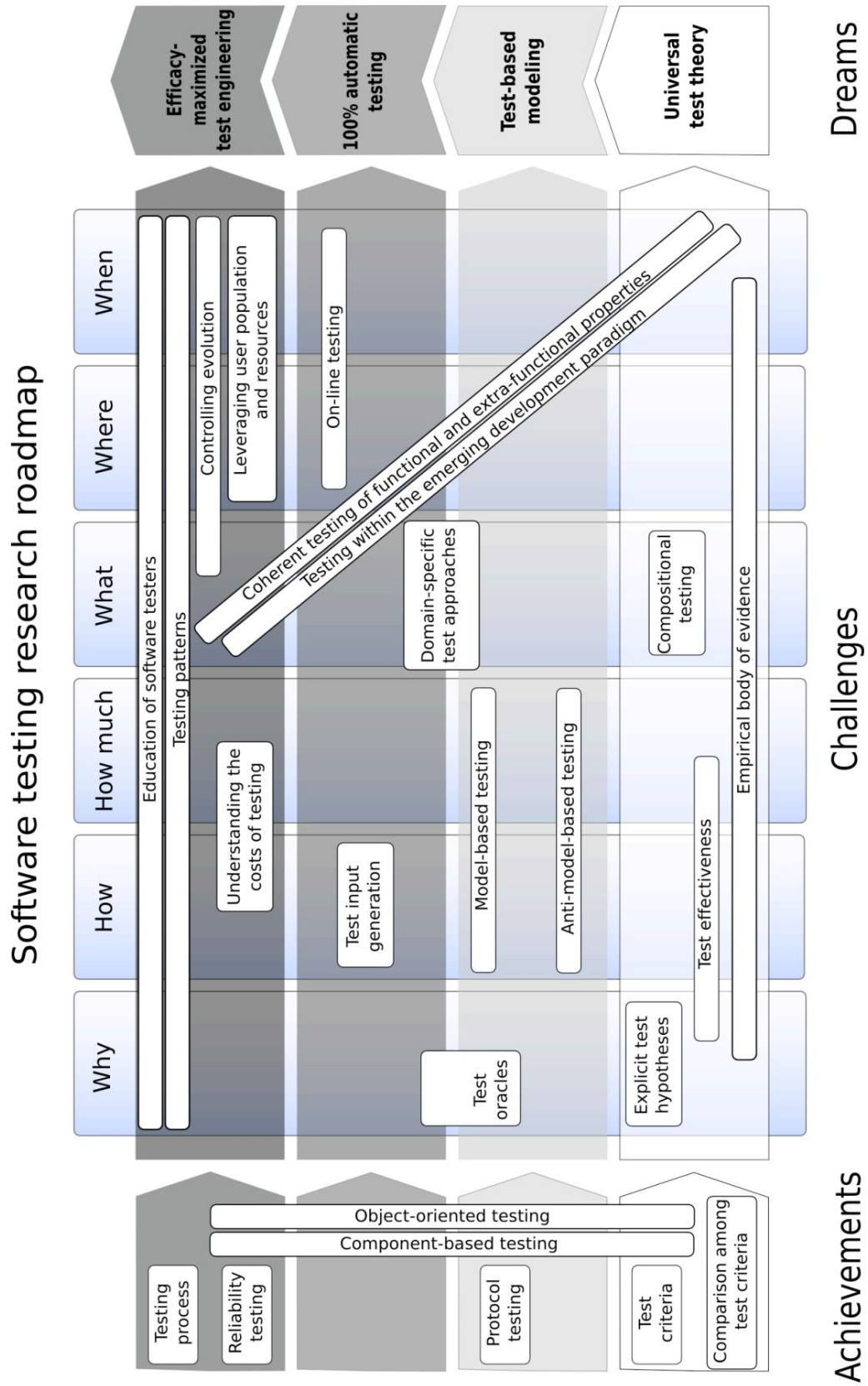

Figura 1. Roteiro

esquerda, e os objetivos finais -os sonhos- à sua direita. Quatro tiras horizontais representam as rotas de pesquisa identificadas para os sonhos, a saber:

1. Teoria do teste universal;
2. Modelagem baseada em testes;
3. Teste 100% automático;
4. Engenharia de teste com eficácia maximizada.

As rotas são ordenadas de baixo para cima de acordo com utilidade progressiva: a teoria está na base da modelos, que por sua vez são necessários para automação, que é instrumental para engenharia de teste econômica.

Os desafios se estendem horizontalmente por seis faixas verticais correspondente ao PORQUÊ, COMO, QUANTO, O QUE, ONDE e QUANDO perguntas que caracterizam o software rostos de teste (sem ordem específica).

Os desafios da pesquisa de teste de software encontram seu lugar em este plano, verticalmente dependendo do sonho de longo prazo, ou sonhos, para os quais eles tendem principalmente, e horizontalmente de acordo com qual pergunta, ou perguntas, do caracterização de teste de software em que eles se concentram principalmente.

No restante deste artigo, discutiremos os elementos (conquistas, desafios, sonhos) desse roteiro. Frequentemente, compararemos este roteiro com seu antecessor de 2000 por Harrold [43], ao qual nos referiremos daqui em diante como FOSE2000.

4. Você está aqui: Conquistas

Antes de delinear as futuras rotas de pesquisa de teste de software, tenta-se aqui um instantâneo de alguns tópicos que constituem o corpo de conhecimento em teste de software (por um pronto, guia mais detalhado veja também [8]), ou em que importantes realizações de pesquisa foram estabelecidas. No roteiro da Figura 1, estes estão representados no lado esquerdo.

As origens da literatura sobre teste de software remontam ao início dos anos 70 (embora se possa imaginar que a própria noção de teste nasceu simultaneamente com as primeiras experiências de programação): Hetzel [44] data a primeira conferência dedicada ao teste de programas em 1972. O teste foi concebido como um arte, e foi exemplificado como o "destrutivo" processo de execução de um programa com a intenção de encontrar erros, em oposição ao design que constituía o Festa. É desses anos o aforismo mais citado de Dijkstra sobre teste de software, que só pode mostrar a presença de falhas, mas nunca a sua ausência [25].

Os anos 80 viram a suposição de testar o status de um disciplina projetada, e uma mudança de visão de seu objetivo de apenas a descoberta de erros para uma abordagem mais abrangente e positiva visão de prevenção. O teste é agora caracterizado como *uma ampla e atividade contínua ao longo do processo de desenvolvimento* ([44], pág.6), cujo objetivo é a medição e avaliação de atributos e capacidades de software, e Beizer afirma: *Mais do que o ato de testar, o ato de* projetar *testes é um dos melhores preventores de bugs conhecidos* ([3], pg. 3).

Processo de teste. De fato, muitas pesquisas no início anos amadureceu em técnicas e ferramentas que ajudam tornar esse "pensamento de design de teste" mais sistemático e incorporá-lo no processo de desenvolvimento. Vários testes modelos de processo foram propostos para adoção industrial, entre os quais provavelmente o "modelo V" é o mais popular. Todas as suas muitas variantes compartilham a distinção de pelo menos o Níveis de unidade, integração e sistema para testes.

Mais recentemente, a implicação do modelo V de uma processo de teste formalmente documentado tem sido argumentado por alguns como sendo ineficiente e desnecessariamente burocrática, e em contraste, processos mais *ágeis* têm sido defendidos. No que diz respeito aos testes em particular, um modelo diferente que está ganhando atenção é *o desenvolvimento orientado a testes* (TDD)[46], um dos principais práticas *de programação extremas .*

O estabelecimento de um processo adequado para testes foi listado no FOSE2000 entre os tópicos de pesquisa fundamentais e, de fato, esta continua sendo uma pesquisa ativa hoje.

Critérios de teste. Extremamente rico é o conjunto de critérios de teste elaborados por pesquisas anteriores para ajudar na identificação sistemática de casos de teste. Tradicionalmente, estes são distinguidos entre a caixa branca (também conhecida como estrutural) e a caixa preta (também conhecida como funcional), dependendo se o código-fonte é ou não explorados na condução dos testes. Uma classificação mais refinada pode ser feita de acordo com a fonte de onde o teste casos são derivados [8], e muitos livros e artigos de pesquisa (por exemplo, [89]) existem que fornecem descrições abrangentes dos critérios existentes. De fato, tantos critérios entre que escolher agora existem, que o verdadeiro desafio se torna a capacidade de fazer uma escolha justificada, ou melhor, de entender como eles podem ser combinados de forma mais eficiente. Recentemente anos, a maior atenção tem sido voltada para modelos baseados em teste, consulte a Seção 5.2.

Comparação entre critérios de teste. Em paralelo com o investigação de critérios para seleção de testes e para adequação do teste, muitas pesquisas abordaram a avaliação do eficácia relativa dos vários critérios de teste, e especialmente dos fatores que tornam uma técnica melhor do que outro na descoberta de falhas. Estudos anteriores incluíram vários comparações analíticas entre diferentes técnicas (por exemplo, [31, 88]). Esses estudos permitiram estabelecer uma hierarquia de subsunção de relativo rigor entre critérios comparáveis e compreender os fatores que influenciam a probabilidade de encontrar falhas, concentrando-se mais em particular na comparação de partição (ou seja, sistemática) com testes aleatórios. "Demonstrar a eficácia das técnicas de teste" foi de fato identificado como uma pesquisa fundamental desafio no FOSE2000, e ainda hoje esse objetivo exige para mais pesquisas, onde a ênfase está agora em

avaliação pírica.

Testes orientados a objetos. De fato, em um determinado período, o paradigma dominante de desenvolvimento catalisou a pesquisa de testes para abordagens adequadas, conforme desenvolvemos na Seção 5.5. Na década de 90 o foco era o teste de software orientado a objetos (OO). Rejeitou o mito que a modularidade e a reutilização aprimoradas trazidas pela A programação OO pode até evitar a necessidade de testes, pesquisadores logo perceberam que não só tudo já aprendi sobre teste de software em geral também aplicado a OO código, mas também o desenvolvimento OO introduziu novos riscos e dificuldades, aumentando assim a necessidade e a complexidade de teste [14]. Em particular, entre os principais mecanismos do desenvolvimento OO, o encapsulamento pode ajudar a esconder bugs e torna o teste mais difícil; herança requer extenso reteste de código herdado; e polimorfismo e vinculação dinâmica exigem novos modelos de cobertura. Além disso, apropriado estratégias para testes de integração incremental eficazes são necessários para lidar com o espectro complexo de possíveis e dependências dinâmicas entre classes.

Testes baseados em componentes. No final dos anos 90, o desenvolvimento baseado em componentes (CB) surgiu como a abordagem definitiva que renderia um rápido desenvolvimento de software com menos Recursos. Os testes dentro deste paradigma introduziram novos desafios, que distinguiríamos entre e teórico em espécie. Do lado técnico, os componentes deve ser genérico o suficiente para ser implantado em diferentes plataformas e contextos, portanto, o usuário do componente precisa reteste o componente no sistema montado onde está implantado. Mas o problema crucial aqui é enfrentar a falta de informações para análise e teste de componentes desenvolvidos externamente. Na verdade, enquanto as interfaces de componentes são descritos de acordo com modelos de componentes específicos, estes não fornecem informações suficientes para testes funcionais. Portanto, a pesquisa tem defendido que as informações apropriadas, ou mesmo os próprios casos de teste (como no *teste interno),* sejam empacotados junto com o componente para facilitar teste pelo usuário do componente, e também que o "contrato" que os componentes cumprem devem ser explicitados, para permitir a verificação.

O teste de sistemas baseados em componentes também foi listado como um desafio fundamental no FOSE2000. Para um mais recente pesquisa ver [70].

O que continua sendo um problema permanente em aberto é o lado teórico do teste CB: como podemos inferir propriedades interessantes de um sistema montado, a partir dos resultados de testando os componentes isoladamente? Os fundamentos teóricos do *teste composicional* ainda permanecem uma importante pesquisa desafio destinado a durar, e discutimos algumas direções para pesquisa na Seção 5.1.

Teste de protocolo. Os protocolos são as regras que regem a comunicação entre os componentes de um sistema, e estes precisam ser especificados com precisão para facilitar a interoperabilidade. O teste de protocolo visa verificar a conformidade de implementações de protocolo em relação a suas especificações. Estes últimos são divulgados por organizações padrão, ou por consórcios de empresas. Em certos casos, também é lançado um conjunto de testes de conformidade padrão.

Impulsionada pela pressão de permitir a comunicação, a pesquisa em teste de protocolo prosseguiu ao longo de um e, de certa forma, trilha privilegiada em relação ao software teste. De fato, graças à existência de especificações precisas baseadas no estado do comportamento desejado, a pesquisa poderia muito cedo desenvolver métodos formais avançados e ferramentas para testando a conformidade com as especificações padrão estabelecidas [16].

Como esses resultados foram concebidos para um campo de aplicação restrito e bem definido, eles não se aplicam prontamente a testes de software em geral. No entanto, o mesmo problema original de garantir a interação adequada entre componentes remotos e serviços surge hoje em uma escala mais ampla para qualquer software moderno; portanto, a pesquisa de teste de software poderia frutificar plenamente com o teste de protocolo sobre o hábito de adotar especificações formais padronizadas, que é a tendência em aplicativos modernos orientados a serviços. Vice-versa, enquanto cedo protocolos eram simples e facilmente tratáveis, hoje o foco está mudando para níveis mais altos de protocolos de comunicação, e daí a praga da complexidade mais típica do software os testes também começam a se tornar urgentes aqui. Portanto, o separação conceitual entre testes de protocolo e testes gerais problemas de teste de software está desaparecendo progressivamente.

Teste de confiabilidade. Dada a onipresença do software, sua confiabilidade, ou seja, a probabilidade de operação livre de falhas para um período de tempo especificado em um ambiente especificado, impactos hoje qualquer produto tecnológico. O teste de confiabilidade reconhece que nunca podemos descobrir a última falha e, portanto, usando o perfil operacional para conduzir os testes, ele tenta eliminar aquelas falhas que se manifestariam com mais frequência: intuitivamente, o testador imita como os usuários empregará o sistema. A confiabilidade de software geralmente é inferida com base em modelos de *confiabilidade:* diferentes modelos devem ser utilizado, dependendo se as falhas detectadas são removidas, caso em que a confiabilidade aumenta ou não, quando confiabilidade é apenas certificada.

A pesquisa em confiabilidade de software cruzou a pesquisa em testes de software de muitas maneiras frutíferas. Modelos de confiabilidade de software foram estudados ativamente nos anos 80 e anos 90 [58]. Esses modelos agora estão maduros e podem ser projetados no processo de teste, fornecendo orientação quantitativa sobre como e quanto testar. Por exemplo, isso foi feito por Musa em sua abordagem Software-Reliability-Engineered Testing (SRET) ([58], Cap.6), e também é defendida no processo de desenvolvimento de Salas Limpas, que busca a aplicação de abordagens de teste estatístico para obter certificados medidas de confiabilidade [69].

Infelizmente, a prática de testes de confiabilidade não

prosseguiu na mesma velocidade dos avanços teóricos em confiabilidade de software, provavelmente por ser (percebida como) uma atividade complexa e cara, mas também pela dificuldade inerente de identificar o perfil operacional necessário [41].
No entanto, hoje a demanda por confiabilidade e outras qualidades de confiabilidade está crescendo e, portanto, surge a necessidade de abordagens práticas para testar coerentemente o comportamento funcional e extra funcional de sistemas modernos de software intensivo, conforme discutido mais adiante na Seção 5.5. Para futuros desafios em testes de confiabilidade nos referimos ao roteiro de Lyu [57].

5. As rotas

Nesta seção descrevemos os sonhos de teste de software pesquisa, e para cada um deles alguns desafios relevantes para ser endereçadas para avançar o estado da arte mais próximo do sonho em si.

5.1. Sonho: Teoria do teste universal

Um dos sonhos de longa data da pesquisa de teste de software seria construir uma teoria abrangente e sólida que é útil para fazer backup e nutrir a tecnologia de teste. Por pedindo uma teoria de teste "universal", quero dizer uma teoria coerente e estrutura rigorosa à qual os testadores podem se referir para entender os pontos fortes e limitações relativos das técnicas de teste existentes, e ser orientado na seleção da mais adequada, ou mistura dos mesmos, dadas as presentes condições.

O trabalho seminal na teoria de teste de software remonta a final dos anos 70, quando as noções relacionadas de um "confiável" [45] ou um conjunto de testes "ideal" [36] foram introduzidos pela primeira vez. Graças a neste trabalho pioneiro, temos argumentos lógicos para corroborar o fato bastante óbvio de que o teste nunca pode ser exato [25]. Mas tal conhecimento em *si,* além do alerta que apesar de muitos testes terem passado, o software ainda pode ser defeituoso, fornece pouca orientação sobre o que é então que podemos concluir sobre o software testado após ter aplicado uma técnica selecionada, ou indo ainda mais longe, sobre como poderíamos ajustar dinamicamente nossa estratégia de teste à medida que proceder com a acumulação dos resultados dos testes, tendo em conta o que observamos.

O sonho seria ter uma máquina de teste que amarra uma declaração do objetivo para testar com o mais eficaz técnica, ou combinação de técnicas, para adotar, juntamente com as suposições subjacentes que precisamos fazer. Para alcançar esse sonho, a pesquisa precisa enfrentar vários desafios.

Desafio: hipóteses de teste explícitas

Em última análise, dado que o teste é necessariamente baseado em aproximações (lembre-se que começamos com a afirmação de que testar equivale a amostrar algumas execuções), essa teoria universal também deve explicitar para cada técnica quais são suas suposições subjacentes, ou *hipóteses de teste:* formalizado pela primeira vez em [6], o conceito de uma hipótese de teste justifica a prática de teste comum e intuitiva por trás do seleção de cada conjunto de teste finito, pelo qual uma amostra é retirada como representante de várias execuções possíveis. Com Com exceção de algumas abordagens de teste formais, as hipóteses de teste geralmente são deixadas implícitas, embora seja de extrema importância importância de torná-los explícitos. Desta forma, se realizarmos testes "exaustivos" de acordo com o critério de teste selecionado, a partir da conclusão com sucesso da campanha de testes poderíamos concluir justificadamente que o software está *correto sob as hipóteses declaradas:* ou seja, ainda sabemos que realmente o software pode estar com defeito, mas também sabemos o que assumiram ser verdadeiros na origem e poderiam, em vez disso, ser falso. Esta noção é semelhante à de um "modelo de falha", que é usado no domínio do teste de protocolo, onde diz-se que um conjunto de testes fornece garantia de cobertura de falhas para um dado modelo de falha.

Um resumo das hipóteses de teste por trás das mais comuns abordagens de teste é dada, por exemplo, por Gaudel [34], que menciona entre outras Hipótese de Uniformidade para os critérios de partição de caixa preta (supõe-se que o software se comporte uniformemente dentro de cada subdomínio de teste) e Hipótese de Regularidade, usando uma função de tamanho sobre os testes. Tal pesquisa deve ser alargado para abranger outros critérios e abordagens.
As hipóteses de teste devem ser modularizadas pelo objetivo do teste: diferentes teorias/hipóteses seriam necessárias ao testar a confiabilidade, ao testar a depuração e em breve.

Ao explicitar nossas suposições, esse desafio multas o PORQUE observamos algumas execuções.

Desafio: testar a eficácia

Para estabelecer uma teoria útil para teste, precisamos avaliar a eficácia dos critérios de teste existentes e novos. Embora como dito entre as Conquistas, diversos estudos comparativos tenham sido realizados para este fim, o Fose2000 já sinalizou que pesquisas adicionais eram necessárias para fornecer *evidências analíticas, estatísticas ou empíricas do eficácia dos critérios de seleção de teste em revelar falhas,* para *entender as classes de falhas para as quais o critérios são úteis.* Esses desafios ainda estão vivos. Em particular, agora é geralmente aceito que é sempre mais eficaz usar uma combinação de técnicas, em vez de aplicar apenas uma, mesmo se considerada a mais poderosa, porque cada técnica pode visar diferentes tipos de falhas, e sofrem de um *efeito de saturação* [58].

Vários trabalhos contribuíram para uma melhor compreensão das limitações inerentes de diferentes abordagens de teste, começando com o artigo seminal de Hamlet e Taylor discutindo teste de partição e seus pressupostos subjacentes [41]. Ainda mais trabalho é necessário, notadamente para contextualizar tais comparações com a complexidade dos testes do mundo real (por exemplo, Zhu e He [90] analisam a adequação dos testes sistemas concorrentes), bem como refinar suposições no

bases de tais comparações, para levar em conta os progressos na automação de testes. Por exemplo, mesmo a controvérsia convencional sobre os méritos relativos de métodos *sistemáticos versus aleatórios* técnicas é hoje revitalizada pelo surgimento sofisticado métodos para automatizar a geração de testes aleatórios (que são discutido na Seção 5.3).

Este desafio aborda o PORQUÊ, COMO e COMO MUITOS testes, em termos de falhas (quais e como muitos) visamos.

Desafio: Teste de composição

A complexidade cada vez maior do software torna os testes difícil e impede o progresso em direção a qualquer sonho de pesquisa, teoria de teste incluída. Tradicionalmente, a complexidade do teste tem sido abordada pela antiga estratégia *divide et impera* , ou seja, a teste de um grande sistema complexo é decomposto no testes separados de suas "peças" de composição. Muitas pesquisas anteriores abordaram técnicas e ferramentas para ajudar nas estratégias de teste incremental na organização e execução de agregações de componentes progressivamente diferentes. Por exemplo, diferentes estratégias têm sido propostas para gerar o *ordem de teste* que é mais eficiente em minimizar a necessidade de stubs e andaimes, veja [18] para uma comparação recente. O problema tornou-se particularmente relevante hoje com o surgimento do paradigma de desenvolvimento CB, como já discutido no FOSE2000, e ainda mais com o aumento adoção de composições de sistemas dinâmicos.

Portanto, precisamos de um capítulo de teoria de teste abordando testes composicionais: precisamos entender como podemos *reutilizar os resultados do teste* observados no teste separado do peças individuais (sejam elas Unidades ou Componentes ou Subsistemas), em particular quais conclusões podem ser inferidas sobre o sistema resultante da composição e quais casos de teste adicionais devem ser executados na integração. Diversos direções promissoras de estudo têm sido empreendidas em diferentes contextos. Por exemplo, Hamlet propôs uma simples teoria fundamental para *confiabilidade de software baseado em componentes* [40], recentemente estendida com a noção de estado [39], mas trabalho ainda é necessário para torná-lo geralmente aplicável.

Blundell e coautores [15] estão investigando a aplicação ao teste de raciocínio *de garantia de* suposição, uma técnica de verificação usada para inferir propriedades globais do sistema por verificando componentes individuais isoladamente. Desde ser capaz de verificar um componente individualmente, precisamos fazer suposições sobre seu contexto, a verificação assume-garantia verifica se um componente garante uma propriedade ao somar que o contexto se comporta corretamente, e então simetricamente o contexto é verificado assumindo que o componente está correto. A promessa de testes de garantia presumida seria que observando os traços de teste dos componentes individuais pode-se inferir comportamentos globais.

A comunidade de testes de protocolo também está investigando ativamente os testes de composição. Por exemplo, van der Bijl e coautores [81] analisaram formalmente as composições paralelas ção de dois componentes de comunicação dados, com base na teoria do ioco-test [79], que funciona na transição rotulada Sistemas. Em particular, se dois componentes foram testados separadamente e provaram ser iococorretos, sua integração também é iococorreta? Os autores mostram que, em geral, isso não pode ser concluído, mas a resposta pode ser afirmativa para componentes cujas entradas são completamente especificadas [81]. Gotzhein e Khendek [37] em vez disso consideraram o código de cola para a integração de componentes de comunicação, produziram um modelo de falha para ele e desenvolveram um procedimento para encontrar os casos de teste para a cola.

Este desafio está claramente relacionado ao QUE testamos.

Desafio: Corpo de evidências empíricas

Hoje a importância da experimentação para o avanço da maturidade da disciplina de engenharia de software certamente não precisa ser sublinhado (Siøberg e coautores [77] discutem em profundidade os desafios de pesquisa enfrentados pelos métodos empíricos). Em todos os tópicos de pesquisa em engenharia de software, estudos empíricos são essenciais para avaliar as técnicas propostas. e práticas, para entender como e quando eles funcionam, e para melhorá-los. Isso é obviamente verdadeiro para testes, pois bem, em que *a experimentação controlada é uma metodologia de pesquisa indispensável* [26].

No FOSE2000, Harrold identificou a esse respeito as seguintes necessidades: experimentos controlados para demonstrar técnicas; recolha e disponibilização pública de conjuntos de temas experimentais; e experimentação industrial. Tudo tais necessidades podem ser confirmadas hoje, e uma revisão mais recente dos experimentos de técnicas de teste [48] infelizmente concluiu que *mais da metade do conhecimento* existente (técnica de teste) *é baseado em impressões e percepções e, portanto, desprovido de qualquer fundamento formal.*

De fato, experimentando, devemos ter como objetivo produzir um corpo empírico de conhecimento que está na base construindo e evoluindo a teoria para teste. Nós precisamos examinar fatores que podem ser usados para estimar antecipadamente onde falhas residem e por quê, para que os recursos de teste possam ser alocado. E para isso precisamos ter experimentos significativos, em termos de escala, dos sujeitos usados, e de contexto, o que nem sempre é realista. Banalmente, para todos três aspectos, a barreira é o custo: estudos empíricos cuidadosos em produtos de grande escala, dentro de contextos do mundo real (como como [66]), e possivelmente replicado por vários profissionais os testadores para obter resultados geralmente válidos são, obviamente, proibitivamente caros. Uma saída possível para superar tal barreira poderia ser a de juntar as forças de várias pesquisas grupos, e realizando distribuídos e amplamente replicados experimentar. A ideia seria, grosso modo, lançar uma espécie de iniciativa de "Experiências Abertas", da mesma forma que vários projetos Open Source foram conduzidos com sucesso. A consciência da necessidade de unir forças está se espalhando, e alguns esforços já estão sendo feitos para construir repositórios de dados compartilhados, como em [26], ou experiências distribuídas

testbeds mentais, como a coleção PlanetLab [68] de
mais de 700 máquinas conectadas ao redor do mundo.

Este é um desafio fundamental que abrange todos os seis caracterizando perguntas.

5.2. Sonho: modelagem baseada em testes

Atualmente, grande parte da pesquisa se concentra em testes baseados em modelos, que discutiremos a seguir. A ideia principal
é usar modelos definidos na construção de software para conduzir o processo de teste, em particular para gerar automaticamente os casos de teste. A abordagem pragmática que a pesquisa de teste toma é a de seguir o que é a tendência atual na modelagem: qualquer que seja a notação usada, digamos, por exemplo, UML ou Z, tentamos adaptar a ele uma técnica de teste tão eficaz quanto possível.

Mas se nos for permitido considerar o sonho, do ponto de vista do testador, a situação ideal seria reverter isso abordagem em relação ao que vem primeiro e o que vem depois: em vez de pegar um modelo e ver como podemos melhor explorá-lo para teste, vamos considerar como devemos idealmente construir o modelo para que o software possa ser efetivamente testado. Não seria bom se os desenvolvedores - totalmente cientes da importância e dificuldade de testes completos baseados em modelos se preocupam antecipadamente com testes e derivam modelos apropriados já aprimorado com informações instrumentais para testes? Esta é a motivação pela qual estamos aqui invertendo a atual visão de "testes baseados em modelos" em direção ao sonho de "modelagem baseada em testes".

É certo que este é apenas um novo termo para uma ideia antiga, pois na verdade já podemos encontrar várias direções de pesquisa que mais ou menos explicitamente têm trabalhado para abordar esse sonho. Por um lado, essa noção de modelagem baseada em testes está intimamente relacionada, na verdade, um fator da velha ideia de "Design-para-testabilidade", que se preocupa principalmente com projetar software para melhorar a Controlabilidade (das entradas) e a Observabilidade (das saídas). Mas também relacionado pode ser visto abordagens anteriores para testar com base em afirmações, e mais recentes baseados em Contratos. Asserções em particular foram desde cedo reconhecidas como uma ferramenta útil para testando, pois podem verificar em tempo de execução o estado interno de um programa. Descendentes das assertivas, os contratos foram originalmente introduzido no nível de classes para software OO, e foram então adotados para componentes: intuitivamente, um contrato estabelece um acordo "legal" entre duas partes que interagem, que se expressa por meio de três tipos diferentes de afirmações: pré-condições, pós-condições e invariantes. O passo para usar esses contratos como referência para testes é curto, e muita pesquisa interessante está sendo com resultados promissores, por exemplo, [20, 52].

Desafio: teste baseado em modelo

As tendências frequentemente citadas neste artigo de níveis crescentes de complexidade e necessidades de alta qualidade estão impulsionando o custo de testar mais alto, a ponto de as práticas tradicionais de teste se tornarem antieconômicas, mas felizmente no outro extremo, o uso crescente de modelos no desenvolvimento de software rende perspectiva de remover a principal barreira à adoção de testes baseados em modelos, que são habilidades de modelagem (formais).

O teste baseado em modelo é na verdade uma espécie de filme de *volta para o futuro* para teste de software. De fato, a ideia de testes baseados em modelos existe há décadas (Moore [62] iniciou a pesquisa sobre a geração de testes baseados em FSM em 1956!), mas foi nos últimos anos que vimos um aumento no interesse em aplicá-lo a aplicações reais (para um introdução às diferentes abordagens e ferramentas em testes baseados em modelos ver, por exemplo, [80]).

No entanto, a adoção industrial de testes baseados em modelos permanece baixo e os sinais do avanço previsto pela pesquisa são fracos. Portanto, além dos desafios teóricos, os pesquisadores estão hoje focando em como vencer o barreiras à ampla adoção. Existem técnicas importantes e questões relacionadas ao processo pendentes.

Uma questão amplamente reconhecida é como podemos *combinar* diferentes estilos de modelagem (como baseado em transição, pré/pós-baseado em condição e baseado em cenário). Por exemplo, precisamos encontrar maneiras eficazes de compor abordagens baseadas em estado e baseadas em cenário [9, 38]. Na Microsoft, onde os modelos baseados teste tem sido defendido há vários anos, mas com acompanhamento limitado, uma abordagem *multiparadigmática* [38] é agora perseguido para favorecer uma adoção mais ampla. A ideia é que modelos oriundos de diferentes paradigmas e expressos em qualquer notação pode ser usado perfeitamente dentro de um integrado meio Ambiente. A lição aprendida é de fato que forçar os usuários usar uma nova notação não funciona, em vez disso, o núcleo da uma abordagem de teste baseada em modelo deve ser agnóstica e permitir os desenvolvedores usam notações de programação e ambientes existentes [38]. Também precisamos de maneiras de combinar modelos baseados em critérios com outras abordagens; por exemplo, uma ideia promissora é usar testes sobre simulações [72] para otimizar o teste suite e para impulsionar os testes.

As questões relacionadas ao processo dizem respeito à necessidade de integrar prática de teste baseada em modelo em processos de software atuais: talvez as questões cruciais aqui sejam as duas necessidades de gerenciamento de teste de fazer modelos de teste como abstratos possível, mantendo a capacidade de gerar testes executáveis de um lado; e de manter a rastreabilidade de requisitos para testes ao longo do processo de desenvolvimento, o outro. Finalmente, também precisamos de ferramentas de força industrial para autoria e modelagem interativa, que podem ajudar a reduzir o educação inadequada dos testadores atuais (ou talvez os requisitos excessivos de especialização das técnicas propostas).

Um caso especial de teste baseado em modelo é a *conformidade teste,* ou seja, verificar se o sistema em teste está de acordo com sua especificação, sob alguma relação definida (que está estritamente relacionado com as hipóteses de teste anteriormente discutido). A partir da década de 70, muitos algoritmos

foi proposto; uma ampla visão geral recente dos atuais
desafios é dado no tutorial de Broy e coautores sobre
teste baseado em modelo para sistemas reativos [21]. Os resultados
alcançados até agora são impressionantes em termos teóricos, mas
muitos dos métodos propostos dificilmente são aplicáveis a sistemas realistas,
embora várias ferramentas tenham sido produzidas
e alguns deles são aplicados em domínios especializados. UMA
uma boa visão geral de ferramentas para testes de conformidade baseados
em modelos construídos em uma teoria sólida é fornecida por Belinfante e
coautores [4], que destacam a necessidade de melhorar e facilitar
a aplicação da teoria.

Este desafio refere-se a COMO selecionamos qual teste execuções a observar, e em parte ao QUANTO de eles.

Desafio: teste baseado em antimodelo

Paralelamente aos testes baseados em modelos, vários esforços estão sendo
dedicado a novas formas de teste que estão diretamente
na análise de execuções de programas, e não em uma
modelo a priori. Em vez de pegar um modelo, invente a partir dele
um plano de teste e, portanto, comparar os resultados do teste com o
modelo, essas outras abordagens coletam informações da execução do
programa, seja após solicitar ativamente alguma execução, ou passivamente
durante a operação, e tentam sintetizar
a partir destes algumas propriedades relevantes de dados ou de comportamento.
Pode haver casos em que os modelos simplesmente não existem
ou não são acessíveis, como para COTS ou componentes legados; outros
casos em que a arquitetura do sistema global
não é decidido a priori, mas é criado e evolui dinamicamente ao longo da
vida de um sistema; ou, um modelo é originalmente
criado, mas durante o desenvolvimento torna-se progressivamente
menos útil, pois sua correspondência com a implementação não é aplicada e
é perdida. Assim, simetricamente a
teste baseado em modelo, temos que (explícita ou implicitamente)
um modelo é derivado a posteriori por meio de testes, aos quais nos referimos
como *testes baseados em anti-modelo,* conforme previsto em [11]. Por este
termo nos referimos a todas as várias abordagens que por meio de
teste, engenharia reversa de um modelo, na forma de uma invariante sobre
as variáveis do programa, ou na forma de uma máquina de estado, ou um
sistema de transição rotulado, ou uma sequência
diagrama, e assim por diante, e então verificar tal modelo para detectar
se o programa se comporta adequadamente.

O teste baseado em antimodelo pode contar com os grandes avanços da
análise dinâmica de programas, que é uma ferramenta muito ativa
disciplina de pesquisa hoje, como discutido por Canfora e Di
Penta [22].

Precisamos ser capazes de inferir propriedades do sistema raciocinando
sobre um conjunto limitado de traços observados, ou mesmo traços parciais,
já que podemos observar os componentes que formam o sistema. Em um
trabalho recente, Mariani e Pezze [59] propõem a técnica BCT para derivar
modelos de comportamento para monitorados.
componentes COTS. Em sua abordagem os modelos comportamentais
consistem em ambos os modelos de E/S, obtidos por meio do
conhecido detector invariante dinâmico Daikon [30], e
modelos de interação, na forma de Autômatos de Estados Finitos.
Curiosamente, esses modelos derivados podem ser usados posteriormente
para testes baseados em modelos se e quando os componentes forem
substituídos por novos. Um desafio relacionado é manter a
modelos derivados dinamicamente atualizados: dependendo da
tipo de atualização para o sistema, também o modelo pode precisar ser
refinado, como também observam Mariani e Pezze, traçando algumas
estratégias possíveis.

Este desafio também se refere ao COMO e ao COMO MUITO observamos de execuções de testes.

Desafio: Testar oráculos

Estritamente relacionado ao planejamento de testes, e especificamente ao
problema de como derivar os casos de teste, é a questão de decidir se um
resultado de teste é aceitável ou não. este
corresponde ao chamado "oráculo", idealmente um mágico
método que fornece as saídas esperadas para cada dado
casos de teste; mais realista, um motor/heurística que pode
emitir um veredicto de aprovação/reprovação sobre as saídas de teste observadas.

Embora seja óbvio que uma execução de teste para a qual
não são capazes de discriminar entre o sucesso ou o fracasso é
um teste inútil, e embora a criticidade deste problema
foi levantado muito cedo na literatura [85], o problema do oráculo tem
recebido pouca atenção pela pesquisa e na prática
poucas soluções alternativas ainda existem para a visão humana. Mas
tal estado de coisas que já hoje não é satisfatório,
com a crescente complexidade e criticidade do software
aplicativos está destinado a se tornar um obstáculo de bloqueio para
automação de teste confiável (na verdade, os oráculos de teste desafiam
também sobrepõe a rota para a automação de teste). De fato,
a precisão e a eficiência dos oráculos afetam muito o custo/eficácia dos
testes: não queremos que as falhas nos testes passem
despercebido, mas por outro lado também não queremos ser
notificados de muitos falso-positivos, que desperdiçam recursos importantes.
Precisamos encontrar métodos mais eficientes para realizar e automatizar
oráculos, modular a informação que
está disponível.

Um levantamento crítico das soluções oracle é fornecido pela Baresi
e Young [1], que concluem destacando áreas onde
espera-se o progresso da pesquisa, que tomamos emprestado e expandimos
abaixo: *Estado e comportamento concreto vs. abstrato:* testes baseados em
modelos prometem aliviar o problema do oráculo, já que
o mesmo modelo pode atuar como oráculo; entretanto, para oráculos
baseados em descrições abstratas do comportamento do programa, o
permanece o problema de preencher a lacuna entre o concreto
entidades observadas e as entidades abstratas especificadas; *Parcialidade:*
oráculos parciais plausíveis são a única solução viável para
automação oracular: o desafio é encontrar o melhor equilíbrio entre precisão
e custo; *Quantificação:* para teste ou acles implementados via linguagens de
especificação executáveis
um compromisso entre expressividade e eficiência deve
ser buscado, até agora não há um equilíbrio ideal claro nem qualquer
abordagem totalmente satisfatória para acomodar quantificadores;
Oráculos e seleção de casos de teste: idealmente, os oráculos devem ser

ortogonal à seleção de casos de teste; no entanto, em modelos
testar os modelos disponíveis são frequentemente usados para derivar
classes e oráculos de teste específicos da classe de teste juntos.

Este desafio refere-se à pergunta POR QUE, no sentido do que testamos.

5.3. Sonho: teste 100% automático

A automação de longo alcance é uma das maneiras de manter a análise e os testes de qualidade alinhados com a quantidade crescente e complexidade do software. Pesquisa em engenharia de software coloca grande ênfase na automação da produção de software, com uma grande quantidade de ferramentas de desenvolvimento modernas gerando quantidades de código cada vez maiores e mais complexas com menos esforço. O outro lado da moeda é o grande perigo de que o métodos para avaliar a qualidade do software assim produzido, em métodos de teste específicos, não pode acompanhar o ritmo de tais métodos de construção de software.

Grande parte da pesquisa atual de testes visa melhorar o grau de automação alcançável, seja desenvolvendo técnicas avançadas para gerar as entradas de teste (esta
desafio é expandido abaixo), ou, além da geração de teste, encontrando procedimentos de suporte inovadores para automatizar o processo de teste.

O sonho seria um poderoso ambiente de teste integrado que, por si só, à medida que um software é concluído e implantado, pode cuidar automaticamente de possivelmente instrumentá-lo e gerar ou recuperar os dados necessários.
código de andaime (drivers, stubs, simuladores), gerando o casos de teste mais adequados, executando-os e finalmente emitindo um relatório de teste. Essa ideia, embora quimérica, atraiu seguidores, por exemplo, na primeira iniciativa patrocinada pela DARPA para Teste Perpétuo (também mencionado no FOSE2000) e mais recentemente na abordagem de teste contínuo de Saff e Ernst [74], que visa exatamente executar testes em segundo plano em a máquina do desenvolvedor enquanto eles programam.

Passos bastante promissores foram dados recentemente para realização desse sonho de *testes unitários,* amplamente reconhecida como uma fase essencial para garantir a qualidade do software, pois ao escrutinar unidades individuais isoladamente pode detectar precocemente até mesmo falhas sutis e profundamente ocultas que dificilmente seria encontrado em testes de sistema. Infelizmente, o teste de unidade geralmente é mal executado ou ignorado porque muito caro. Precisamos de abordagens para torná-lo viável dentro dos processos de desenvolvimento industrial. Um componente importante do alto custo do teste de unidade é a enorme quantidade de codificação extra necessária para simular o ambiente onde a unidade será executada e para realizar o
necessária verificação funcional das saídas da unidade. Para aliviar tais tarefas, de grande sucesso entre desenvolvedores foram os frameworks pertencentes à família de XUnit. Dentre destes, o mais bem sucedido é JUnit [47], que permite automatizar a codificação de casos de teste Java e seu gerenciamento mento, e tem favorecido a disseminação do já mencionado desenvolvimento orientado a testes.

No entanto, tais frameworks não auxiliam na geração de testes e simulação do ambiente. Nós gostaríamos de empurrar automação ainda mais, como por exemplo na abordagem Directed Automated Random Testing (DART) [35], que automatiza o teste de unidade por: extração de interface automatizada por análise estática de código-fonte; geração automatizada de driver de teste aleatório para esta interface; e análise dinâmica do comportamento do programa durante a execução do teste aleatório casos, visando gerar automaticamente novas entradas de teste que pode direcionar a execução ao longo de caminhos alternativos do programa.

Outro exemplo é fornecido pela noção de “software agitação” [17], uma técnica de teste de unidade automática suportada pela ferramenta comercial Agitator, que combina diferentes análises, como execução simbólica, resolução de restrições, e geração de entrada aleatória direcionada para geração dos dados de entrada, juntamente com o já citado sistema Daikon [30].

Ainda outra abordagem é constituída por Microsoft Param eterized Unit Tests (PUT) [78], ou seja, testes de unidade codificados que não são fixos (como acontece com os programados no XUnit frameworks), mas dependem de alguns parâmetros de entrada. PUTs pode descrever o comportamento abstrato de forma concisa, usando técnicas de execução simbólica e resolvendo restrições. encontre entradas para PUTs que alcancem alta cobertura de código.

Os três exemplos citados certamente não são exaustivos da etapa bastante ativa e frutífera que a automação de testes está desfrutando atualmente. A tendência subjacente comum que surge é o esforço de *combinar* e de forma eficiente a engenharia avanços provenientes de *diferentes tipos de análise,* e isso, juntamente com o aumento exponencial dos recursos computacionais disponíveis, pode ser a direção realmente vencedora para o sonho de 100% de automação.

Desafio: geração de entrada de teste

A pesquisa em geração automática de entradas de teste sempre foi muito ativa e atualmente tantos tecnologias estão sob investigação que mesmo dedicando o todo o artigo apenas para este tópico não renderia espaço para cobri-lo adequadamente. O que desanima é que até hoje todo esse esforço produziu impacto limitado na indústria, onde a atividade de geração de testes permanece em grande parte manual (como relatado, por exemplo, em ISSTA 2002 Painel[7]). Mas, finalmente, a combinação de progressos teóricos nas tecnologias subjacentes, como execução, verificação de modelos, prova de teoremas, estática e análises dinâmicas, com avanços tecnológicos em modelagem padrões de força da indústria e com poder computacional disponível parece tornar este objetivo mais próximo e tem revitalizou a fé dos pesquisadores.

Os resultados mais promissores são anunciados para vir de três direções, e especialmente de sua convergência mútua: a já amplamente discutida abordagem baseada em modelos, maneiras “inteligentes” de aplicar a geração aleatória, e um rico

variedade de técnicas baseadas em pesquisa usadas tanto para
e geração de teste de caixa preta.

No que diz respeito *à geração de testes baseados em modelos,* claramente a pesquisa está colocando grandes expectativas nessa direção, uma vez que a ênfase no uso de modelos (formais) para orientar os testes reside exatamente no potencial de derivação automatizada de
casos de teste. Referências a trabalhos em andamento já foram fornecido ao falar de desafios de teste baseado em modelo.
Muitas das ferramentas existentes são baseadas em estado e não lidam com com dados de entrada. Pesquisas são necessárias para entender como podemos incorporar modelos de dados dentro de
abordagens baseadas no estado; uma direção poderia ser a introdução do simbolismo sobre os modelos baseados no Estado, que
poderia evitar a explosão do espaço de estados durante a geração do teste e preservaria as informações presentes nos dados
definições e restrições para uso durante a seleção do teste
processo. Por exemplo, tal abordagem está sendo realizada
pela abordagem de Sistemas de Transição Simbólica [33], que aumentar os sistemas de transição com uma noção explícita de dados e fluxo de controle dependente de dados. Também precisamos aumentar a eficiência e o potencial da geração automatizada de testes reutilizando dentro de abordagens baseadas em modelos os mais recentes avanços em provas de teoremas, verificação de modelos e técnicas de satisfação de restrições. Em particular, desde meados
Nos anos 70, a execução simbólica pode ser considerada a abordagem mais tradicional para a geração automatizada de dados de teste. Tal abordagem, que foi posta de lado há algum tempo, porque
das muitas dificuldades subjacentes, é hoje revitalizado por
o ressurgimento de linguagens fortemente tipadas e pelo desenvolvimento de solucionadores de restrições mais poderosos; Lee e
coautores [53] pesquisam os desenvolvimentos mais promissores.

No que se refere à *geração de testes aleatórios,* esta era considerada uma abordagem rasa, no que diz respeito às técnicas sistemáticas, consideradas mais abrangentes e capazes de
encontrar casos de canto importantes que provavelmente seriam ignorados por técnicas aleatórias. No entanto, estudos anteriores compararam principalmente implementações de espantalho de
testes a implementações sofisticadas de técnicas sistemáticas. Hoje, vários pesquisadores estão propondo implementações inteligentes de testes aleatórios que parecem superar
geração sistemática de testes, senão em termos de viabilidade.
A ideia subjacente de tais abordagens é que o
geração é melhorada dinamicamente, explorando feedback informações coletadas à medida que os testes são executados. Por exemplo, Sen e coautores construíram em cima da abordagem DART citada, uma noção de “testes concólicos” [75], que é a
combinação de testes concretos (aleatórios) com execução simbólica. As execuções concretas e simbólicas são executadas em
paralelos e “ajudam-se” uns aos outros. Em vez disso, Pacheco e coau thors [67] selecionam aleatoriamente um caso de teste, executam-no e verificam contra um conjunto de contratos e filtros.

A direção mais promissora, então, é descobrir maneiras eficientes de combinar os respectivos pontos fortes de sistemas sistemáticos. (baseado em modelo) e testes aleatórios.

Finalmente, no que diz respeito *à geração de testes baseados em busca,* este consiste em explorar o espaço de soluções (o teste
casos) para um critério de teste selecionado, usando técnicas metaheurísticas que direcionam a busca para o potencial
áreas mais promissoras do espaço de entrada. A característica atraente é que esta abordagem parece ser frutífera
a uma gama ilimitada de problemas; uma pesquisa recente é fornecida por McMinn [60]. A geração de dados de teste baseada em pesquisa é apenas uma aplicação possível de engenharia de software baseada em busca [42].

Este desafio aborda como as observações são gerado.

Desafio: abordagens de teste específicas de domínio

As linguagens específicas de domínio emergem hoje como um eficiente solução para permitir que especialistas dentro de um domínio expressem especificações abstratas mais próximas de suas exigências de expressão, e que podem então ser traduzidas automaticamente
em implementações otimizadas. Os testes também podem se beneficiar ao restringir o escopo de aplicação às necessidades de
um domínio específico.

A pesquisa deve abordar como o conhecimento do domínio pode melhorar o processo de teste. Precisamos estender abordagens específicas de domínio para o estágio de teste e, em particular, para encontre métodos e ferramentas específicos de domínio para automatizar testes por push. Testes específicos de domínio podem usar tipos especializados de abordagens, processos e ferramentas. Estes, por sua vez, precisam para fazer uso de modelagem e transformação personalizáveis ferramentas, portanto, o desafio também se sobrepõe à rota de modelagem baseada em teste.

Técnicas de teste para domínios específicos foram investigadas, por exemplo, para bancos de dados, para usabilidade de GUI, para web aplicações, para aviônica, para sistemas de telecomunicações; mas poucos trabalhos têm como foco o desenvolvimento de existem metodologias para explorar o conhecimento do domínio. Um interessante trabalho pioneiro deve-se a Reyes e Richardson [71], que logo desenvolveu um framework, chamado Siddhartha, para desenvolver drivers de teste específicos de domínio. Siddhartha implementou um método iterativo orientado a exemplos para desenvolver tradutores específicos de domínio das Especificações de teste para um driver específico de domínio. No entanto, exigia a
entrada do testador na forma de um driver geral, de exemplo codificado manualmente. Mais recentemente, Sinha e Smidts desenvolveram a técnica HOTTest [76], que se refere a um
linguagem específica de domínio tipada para modelar o sistema sob teste e demonstra como isso permite extrair e incorporar automaticamente requisitos específicos de domínio no teste
modelos. Acredito que tais esforços de pesquisa mostram resultados promissores ao demonstrar as melhorias de eficiência de abordagens específicas de teste de domínio e, espero, pesquisas adicionais seguirá.

Esse desafio refere-se ao tipo de aplicação que está sendo observada, ou seja, a pergunta O QUE.

Desafio: Testes online

Em paralelo com a visão tradicional do teste como uma atividade realizada antes do lançamento para verificar se um programa vai se comportar como esperado, um novo conceito de teste está surgindo em torno da ideia de monitorar o comportamento de um sistema na operação da vida real, usando análise dinâmica e autoteste técnicas.

Na verdade, o monitoramento de tempo de execução está em uso há mais de 30 anos, mas um interesse renovado surge devido à crescente complexidade e natureza onipresente dos sistemas de software. A terminologia não é uniforme e termos diferentes, como como monitoramento, testes em tempo de execução, testes on-line são usados em a literatura (Delgado e coautores [24] apresentam uma recente taxonomia). Todas as abordagens compartilham o objetivo de observar a comportamento do software em campo, com o objetivo de determinar se cumpre com o comportamento pretendido, detectando avarias ou também problemas de desempenho. Em alguns casos tenta-se uma recuperação online, noutros casos a análise é realizado off-line para produzir um perfil ou obter números de confiabilidade.

Uma característica distintiva dos testes on-line é que nós não precisamos conceber uma suíte de testes para estimular o sistema em teste, pois nos limitamos a observar passivamente o que acontece. De fato, em testes de protocolo de comunicação, as abordagens de monitoramento são chamadas de *testes passivos.* A troca de mensagens ao longo dos canais reais é rastreada, e o padrões são comparados com os especificados.

Em princípio, a "passividade" inerente de qualquer abordagem de teste on-line as torna menos poderosas do que as abordagens proativas. Todas as abordagens podem ser refeitas para verificar os rastreamentos de execução observados contra asserções que expressam propriedades desejadas ou contra invariantes de especificação. Por exemplo, Bayse e coautores [2] desenvolveram uma ferramenta que suportam testes passivos contra invariantes derivados de FSMs; eles distinguem entre invariantes simples e invariantes de obrigação. Mais em geral, a eficácia dos testes on-line dependerá da identificação das afirmações de referência. Além disso, a coleta de vestígios pode degradar o sistema atuação. Precisamos entender quais são os bons locais e o momento certo para sondar o sistema.

No meio do caminho entre os testes clássicos antes da implantação e o monitoramento passivo em campo, também poderíamos conceber testes proativos em campo, ou seja, estimular ativamente a aplicação após a implantação, seja quando algum eventos acontecem, por exemplo, um componente é substituído, ou em intervalos programados. Uma ideia semelhante é explorada no chamado framework de "audição" [10], propondo uma admissão fase de testes para web services.

Outra questão diz respeito à capacidade de realizar os testes no campo, especialmente para aplicações embarcadas que devem ser implantado em um ambiente com recursos limitados, onde a sobrecarga exigida pela instrumentação de teste não poderia ser viável. Uma nova direção de pesquisa interessante tem sido tomadas por Kapfhammer et ai. [49], que estão desenvolvendo o Ferramenta Juggernaut para testar aplicativos Java em um ambiente restrito. A ideia original deles é explorar as informações de execução, até então usadas para ajustar a suíte de testes, também para adaptando o ambiente de teste (eles usam em particular o descarregamento de código adaptativo para reduzir os requisitos de memória). Tal idéia é atraente e certamente pode encontrar muitos outros úteis formulários.

Este desafio diz respeito principalmente ao ONDE e QUANDO observar as execuções dos testes, com atenção especial para sistemas em evolução dinâmica.

5.4. Sonho: Engenharia de teste com eficácia maximizada

O objetivo final da pesquisa de teste de software, hoje como estava no FOSE2000, continua sendo o de engenharia econômica "métodos de teste práticos, ferramentas e processos para desenvolvimento de software de alta qualidade" [43].

Todas as questões teóricas, técnicas e organizacionais levantadas até agora devem ser reconciliadas em um processo de teste viável produzindo a máxima eficiência e eficácia (tanto resumido pelo termo eficácia). Além disso, a inerente tecnicismos e sofisticação de soluções avançadas propostas por pesquisadores devem ser escondidas atrás de soluções fáceis de usar ambientes integrados. Essa visão torna-se um empreendimento tão desafiador que nós a qualificamos como a mais alta sonho de pesquisa de teste de software.

O principal obstáculo para tal sonho, que mina todas as desafios de pesquisa mencionados até agora, é a crescente complexidade dos sistemas modernos. Esse crescimento da complexidade afeta não apenas o sistema em si, mas também os ambientes em que esses sistemas são implantados, fortemente caracterizados por variabilidade e dinamicidade.

Estratégias para alinhar o processo de desenvolvimento para maximizar a eficácia dos testes pertencem ao *design para testabilidade.* Já mencionamos a testabilidade ao falar de modelos e artefatos de pré-código que podem ser aprimorados para facilitar o teste. No entanto, testabilidade é um conceito mais amplo do que apenas como o sistema é modelado, envolve também características da implementação, bem como da técnica de teste si mesmo e seu ambiente de suporte. De fato, o design para testabilidade tem sido atribuído pelos profissionais como o principal direcionador de custo em testes [5].

A engenharia de teste com eficácia maximizada passa por muitos desafios, alguns dos quais são discutidos abaixo.

Desafio: Controlar a evolução

A maioria das atividades de teste realizadas na indústria envolve o reteste de código já testado para verificar se as alterações no o programa ou no contexto não afetou negativamente o sistema correção. Conforme apontado no FOSE2000, por causa da alto custo de *testes de regressão,* precisamos de técnicas eficazes para reduzir a quantidade de reteste, para priorizar a regressão

casos de teste e automatizar a reexecução dos casos de teste.

Em geral, precisamos de estratégias para escalar os testes de regressão para grandes sistemas compostos. Nós já discutimos questões teóricas por trás do teste de composição (consulte a Seção 5.1); também precisamos de abordagens práticas para testar as propriedades globais do sistema à medida que algumas partes do sistema são modificadas. Por exemplo, dado um componente que é originalmente projetado com uma arquitetura, precisamos entender como para testar se este pedaço de software evolui de acordo com sua arquitetura. Tal problema também é central para o teste de *famílias de produtos.*

Uma ideia relacionada é a *fatoração de teste,* que consiste em converter um teste de sistema de longa duração em uma coleção de muitos pequenos testes unitários. Esses testes unitários podem exercitar uma pequena parte o sistema exatamente da mesma maneira que o teste do sistema fizeram, mas sendo mais focados, eles podem identificar falhas em partes selecionadas específicas do sistema. A fatoração de teste é hoje investigado ativamente [73, 65, 28] uma vez que promete melhorias de ordem de magnitude na execução de testes de regressão.

Uma suposição básica comum de muitas abordagens existentes para testes de regressão é que um artefato de software pode ser assumido como referência, por exemplo, os requisitos especificação ou a arquitetura de software. Para aplicativos de software modernos que evoluem continuamente, muitas vezes dinamicamente de maneiras que não podemos saber de antemão, nem esses artefatos de pré-código nem mesmo o próprio código-fonte podem estar disponíveis, e o paradigma de teste está mudando de regressão testando para um dos testes contínuos em tempo de execução.

Portanto, uma questão crucial diz respeito a como manter o controle da qualidade do software que evolui dinamicamente no campo. Precisamos entender qual é o significado do teste de regressão em um contexto tão evolutivo, e como podemos pode modificar e estender a ideia básica de regressão seletiva testando, ou seja, com que frequência precisamos verificar a execução vestígios? como podemos comparar os traços tomados em diferentes intervalos temporais e entender se a evolução não trazer algum mau funcionamento?

Este desafio diz respeito amplamente ao QUE, ONDE e QUANDO reproduzimos algumas execuções seguindo a evolução do software.

Desafio: Aproveitar a população de usuários e os recursos

Já mencionamos a tendência emergente de validação contínua após a implantação, por meio de abordagens de teste (consulte a Seção 5.3), quando as técnicas tradicionais de teste off-line se tornam ineficazes. Desde software sistemas intensivos podem se comportar de maneira muito diferente em ambientes e configurações variados, precisamos de maneiras práticas de ampliar os testes on-line para cobrir o amplo espectro de comportamentos possíveis. Uma abordagem crescente para enfrentar esse desafio é aumentar as atividades internas de garantia de qualidade usando dados coletados dinamicamente do campo. Isto é promissor na medida em que pode ajudar a revelar espectros de uso real e expor problemas reais nos quais focar as atividades de teste laços e em que faltam testes. Por exemplo, dando cada usuário uma configuração padrão diferente, a base de usuários pode ser aproveitado para expor mais rapidamente conflitos ou problemas de configuração, como em [87]. E perfis em campo também pode ser usado para melhorar um determinado conjunto de testes, como em [56, 64, 29]. Embora também algumas iniciativas comerciais começam a aparecer, como o Customer Experience da Microsoft Programa de Melhoria [61], esses esforços ainda estão em seu infância, e um importante desafio de pesquisa deixado em aberto é como definir técnicas eficientes e eficazes para desencadear o potencial representado por um grande número de usuários, executando aplicativos semelhantes, em máquinas interconectadas. este desafio de alto nível envolve vários desafios mais específicos, entre os quais:

- Como podemos coletar dados de tempo de execução de programas executados no campo sem impor muita sobrecarga?

- Como podemos armazenar e minerar o material coletado (potencialmente enorme) quantidade de dados brutos para extrair efetivamente em formação?

- Como podemos usar efetivamente os dados coletados para aumentar e melhorar os testes e a manutenção internos Atividades?

Este desafio propõe que os usuários instanciem o ONDE e QUANDO examinar as execuções de software.

Desafio: padrões de teste

Já mencionamos no sonho de uma útil teoria do teste, que precisamos entender a eficácia relativa das técnicas de teste nos tipos de falhas que elas visam. Para a engenharia do processo de teste, precisamos coletar evidências para que tais informações sejam capazes de encontrar o padrão mais eficaz para testar um sistema. Isso é feito rotineiramente, quando, por exemplo, testes funcionais baseados nos requisitos são combinados com medidas de adequação de cobertura de código. Outra recomendação recorrente é combinar testes operacionais com verificação específica de caso especial entradas. No entanto, tais práticas precisam ser apoiadas por um esforço sistemático para extrair e organizar provou soluções eficazes para testar problemas em um catálogo de *padrões de teste,* de forma semelhante ao que é agora um esquema bem estabelecido para abordagens de projeto.

Os padrões oferecem soluções comprovadas para problemas recorrentes, ou seja, tornam explícitos e documentam perícia em resolução de problemas. Como os testes são reconhecidos como caros e propensos a esforço, é altamente desejável tornar explícito quais são os procedimentos bem-sucedidos.

Um esforço relacionado é a caracterização de Vegas e coautores esquema de como as técnicas de teste são selecionadas [82]. Eles pesquisaram o tipo de conhecimento que os profissionais usam para escolher as técnicas de teste para um projeto de software e produziram uma lista formalizada dos parâmetros relevantes. No entanto, as organizações que usam o esquema proposto podem não descartar todas as informações necessárias, portanto, mais recentemente eles também estão investigando as fontes de informação, e

como essas fontes devem ser usadas [82]. Estudos semelhantes são necessários para formalizar e documentar práticas bem-sucedidas para qualquer outra atividade relacionada a testes.

Na verdade, este desafio abrange todas as seis questões.

Desafio: Compreender os custos dos testes

Uma vez que o teste não ocorre em abstrato, mas dentro de o mundo concreto, com seus riscos e restrições de segurança e econômicas, em última análise, queremos ser capazes de vincular o processo de teste e técnicas com seu custo.

Todo e qualquer artigo de pesquisa sobre teste de software começa de alegar que o teste é uma atividade muito cara, mas carecemos de referências atualizadas e confiáveis; é de alguma forma desanimador que ainda hoje se faça referência a quantificar a alta custo do teste citam livros didáticos que datam de mais de vinte anos atrás. Isso pode ser reconhecidamente devido à sensibilidade dos dados de falha, que são confidenciais da empresa. No entanto, para transferir de forma útil os avanços da pesquisa para a prática, precisamos ser capazes de quantificar os custos diretos e indiretos de técnicas de teste de software.

Infelizmente, a maioria das pesquisas em teste de software leva uma posição de valor neutro, como se cada falha encontrada fosse igualmente importante ou tem o mesmo custo, mas isso obviamente não é verdadeiro; precisamos de maneiras de incorporar valor econômico ao processo de teste, para ajudar os gerentes de teste a aplicar seu julgamento e selecionar as abordagens mais apropriadas. Boehm e colegas introduziram o *software baseado em valor* paradigma de *engenharia* (VBSE) [12], no qual frameworks para apoiar as decisões dos gerentes de software são buscavam aumentar o valor dos sistemas de software entregues. Em particular, vários aspectos da qualidade de software como garantia foram investigados, incluindo valores baseados e testes baseados em risco, por exemplo, [13]. O VBSE diz respeito principalmente ao gerenciamento de processos, por exemplo, teste de wrt, diferentes tipos de funções de utilidade das partes interessadas são considerados ao trade-off tempo de entrega versus valor de mercado. Nós gostaríamos também precisam ser capazes de incorporar funções de estimativa de a relação custo/eficácia das técnicas de teste disponíveis. o questão-chave é: dado um orçamento de teste fixo, como ser empregado de forma mais eficaz?

Este desafio aborda claramente principalmente o COMO e QUANTO de testes.

Desafio: Formação de testadores de software

Finalmente, para teste de software como para qualquer outro software atividade de engenharia, um recurso crucial continua sendo o humano fator. Além da disponibilidade de técnicas avançadas e ferramentas e de processos eficazes, a habilidade, o comprometimento e a motivação dos testadores podem fazer uma grande diferença entre um processo de teste bem-sucedido ou ineficaz. Pesquisa sobre seu lado deve se esforçar para produzir soluções de engenharia eficazes que sejam facilmente integradas ao desenvolvimento e não não requerem conhecimentos técnicos profundos. Mas também precisamos trabalham em paralelo para potencializar o potencial humano. este é feito tanto pela educação quanto pela motivação. Os testadores devem ser educado para entender as noções básicas de teste e as limitações e as possibilidades oferecidas pelos técnicas. Embora seja a pesquisa que pode avançar o estado de a arte, é somente pela conscientização e adoção desses resultados pela próxima geração de testadores que também podemos avançar o estado da prática. A educação deve ser contínua, para manter o ritmo com os avanços na tecnologia de teste. A educação por si só apresenta vários desafios, como discutido em [54].

É evidente que a educação deve abranger todas as características aspectos do teste.

5.5. Desafios transversais

Por desafios transversais identificamos algumas pesquisas tendências que passam por todos os quatro sonhos identificados. Em particular, discutimos aqui dois desafios transversais.

Desafio: Testar dentro do desenvolvimento emergente paradigma

A história da pesquisa em engenharia de software é faseada pelo subsequente surgimento de novos paradigmas de desenvolvimento, que prometem liberar maior qualidade e menos software caro. Hoje, a moda é Service-oriented Computação e muitos desafios interessantes surgem para o teste de aplicativos orientados a serviços.

Existem várias semelhanças com os sistemas CB, e como em CB testes, os serviços podem ser testados de diferentes perspectivas, dependendo de quem é a parte interessada envolvida [23]. o desenvolvedor de serviço, que implementa um serviço, o serviço provedor, que implanta e disponibiliza, e o integrador de serviços, que compõe serviços eventualmente disponibilizados por terceiros, acessam diversos tipos de informações e têm necessidades de teste diferentes. Exceto para o desenvolvedor de serviço, técnicas de teste de caixa preta precisam ser aplicadas, porque detalhes de projeto e implementação de serviços não estão disponíveis.

Um aspecto peculiar dos serviços é que eles são forçados disponibilizar uma descrição padrão em formato processável por computador para permitir a pesquisa e a descoberta. Então, dado que muitas vezes essa é a única informação disponível para análise, a pesquisa está investigando como explorar essa especificação para fins de teste. Atualmente, esta descrição inclui apenas a interface de serviço em termos de assinatura de métodos fornecidos (por exemplo, a definição WSDL para Serviços da Web). Claramente, as assinaturas de métodos fornecem expressividade para fins de teste e, de fato, pesquisadores visam enriquecer tal descrição para permitir testes mais significativos.

Para promover a interoperabilidade, uma primeira preocupação é para garantir que os serviços cumpram os protocolos padronizados estabelecidos para troca de mensagens. Por exemplo, diretrizes foram lançadas pela organização WS-I (Web Services Interoperability), juntamente com ferramentas de teste para

monitorar e verificar se a mensagem trocada está de acordo com
As diretrizes. Tal abordagem é certamente necessária
para assegurar a interoperabilidade, mas não suficiente, em particular
não testa o comportamento dinâmico e não se preocupa com
a verificação de propriedades extra-funcionais.

Conceitualmente, testes off-line e on-line de serviços podem
ser distinguido. No que diz respeito aos testes off-line, os dois
abordagens que surgem são baseadas em modelos e mutações. Por
testes de serviços baseados em modelo, a ideia geral é assumir que os
desenvolvedores de serviços e/ou integradores fazem
modelos disponíveis adequados para a derivação automática de
casos de teste. Precisamos, então, adaptar a riqueza das
abordagens para testes baseados em modelo para o contexto e as restrições
dos serviços, e várias propostas estão sendo feitas,
por exemplo, [55, 32, 50]. Estratégias de mutação adaptadas aos serviços
em vez disso, prever a mutação das mensagens de entrada, conforme
apresentado em [23].

O teste on-line (discutido em 5.3) assume uma importância especial para
o teste de serviços, uma vez que monitorar a execução no mundo real é a
única forma de observar o comportamento da aplicação. Uma aplicação
orientada a serviços geralmente resulta da integração de vários serviços,
controlados e
pertencentes a diferentes organizações. Como resultado, o controle de
um aplicativo é distribuído e os serviços que o compõem
descobrir uns aos outros em tempo de execução e podem mudar sem prévio
aviso prévio, para que ninguém possa prever o receptor ou o provedor de
um determinado serviço. O monitoramento de serviços introduz muitos
problemas sutis, relativos à degradação do desempenho, produção de efeitos
colaterais indesejados e custo. Precisamos entender por um lado como
podemos observar a execução em
uma rede distribuída sem (muito) afetar negativamente o
performance do sistema; por outro, precisamos de meios para raciocinar
no nível abstrato da composição do serviço e entender
o que e quando precisamos verificar.

Desafio: Testes coerentes de propriedades funcionais e extrafuncionais

De longe, a maior parte da literatura de teste de software aborda
teste de funcionalidade, ou seja, verificar se o comportamento observado está
de acordo com a lógica das especificações. Mas
isso não é suficiente para garantir a real utilidade e adequação ao propósito
do software testado: tão importante quanto,
um software de bom comportamento deve cumprir propriedades
extrafuncionais, dependendo do domínio de aplicação específico. No tably,
enquanto o teste de funcionalidade convencional não fornece nenhuma noção
de tempo, muitos recursos do exibido
comportamento de um pedaço de software pode depender de *quando* o
resultados são produzidos, ou em *quanto tempo* eles levam para serem
produzidos. Da mesma forma, enquanto o teste de funcionalidade não aborda
uso de recursos e cargas de trabalho, em domínios específicos, como
telecomunicações, a questão do desempenho representa uma grande
categoria de falha [84].

Gostaríamos de ter abordagens de teste para serem aplicadas em
tempo de desenvolvimento que poderia fornecer feedback tão cedo quanto
possível. A pesquisa em andamento que pode abrir o caminho não é
muito, e as abordagens adotadas podem ser classificadas em baseadas em
modelos e genéticas. Entre os primeiros, precisamos de
maneiras de aprimorar modelos com restrições extrafuncionais desejadas.
Nessa direção, pesquisadores da Universidade de Aalborg [51] têm investigado
há muito tempo a extensão da teoria de teste de conformidade existente para
configuração temporizada, produzindo
uma ferramenta que pode gerar casos de teste de Timed Automata e
executá-los monitorando os traços produzidos.

As abordagens baseadas em modelos são certamente um instrumento
importante também para sistemas embarcados de tempo real, mas
provavelmente tomar um longo curso antes de ser semelhante a grande escala
aplicação, também tendo em conta as muitas questões técnicas que
precisam ser modelados, como dependência do ambiente, distribuição,
restrições de recursos. Assim, é aconselhável procurar
em paralelo para abordagens inovadoras: por exemplo, Wegener
e Grochtmann [83] propuseram o uso de
algoritmos. Eles reduzem os testes em tempo real ao problema de otimização
de encontrar o melhor e o pior caso de valores de tempo de execução. Tal
ideia poderia ser estendida a outros
propriedades extra-funcionais, traduzindo apropriadamente o
restrição em um problema de otimização.

6. conclusões

Acreditamos que o teste de software é uma atividade dinâmica, difícil e
disciplina de pesquisa ricamente articulada, e espero que este artigo tenha
fornecido uma visão geral útil das atuais e futuras
desafios. Cobrir em um artigo todas as direções de pesquisa em andamento
e vistas é impossível; nós privilegiamos
amplitude contra profundidade, e a contribuição deste artigo deve ser vista
mais como uma tentativa de descrever um roteiro abrangente e extensível, no
qual qualquer
futuro desafio de pesquisa para teste de software pode encontrar seu
Lugar, colocar. A imagem que surge deve ser tomada como um tecido de
trabalho em andamento que a comunidade pode querer adaptar
e expandir.

É óbvio que os objetivos desse roteiro que
foi resolvido como os sonhos estão destinados a permanecer assim. No
entanto, em um roteiro de pesquisa, a coisa real não é o rótulo
na chegada, mas os caminhos ao longo das rotas traçadas. Então,
o que realmente é importante que os pesquisadores se concentrem para assinar
progresso são os chamados desafios, e certamente os
roadmap fornece muitos deles, alguns em um mais maduro
palco, outros apenas começando a aparecer.

O que é garantido é que os pesquisadores de teste de software
não correm o risco de ficar sem o emprego. O teste de software é e
continuará sendo uma atividade fundamental da engenharia de software: não
obstante os avanços revolucionários na
forma como é construído e empregado (ou talvez exatamente porque
de), o software sempre precisará ser eventualmente testado e
monitorou. E como amplamente discutido neste artigo, por
certeza de que precisaremos tornar o processo de teste mais ef

eficaz, previsível e sem esforço (o que coincide com o último dos quatro sonhos de teste).

Infelizmente, o progresso pode ser retardado pela fragmentação de pesquisadores de teste de software em várias comunidades desarticuladas: por exemplo, diferentes eventos foram estabelecido pelas comunidades como o loci onde se reúnem para discutir os últimos resultados, como o *ACM International Simpósio sobre Teste e Análise de Software* (ISSTA), ou os eventos da *IFIP International Conference on the Testing of Communication Systems* (TESTCOM), só para citar alguns, mostrando pouca sobreposição entre membros do PC, participação, conhecimento mútuo e citações (o que é uma pena). Dentro além dos desafios científicos enfrentados pela pesquisa de testes, que foram discutidos na Seção 5, então gostaríamos também um desafio, que é oportunista: chegou a hora vêm que as diferentes comunidades de pesquisa de teste existentes eventualmente convergirão e conciliarão as respectivas conquistas e esforços, pois isso certamente seria o maior benefício para o avanço do estado da arte1.

Uma observação conclusiva necessária diz respeito às muitas relações frutíferas entre teste de software e outras pesquisas áreas. Ao focar nos problemas específicos de software testes, de fato negligenciamos muitas oportunidades interessantes que surgem na fronteira entre o teste e outros disciplinas. Alguns foram mencionados neste artigo, por exemplo, técnicas de verificação de modelos, veja [27] (por exemplo, para conduzir testes baseados em modelo) ou o uso de abordagens, veja [42], para geração de entrada de teste, ou a aplicação de técnicas de teste para avaliar atributos de desempenho, veja [86]. Acreditamos que realmente muitas são as aberturas que pode surgir de uma abordagem mais holística para teste de software pesquisa, e em [19] os leitores podem certamente encontrar e apreciar muitas novas sinergias interessantes que abrangem todo o disciplinas de pesquisa de engenharia de software.

7. Agradecimentos

Resumindo o campo bastante amplo e ativo do software testar a pesquisa tem sido um desafio difícil. Enquanto eu permaneço único responsável por imprecisões ou omissões, existem muitas pessoas a quem estou em dívida. Primeiro, com o objetivo de sendo o mais abrangente e imparcial possível, perguntei vários colegas para me enviarem uma declaração do que eles considerado o maior desafio pendente enfrentado pela pesquisa de teste de software, e uma referência a trabalhos relevantes (diferentemente de outros autores ou de seu próprio artigo) que este artigo não poderia deixar de citar. Dos muitos que eu convidei, eu calorosamente obrigado por contribuir: Ana Cavalli, SC Cheung, Sebastian Elbaum, Mike Ernst, Mark Harman, Bruno Legeard, Alex Orso, Mauro Pezze, Jan Tretmans, Mark Utting, Margus Veanes; suas contribuições foram editadas e incorporadas ao jornal. Também gostaria de agradecer a Lars Frantzen, Eda Marchetti, Ioannis Parissis e Andrea Polini pela discussão sobre alguns dos temas apresentados. Daniela Mulas e Antonino Sabetta ajudaram na elaboração do Roteiro figura. Também gostaria de agradecer sinceramente a Lionel Briand e Alex Wolf por me convidar e por fornecer valiosos adendo.

Este trabalho foi parcialmente financiado pela Marie Curie TAROT Network (MRTN-CT-2004-505121).

1Notavelmente, entre seus objetivos a atual Rede Marie Curie TAROT http://www.int-evry.fr/tarot/ tem o objetivo de juntar pesquisadores das comunidades de teste de software e teste de protocolo.

Referências

[1] L. Baresi e M. Young. Teste oráculos. Relatório técnico, Departamento de Comp. e Ciência da Informação, Univ. do Óregon, 2001. http://www.cs.uoregon.edu/michal/pubs/oracles.html.

[2] E. Bayse, AR Cavalli, M. Nu´nez e F. Za¨ydi. Um passivo abordagem de teste baseada em invariantes: aplicação ao wap. *Redes de Computadores,* 48(2):235-245, 2005.

[3] B. Beizer. *Técnicas de Teste de Software (2ª ed.).* Van Nos trand Reinhold Co., Nova York, NY, EUA, 1990.

[4] A. Belinfante, L. Frantzen e C. Schallhart. Ferramentas para teste geração de casos. Em *[21].*

[5] S. Berner, R. Weber e R. Keller. Observações e lições aprendidas com testes automatizados. Em *Proc. 27ª Int. Conf. sobre Sw. Eng.,* páginas 571-579. ACM, 2005.

[6] G. Bernot, MC Gaudel e B. Marre. Teste de software baseado em especificações formais: uma teoria e uma ferramenta. *Softw. Eng. J.,* 6(6):387-405, 1991.

[7] A. Bertolino. Painel ISSTA 2002: a pesquisa ISSTA é relevante para usuários industriais? Em *Proc. ACM/ SIGSOFT Int. Sintoma sobre Teste e análise de software,* páginas 201–202. Imprensa ACM, 2002.

[8] A. Bertolino e E. Marchetti. Teste de software (cap.5). Em P. Bourque e R. Dupuis, editores, *Guia para o Corpo de Conhecimento de Engenharia de Software SWEBOK, 2004 Versão,* páginas 5–1–5–16. Sociedade de Computação IEEE, 2004. http://www.swebok.org.

[9] A. Bertolino, E. Marchetti e H. Muccini. Apresentando uma abordagem razoavelmente completa e coerente para testes baseados em modelos. *Eletr. Teórico das Notas. Computar. Sci.,* 116:85-97, 2005.

[10] A. Bertolino e A. Polini. A estrutura de audição para testar a interoperabilidade de serviços da Web. Em *Proc. EUROMICRO '05,* páginas 134-142. IEEE, 2005.

[11] A. Bertolino, A. Polini, P. Inverardi e H. Muccini. Para as alas de testes baseados em anti-modelo. Em *Proc. DSN 2004 (Ext. resumo),* páginas 124-125, 2004.

[12] S. Biffl, A. Aurum, B. Boehm, H. Erdogmus e P. Gru enbacher, editores. *Engenharia de Software Baseada em Valor.* Springer-Verlag, Heidelberg, Alemanha, 2006.

[13] S. Biffl, R. Ramler e P. Gruenbacher. Gerenciamento baseado em valor de teste de software. Em *[12].*

[14] Pasta RV. *Testando Modelos, Padrões e Ferramentas de Sistemas Orientados a Objetos.* Addison Wesley Longman, Inc., Leitura, MA, 2000.

[15] C. Blundell, D. Giannakopoulou e CS Pasareanu.
Assume-garantia de testes. Em *Proc. SAVCBS '05,* páginas 7–14. ACM Press, 2005.

[16] GV Bochmann e A. Petrenko. Teste de protocolo: revisão de métodos e relevância para teste de software. Em *Proc. ACM/ SIGSOFT Int. Sintoma Teste e análise de software,* páginas 109–124, 1994.

[17] M. Boshernitsan, R. Doong e A. Savoia. Do Daikon ao Agitator: lições e desafios na construção de uma ferramenta comercial para testes de desenvolvedores. Em *Proc. ACM/ SIGSOFT Int. Sintoma Teste e análise de software,* páginas 169–180. ACM Press, 2006.

[18] L. Briand, Y. Labiche e Y. Wang. Uma investigação de estratégias de ordem de teste de integração de classes baseadas em gráficos. *Trans. IEEE Softw. Eng.,* 29(7):594-607, 2003.

[19] L. Briand e A. Wolf, editores. *Futuro da Engenharia de Software 2007.* IEEE-CS Press, 2007.

[20] LC Briand, Y. Labiche e MM Sowka. Testes de usuário automatizados e baseados em contrato de componentes comerciais prontos para uso. Em *Proc. 28ª Int. Conf. na Sw. Eng.,* páginas 92–101. ACM Press, 2006.

[21] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker e A. Pretschner, editores. *Testes Baseados em Modelos de Sistemas Reativos - Palestras Avançadas, LNCS 3472.* Springer Verlag, 2005.

[22] G. Canfora e M. Di Penta. As próximas novas fronteiras de engenharia reversa. Em *[19].*

[23] G. Canfora e M. Di Penta. Serviços de teste e sistemas centrados em serviços: desafios e oportunidades. *Profissional de TI,* 8(2):10–17, março/abril de 2006.

[24] N. Delgado, AQ Gates e S. Roach. Uma taxonomia e um catálogo de ferramentas de monitoramento de falhas de software em tempo de execução. *Trans. IEEE Softw. Eng.,* 30(12):859-872, 2004.

[25] E. Dijkstra. Notas sobre programação estruturada. Relatório Técnico 70-WSK03, Univ. Tecnológica. Eindhoven, 1970. http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF.

[26] H. Do, S. Elbaum e G. Rothermel. Apoiando a experimentação controlada com técnicas de teste: Uma infraestrutura e seu impacto potencial. *Softw. Eng.,* 10(4):405-435, 2005.

[27] M. Dwyer, J. Hatcliff, C. Pasareanu, Robby e W. Visser. Análise formal de software: tendências emergentes na verificação de modelos de software. Em *[19].*

[28] S. Elbaum, HN Chin, MB Dwyer e J. Dokulil. Esculpindo casos de teste unitários diferenciais a partir de casos de teste do sistema. Em *Proc. 14º ACM/ SIGSOFT Int. Sintoma em Fundações de Sw Eng.,* páginas 253-264. ACM Press, 2006.

[29] S. Elbaum e M. Diep. Profiling software implantado: Como avaliar estratégias e oportunidades de teste. *Trans. IEEE Softw. Eng.,* 31(4):312-327, 2005.

[30] MD Ernst, JH Perkins, PJ Guo, S. McCamant, C. Pacheco, MS Tschantz e C. Xiao. O sistema Daikon para detecção dinâmica de invariantes prováveis. *Ciência da Programação de Computadores,* a aparecer.

[31] P. Frankl e E. Weyuker. Melhorias comprovadas em testes de ramificações. *Trans. IEEE Softw. Eng.,* 19(10):962-975, 1993.

[32] L. Frantzen, J. Tretmans e R. d. Vries. Rumo ao teste baseado em modelo de serviços da web. Em *Proc. Int. Workshop sobre Web Services - Modelagem e Teste (WS-MaTe2006),* páginas 67–82, 2006.

[33] L. Frantzen, J. Tretmans e T. Willemse. Uma estrutura simbólica para testes baseados em modelo. Em *Proc. FATES/RV,* LNCS 4262, páginas 40–54. Springer-Verlag, 2006.

[34] M.-C. Gaudel. Métodos e testes formais: Hipóteses e aproximações de exatidão. Em *Proc. FM 2005, LNCS 3582,* páginas 2–8. Springer-Verlag, 2005.

[35] P. Godefroid, N. Klarlund e K. Sen. DART: Teste aleatório automatizado dirigido. Em *Proc. ACM SIGPLAN PLDI'05,* páginas 213–223, 2005.

[36] JB Goodenough e SL Gerhart. Em direção a uma teoria de seleção de dados de teste. *Trans. IEEE Softw. Eng.,* 1(2):156-173, junho de 1975.

[37] R. Gotzhein e F. Khendek. Teste de composição de sistemas de comunicação. Em *Proc. IFIP TestCom 2006, LNCS 3964,* páginas 227–244. Springer Verlag, maio de 2006.

[38] W. Grieskamp. Testes baseados em modelos multiparadigmáticos. Em *Proc. FATES/RV,* páginas 1–19. LNCS 4262, 15 a 16 de agosto de 2006.

[39] D. Hamlet. Teste de subdomínio de unidades e sistemas com estado. Em *Proc. ACM/ SIGSOFT Int. Sintoma sobre Teste e Análise de Software,* páginas 85–96. ACM Press, 2006.

[40] D. Hamlet, D. Mason e D. Woit. Teoria da confiabilidade de software baseada em componentes. Em *Proc. 23ª Int. Conf. na Sw. Eng.,* páginas 361–370, Washington, DC, EUA, 2001.
Sociedade de Computação IEEE.

[41] D. Hamlet e R. Taylor. O teste de partição não inspira confiança. *Trans. IEEE Softw. Eng.,* 16(12):1402-1411, 1990.

[42] M. Harman. O estado atual e o futuro da tecnologia baseada em pesquisa Engenharia de software. Em *[19].*

[43] MJ Harrold. Teste: um roteiro. Em A. Finkelstein, editor, *The Future of Software Engineering,* páginas 61–72. IEEE Computer Society, 2000. Em conjunto com ICSE2000.

[44] W. Hetzel. *O Guia Completo para Teste de Software, 2º Edição.* QED Inf. Sc., Inc., 1988.

[45] W. Howden. Confiabilidade da estratégia de teste de análise de caminho. *Trans. IEEE Softw. Eng.,* SE-2(3):208-215, 1976.

[46] D. Janzen, H. Saiedian e L. Simex. Conceitos de desenvolvimento orientados a testes, taxonomia e direção futura. *Computador,* 38(9):43–50, setembro de 2005.

[47] JUnit.org. http://www.junit.org/index.htm.

[48] N. Juristo, AM Moreno e S. Vegas. Revendo 25 anos de experimentos de técnicas de teste. *Softw. Eng.,* 9(1-2):7-44, 2004.

[49] GM Kapfhammer, ML Soffa e D. Mosse. Testes em ambientes de execução com restrição de recursos. Em *Proc. dia 20 IEEE/ ACM Int. Conf. em Engenharia de Software Automatizada,* Long Beach, Califórnia, EUA, novembro de 2005. ACM Press.

[50] C. Keum, S. Kang, I.-Y. Ko, J. Baik e Y.-I. Choi. Gerando casos de teste para web services usando máquina de estado finito estendida. Em *Proc. IFIP TestCom 2006, LNCS 3964,* páginas 103–117. Springer Verlag, 2006.

[51] KG Larsen, M. Mikucionis, B. Nielsen e A. Skou. Testando software embarcado em tempo real usando UPPAAL-TRON: um estudo de caso industrial. Em *Proc. 5º ACM Int. Conf. em Embed ded Softw.,* páginas 299–306. ACM Press, 2005.

[52] Y. Le Traon, B. Baudry e J.-M. Jez´equel. Projeto por contrato para melhorar a vigilância do software. *Trans. IEEE Softw. Eng.,* 32(8):571-586, 2006.

[53] G. Lee, J. Morris, K. Parker, GA Bundell e P. Lam.
Usando a execução simbólica para guiar a geração de testes: Pesquisa artigos. *Softw. Teste. Verif. Reliab.,* 15(1):41-61, 2005.

[54] TC Lethbridge, J. Daz-Herrera, RJ LeBlanc., e J. Thompson. Melhorando a prática de software através da educação: Desafios e tendências futuras. Em *[19].*

[55] Z. Li, W. Sun, ZB Jiang e X. Zhang. BPEL4WS teste unitário: Framework e implementação. Em *Proc. do ICWS'05,* páginas 103–110, 2005.

[56] B. Liblit, A. Aiken, AX Zheng e MI Jordan. Incomodar isolamento via amostragem de programa remoto. Em *Proc. ACM SIG PLAN Conf. Projeto e Implementação de Linguagem de Programação,* páginas 141–154. ACM Press, 2003.

[57] M. Lyu. Engenharia de confiabilidade de software: um roteiro. Dentro *[19].*

[58] M. Lyu (ed.). *Manual de Engenharia de Confiabilidade de Software.* McGraw-Hill, Nova York, e IEEE CS Press, Los Alamitos, 1996.

[59] L. Mariani e M. Pezze. Detecção dinâmica de componentes COTS incompatibilidade de componentes. *Software IEEE,* para aparecer.

[60] P. McMinn. Geração de dados de teste de software baseada em pesquisa: um questionário. *Teste, Verificação e Confiabilidade de Software,* 14(2):105–156, setembro de 2004.

[61] Pesquisa da Microsoft. Melhoria da experiência do cliente programa, 2006. http://www.microsoft.com/products/ceip/.

[62] EF Moore. Gedanken-experimentos em máquinas sequenciais. *Estudos de autômatos,* páginas 129-153, 1956.

[63] NIST. Os impactos econômicos da falta de infraestrutura para teste de software, maio de 2002. http://www.nist.gov/director/prog-ofc/report02-3.pdf.

[64] A. Orso, T. Apiwattanapong e MJ Harrold. Aproveite os dados de campo antigos para análise de impacto e teste de regressão. Em *Proc. Reunião conjunta do Soft Europeu. Eng. Conf. e ACM/ SIGSOFT Symp. em Fundamentos de Soft. Eng. (ESEC/ FSE'03),* páginas 128-137, 2003.

[65] A. Orso e B. Kennedy. Captura seletiva e reprodução de execuções do programa. Em *Proc. 3ª Int. Workshop do ICSE sobre Análise Dinâmica (WODA 2005),* páginas 29–35, St. Louis, MO, EUA, maio de 2005.

[66] TJ Ostrand, EJ Weyuker e RM Bell. Previsão a localização e o número de falhas em grandes sistemas de software. *Trans. IEEE Softw. Eng.,* 31(4):340-355, 2005.

[67] C. Pacheco, SK Lahiri, MD Ernst e T. Ball. Geração de teste aleatório direcionado por feedback. Relatório Técnico MSR TR-2006-125, Microsoft Research, Redmond, WA.

[68] L. Peterson, T. Anderson, D. Culler e T. Roscoe. UMA plano para introduzir tecnologia disruptiva na internet. *SIGCOMM Comput. Comum. Rev.,* 33(1):59-64, 2003.

[69] J. Poore, H. Mills e D. Mutchler. Planejamento e certificação confiabilidade do sistema de software. *Software IEEE,* páginas 87–99, Janeiro de 1993.

[70] MJ Rehman, F. Jabeen, A. Bertolino e A. Polini. Testando componentes de software para integração: um levantamento de problemas e técnicas. *Teste de Software, Verificação e Confiabilidade,* para aparecer.

[71] A. Reyes e D. Richardson. Siddhartha: um método para desenvolver geradores de drivers de teste específicos de domínio. Em *Proc. 14ª Int. Conf. em Engenharia de Software Automatizada,* páginas 81 – 90. IEEE, 12-15 de outubro de 1999.

[72] MJ Rutherford, A. Carzaniga e AL Wolf. Critérios de adequação de testes baseados em simulação para sistemas distribuídos. Em *Proc. 14º ACM/ SIGSOFT Int. Sintoma sobre Fundações da Sw Eng.,* páginas 231-241. ACM Press, 2006.

[73] D. Saff, S. Artzi, JH Perkins e MD Ernst. Automático fatoração de teste para Java. Em *Proc. 20º Int. Conf. em Engenharia de Software Automatizada,* páginas 114–123, Long Beach, CA, EUA, 9 a 11 de novembro de 2005.

[74] D. Saff e M. Ernst. Uma avaliação experimental de testes contínuos durante o desenvolvimento. Em *Proc. ACM/ SIGSOFT Int. Sintoma em Teste e Análise de Software,* páginas 76–85. ACM, 12-14 de julho de 2004.

[75] K. Sen, D. Marinov e G. Agha. CUTE: Uma unidade concólica motor de teste para C. Em *reunião conjunta da União Europeia Suave. Eng. Conf. e ACM/ SIGSOFT Symp.on Foundations de Suave. Eng. (ESEC/ FSE'05),* páginas 263–272. ACM, 2005.

[76] A. Sinha e C. Smidts. HOTTest: Uma técnica de design de teste baseada em modelo para testes aprimorados de aplicativos específicos de domínio. *ACM Trans. Softw. Eng. Methodol.,* 15(3):242–278, 2006.

[77] D. Sjøberg, T. Dyba e M. Jørgensen. O futuro dos métodos empíricos na pesquisa em engenharia de software. Em *[19].*

[78] N. Tillmann e W. Schulte. Testes unitários recarregados: Testes unitários parametrizados com execução simbólica. *IEEE Softw.,* 23(4):38-47, 2006.

[79] J. Tretmans. Geração de testes com entradas, saídas e quiescência repetitiva. *Software – Conceitos e Ferramentas,* 17:103–120, 1996.

[80] M. Utting e B. Legeard. *Testes Práticos Baseados em Modelos - Uma Abordagem de Ferramentas.* Morgan e Kaufmann, 2006.

[81] M. van der Bijl, A. Rensink e J. Tretmans. Composicional testando com ioco. Em *Proc. FATES 2003, LNCS 2931,* 2003.

[82] S. Vegas, N. Juristo e V. Basili. Experiências de embalagem para melhorar a seleção de técnicas de teste. *O Jornal de Systems and Software,* 79(11):1606–1618, novembro de 2006.

[83] J. Wegener e M. Grochtmann. Verificação de restrições temporais de sistemas de tempo real por meio de testes evolutivos. *Real-Time Syst.,* 15(3):275-298, 1998.

[84] E. Weyuker e F. Vokolos. Experiência com desempenho teste de sistemas de software: questões, uma abordagem e caso estudar. *Trans. IEEE Suave. Eng.,* 26(12):1147-1156, 2000.

[85] EJ Weyuker. Sobre testar programas não testáveis. *The Computer Journal,* 25(4):465-470, 1982.

[86] M. Woodside, G. Franks e D. Petriu. O futuro da engenharia de desempenho de software. Em *[19].*

[87] C. Yilmaz, AMA Porter, A. Krishna, D. Schmidt, A. Gokhale e B. Natarajan. Preservando as propriedades críticas de sistemas distribuídos: uma abordagem orientada por modelo. *IEEE Software,* 21(6):32–40, 2004.

[88] H. Zhu. Uma análise formal da relação de subsoma entre critérios de adequação de teste de software. *Trans. IEEE Softw. Eng.,* 22(4):248-255, 1996.

[89] H. Zhu, PAV Hall e JHR May. Teste de unidade de software cobertura e adequação. *Computação ACM. Surv.,* 29(4):366–427, 1997.

[90] H. Zhu e X. He. Uma teoria da observação do comportamento em testes de software. Relatório Técnico CMS-TR-99-05, 24, 1999.