PROCEDIMENTO PARA APOIO À SELEÇÃO DE CRITÉRIOS DE PARADA PARA TESTES DE SOFTWARE

Victor Vidigal Ribeiro

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia de Sistemas e Computação.

Orientador: Guilherme Horta Travassos

Rio de Janeiro
Agosto de 2013

PROCEDIMENTO PARA APOIO À SELEÇAO DE CRITÉRIOS DE PARADA PARA TESTES DE SOFTWARE

Victor Vidigal Ribeiro

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Examinada por:

_____

Prof. Guilherme Horta Travassos, D.Sc.

_____

Prof. Toacy Cavalcante de Oliveira, D.Sc.

_____

Prof. Arndt von Staa, Ph.D.

RIO DE JANEIRO, RJ – BRASIL

AGOSTO DE 2013

Ribeiro, Victor Vidigal

Procedimento para Apoio à Seleção de Critérios de Parada para Testes de Software / Victor Vidigal Ribeiro – Rio de Janeiro: UFRJ/COPPE, 2013.

XIII, 200 p.: il.; 29,7 cm.

Orientador: Guilherme Horta Travassos.

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia de Sistemas e Computação, 2013.

Referências Bibliográficas: p. 112-117.

1. Engenharia de Software. 2. Verificação, Validação & Testes. 3. Critérios de Parada. 4. Engenharia de Software Experimental. I. Travassos, Guilherme Horta II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III. Título.

Aos meus pais, pelos exemplos de vida e apoio.
Ao meu irmão, pela amizade. À Thalita, pela compreensão e paciência.

Agradecimentos

Primeiro a Deus, pois sem ele nada seria possível.

Aos meus pais, Wilson e Maria Elisia, pelo amor incondicional e por estarem sempre ao meu lado, mesmo que não fisicamente, nos momentos mais difíceis.

Ao meu irmão, Heitor, pelo otimismo e por me fazer enxergar sempre um ponto de vista positivo nas diversas situações da vida.

À minha noiva, Thalita, pela paciência, compreensão, conselhos e sempre me apoiar em minhas decisões.

A todos meus familiares que sempre estavam disponíveis para qualquer tipo de ajuda.

Ao meu orientador, prof. Guilherme Horta Travassos, que sempre esteve presente, disponível e comprometido com o processo de aprendizagem e com a qualidade desta dissertação. Espero que possamos realizar muitos trabalhos juntos.

A todos os professores que tive a oportunidade de conhecer através das disciplinas. Jayme Szwarcfiter, Claudia Werner, Jano Souza, Geraldo Xexeo, Toacy Oliveira. Esses foram grandes colaboradores para meu processo de formação.

Aos meus amigos e companheiros da COPPE, alguns já egressos, Karen Nakazato, Leonardo Mota, Breno França, Ciro Grippe, Paulo Sérgio, Jobson Massolar, Vitor Lopes, Rafael Maiani, Marco Antônio Araújo, Talita Ribeiro, Gleisson Freitas pelo ótimo ambiente de trabalho e por todos os excelentes momentos vividos durantes esses anos.

Ao pessoal administrativo do PESC, Claudia Prata, Maria Mercedes, Solange Santos, Sônia Galliano e Guitierrez da Costa pelo carinho e atenção.

A CAPES e a Fundação COPPETEC, pelo apoio financeiro durante o mestrado.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

PROCEDIMENTO PARA APOIO À SELEÇÃO DE CRITÉRIOS DE PARADA PARA TESTES DE SOFTWARE

Victor Vidigal Ribeiro

Agosto/2013

Orientador: Guilherme Horta Travassos

Programa: Engenharia de Sistemas e Computação

Este trabalho apresenta um procedimento para apoio à seleção de critérios de parada para testes de software tendo em vista as características apresentadas por um projeto de software e características equivalentes apresentadas pelos critérios de parada organizados em um corpo de conhecimento. Para isso, uma base de conhecimento contendo os critérios de parada para testes de software foi organizada com informações extraídas a partir de uma *quasi*-revisão sistemática da literatura técnica de Engenharia de Software, que também serviu como fonte para a identificação dos atributos utilizados para a comparação de projetos de software com os critérios de parada. A partir deste ponto, os critérios de parada são pré-selecionados de acordo com restrições gerenciais estabelecidas pelas caracteristicas do projeto de software. Em seguida, para cada critério pré-selecionado é calculado um grau de adequação que sugere o quanto um critério de parada é aderente ao projeto de software. O grau de adequação é calculado através da comparação dos atributos comuns que caracterizam o projeto de software e os critérios de parada de teste. Visando facilitar sua utilização, o procedimento para apoio à seleção de critérios de parada para testes de software foi implementado como uma funcionalidade para a ferramenta de gerenciamento e acompanhamento de testes Maraká. Para demonstrar seu funcionamento, uma prova de conceito envolvendo diferentes projetos é apresentada, indicando a possibilidade de uso do procedimento em diferentes contextos e perfis de projeto de software.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

PROCEDURE TO SUPPORT THE SELECTION OF TESTING STOP CRITERIA

Victor Vidigal Ribeiro

August/2013

Advisor: Guilherme Horta Travassos

Department: Computer Science and Systems Engineering

This work proposes a procedure to support the stopping criterion selection for software testing regarding the matching of characteristcs presented by a software project and the stopping criteria organized into a body of knowledge. Such body of knowledge has been organized to contain relevant information acquired through a *quasi*-systematic literature review concerned with 74 stopping criteria for software testing. The stored information includes the attributes used for comparison between software projects and stopping criteria. By considering such attributes, the stopping criteria can be filtered according to the defined characteristics of the software project. Next, for each filtered criterion, an adequacy degree suggesting the level of adequacy (conceptual distance) between a stopping criterion and the software project is calculated by comparing the equivalente attributes characterizing both the software project and stopping criterion. Aiming to make easier its use, the proposed procedure has been implemented and integrated into a test management and monitoring CASE tool called Maraka. Its use has been exemplified by a proof of concept using different projects contexts including one real application.

ÍNDICE

ÍNDICE DE FIGURAS

ÍNDICE DE TABELAS

1 Introdução

Neste capítulo é apresentado o contexto do trabalho, a motivação desta pesquisa e a questão de investigação. São também apresentados os objetivos, a metodologia de pesquisa adotada e a organização desse texto.

1.1 Motivação e Contexto

Diversas tecnologias podem ser utilizadas para apoiar projetos de software, por exemplo, ferramentas, métodos, produtos, técnicas. O objetivo dessas tecnologias é apoiar o desenvolvimento de forma a tornar os projetos mais eficazes e eficientes e levar à construção de software com mais qualidade e que respeitem as restrições de prazo e custo para seu desenvolvimento. Entretanto, a existência de diferentes tecnologias faz com que, ao se construir um software, o engenheiro de software precise decidir sobre quais tecnologias devem ser utilizadas em um projeto particular.

As diferentes combinações de tecnologias que podem ser utilizadas em um projeto de software têm sido objeto de estudos desde o início da computação. A investigação sobre essas questões se intensificou quando, em 1991, BASILI e ROMBACH (1991) propuseram uma abordagem de apoio à seleção e reuso de diferentes tipos de tecnologias de software. Nesta linha, BERTOLINO (2004) afirma que a definição das técnicas mais adequadas para apoiar as tarefas de desenvolvimento é uma questão que ainda carece de estudos. Além disso, afirma também que a riqueza de informações no momento da decisão influencia positivamente no processo de seleção e que a falta dessas informações pode resultar em escolhas inadequadas que levam a um efeito negativo na qualidade do software. Entretanto, observa-se que a atividade de seleção de tecnologias tem influência direta na efetividade do processo e qualidade do produto final (VEGAS e BASILI, 2005).

Nesse contexto, o principal desafio é entender os fatores que fazem com que uma tecnologia seja mais adequada a um determinado projeto de software, conseguir informações para determinar esses fatores, e assim utilizar esses fatores para a seleção dessa tecnologia. Os fatores que podem influenciar nessa escolha podem variar, por exemplo, entre o conhecimento técnico e a habilidade da equipe do projeto sobre determinadas tecnologias e sobre o domínio do problema, cronograma do

projeto, esforço e custo determinados para o projeto, aspectos políticos e objetivos comerciais.

VEGAS e BASILI (2005) consideram que a decisão a respeito de qual tecnologia adotar é um risco a ser gerenciado em um projeto de software e que as principais razões pelas quais engenheiros de software não estão aptos a realizarem boas escolhas durante a seleção de tecnologias são:

- Informações sobre as diferentes tecnologias estão distribuídas em diferentes fontes: livros, artigos científicos, repositórios;
- Carência de diretrizes para apoiar a seleção de tais tecnologias para um projeto de software, e;
- Pouco conhecimento científico sobre tais tecnologias e sobre seu uso em projetos de software.

Existem alguns estudos relacionados à seleção de tecnologias, como por exemplo, VEGAS e BASILI (2005) que propuseram uma abordagem de apoio à seleção de técnicas de teste, denominada Esquema de Caracterização. Nessa abordagem as técnicas de teste, bem como atributos que caracterizam essas técnicas, são armazenadas em um repositório e, então, é possível gerar um catálogo de técnicas de teste sugeridas para serem utilizadas um determinado projeto.

Já WOJCICKI e STROOPER (2007) propuseram uma abordagem de apoio à seleção de técnicas de Verificação e Validação (V&V) que, além de sugerir técnicas de V&V também avaliam a combinação das técnicas escolhidas em relação à compreteza (tipo de falhas reveladas pelas técnicas) das técnicas.

Também relacionados à escolha de tecnologias DIAS NETO (2009) se fundamenta na abordagem proposta por VEGAS e BASILI (2005) e propõe um procedimento para seleção de técnicas de teste baseado em modelos na qual as técnicas são caracterizadas através de atributos e compõe um corpo de conhecimento. Assim, dado um projeto de software o procedimento é capaz de sugerir as técnicas mais adequadas.

SANTA ISABEL (2010) também utilizou o trabalho de VEGAS e BASILI (2005) e DIAS NETO (2009) para criar um procedimento de apoio à seleção de técnicas de teste para aplicações web.

No caso deste trabalho, um problema relacionado à seleção de tecnologias de software foi vivenciado no projeto de um sistema de informação financeira, web, larga escala, com equipes geograficamente distribuídas. Nesse projeto, o problema estava especificamente ligado às atividades de teste de software. Devido ao tamanho, complexidade e necessidade de qualidade do software as atividades de teste estavam consumindo muito esforço e não se tinha ideia da quantidade de testes necessária

para atingir a qualidade esperada. Ou seja, não havia diretrizes, parâmetros ou critérios concretos para determinar o momento mais adequado para parar os testes de software. Dessa forma, os testes poderiam estar sendo conduzidos de forma insuficiente, sem atingir seus objetivos, ou de maneira excessiva, desperdiçando recursos.

A experiência com o planejamento e execução dos testes de software para projetos com diferentes perfis e objetivos indica que critérios de parada podem ser influenciados por diferentes variáveis de contextos, envolvendo, entre outras, a natureza do projeto, a categoria do software, sua criticalidade, os objetivos da organização e a indisponibilidade de recursos financeiros no projeto ou prazo para as atividades de testes. Entretanto, a compreensão mais ampla de critérios que podem ser utilizados para determinar o momento mais adequado para parar os testes motivou o início dessa pesquisa. Por outro lado, quais seriam os critérios de teste disponíveis na literatura técnica de engenharia de software que poderiam ser considerados para este projeto?

Dessa forma, a pesquisa se iniciou através de uma *quasi*-revisão sistemática da literatura onde se procurou identificar se existiam critérios de parada para testes de software, quais eram esses critérios e em que situações poderiam ser aplicados. O resultado dessa revisão mostrou a existência de uma grande quantidade de critérios de parada que podem ser aplicados em diferentes tipos de projetos de software.

Se por um lado, a disponibilidade de muitos critérios de parada indica que pode ser possível encontrar uma solução para o problema, por outro, pode trazer problemas se estes critérios não puderem ser organizados de forma consistente e coerente visando sua rápida identificação. Assim, foi observado que seria útil desenvolver um procedimento para auxiliar na seleção de critérios de parada para testes de software.

Portanto, esse trabalho propõe um procedimento para auxílio à seleção de critérios de parada para testes em projetos de software. Esse procedimento é baseado no mecanismo de seleção de técnicas de teste proposto por VEGAS e BASILI (2005) que posteriormente foi adaptado por DIAS NETO (2009) para auxiliar na seleção de técnicas de testes baseado em defeitos e utilizado por SANTA ISABEL (2011) em um contexto no qual era desejado selecionar técnicas de testes para sistemas web.

Assim, o procedimento proposto nesse trabalho consiste em oganizar um corpo de conhecimento sobre critérios de parada de testes de software, evoluir o procedimento de comparação entre as características relacionadas a um projeto de software e as características de critérios de parada contidos no corpo de conhecimento organizado através de atributos que caracterizam o projeto de software e os critérios de parada e, por fim, disponibilizar o procedimento desenvolvido como

uma funcionalidade integrada a ferramenta Maraká, de apoio ao gerenciamento e acompanhamento dos testes de software.

1.2 Problema

O problema tratado nesta dissertação está relacionado à identificação do critério (tecnologia) a ser utilizado para apoiar a decisão sobre o momento mais adequado de parada dos testes de software.

Apesar dos benefícios auferidos pelos testes de software e das expectativas relacionadas à sua aplicação, os testes apenas conseguem mostrar que existem defeitos no software, mas não demonstram que o software está livre deles (DIJKSTRA, 1972). Essa característica é evidenciada ao se observar que, por questões de prazo e custo, torna-se usualmente inviável exercitar o software utilizando todas as possíveis combinações de entradas e avaliar todas as saídas geradas, ou seja, é inviável testar um software por completo devido a explosão combinatória. BINDER (2006) exemplifica esse cenário estimando que demoraria 317 anos para testar exaustivamente um software que verifica se as coordenadas para formação de três linhas, passadas como parâmetro de entrada para um programa, formam um triângulo. MALDONADO *et al.* (2007) também realizam cálculos parecidos e estimam que demoraria cerca de 5.849.424 séculos para testar exaustivamente (utilizando todas as possíveis combinações de entrada e avaliando todas as saídas) um software que realiza a soma de dois números inteiros. Além disso, teste de software é uma atividade custosa. Alguns autores afirmam que cerca de 50% do custo total de desenvolvimento do software é gasto com atividades de testes (YAMADA e OSAKI, 1985) (CAMUFFO e MORSELLI, 1990) (MATHUR, 2008).

Portanto, não é possível afirmar que um software está totalmente livre de defeitos devido a inviabilidades técnicas e restrições de prazo e custo. Assim, surge um impasse, pois se não é viável testar um software por completo sempre pode haver um caso de teste que revelaria defeitos caso fosse executado. Dessa forma, podemos depreender que uma questão a ser estudada são os critérios de parada que podem ser utilizados para apoiar a decisão de quando parar os testes de software e uma maneira de apoiar sua escolha para um determinado projeto de software.

1.3 Questão de Pesquisa

A partir do problema exposto na seção anterior a questão de pesquisa para esta dissertação pode ser formulada como:

Quais são os critérios de parada disponíveis na literatura técnica e como identificar qual o critério de parada mais adequado para ser utilizado em um determinado projeto de software?

1.4 Objetivos

O objetivo principal deste trabalho consiste em propor um procedimento para a seleção de critérios de parada para testes de software, conforme definido na questão de pesquisa formulada na seção 1.3. Esse objetivo pode ser mais bem detalhado nos seguintes sub-objetivos:

- Definir atributos de caracterização para projetos de software que estejam relacionados às atividades de testes e auxiliem na escolha de um critério de parada;
- Definir atributos de caracterização para critérios de parada para testes de software que possam ser relacionados aos atributos de caracterização do projeto;
- Definir um corpo de conhecimento que funcione como um repositório de critérios de parada, caracterizados através dos atributos definidos anteriormente;
- Definir um algoritmo para apoiar a avaliação de adequação entre um projeto de software e os critérios de parada contidos no corpo de conhecimento, e;
- Disponibilizar um corpo de conhecimento e o algoritmo mencionados anteriormente integrados em uma ferramenta de apoio ao processo de teste de software.

Além disso, um dos objetivos da dissertação é realizar uma avaliação preliminar (prova de conceito) de viabilidade da aplicação do procedimento em casos representando projetos específicos.

1.5 Metodologia de Trabalho

Esse trabalho seguiu uma metodologia de pesquisa fundamentada na abordagem baseada em evidência proposta por SPÍNOLA *et al.* (2008) que por sua vez surgiu de uma evolução da metodologia originalmente proposta por *SHULL et al.* (2001) e estendida por MAFRA *et al.* (2006).

Essa metodologia é composta pela etapa de concepção, na qual acontece a execução de estudo secundário para verificar a existência de evidência na literatura técnica e a proposta inicial em caso de não existência de tecnologia que possa ser evoluída. Na segunda fase (de avaliação) é avaliado se a tecnologia é viável, se sua

aplicação prática faz sentido e se a tecnologia é adequada em um contexto industrial. A Figura 1-1 demonstra a estrutura da metodologia de pesquisa em que essa dissertação foi baseada.

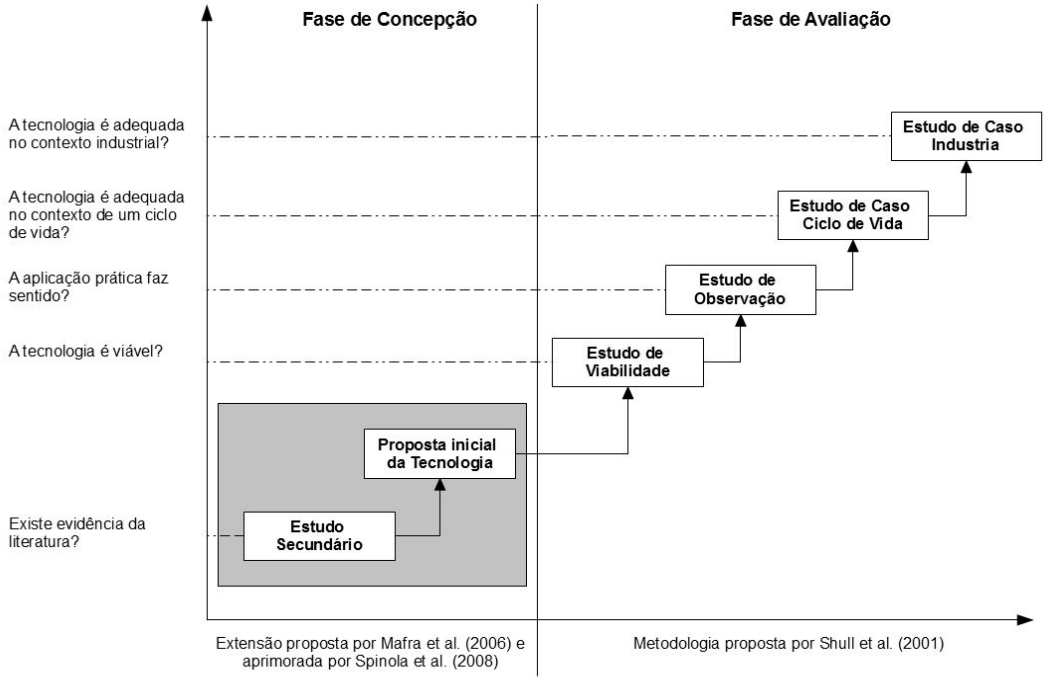


Figura 1-1 – Metodologia de pesquisa (SPÍNOLA *et al.* 2008)

Neste trabalho as atividades da metodologia que foram executadas foram: estudo secundário, proposta inicial da tecnologia e estudo de viabilidade. A Figura 1-2 demonstra as atividades executadas e o texto a seguir descreve cada uma dessas atividades.

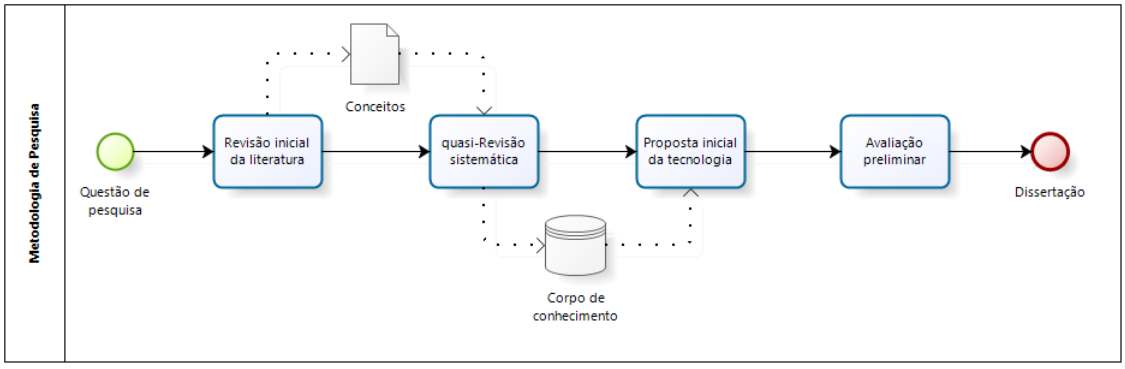


Figura 1-2 – Metodologia de pesquisa instanciada

- **Revisão inicial da literatura técnica:** o objetivo da execução dessa atividade foi identificar os conceitos básicos relacionados a critérios de

paradas para testes de software de forma que seu resultado apoiasse a definição de um protocolo de *quasi*-revisão sistemática. Essa revisão inicial foi realizada em uma única máquina de busca de artigos científicos: IeeeXplore

- ***quasi*-Revisão sistemática:** nessa atividade o protocolo da *quasi*-revisão sistemática da literatura foi elaborado e executado. O conhecimento adquirido na revisão inicial da literatura técnica bem como a consulta a um especialista da área de testes de software foram utilizados na elaboração do protocolo. Nessa revisão quatro máquinas de busca de artigos científicos foram utilizadas: Scopus, IeeeXplore, EI Compendex e Web of Science
- **Corpo de conhecimento:** o corpo de conhecimento consiste em um repositório de critérios de parada caracterizados e gerado a partir dos formulários de extração de dados da *quasi*-revisão sistemática da literatura. Esse corpo de conhecimento é utilizado como fonte de dados para o procedimento desenvolvido
- **Proposta inicial de tecnologia:** o objetivo dessa atividade foi propor um procedimento para auxiliar a seleção de critérios de parada para testes de software. O procedimento proposto foi implementado na infraestrutura de gerenciamento de testes denominada Maraká (DIAS NETO, 2009).
- **Avaliação preliminar da tecnologia:** nessa atividade o procedimento proposto passa por uma avaliação preliminar na qual uma prova de conceito é executada com a finalidade de observar a viabilidade de utilização do procedimento em projetos de software. Essa prova de conceito consistiu na execução do procedimento em cenários representados por tipos variados de projetos de software.

1.6 Organização

Esse capítulo apresentou o contexto e a motivação para realização dessa pesquisa, bem como as questões de pesquisa, os objetivos planejados e a metodologia utilizada na definição e avaliação da abordagem proposta. Esses tópicos serão refinados ao longo dos seguintes capítulos:

- **Capítulo 2 – Contextualização sobre Critérios de Parada para Testes de Software:** este capítulo provê a contextualização sobre o tópico teste de software descrevendo os principais conceitos relacionados ao tema além de expor os níveis, etapas e técnicas que podem ser utilizadas nessa fase de construção de software. Além disso, é apresentada uma visão geral sobre

critérios de parada para testes de software, um histórico dos critérios de parada, um exemplo de aplicação bem como limitações dos critérios de parada.

- **Capítulo 3 – *quasi*-Revisão Sistemática da Literatura sobre Critérios de Parada para Testes de Software:** este capítulo descreve a elaboração do protocolo da *quasi*-revisão sistemática bem como sua execução. O capítulo descreve detalhes da revisão como definição dos artigos de controle, seleção de fontes, seleção dos estudos, resultados da extração dos dados, análise desses resultados e, por fim, expõe as conclusões inferidas dos resultados obtidos.
- **Capítulo 4 – Procedimento para Apoio à Seleção de Critérios de Parada para Testes de Software:** nesse capítulo é descrito o procedimento adaptado para auxílio à seleção de critérios de parada para testes de software. É demonstrado como os critérios de parada foram divididos em duas categorias: apriorísticos e não apriorísticos e como os projetos de software são caracterizados para que a execução do procedimento seja possível. Além disso, o capítulo descreve o cálculo realizado para definição da adequação entre o projeto de software e critérios de parada bem como um exemplo demonstrando esse cálculo.
- **Capítulo 5 – Apoio Computacional para Seleção de Critérios de Parada para Testes de Software:** o procedimento descrito no Capítulo 4 foi implementado como um componente em uma infraestrutura para teste de software denominada Maraká (DIAS NETO, 2009). Assim, esse capítulo descreve a implementação realizada demonstrando as configurações possíveis bem como a execução do procedimento.
- **Capítulo 6 – Prova de Conceito:** esse capítulo tem a finalidade de apresentar uma prova de conceito executada com a finalidade de avaliar previamente a possibilidade de aplicação do procedimento. Essa prova de conceito é composta por seis casos relacionados a projetos de software criados em contextos diferentes. O primeiro caso exemplifica a aplicação do procedimento em um projeto fictício, os quatro casos seguintes apresentam o procedimento sendo aplicado em projetos previamente utilizados em estudo experimental e, no último caso de observação, o procedimento é aplicado em um projeto real de um sistema de informação web larga escala.
- **Capítulo 7 – Conclusões:** apresenta as considerações finais deste trabalho expondo as contribuições da dissertação, suas limitações e trabalhos futuros.

Além dos capítulos que compõe essa dissertação três documentos anexos em forma de apêndices também fazem parte do trabalho:

- **Apêndice A – Protocolo da *quasi*-Revisão Sistemática:** apresenta o protocolo elaborado com o planejamento da *quasi*-revisão sistemática executada e descrita no Capítulo 3
- **Apêndice B – Modelo de Formulário de Extração de Dados:** representa o modelo do formulário de extração de dados utilizado na *quasi*-revisão sistemática. Esse modelo foi utilizado para extrair os dados de todos os artigos incluídos na revisão e serviu de base para a organização do corpo de conhecimento sobre critérios de parada para testes de software
- **Apêndice C – Extração dos Dados dos Critérios de Parada:** descreve cada um dos critérios de parada extraídos e resultantes da execução da *quasi*-revisão sistemática. O conteúdo desse apêndice representa o corpo de conhecimento que, por questões computacionais, foi traduzido para um banco de dados relacional para viabilizar a implementação do procedimento.

2 *Contextualização sobre Critérios de Parada para Testes de Software*

Neste capítulo são apresentados os conceitos para um melhor entendimento sobre testes de software e, posteriormente, sobre critérios de parada para testes de software. As definições apresentadas intencionam assegurar uma mesma perspectiva no entendimento do conteúdo expresso nesta dissertação. É apresentado também um resumo histórico sobre os critérios de parada e algumas de suas limitações.

2.1 Introdução

A importância do software nem sempre foi compreendida e, por esse motivo, inicialmente o software era desenvolvido de maneira empírica e utilizando o mínimo de técnicas. Dessa forma, a maioria do software não atendia as expectativas em termos de custo, prazo de entrega e qualidade. Além disto, por ser menos complexo, o software muitas vezes era desenvolvido por um único desenvolvedor.

Entretanto, a importância do software foi sendo reconhecida ao longo dos anos. Diversos autores expressam o importante papel que o software desempenha na sociedade atual. Por exemplo, SOMMERVILLE (2003) trata da importância do software enfatizando que o software está em todos os lugares e que a maioria dos equipamentos elétricos contém algum tipo de software. O software é utilizado na indústria manufatureira, nas escolas e universidades, nos setores de assistência à saúde, finanças e no governo.

PRESSMAN (2006) menciona que softwares de computadores podem ser considerados atualmente a tecnologia mais importante. Esta tecnologia se tornou indispensável para negócios, ciência e engenharia. Além disto, o software permitiu a criação de novas tecnologias, a extensão de tecnologias existentes e o declínio de outras tecnologias. Como exemplos dessas mudanças podemos citar a engenharia genética, telecomunicações e tipografia respectivamente. O software também é considerado o grande impulsionador da revolução do computador pessoal e guia a grande rede mundial: a internet.

PFLEEGER (2006) também trata da importância do software quando diz que estamos rodeados por ele. O software está nas torradeiras de preparo do pão programadas com o tempo, no controle da eletricidade entregue nas residências, são utilizados para pagar nossa conta de luz, para comprar produtos e também em sistemas críticos que monitoram a saúde e o bem estar.

Com o reconhecimento da importância e aumento da complexidade do software surgiu a necessidade de produção de software com mais qualidade e a criação de técnicas que tornem mais fácil, rápido e menos dispendioso construir e manter programas de computador. É nesse contexto que nasceu a engenharia de software (PRESSMAN, 2006).

Uma das primeiras definições sobre engenharia de software foi proposta por BAUER (1969). Foi definido que a engenharia de software é a criação e utilização de sólidos princípios de engenharia a fim de obter softwares econômicos que sejam confiáveis e que trabalhem de forma eficiente em máquinas reais. Outra definição mais abrangente considera a engenharia de software como sendo a aplicação de uma abordagem sistemática, disciplinada e quantificável, para o desenvolvimento, operação e manutenção de software (IEEE 610, 1990). Desde então, diversas definições de engenharia de software surgiram ao longo da história. Todas essas definições, mesmo que implicitamente, contêm o conceito de qualidade como uma propriedade necessária ao software engenheirado.

Dessa forma, uma área de pesquisa específica e identificada como Verificação, Validação & Testes (VV&T) foi criada visando a garantir a qualidade do software engenheirado (PRESSMAN, 2006). Este trabalho está diretamente relacionado às atividades de teste estudadas nesta área de pesquisa da engenharia de software.

Os testes de software serão mais bem explicados a partir da seção 2.3. Entretanto, antes de dar início ao assunto, é preciso que algumas definições sejam realizadas para permitir uma discussão mais consistente e o entendimento do contexto onde este trabalho se insere.

2.2 Definições relacionadas a testes de software e critérios de parada

Diversos conceitos estão envolvidos quando se discutem testes de software e critérios de parada. Os conceitos contemplam diferentes características e envolvem, por exemplo, a definição de confiabilidade, formas de se medir confiabilidade e a compreensão de defeito, falta e falha. Dessa forma, as definições a seguir serão utilizadas ao longo do texto:

- **Falta (*fault*)**: um passo, processo ou definição de dados incorretos em um programa de computador (IEEE 610, 1990).
- **Falha (*failure*)**: é a inabilidade do sistema ou componente de executar suas funções da maneira como foi especificado. Ou seja, é o comportamento operacional do software diferente do esperado pelo usuário. Pode se dizer também que é a manifestação de uma falta (*fault*) (IEEE 610, 1990). Uma falta pode gerar diversas falhas e diversas faltas podem gerar uma falha (HUANG e LIN, 2010)
- **Defeito (*defect*)**: o termo defeito pode ser utilizado de maneira genérica para designar faltas ou falhas (LYU, 1996)
- **Confiabilidade (*reliability*):** probabilidade de um software executar livre de falhas por um período de tempo definido em um ambiente específico (IEEE 179, 1983). Também pode ser definida como a capacidade de um sistema ou componente de executar suas funcionalidades sob condições determinadas por um período de tempo definido (IEEE 610, 1990)
- **Latência de falha:** período de tempo desde a ocorrência da falta até a manifestação da falha (DENARO *et al.*, 2004).
- **Tempo (*time*):** em relação à confiabilidade o tempo pode ser medido de três maneiras: 1) tempo de execução (*execution time*) que é o tempo de CPU gasto pelo computador para executar o software. O tempo de execução é o mais indicado para medir a confiabilidade; 2) tempo de calendário (*calendar time*) é o tempo normalmente utilizado pelas pessoas e contato em anos, meses, dias; 3) tempo de *clock* (*clock time*) é o tempo gasto do início ao fim do funcionamento do computador na execução do software (LYU, 1996)
- **Tempo médio para reparo:** é o tempo que vai da observação da falha até a reparação do sistema (LYU, 1996)
- **Depuração (*debug*):** é o processo de localização, análise e correção das faltas de um software (SHOOMAN, 1983)
- **Depuração perfeita:** um processo de testes que possui depuração perfeita considera que quando uma falha é encontrada, a falta que gerou esta falha é corrigida imediatamente e, sem a introdução de novas falhas (XIE e YANG, 2003)
- **Funções de falhas:** depois de definida a base de tempo, as falhas podem ser expressas de diversas maneiras. É importante destacar que é possível converter a ocorrência de falhas de uma unidade para outra (LYU, 1996):

- **Função cumulativa de falhas ou função de valor médio:** é a média das falhas acumuladas associadas a cada ponto no tempo (LYU, 1996)
- **Função de intensidade de falhas:** representa a taxa de mudança da função cumulativa de falhas (LYU, 1996). Também pode ser definida como o número de falhas por unidade de tempo (PHAM, 2003)
- **Função de taxa de falhas ou taxa de ocorrência de falhas:** é a probabilidade de ocorrer uma falha em um intervalo de tempo especificado (LYU, 1996)

- **Modelos de confiabilidade do software:** é definido formalmente como a especificação da forma geral da dependência do processo de falhas sobre os fatores que a afetam: 1) inserção de defeitos; 2) remoção de defeitos; 3) ambiente operacional (PHAM, 2003). Também se pode dizer, informalmente, que modelos de confiabilidade são utilizados para medir ou predizer a confiabilidade do software
- **Medição da confiabilidade de software:** dois tipos de atividades estão incluídas na medição da confiabilidade (LYU, 1996):
  - **Estimativa de confiabilidade:** esta atividade utiliza técnicas de inferência estatística para determinar a confiabilidade **atual** do software. Para realizar a estimativa são utilizados dados de falhas obtidos durante as atividades de teste ou durante a operação do sistema
  - **Predição de confiabilidade:** determina a confiabilidade **futura** do software baseado em métricas e medidas disponíveis. Dependendo do estágio de desenvolvimento, a atividade de predição envolve duas técnicas diferentes:
    - **Previsão de confiabilidade (*reliability prediction*):** se os dados de falhas estão disponíveis, o software está na fase de testes ou na fase operacional. Nesse caso, as técnicas de estimativa podem ser utilizadas para parametrizar e verificar os modelos de confiabilidade que realizam a predição de confiabilidade e posteriormente predizer a confiabilidade utilizando esses modelos
    - **Previsão antecipada de confiabilidade (*early reliability prediction*):** se dados de falhas não estão disponíveis, o software se encontra nas fases iniciais (projeto, codificação). Nesse caso, as medidas obtidas para as métricas do processo

de desenvolvimento e as características do software podem ser utilizadas para determinar a confiabilidade do software

- **Perfil operacional:** o conjunto de operações que o software pode executar juntamente com a probabilidade da operação ocorrer (MUSA, 1993). Em outras palavras, pode-se dizer que é uma representação probabilística da forma como os usuários utilizam as funcionalidades do software

2.3 Testes de Software

Existem diversas definições na literatura técnica para testes de software. Por exemplo, MYERS (1979) define teste de software como sendo a execução de um programa utilizando uma combinação de dados de entrada e posteriormente a análise da saída com a intensão de encontrar um defeito.

Testes de software também podem ser definidos como uma técnica dinâmica de verificação e validação e consistem na execução do software com o objetivo de revelar *falhas*. Com o conhecimento das *falhas* do software, é possível corrigir as *faltas* que geraram essas *falhas* através do processo de depuração. Com menos *faltas*, assume-se que o software falhe menos e, assim há um aumento da confiança do software (adaptado de ROCHA *et al.*, 2001).

Posteriormente, MYERS (2004) redefine seu próprio conceito de teste de software como sendo o processo, ou conjunto de processos, que tem por intuito certificar que um programa faz aquilo que foi desenvolvido para fazer e que não executa nada involuntariamente.

Através dos conceitos anteriores é possível perceber que os testes de software sozinhos não trazem mais qualidade ao software. Isto porque, as atividades de testes não contemplam a correção das faltas, apenas a detecção das falhas geradas por essas faltas. Portanto, pode-se afirmar apenas que os testes podem ser utilizados para avaliar a qualidade do software. A correção das faltas ocorre através do processo chamado depuração.

2.3.1 Níveis de teste

As atividades de testes são consideradas complexas e por esse motivo são divididas em níveis. Esses níveis são conhecidos também como fases de teste. Além disto, as faltas contidas no software podem estar em diferentes níveis de abstração e podem ser reveladas apenas por diferentes níveis de testes.

Os *Testes de Unidade* costumam ser o primeiro nível de teste. Nesse nível as menores unidades de um software são tratadas. Como unidades podem ser

consideradas funções, procedimentos, métodos ou classes. É esperado que nesse nível sejam identificados defeitos relacionados a algoritmos incorretos ou mal implementados, estruturas de dados incorretas, ou simples erros de programação como a utilização de variáveis incorretas. É possível testar cada unidade à medida que elas vão sendo construídas sem que se tenha ainda todo o sistema de software completo, pois cada unidade pode ser testada separadamente.

Depois que cada unidade é testada separadamente, o *Teste de Integração* pode ocorrer. Nesse nível é testado se as unidades continuam com o comportamento esperado após serem colocadas para trabalhar em conjunto. Ou seja, são testadas as interações entre as unidades que compõem o software. Esta fase de teste exige um grande conhecimento das estruturas internas do software e de como essas estruturas permitem a comunicação entre as diferentes unidades.

Com as unidades testadas separadamente (*Testes de Unidade*) e certificado que elas estão trabalhando corretamente em conjunto (*Testes de Integração*), o *Teste de Sistema* pode ser iniciado. Esse nível de teste tem o objetivo de avaliar se as funcionalidades implementadas estão de acordo com a especificação do software. São testados aspectos de correção, completude, concorrência e os requisitos não funcionais como segurança, performance e robustez podem ser verificados. Diferentemente das fases anteriores, esse nível de teste pode ser executado sem que se tenha um conhecimento aprofundado sobre as estruturas internas no software.

Após a liberação do software, cada alteração efetuada causa um risco de inserção de defeitos no software. Portanto, é preciso ter uma fase de testes que contemple a etapa de manutenção do software. Esta fase é chamada de *Teste de Regressão* e tem o objetivo de executar testes que mostrem que os novos requisitos implementados ou alterados funcionam como esperado. Além disto, também é avaliado se os requisitos que antes funcionavam continuam corretos (MALDONADO *et al.*, 2007).

2.3.2 Etapas da Atividade de Teste

Além dos níveis mencionados anteriormente, os testes também podem ser divididos em etapas: *Planejamento*, *Projeto*, *Criação de Casos de Testes*, *Execução e Análise dos Resultados* (MALDONADO *et al.*, 2007).

A etapa de *Planejamento* tem como objetivo elaborar o documento denominado plano de teste que define o escopo dos testes, ou seja, as funcionalidades a serem testadas e os critérios que devem ser utilizados. Define a abordagem geral, projeto de teste, resultado das execuções, critérios de conclusão, critérios de parada, ambiente

de teste, responsabilidades, pessoal e treinamento, cronograma, riscos, contingências e aprovações.

Na etapa de *Projeto* é criado um documento descrevendo mais detalhadamente a abordagem de teste (definida de forma geral no plano de teste), as ferramentas, descrições dos métodos ou técnicas utilizadas e também o critério de aceitação ou rejeição do teste. Nessa etapa é criado também o procedimento (roteiro) que descreve os passos necessários para a execução de um ou um grupo de casos de teste.

Na etapa de criação dos *Casos de Testes* é definido o conjunto de valores de entrada, restrições de uso e valores de saída esperados. Cada caso de teste deve conter uma caracterização, a especificação das entradas, resultados esperados para cada entrada, definição dos recursos necessários, especificação das restrições de uso e a definição das dependências existentes entre casos de testes.

A próxima etapa é a *Execução* dos testes de acordo com o que foi especificado nas etapas anteriores. Além da execução do software acontece o monitoramento das atividades relacionadas, registro dos incidentes encontrados pelos testes e apresentação dos resultados finais dos testes além da configuração do ambiente de teste. A execução de todos os procedimentos de teste para uma versão do software em um determinado ambiente de teste é chamada de rodada ou bateria de testes.

Depois de executar os testes é preciso realizar a *Análise dos Resultados* para observar se os testes obtiveram sucesso. Nesta fase, métricas podem ser coletadas para serem utilizadas posteriormente (BINDER, 2006).

2.3.3 Técnicas e Critérios de Cobertura para Teste de Software

Existem duas técnicas principais relacionadas aos testes de software: Funcional e Estrutural. A técnica de teste funcional considera o programa como uma caixa fechada e tenta exercitar as diferentes possibilidades de entrada comparando as saídas retornadas com as saídas previstas (DI LUCCA e FASOLINO, 2006). Já a técnica estrutural considera as estruturas internas do sistema e, portanto, considera o programa como uma caixa aberta. Esta técnica trabalha sobre o código fonte de forma que os caminhos lógicos e condições são avaliados (MALDONADO *et al.*, 2007).

Além das técnicas de testes mencionadas existem critérios de cobertura que podem ser aplicados para identificar partes do software que devem ser executadas para melhorar a eficiência dos testes, ou seja, ajudar na criação de testes que têm maior probabilidade de revelar falhas. Pode-se dizer que a cobertura dos testes determina o percentual de elementos testados em um programa. Além disso, alguns

desses critérios são específicos para determinadas técnicas de testes e outras podem ser utilizadas em técnicas diferentes. Alguns critérios são descritos a seguir.

2.3.3.1 Particionamento em Classes de Equivalência

Esse critério de teste propõe a divisão do domínio de entrada de um sistema em classes de dados e que os casos de testes sejam baseados nessas classes. Assim, cada domínio tende a descobrir uma classe de erro reduzindo a quantidade de casos de teste para o software. Dessa forma, não é preciso ter vários testes para uma mesma classe de erro. Esse critério também afirma que as classes inválidas devem ser testadas (MALDONADO *et al.*, 2007).

Por exemplo, caso determinado parâmetro de entrada para um sistema receba um número inteiro que possa variar de -100 a 100 podem ser criadas três classes de equivalência sendo duas inválidas e uma válida: inteiros < -100 (inválida), inteiros > 100 (inválida) e inteiros entre -100 e 100 (válida). Dessa forma, casos de testes com valores de entrada equivalentes a cada classe de equivalência devem ser criados (PRESSMAN, 2006).

O particionamento em classes de equivalência pode ser utilizado tanto para a técnica funcional quanto a estrutural.

2.3.3.2 Análise do Valor Limite

Esse critério geralmente é utilizado em conjunto com o particionamento em classes de equivalência e auxilia na escolha dos valores utilizados nos casos de testes. É proposto que os valores de cada classe de equivalência sejam escolhidos nas fronteiras dessas classes, pois é nesses pontos em que se encontram a maior quantidade de erros (MYERS, 2004).

PRESSMAN (2006) define diretrizes relacionadas à análise de valor limite:

- Caso valores A e B especifiquem os valores mínimos e máximos de entrada, casos de testes devem ser projetados com valores de entrada igual a A, igual a B, imediatamente menor que A, imediatamente maior que A, imediatamente menor que B e imediatamente maior que B.
- Quando vários valores são especificados em uma condição de entrada, casos de testes devem exercitar os valores mínimos e máximos. Valores imediatamente acima e abaixo do valor mínimo e máximo especificados também devem ser testados.
- As diretrizes anteriores devem ser aplicadas aos valores de saída.
- Casos de teste devem exercitar os limites das estruturas de dados (por exemplo, vetores) do software.

Utilizando o mesmo exemplo da seção anterior no qual um parâmetro de entrada pode variar entre -100 e 100, os valores escolhidos de cada classe de equivalência seriam: -101, -100, -99, 99, 100, 101.

Como esse critério é complementar ao particionamento em classes de equivalência, ele também pode ser utilizado com as técnicas funcional e estrutural.

2.3.3.3 Grafo Causa-Efeito

Os critérios anteriormente apresentados não exploram combinações dos dados de entrada, ou seja, fornecem apenas um método para escolha de valores de entrada para serem utilizados separadamente nos casos de testes. Neste sentido, contemplando essa deficiência, o critério Grafo Causa-Efeito ajudam na definição de um conjunto de casos de teste com a finalidade de explorar ambiguidades e incompletude nas especificações. A derivação de casos de testes através desse critério é realizada em seis passos (MYERS, 2004):

1. Dividir a especificação do software em partes para diminuir a complexidade dos grafos gerados;
2. Identificar as causas e efeitos na especificação. As causas representam as entradas, estímulos ou qualquer coisa que provoque uma resposta do sistema em teste e os efeitos correspondem às saídas, mudanças no estado do sistema ou qualquer resposta observável. Uma vez identificados, cada um deve ser atribuído um único número;
3. Analisar a semântica da especificação e transforma-la em um grafo booleano que liga as causas e efeitos (grafo causa-efeito);
4. Adicionar anotações no grafo que descrevam combinações de causas e efeitos impossíveis devido a restrições sintáticas ou do ambiente;
5. Converter o grafo em uma tabela de decisão, na qual cada coluna representa um caso de teste;
6. Converter as colunas da tabela de decisão em casos de teste;

2.3.3.4 Complexidade ciclomática

Apesar de não ser um critério de teste, a complexidade ciclomática é muito utilizada em conjunto com os critérios apresentados anteriormente. Complexidade ciclomática é uma métrica de software proposta por MCCABE (1976) e mede a quantidade de fluxos de execução independentes a partir do código fonte. Dessa forma, a métrica fornece a quantidade mínima de casos de testes necessários para exercitar cada fluxo do programa.

O valor da complexidade ciclomática de um grafo G pode ser calculada de três formas:

- **CC (G) = E – N + 2**: onde **CC (G)** é a complexidade ciclomática que está sendo calculada; **E** é igual ao número de ramos do grafo **G**; **N** o número de nós;
- **CC (G) = P + 1**: onde **CC(G)** é a complexidade ciclomática que está sendo calculada; **P** é o número de nós predicados;
- **Contagem da quantidade de regiões do grafo G**: outra forma de cálculo da complexidade ciclomática é a contagem da quantidade de regiões do grafo gerado incluindo a área externa, ou seja, não delimitada pelo grafo.

2.4 Critérios de Parada para Testes de Software

Com o estudo mais aprofundado sobre testes de software, percebeu-se que as disponibilidades orçamentárias, de cronograma e de recursos dos projetos, aliada a inviabilidade prática do teste exaustivo levam os engenheiros de software a refletir sobre que critérios podem ser usados para definir que momento as atividades de teste podem parar em um projeto de desenvolvimento de software sem prejudicar o atendimento aos requisitos de qualidade.

Algumas características dos testes que dão ênfase à importância do problema de indecisão de quando parar os testes são listadas a seguir.

- **Teste de software é uma atividade cara:** diversos autores afirmam que as atividades relacionadas ao teste de software consomem cerca de 50% do orçamento de um projeto (YAMADA e OSAKI, 1985) (CAMUFFO e MORSELLI, 1990) (MATHUR, 2008)
- **Testes não provam que um software está correto:** testes de software podem mostrar apenas a presença de defeitos, nunca sua ausência (DIJKSTRA, *et al.*, 1972).
- **Testes exaustivos são inviáveis:** não é viável executar todas as possíveis combinações de entrada e analisar as saídas geradas por essas entradas. Isto porque a aplicação de testes exaustivos demoraria um tempo demasiadamente grande.

Portanto, como é inviável testar todos os fluxos de execução do software utilizando todas as possibilidades de entrada sempre pode haver um caso de teste que revelaria uma falha. Assim, surge um problema: qual o momento ideal para parar os testes e liberar o software para utilização?

Esse problema vem sendo tratado através de estudos que têm por objetivo determinar quais são os critérios que podem ser utilizados para se chegar ao momento ótimo de parada dos testes de software, geralmente descritos como critérios de parada para testes de software.

2.4.1 Visão Geral

Desde a década de 70 (OKUMOTO e GOEL, 1979b) critérios de parada para testes de software vêm sendo propostos e, em sua maioria, são baseados em duas características principais: confiabilidade e/ou custo.

Frequentemente critérios de parada são baseados em modelos de confiabilidade nos quais se propõe prever ou determinar a confiabilidade do software. Sabida a confiabilidade do software, ou em que momento o software atinge determinada confiabilidade, é possível decidir em parar ou continuar com os testes.

Outros critérios de parada também consideram o custo como fator decisivo para escolha do momento de parada. Nesta linha, existem critérios de parada que propõem que a confiabilidade seja fixada e então se minimiza o custo, ou seja, dada uma confiabilidade desejada, qual menor custo para atingir esta confiabilidade. Por outro lado, existem critérios que fixam o custo e maximiza a confiabilidade, ou seja, com um custo pré-determinado, qual a maior confiabilidade que pode ser atingida.

De forma geral, os critérios de parada para testes de software trabalham com o conceito de que ao longo das atividades de testes os defeitos são encontrados e corrigidos posteriormente através do processo de depuração. Dessa forma, os defeitos contidos no software tendem a diminuir aumentando a confiabilidade. No início do processo, a detecção de defeitos é intensa, entretanto tende a diminuir ao longo dos testes. Assim, chega-se a um momento em que o esforço para encontrar um defeito é maior do que o benefício da correção do mesmo. Ou seja, o teste passa a ser mais custoso que a falha gerada pela falta não corrigida. É este ponto de equilíbrio entre o esforço do teste e o custo de falha que os critérios de parada propõem encontrar.

A Figura 2-1 exibe o comportamento padrão dos custos relacionados à confiabilidade do software. O *Custo de Falha* representa o custo de liberação de software não confiável, ou seja, o quanto custaria caso as falhas não removidas fossem reveladas em produção. O *Custo de Teste* apreende todo custo gasto com testes para revelar as falhas em um dado momento. Já o *Custo Global* representa a soma dos dois custos anteriores (PETERS e PEDRYCZ, 2001).

É possível perceber que a probabilidade de falha tende a diminuir levando consigo o *Custo de Falha*. O *Custo de Teste* tem o comportamento contrário ao *Custo de Falha*, no início é baixo e vai aumentando ao longo do processo de teste. Além

disso, a taxa de mudança do Custo de Teste também tende a aumentar ao longo do processo de testes. O *Custo Global* é influenciado diretamente pelo *Custo de Teste* e o *Custo de Falha*. No início, tende a cair mais rapidamente até que atinge um ponto de equilíbrio e começa a aumentar.

Esse comportamento ocorre, pois no início do processo de testes há muitos defeitos no software e, dessa forma, os testes tendem a ser bastante eficientes. Com a correção dos defeitos encontrados o custo global do software tende a cair, pois esses defeitos deixam de acontecer com o software em produção. Entretanto, no decorrer dos testes, é atingido um ponto em que o custo dos testes ultrapassa o benefício de correção dos defeitos. Isto porque, como há poucos defeitos no software, o esforço para sua detecção passa a ser muito grande. Dessa forma, caso os testes continuem, o custo global passa a subir.

Com base nisto, o ideal é encontrar o equilíbrio entre o *Custo de Teste* e o *Custo de Falha* que resulte no menor *Custo Global*. É importante ressaltar que o menor *Custo Global* não está exatamente na linha de interseção entre *Custo de Falha* e *Custo de Teste*, mas sim próximo a essa linha.

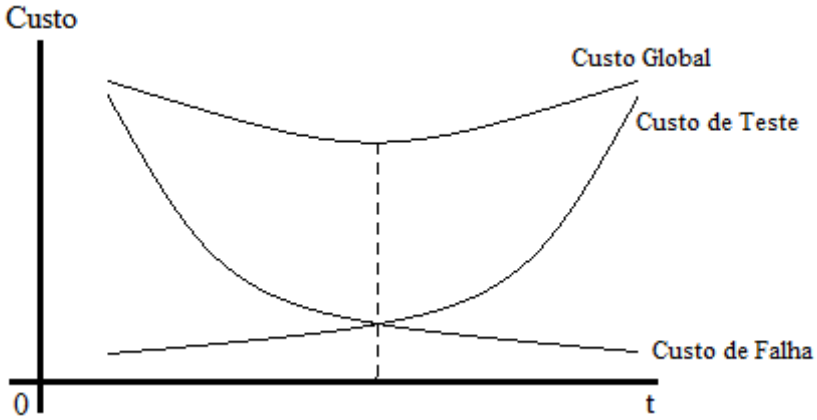


Figura 2-1 – Custos da confiabilidade de software (adaptada de MALDONADO *et al.*, 2007)

2.4.2 Histórico dos Critérios de Parada

Um dos primeiros critérios de parada publicado na literatura técnica foi proposto por OKUMOTO e GOEL (1979b). Este critério de parada é baseado em um modelo de confiabilidade e um modelo de custo proposto pelo mesmo autor. Dessa forma, o critério de parada proposto tem o objetivo de determinar qual o tempo de teste ideal para a liberação do software para produção e considera em seus cálculos a confiabilidade ou o custo. Ou seja, é possível determinar o momento de parada

minimizando o custo do software ou avaliando se o software atingiu determinada confiabilidade.

Posteriormente, o critério de parada anterior foi estendido por YAMADA e OSAKI (1985) no qual a confiabilidade e o custo são considerados simultaneamente. Nesse caso, o critério de parada tem a função de determinar qual o menor custo para se atingir uma confiabilidade pré-determinada. OHTERA e YAMADA (1990) também se basearam no critério proposto por OKUMOTO e GOEL (1979b) para propor outro critério de parada. Eles perceberam que o fenômeno de detecção de falhas era diferente na fase de teste e na fase de produção. Assim, criaram um critério de parada para contemplar o fenômeno de detecção de defeitos de acordo com a fase em que o software se encontra.

Outros autores também se basearam no trabalho de OKUMOTO e GOEL (1979b), como por exemplo, LEUNG (1992) que propõe um critério de parada no qual o orçamento para testes é pré-determinado e PHAM e ZHANG (1999) que apresentam um critério de parada que considera o tempo de garantia do software. Durante a garantia, o custo das falhas é de responsabilidade do desenvolvedor do software.

Em paralelo aos trabalhos que se basearam em OKUMOTO e GOEL (1979b) outros modelos também foram estudados. Por exemplo, KOCH e KUBAT (1983) utilizaram o modelo de detecção de falhas de JELINSKI e MORANDA (1972) para criar um critério de parada específico para empresas que utilizam o próprio software que desenvolvem.

Esse mesmo modelo também foi utilizado por CHO e PARK (1994) para criar um critério de parada que inclui o cliente/usuário no processo de decisão. Este critério de parada assume que os testes são divididos em duas fases: testes internos (executados pelo desenvolvedor) e testes de aceitação (executados com os clientes/usuários). Este critério de parada tem a função de decidir o quanto de teste interno é preciso para que, durante os testes de aceitação, menos falhas sejam encontradas.

BOLAND e CHUIV (2007) também evoluíram o modelo de JELINSKI e MORANDA (1972) para criar um critério de parada que não considera depuração perfeita. Ou seja, considera o tempo gasto na correção de defeitos e que o processo de correção de um defeito pode incluir mais defeitos no software.

Diversos outros modelos com suas próprias características e com finalidade de atender problemas específicos ou considerar novas características do processo de teste também foram propostos. DALAL e MCINTOSH (1994) sugerem um critério de parada para softwares que possuem intensa alteração do código fonte durante a fase de teste. Este critério é aplicado através de um modelo de confiabilidade que calcula o

tempo de teste necessário até a entrega do software. HOU *et al.* (1997) publicaram um critério de parada que pode ser aplicado em softwares que possuem prazo para entrega programado. Este critério é baseado em um modelo de confiabilidade que considera que as falhas do software seguem a distribuição hipergeométrica. OZEKICI *et al.* (2000) propuseram um critério de parada que considera o perfil operacional do software. Sabendo o perfil operacional, é possível concentrar os testes nas partes mais utilizadas do software diminuindo os custos de testes.

Algumas tentativas de utilização de diferentes tecnologias ou áreas de conhecimento também podem ser encontradas na literatura técnica. Por exemplo, SHINOHARA *et al.* (1998) utilizam redes neurais artificiais para determinar o momento ótimo de parada dos testes.

É possível perceber também que muitas pesquisas foram voltadas para que os critérios de parada deixassem de considerar depuração perfeita. Por exemplo, SCHNEIDEWIND (2001), GOKHALE (2003), XIE e YANG (2003), GOKHALE *et al.* (2006) e HUANG e LIN (2006) propuseram critérios de parada especificamente considerando esta característica do processo de testes.

Outros critérios que também podem ser citados são os critérios de HUANG e LIN (2010) e HSU *et al.* (2011). O primeiro critério é proposto sobre um modelo de confiabilidade que tem como características principais a consideração de um fator de compressão de testes e considera que uma falta pode gerar diversas falhas e que uma falha pode ser gerada por diversas faltas. Fator de compressão de testes significa que durante a fase de testes erros são encontrados mais rapidamente porque a variação dos dados de entrada de testes é controlada. O segundo critério é proposto sobre um modelo de confiabilidade que considera fator de redução de faltas (FRF – *Fault Reduction Factor*). Esse fator de redução de faltas é definido como o número líquido de faltas removidas por falha.

Portanto, é possível observar através da evolução histórica apresentada que os critérios de parada tentam prever de alguma forma como as falhas estão distribuídas no software e assim determinar qual o momento ótimo de parar os testes. Algumas características do processo de teste do software ou do contexto em que o critério é aplicado também influenciam no processo de tomada de decisão. Nesse sentido, diversos critérios que consideram características e contextos diferentes foram propostos.

Todos os critérios de parada encontrados durante a execução deste trabalho podem ser vistos no Apêndice C.

2.4.3 Exemplo de Critério de Parada

Um dos critérios mais reutilizados como base para a evolução de outros critérios de parada foi proposto por OKUMOTO e GOEL (1979b). Este critério de parada é baseado na modelagem do fenômeno de detecção de falhas no software como um NHPP (*Non Homogeneous Poisson Process*) proposto anteriormente por OKUMOTO e GOEL (1979a). Além disto, é proposto também um modelo de custo para o software. Assim, o critério de parada tem a finalidade de encontrar o custo mínimo ou determinar a confiabilidade do software.

O modelo de confiabilidade proposto é representado pela fórmula descrita na Figura 2-2.

$$R = \exp\left[-m(x) * e^{-bs}\right]$$

Onde,
- **R**: confiabilidade do software
- **m(x)**: é a função de valor médio que representa o número de falhas acumuladas.
- **b**: risco de uma falha ocorrer
- **s**: tempo de teste

Figura 2-2 – Modelo de confiabilidade (OKUMOTO e GOEL, 1979a)

Após propor o modelo de confiabilidade, os autores apresentam o modelo simplificado de custo utilizado no critério de parada (Figura 2-1).

$$c(t,T) = c_1 m(T) + c_2[m(t) - m(T)] + c_3 T$$

Onde,
- c_1: custo de correção de um defeito durante a fase de testes
- c_2: custo de correção de um defeito com o software em produção ($c_1 < c_2$)
- c_3: custo por unidade de tempo devido ao atraso na entrega do software
- **t**: tamanho do ciclo de vida do software
- **T**: momento de liberação do software

Figura 2-3 – Modelo simplificado de custo (OKUMOTO e GOEL, 1979b)

Depois de propor os dois modelos apresentados anteriormente, os autores OKUMOTO e GOEL realizam uma série de operações matemática sobre essas fórmulas e chegam a uma equação que é capaz de determinar o tempo ótimo para liberação do software. Assim, o critério de parada é definido através da fórmula demonstrada na Figura 2-4.

$$T_0 = \frac{1}{b} \ell n \frac{ab(c_2 - c_1)}{c_3}$$

Onde,
- **a**: número estimado total de defeitos

- **b**: risco de uma falha ocorrer
- $\mathbf{c_1}$: custo de correção de um defeito durante a fase de testes
- $\mathbf{c_2}$: custo de correção de um defeito com o software em produção ($c_1 < c_2$)
- $\mathbf{c_3}$: custo por unidade de tempo devido ao atrazo na entrega do software

Figura 2-4 – Tempo ótimo para liberar software (OKUMOTO e GOEL, 1979b)

Utilizando a fórmula anterior, é possível calcular quanto tempo de teste é o ideal para que o software seja liberado para produção. A Figura 2-5 demonstra um exemplo de cálculo do momento ótimo de parada com base em um conjunto de dados retirados de GOEL (1978).

Parâmetros utilizados para cálculo
- **a**: 1348
- **b**: 0,124
- $\mathbf{c_1}$: $1 por defeito
- $\mathbf{c_2}$: $5 por defeito
- $\mathbf{c_3}$: $100 por defeito
- **t**: 100 semanas

Aplicando fórmula demonstrada na Figura 2-4:

$$T_0 = \frac{1}{0,124} \ln \frac{1348 \cdot 0,124 (5 - 1)}{100}$$

Resolvendo a euqação, o tempo de teste ótimo para o sistema é:

$$T_0 = 15,3 \; Semanas$$

Figura 2-5 – Cálculo de critério de parada para testes de software

O exemplo anterior calcula a quantidade de semanas que seriam necessárias para atingir o ponto ótimo dos testes utilizando o modelo proposto por OKUMOTO e GOEL (1979b). É possível perceber que, desde que se tenham as fórmulas definidas e os valores dos parâmetros requeridos pelas fórmulas, o cálculo do momento de parada pode ser considerado simples. Aplicando a fórmula demonstrada na Figura 2-2 é possível saber a confiabilidade do software. Dessa forma, também é possível determinar se os testes devem parar com base na confiabilidade e não no custo.

Apesar de parecer bastante simples, o critério demonstrado anteriormente possui algumas limitações. Primeiramente, não são definidas as premissas para sua aplicação, ou seja, não define o que é necessário atender para que o critério seja aplicado com sucesso. O critério não considera nenhuma informação sobre o processo de teste. Não considera as fases de testes pelas quais o software está sendo exercitado, não considera os critérios utilizados para construção dos casos de testes. Além disto, o resultado da aplicação do critério é o número de dias que o software

precisa estar em testes. Esse resultado pode ser considerado bastante subjetivo, pois, sem a definição de critérios para a construção dos casos de testes, duas equipes de testes diferentes, no mesmo período de tempo, provavelmente exercitaria de formas diferentes o mesmo software revelando falhas diferentes. Outras limitações dos critérios de parada são demonstradas na seção seguinte.

2.4.4 Limitações dos Critérios de Parada

Os critérios de parada para testes de software apesar de auxiliar na decisão de quando parar os testes possuem algumas limitações que merecem ser consideradas.

Como é possível perceber através do exemplo demonstrado na seção 2.4.2, os critérios de parada geralmente são expressos através de expressões matemáticas. Essas expressões podem ser de difícil assimilação em um contexto empresarial. Dessa forma, apoio ferramental para realizar os cálculos dessas fórmulas pode ser considerado um fator importante para o sucesso da aplicação do critério de parada. GARG *et al.* (2010) expressa esta característica quando fala especificamente dos critérios de parada estatísticos. O autor menciona que não há uma aceitação ampla desses critérios justamente por conta de sua complexidade matemática.

Além da complexidade matemática, outro problema que envolve a aplicação dos critérios de parada é a estimativa dos parâmetros de entrada. Alguns critérios de parada exigem parâmetros de entrada extremamente difíceis de serem estimados. Por exemplo, CHAVEZ (2000) propõe um critério de parada onde o custo de descontentamento do cliente caso ocorra uma falha é utilizado para calcular o ponto de parada.

Outra limitação que pode ser exposta, é que alguns critérios de parada são criados especificamente para um determinado software. Por exemplo, LITTLEWOOD e WRIGHT (1995) propõem um critério de parada específico para um software de monitoramento de uma usina nuclear. Dessa forma, seria arriscada a utilização desses critérios de parada em um software diferente do qual ele foi proposto. Por outro lado, existem critérios de parada que não especificam ou caracterizam o software em que podem ser aplicados. Além disto, esta situação se agrava, pois a grande maioria dos critérios de parada foi proposta sem nenhum tipo de avaliação formal. Dessa forma, pode ser considerado arriscado a utilização dos critérios em outros contextos.

A desconsideração das fases de testes também pode ser considerada como uma limitação dos critérios de parada. Como visto na seção 2.3.1, os testes de software são divididos em níveis ou fases. Essas fases têm suas próprias características e possuem finalidade de encontrar diferentes tipos de falhas. Critérios de parada que consideram as falhas encontradas para calcular o ponto ótimo de

parada e não consideram em que fase de teste essas falhas foram encontradas podem estar somando informações incompatíveis ou analisando os dados sem considerar o contexto em que foram extraídos. Além disto, como não consideram as fases de testes, os critérios de parada calculam apenas quando todos os testes terminam. Entretanto, pode se ter a necessidade de saber quando parar determinada fase de teste para iniciar a próxima. Por exemplo, como determinar qual a quantidade de teste de unidade suficiente para passar para a fase de teste de integração.

Outra limitação dos critérios de parada se refere à não consideração da qualidade dos casos de testes executados. Ou seja, os critérios de parada não consideram como os casos de testes foram gerados, tão pouco consideram a qualidade desses casos de testes. Uma possível solução para esse problema seria utilizar recursos como instrumentação (STAA, 2001) para avaliar a qualidade dos casos de testes quanto à cobertura.

Considerar depuração perfeita também pode ser uma limitação dos critérios de parada. Muitos critérios de parada são propostos sobre modelos de confiabilidade que não consideram o tempo de correção de defeitos e também ignora a probabilidade de inserção de novos defeitos ao se corrigir o software, ou seja, consideram que o processo de depuração é perfeito (MASUDA *et al.*, 1989) (KAPUR, *et al.*, 1994) (TENG e PHAN, 2004). Dessa forma, é preciso ser cuidadoso com esses critérios de parada, pois podem subestimar o ponto ótimo de parada para os testes.

Apesar das limitações expressadas anteriormente, é importante ressaltar que a utilização de alguem critério de parada é preferível que a tomada de decisão ao acaso.

2.5 Conclusão

Este capítulo apresentou alguns conceitos importantes para o entendimento dos critérios de parada, como por exemplo, a definição de defeito, falta, falha, confiabilidade e perfil operacional. Além disto, foi apresentada também a importância do processo de testes nas atividades de desenvolvimento de software bem como algumas características: testes é uma atividade custosa, testes não provam que o software está livre de defeitos e testes exaustivos são inviáveis.

Também foi explicitado que os testes são divididos em níveis ou fases e que é natural a divisão dos testes nas seguintes fases: testes de unidade, testes de integração, testes de sistema e testes de regressão. Cada uma dessas fases podem passar pelas etapas de planejamento, projeto, criação dos casos de testes, execução e análise dos resultados.

Também foram apresentadas algumas técnicas de testes como o particionamento em classes de equivalência, análise do valor limite e a complexidade ciclomática. Essas técnicas auxiliam na criação de testes mais eficientes.

Parte do capítulo foi dedicada especificamente aos critérios de parada para testes de software. Nesta parte, foi demonstrado o contexto no qual os critérios de parada se encaixam bem como sua importância e um histórico dos critérios de parada ao longo dos anos. Além disto, um exemplo de um critério de parada foi demonstrado e, posteriormente, algumas limitações dos critérios foram expostas.

Além disto, algumas limitações dos critérios de parada foram apresentadas. Essas limitações, bem como a necessidade de definir um critério de parada eficiente para os testes de software, levaram à execução da *quasi*-revisão sistemática apresentada no Capítulo 3.

3 *quasi*-Revisão Sistemática sobre Critérios de Parada para Teste de Software

Neste capítulo são apresentados com detalhes informações sobre a quasi-revisão sistemática a respeito de critérios de parada para testes de software executada. A partir desse estudo foram identificados os atributos para caracterização dos projetos de software, atributos para caracterização de critérios de parada e foi montado o corpo de conhecimento sobre critérios de parada para testes de software.

3.1 Introdução

Segundo KITCHENHAN (2004), revisões sistemáticas da literatura baseiam-se em uma estratégia de pesquisa bem definida com o objetivo de coletar o máximo de material bibliográfico relevante sobre determinado assunto. TRAVASSOS *et al.* (2008) definem o conceito de *quasi*-revisão sistemática, um tipo de estudo secundário no qual o objetivo é avaliar e interpretar os trabalhos relevantes na literatura técnica sobre um determinado tópico de interesse. Este tipo de estudo contém uma etapa de planejamento da revisão na qual é criado um protocolo que especifica detalhadamente como a revisão será conduzida. São definidos, por exemplo, a questão central de pesquisa, métodos que serão utilizados para executar a revisão e fontes de dados utilizadas. Além disto, o protocolo serve como documentação da estratégia de busca e define os critérios de inclusão e exclusão para avaliar cada estudo primário que pode ser utilizado. Dessa forma, o protocolo permite que outros pesquisadores possam ter maior confiança nos resultados da revisão, pois permitem observar o grau de rigor que a revisão foi conduzida e sua completeza (BIOLCHINI *et al.*, 2007).

A *quasi*-revisão sistemática da literatura reportada neste capítulo surgiu em um contexto em que era desejado conhecer os critérios de parada para testes de software que já haviam sido propostos. Assim sendo, a *quasi*-revisão sistemática foi planejada seguindo a estrutura PICO (*Population*, *Intervention*, *Comparison*, *Outcome*) proposta por Pai *et al.* (2004) e por não aplicar nenhuma comparação, é possível classificá-la como uma *quasi*-revisão sistemática (Travassos *et al.*, 2008).

As seções a seguir têm a finalidade de definir as partes do protocolo utilizado na *quasi*-revisão sistemática da literatura bem como os dados da execução e as conclusões obtidas com base na análise desses dados. O protocolo da quasi-revisão sistemática se encontra na íntegra no Apêndice A.

3.2 Protocolo da *quasi*-Revisão Sistemática

Assim, os artigos de controle, o problema tratado, a questão de pesquisa, a população, a intervenção e a saída, bem como as respectivas palavras chaves utilizadas nas buscas foram definidos como mostrado a seguir. Além disto, é importante ressaltar que os artigos de controle e as palavras chaves utilizadas foram definidas a partir de uma revisão informal da literatura técnica em conjunto com a opinião de um especialista da área de testes de software e engenharia de software experimental.

3.2.1 Definição dos Artigos de Controle

Através de uma revisão informal da literatura técnica foram selecionados alguns artigos que tratavam de critérios de parada para testes de software. Esses artigos foram considerados de controle, pois propõem critérios de parada para testes de software. Assim cinco artigos de controles foram encontrados.

Em DALAL e MALLAWS (1990), é proposto um critério de parada baseado no custo de se dar continuidade aos testes, o custo de se corrigir defeitos com o software em produção e no descontentamento do cliente em caso de falhas. Este critério foi proposto para sistemas de software em larga escala e é um critério que pode ser aplicado em um contexto em que o software é comercializado para um único cliente.

O segundo artigo de controle utilizado foi publicado por LITTLEWOOD e WRIGHT (1995) e propõe um critério de parada especificamente para um software de missão crítica que controla um reator nuclear. A finalidade deste critério de parada é avaliar se o software atingiu a confiabilidade desejada.

Outro artigo utilizado como controle foi publicado por CHAVEZ (2000). Neste artigo é proposto um critério de parada que visa o balanceamento entre os custos de identificação e correção de defeitos contra os custos de se perder posições no mercado pela não disponibilização do software em determinado momento. Este critério de parada é proposto em um contexto no qual o software desenvolvido é comercializado em larga escala.

MALEVRIS e PETROVA (2000) propõem um critério de parada baseado no tempo entre falhas. Segundo o critério de parada, os testes devem parar quando um intervalo de falhas pré-definido for ultrapassado.

O último artigo de controle utilizado foi proposto por PROWELL (2004) no qual é apresentado um critério de parada baseado na estimativa da confiabilidade do software e o custo de falhas encontradas depois que o software está em fase de produção. O critério define que o software deve continuar a ser testado enquanto o custo com os testes for menor que o ganho por aumentar a confiabilidade. Segundo o autor, este critério pode ser utilizado para sistemas compostos por software e hardware.

3.2.2 Qualidade e Amplitude da Questão

- **Problema**: testes de software são importantes para avaliar a qualidade do software. Entretanto, sabe-se que na prática não é possível provar a correção de um programa que tenha alguma complexidade utilizando testes de software. Assim há sempre a possibilidade de haver defeitos no software e sempre pode haver um caso de teste que identificaria esse defeito. Dessa forma, isto leva ao problema da indecisão do momento de parada dos testes de software.
- **Questão**: quais critérios têm sido utilizados para determinar o momento de parada de teste de software?
- **População**: projetos e processos de software;
    - **Palavras chaves**: *software application, software development, software project, software product, software system, software safety system, safety-critical software, computer software, software componentes, Project management, software measurement, software industry*
- **Intervenção**: teste de software
    - **Palavras chaves**: *software testing, statistical testing, testing requirement, testing process, program testing, reliability requirement, testing the software, testing procedure*
- **Saída**: conjunto de critério que podem ser utilizados para a tomada de decisão de quando o teste de software deve parar
    - **Palavras chaves**: *decision analysis, degree of correctness, failure behaviour, probability of failure, reliability state, software assurance, stop testing, stopping analysis, stopping criteria, stopping criterion,*

stopping rule, testing costs, testing requirement, reliability model, optimal software release, release time

3.2.3 Seleção de Fontes

Como fontes de dados foram considerados artigos disponíveis na web nos idiomas português e inglês e referenciados em máquinas de busca que permitam a pesquisa de strings no *abstract*, título e palavra chave. Também foram pesquisados livros sobre engenharia de software disponíveis na web ou impressos e os anais das conferências listadas abaixo com suas respectivas datas. A escolha por esses anais se deu por questões de disponibilidade.

- **SBES**: 1993, 1994, 1995, 1997, 1999, 2000, 2002, 2003, 2004, 2005, 2006, 2007, 2008
- **SBQS**: 2003, 2004, 2005, 2006, 2008, 2009, 2010
- **ESELAW**: 2008, 2009, 2010

A Tabela 3-1 – Máquinas de busca utilizadas na *quasi-revisão* sistemática demonstra as máquinas de busca onde a string de busca foi executada. Essas máquinas de busca foram por referenciarem grande parte do conhecimento científico da área de engenharia de software.

Tabela 3-1 – Máquinas de busca utilizadas na *quasi*-revisão sistemática

Máquina de busca	Link
Scopus	http://www.scopus.com/
IeeeXplore	http://ieeexplore.ieee.org/
EI Compendex	http://www.engineeringvillage.org/
Web of Science	http://isiknowledge.com/

3.2.4 Seleção dos Estudos

Como é comum em revisões sistemáticas da literatura critérios de inclusão e execução são utilizados para definir quais materiais possuem conteúdo adequado ao tema e devem ser incluídos na revisão. Assim, os critérios de inclusão e exclusão foram definidos como se segue.

- **Critérios de inclusão**:
  - Tratar de testes de software
  - Descrever critérios de parada para processos de testes de software; E
  - Apresentar alguma aplicação dos critérios de parada propostos; E
  - Apresentar referência bibliográfica que caracterize o critério apresentado caso não seja de autoria

- **Critérios de exclusão**:
  - Artigos que não tratam critérios de parada para testes de software; OU
  - Artigos que não apresentem alguma forma de avaliação (experimental, prova de conceito, qualquer outro tipo de estudo ou simples demonstração) sobre o critério proposto; OU
  - Artigos que não estejam disponíveis por meio digital ou impresso; OU
  - Artigos publicados em idiomas diferentes do português ou inglês

Além da classificação anteriormente mencionada, durante a leitura dos artigos foi realizada uma avaliação da qualidade dos artigos. O objetivo desta avaliação é identificar os artigos que possuem uma relação mais estreita com o tema que está sendo investigado e, com isto, terão maior confiabilidade no resultado final. Cada um desses critérios possui uma pontuação diferente que se refere ao quanto o critério é importante para esta *quasi*-revisão sistemática. Ou seja, trata-se de uma avaliação da qualidade do artigo em relação à questão de pesquisa. Os critérios para avaliação da qualidade dos artigos e seus respectivos pesos são listados a seguir.

- O artigo apresenta algum tipo de estudo experimental ou avaliação do critério proposto? (2 pt)
- O artigo apresenta alguma prova de conceito? (1 pt)
- O artigo caracteriza o software em que o critério pode ser aplicado? (2 pt)
- O artigo utilizada metodologia e linguagem que facilita o entendimento? (2 pt)
- O artigo utiliza metodologia adequada? (1 pt)
- O artigo deixa explícitas as condições e restrições de aplicação do critério? (1 pt)

3.2.5 Procedimento para Seleção dos Artigos

Após a definição da string de busca a ser executada nas máquinas de busca, a procura por materiais nos meios impressos mencionados na seção 3.2.3 e a definição dos critérios de inclusão e exclusão descritos na seção 3.2.4, a *quasi*-revisão sistemática pode ser executada. Para respeitar o formalismo do processo de execução um procedimento foi definido. A Figura 3-1 – Procedimento para seleção de artigos, bem como o texto que seguinte descrevem o procedimento adotado para a seleção dos artigos.

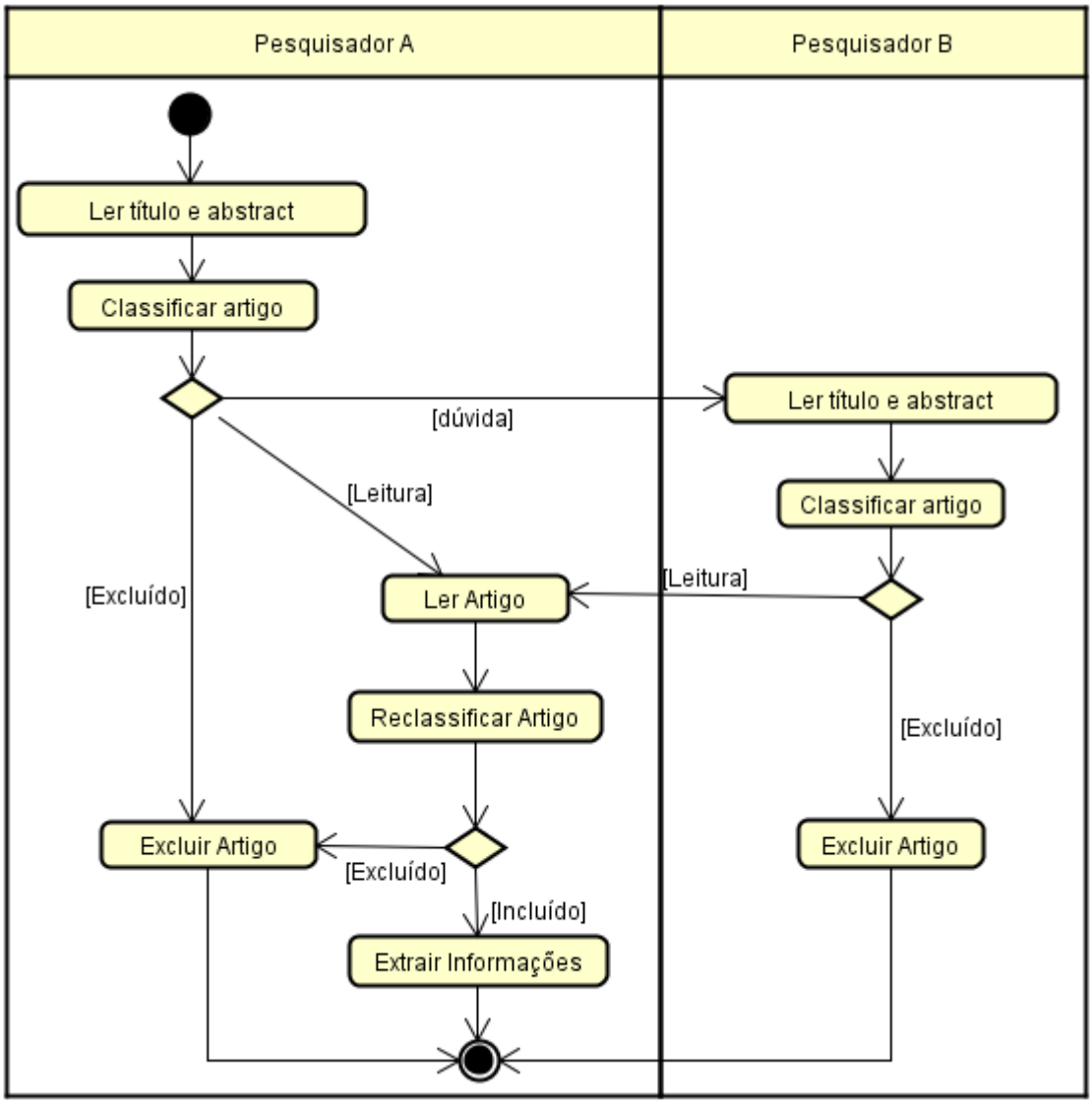


Figura 3-1 – Procedimento para seleção de artigos

A seleção dos artigos foi realizada por dois pesquisadores com níveis de experiência diferentes. O **Pesquisador A**, que participou em 2 projetos que contemplavam atividades de testes e desenvolvou 2 trabalhos acadêmicos sobre o assunto, portanto com menos experiência, e o **Pesquisador B** que possui grande experiência em quasi-revisões sistemáticas da literatura e realizou inúmeras orientações relacionados à testes de software.

O **Pesquisador A** foi responsável pela execução da string de busca nas máquinas de busca e pela procura dos materiais impressos. Depois de ter o conjunto de materiais pré-selecionados, o Pesquisador A realizou a leitura do título e *Abstract* de todos os documentos retornados pelas máquinas de busca classificando os artigos com os seguintes status:

- **Leitura**: documentos que tratam de alguma forma de critérios de parada para testes de software
- **Excluído**: documentos que não tratam de critérios de parada para testes de software
- **Dúvida**: documentos em que houve dúvida se tratam de alguma forma de critérios de parada para testes de software

Após esta classificação inicial, o **Pesquisador B** realizou a leitura do título e *abstract* dos documentos que foram classificados com o status **Dúvida** e reclassificou esses documentos com o status **Leitura** ou **Excluído**.

O próximo passo do processo foi a realização da leitura, pelo **Pesquisador B**, de todos os documentos classificados como **Leitura**. Ao realizar esta atividade, os documentos foram reclassificados da seguinte forma:

- **Incluído**: documentos que atendem aos critérios de inclusão e não atendam aos critérios de exclusão. Esses documentos tiveram suas informações extraídas
- **Excluído**: documentos que não atendem aos critérios de inclusão ou que atendam aos critérios de exclusão. Esses documentos foram excluídos da *quasi*-revisão sistemática

3.2.6 Resultados da Execução

A execução da *quasi*-revisão sistemática foi iniciada em abril de 2011. Após a execução da string de busca nas máquinas de busca um conjunto de artigos foi retornado. A Tabela 3-2 demonstra a quantidade de artigos retornados e separados por máquina de busca. Cabe ressaltar que nenhum conteúdo de livros ou anais de congressos pesquisados foi selecionado durante o processo de procura, pois não foram encontradas informações referentes à critérios de parada nesse tipo de publicação.

Tabela 3-2 – Artigos selecionados para leitura

Máquina de busca	Número de artigos	Controles
Scopus	378	5
IeeeXplore	129	5
EI Compendex	191	5
Web of Science	103	1

Depois disto, o procedimento descrito na seção anterior foi executado e um conjunto de artigos foi selecionado para leitura e outro conjunto foi excluído. Nesse ponto, iniciou-se a leitura dos artigos para a extração dos dados e, durante esse

processo, ainda houve artigos que foram excluídos, pois não foram encontrados, não apresentavam critérios de parada como anunciado no título e *abstract*, não apresentavam experimento ou demonstração do critério apresentado ou estavam em um idioma diferente do especificado. Além disto, foram encontrados três artigos diferentes que propunham o mesmo critério de parada. Dessa forma, a extração dos dados ocorreu efetivamente com setenta e quatro artigos como demonstra a Tabela 3-3.

Tabela 3-3 – Artigos para extração

Status do artigo	Quantidade
Leitura+Controle	121
Não encontrados	16
Não apresentaram critério de parada	23
Não apresenta experimento ou demonstração	3
Artigo em chinês	2
Critério repetido	3
Total de artigos para extração	**74**

3.2.7 Extração dos Dados

A etapa de extração dos dados dos artigos selecionados foi realizada com apoio de um formulário de extração que pode ser encontrado no Apêndice B. A descrição de cada campo contido nesse formulário é realizada a seguir.

- **Campos extraídos diretamente das máquinas de busca**
  - **Título**: o título do artigo
  - **Autores**: autores do artigo
  - **Ano de publicação**: ano de publicação do artigo
  - **Fonte de publicação**: conferência/journal onde o artigo foi publicado
  - ***Abstract***: resumo do artigo
  - **Máquina de busca**: nome da máquina de busca onde foi encontrado o artigo.
- **Campos extraídos a partir do entendimento do artigo**
  - **Caracterização do software**: descrição do tipo de software em que o autor sugere que o critério possa ser aplicado ou que o critério foi aplicado para avaliação.
  - **Níveis de testes**: definem quais níveis/fases de testes o critério de parada está preparado para funcionar. Exemplo de níveis de testes: Testes de aceitação, Testes de integração, Testes de regressão,

Testes de stress, Testes de sistemas, Testes funcionais, Testes de unidade.

- **Modelo Utilizado**: identifica os modelos que o processo de desenvolvimento do software deve prover para que o critério possa ser aplicado. Por exemplo, diagramas da UML, Diagrama de fluxo de dados, Cadeia de Markov.
- **Habilidades Necessárias**: descreve as habilidades que devem ser providas pela equipe de testes para a aplicação do critério de parada.
- **Contexto de Aplicação**: Identifica em qual contexto o critério de parada foi proposto e aplicado. Exemplos de contextos são comercialização em larga escala, comercialização para cliente único, utilização própria.
- **Princípio do critério**: em que se baseia o critério de parada. Por exemplo, modelo de confiabilidade, fluxo de execução do sistema.
- **Objetivo do critério**: o principal objetivo do critério. Por exemplo, maximizar a confiabilidade, minimizar o custo com testes.
- **Considera depuração perfeita**: [Sim] quando uma falha é encontrada, o defeito que provocou essa falha é corrigido imediatamente, instantaneamente e sem provocar novos defeitos; [Não] quando uma falha é encontrada, o defeito que provocou esta falha não é corrigido imediatamente, nem instantaneamente ou pode provocar novos defeitos.
- **Confiabilidade pré-determinada**: [Sim] Confiabilidade é uma entrada fixa para o critério de parada. [Não] Caso contrário.
- **Orçamento para testes pré-determinado**: [Sim] Custo com os testes é uma entrada fixa para o critério de parada. [Não] Caso contrário.
- **Custo com testes é pré-determinado**: [Sim] Custo com os testes é uma entrada fixa para o critério de parada. [Não] Caso contrário.
- **Número de faltas é pré-determinado**: [Sim] Caso o número de faltas deva ser conhecido ou estimado e serve como entrada para o critério. [Não] Caso contrário.
- **Taxa de identificação de falhas é pré-determinada**: [Sim] caso a taxa de identificação de falhas deva ser conhecida ou estimada e serve como entrada para o critério. [Não] Caso contrário.
- **Considera a qualidade dos casos de testes**: [Sim] caso a qualidade dos casos de testes sejam relevantes para o resultado do critério de parda. [Não] Caso contrário.

- **Considera atividade de teste isolada**: [Sim] caso o critério considere que a atividade de teste aconteça isoladamente ao final do processo de codificação. [Não] Caso contrário.
- **Diferencia taxa de falhas**: [Sim] caso a taxa de falhas seja diferenciada na fase de testes e na fase de produção. [Não] Caso contrário.
- **Variações do critério**: Quantidade de variações do critério que são apresentadas no artigo. Por exemplo, um mesmo modelo de confiabilidade pode resultar em dois critérios de parada. Um que minimiza o custo e outro que maximiza a confiabilidade.
- **Modelo de confiabilidade base**: é o modelo de confiabilidade utilizado para definir o critério de parada. Geralmente os modelos de confiabilidade não são nomeados. Portanto, é referenciado o autor que propôs o modelo. Quando há uma evolução do modelo, a referência é feita ao modelo de origem e é especificado que houve uma evolução.
- **Modelo de custo base**: é o modelo de custo no qual o critério é baseado. Geralmente os modelos de custo não são nomeados. Assim sendo, é referenciado o autor que propôs o modelo. Quando há uma evolução do modelo, a referência é feita ao modelo de origem e é especificado que houve uma evolução.
- **Distribuição de falhas**: é a forma como se presume que as falhas estão distribuídas no software. Os modelos de confiabilidade utilizam distribuições diferentes. Por exemplo, distribuição exponencial e Distribuição de Poisson.
- **Unidade de medida de falhas**: definem em qual unidade as falhas são medidas. Os valores possíveis são: Falhas acumuladas (função de valor médio), Intensidade de falhas, Taxa de ocorrência de falhas e Tempo médio entre falhas.
- **Tipo de modelo de confiabilidade**: define o tipo de modelo de confiabilidade que pode ser modelo de estimativa, modelo de predição e modelo de predição em fases iniciais. Os modelos de estimativa determinam a confiabilidade atual do software. Os modelos de predição determinam a confiabilidade futura do software e são utilizados quando o software se encontra nas fases de testes ou operacional, pois nestas fases é possível obter dados sobre as falhas do software. Os modelos de predição em fases iniciais são utilizados quando ainda não se tem dados sobre as falhas.

- **Restrições do critério**: restrições que possam ser impostas ao critério ou ao modelo que dê origem ao critério. Por exemplo, a identificação de falhas é modelada por um NHPP (*Non-Homogeneous Poisson Process*).
- **Parâmetros do Critério**: são os parâmetros de entrada necessários para a aplicação do critério.
- **Critério**: Uma descrição textual do critério e sua representação através de fórmulas e/ou gráficos.

- **Critérios de Avaliação**
  - **A**: O artigo apresenta algum tipo de estudo experimental ou avaliação mais formal do critério proposto? (2 pontos)
    - Um estudo experimental pode ser considerado uma forma mais formal de avaliação. Com estudos experimentais a parcialidade na avaliação dos critérios pode ser mitigada.
  - **B**: O artigo apresenta alguma prova de conceito? (1 ponto)
    - Apesar de não ser uma avaliação tão formal quanto estudos experimentais, provas de conceitos podem ser um critério de avaliação.
  - **C**: O artigo caracteriza o software em que o critério pode ser aplicado? (2 pontos)
    - A caracterização do software é importante, pois um bom critério para um tipo de software pode ser ruim para outros tipos.
  - **D**: O artigo utiliza metodologia e linguagem que facilita o entendimento? (2 pontos)
    - O artigo deve ser escrito com linguagem clara que facilite a interpretação e extração do critério apresentado.
  - **E**: O artigo utiliza terminologia adequada? (1 ponto)
    - É importante que o artigo utilize os termos conceitualmente corretos para evitar danos à interpretação
  - **F**: O artigo deixa explícitas as condições e restrições de aplicação do critério? (1 ponto)
    - Sem a devida avaliação, não é possível afirmar que um mesmo critério seja capaz de ter a mesma eficiência em qualquer ambiente, com qualquer equipe e em qualquer tipo de software. Por esse motivo, é preciso que os artigos deixem explícitos as restrições do critério apresentado.

3.2.8 Análise dos Dados

Depois de extrair os dados de cada artigo e preencher os formulários de extração, os critérios de parada foram agrupados por alguns campos. Com o agrupamento dos dados foi possível inferir informações relativas ao estado atual em que se encontra o conhecimento publicado sobre critérios de parada para testes de software. Além disto, percebeu-se que dos setenta e quatro artigos encontrados alguns propunham variações de um mesmo critério de parada. Por essas variações possuírem características específicas, no agrupamento dos dados de alguns atributos, elas foram consideradas como critérios de paradas diferentes. Por exemplo, alguns artigos propuseram um critério de parada que maximiza a confiabilidade e, no mesmo artigo, apresentaram também uma variação desse critério que maximiza a confiabilidade restrita ao custo com os testes. Nesse caso, foram contabilizadas duas variações do critério de parada.

A primeira análise realizada foi relacionada à caracterização do software em que o critério de parada proposto poderia ser aplicado. Esta análise foi realizada, pois se imaginava que dificilmente um mesmo critério poderia ser utilizado por diversos tipos de software e, mesmo se pudesse ser utilizado, seria inviável avaliar o critério perante diversos tipos de software diferentes. Assim, tentou-se verificar em quais tipos de software determinados critérios poderiam ser aplicados.

Como foi mantida a caracterização proposta nos artigos nos quais os critérios foram encontrados, algumas categorias parecem se sobrepor ou não pertencer ao mesmo nível de abstração. Por exemplo, nada impede que “softwares comerciais” sejam também “sistemas de software em larga escala” ou que “softwares de missão crítica” sejam também “softwares embarcados”. Entretanto, como a finalidade da classificação realizada é verificar se o tipo de software influenciava no critério de parada, a caracterização explicitada no artigo foi mantida.

Como pode ser visto na Figura 3-2, a maioria dos artigos não caracteriza o software em que o critério de parada pode ser aplicado. Outro fato importante é que algumas caracterizações são imprecisas e não dimensionam de forma real o tipo de software em que o critério pode ser aplicado tão pouco apresentam informações sobre o contexto de aplicação. Por exemplo, “sistemas de software em larga escala” podem ser uma caracterização interpretada de formas diferentes por pessoas diferentes em contextos diferentes.

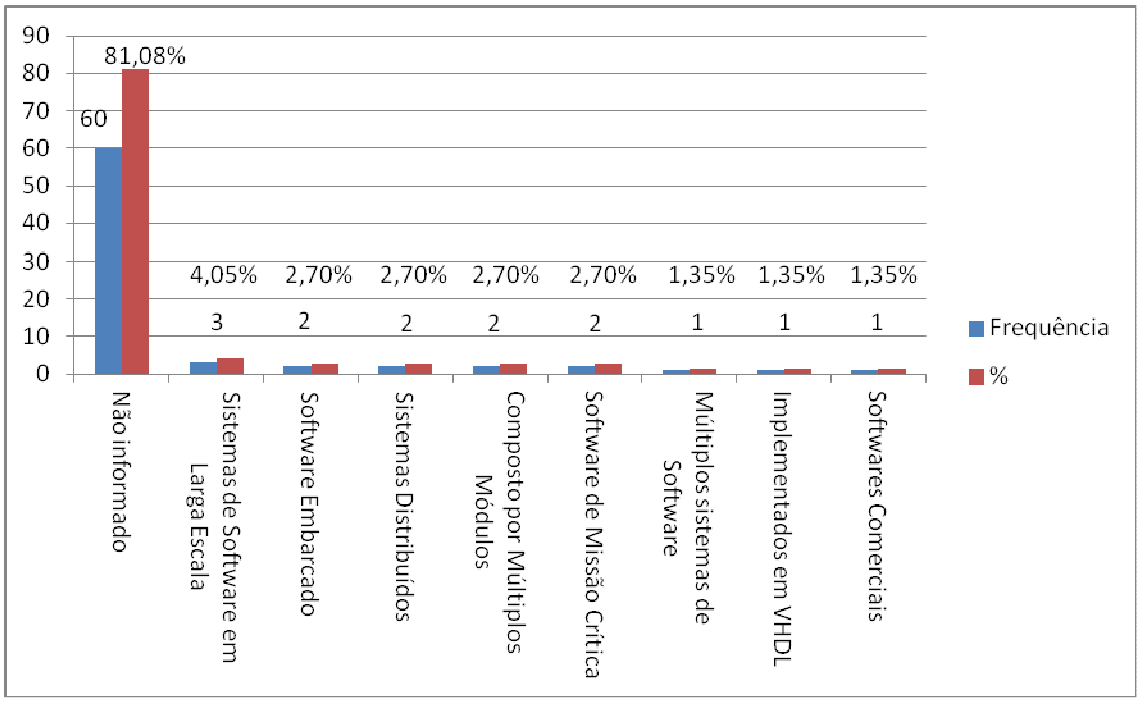


Figura 3-2 – Critérios de parada por caracterização do software

A segunda análise realizada foi relativa ao princípio do critério. Princípio do critério pode ser entendido como o instrumento que deu base ao critério. Através da

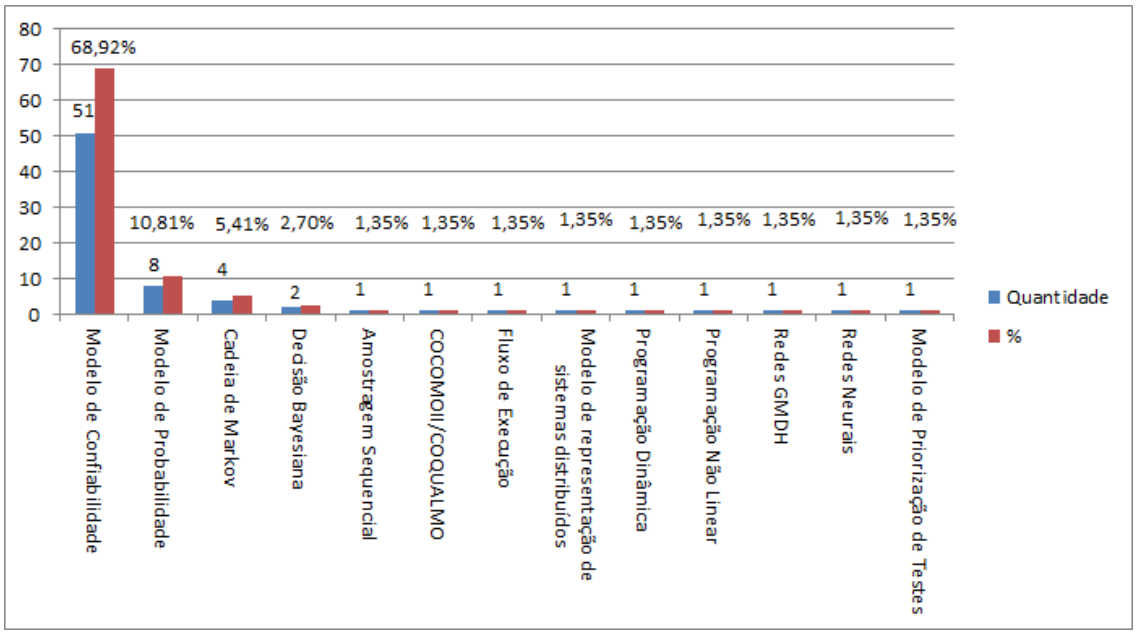


Figura 3-3, é possível perceber que a maioria dos critérios de parada é baseado em modelos de confiabilidade. Isto reflete o fato de que a forma mais utilizada para mensurar a confiabilidade do software seja por meio de modelos de confiabilidade.

Alguns critérios também são propostos sobre modelos de probabilidade que utilizam teorias probabilísticas para determinar o momento de parada dos testes. De certa forma, estes modelos podem ser considerados modelos de confiabilidade, entretanto, não foram classificados dessa forma neste trabalho, pois foi respeitada a conceituação explicitada nos artigos.

Um problema recorrente da utilização de modelos de confiabilidade para apoio da decisão de quando parar os testes, é que a aplicação desses modelos depende de dados de falhas do software. Por sua vez, esses dados dependem da execução do software e são coletados apenas nas fases de testes ou de operação, levando ao reconhecimento tardio do estado da confiabilidade do software podendo causar a parada dos testes no momento inadequado. Outro problema que pode acontecer devido à utilização de modelos de confiabilidade é que esses modelos precisam de uma quantidade relevante de dados de falha para medir a confiabilidade de forma mais precisa. Ao atingir essa quantidade, o ponto ótimo de parada já pode ter sido atingido.

Os critérios baseados em Cadeias de Markov, como por exemplo o critério proposto em WHITTAKER e THOMASON (1994), merecem destaque, pois utilizam o perfil operacional dos usuários para determinar o momento de parada. Nesse caso, uma Cadeia de Markov é utilizada para representar os estados do sistema e a probabilidade de mudança de um estado para outro. Com estas informações é possível priorizar os testes nas funcionalidades mais utilizadas do sistema. Entretanto, o número de funcionalidades de um sistema pode ser muito grande causando a inviabilidade desse tipo de critério.

Outro fato importante de se comentar é que os critérios de parada que não são baseados em modelos de confiabilidade foram propostos para tipos de software muito específicos. Por exemplo, encontrou-se critério de parada para software desenvolvido em linguagem VHDL (Very High Speed Circuits Hardware Description Language) e um critério específico para um software de controle de sensores de monitoramento um reator nuclear.

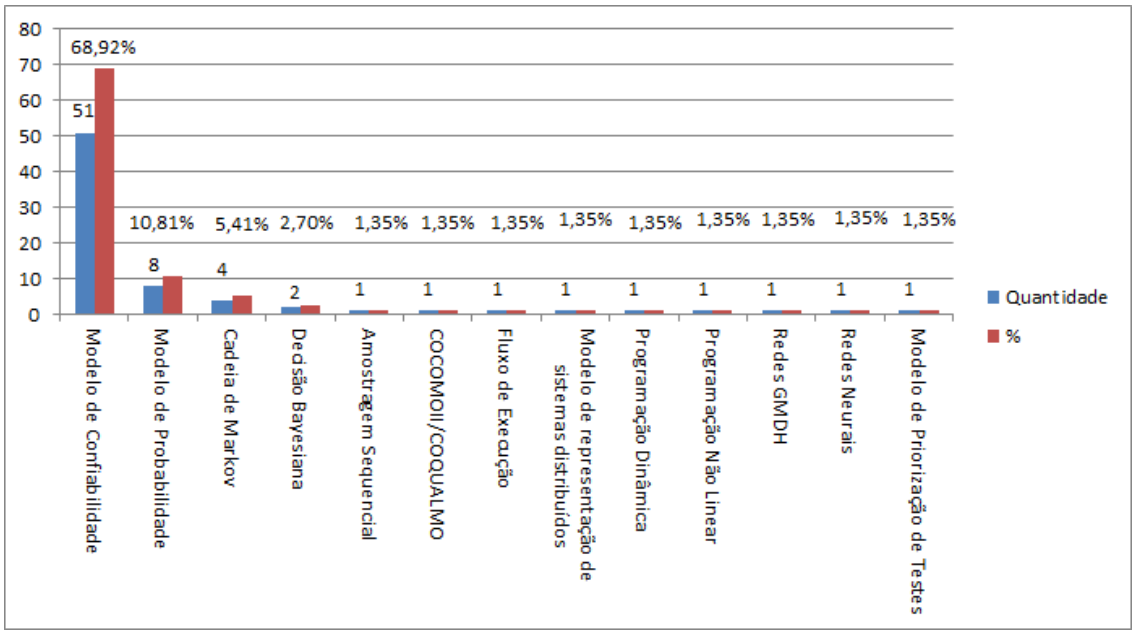


Figura 3-3 – Critérios de parada por princípio

Os critérios de parada também foram agrupados segundo seus objetivos. Por exemplo, alguns critérios são baseados na minimização do custo, outros na maximização da confiabilidade, outros na definição de prazo de entrega. Nesta análise, é possível perceber que a maior parte dos critérios de parada é baseada na minimização do que pode ser chamado de “custo da qualidade”.

O custo da qualidade foi chamado dessa forma, pois é calculado utilizando-se modelos de custo que consideram o custo de atividades e fatores que influenciam na qualidade do software. Por exemplo, modelos de confiabilidade podem considerar o custo de correção de uma falha na fase de testes, custo de correção de uma falha na fase de operação, custo por unidade de tempo para correção/detecção de falhas, custo de configuração dos testes, custo devido à falha do software em produção.

É importante explicitar que se pode imaginar que o custo mínimo para desenvolvimento seria um processo sem teste. Nesse caso, o custo com testes seria igual a zero. Entretanto, caso a atividade de testes não ocorra, mais falhas serão encontradas na fase de produção e talvez causem prejuízos maiores que o gasto com os testes.

Além disto, há uma quantidade relevante de critérios que têm como objetivo atingir uma confiabilidade pré-estabelecida e minimizar o custo restrito a confiabilidade. Nesse último caso, um valor desejado para a confiabilidade é fixado e o custo com testes é calculado de forma que seja mínimo e possa atingir a confiabilidade desejada. A Figura 3-4 demonstra o resultado desta parte da análise.

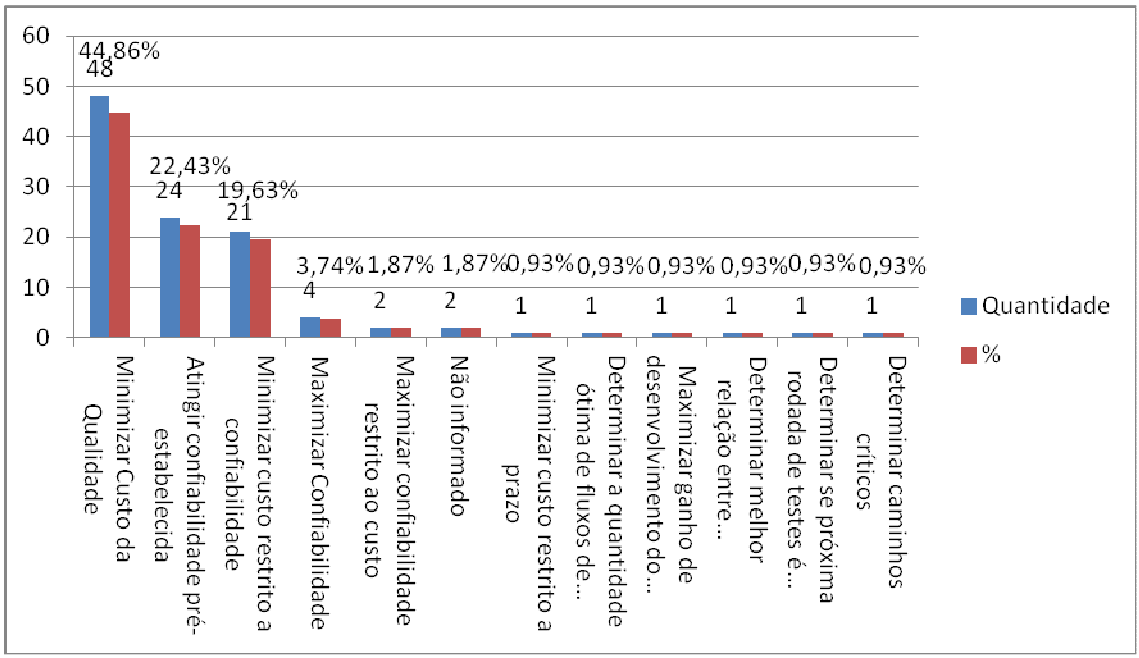


Figura 3-4 – Critérios de parada por objetivo

Ao se pesquisar por critérios de parada para testes de software, um conceito recorrente é o de depuração perfeita. GOEL (1985) discorre sobre este assunto e declara que esta é uma das maiores limitações para a aplicação real dos modelos de confiabilidade. XIE e YANG (2003) definem que um processo de testes que possuir depuração perfeita considera que quando uma falha é encontrada, a falta que gerou esta falha é corrigida imediatamente e, sem a introdução de novas falhas.

Como a depuração perfeita é uma característica que pode influenciar o critério de parada, foi realizado um agrupamento considerando esta característica como mostrado na Figura 3-5. É possível perceber que cerca da metade dos critérios de parada não mencionam se consideram ou não depuração perfeita, o que pode ser considerado um risco. A outra metade dos critérios divide-se em considerar (27,03%) ou não (24,32%) esta característica. Ao utilizar um critério de parada que considera depuração perfeita, assume-se o risco de subestimar o tempo e custos com testes.

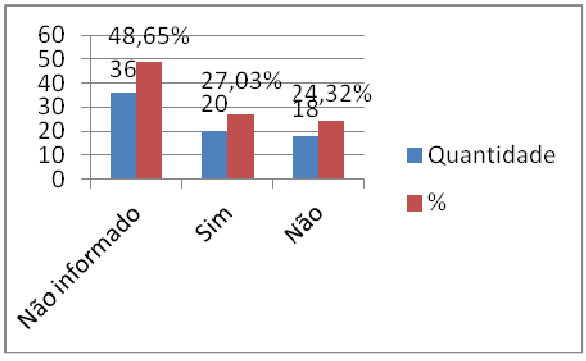


Figura 3-5 – Critérios de parada por depuração perfeita

As pré-suposições dos critérios também foram analisadas, pois é importante saber o que é necessário contemplar para que o critério possa ser aplicado de forma eficaz. Entretanto, como pode ser visto na Figura 3-6, grande parte dos critérios de parada não apresentou nenhuma pré-suposição para utilização. Além disto, as pré-suposições apresentadas por alguns critérios de parada são teóricas ou matemáticas podendo dificultar a aplicação prática do critério. Um exemplo de pré-suposição é “O fenômeno de detecção de defeitos é modelado através do processo não homogêneo de Poisson” ou “Tempo entre falhas é independente”. Dificilmente os responsáveis pelos testes teriam condições de levantar essas informações ou conseguiriam dados necessários para calculá-las ao longo do projeto.

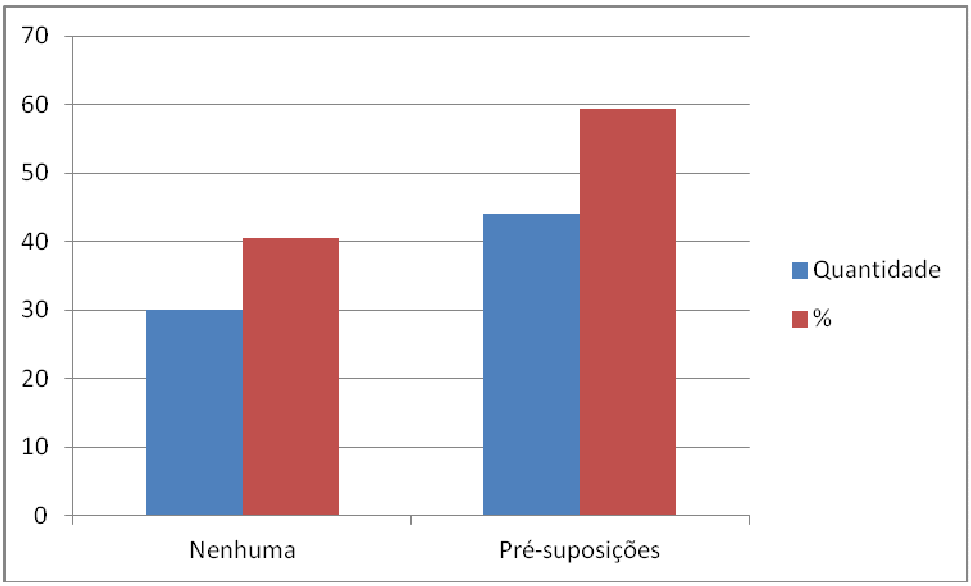


Figura 3-6 – Critérios de parada por pré-suposições

Procurou-se verificar também se os critérios encontrados possuem avaliação experimental. Entretanto, o número de artigos que explicita esta informação é muito pequeno. Assim, passou-se a avaliar se o artigo apresentava uma avaliação experimental ou simples demonstração do critério de parada. Exemplo de avaliações formais consideradas são simulações, comparação com critérios de parada existentes utilizando métodos estatísticos e prova de conceito.

Como demonstrado na Figura 3-7, mesmo considerando qualquer tipo de avaliação, o número de critérios avaliados ainda é pequeno. Apenas 27,03% dos critérios de parada encontrados foram formalmente avaliados. Isto pode ser uma indicação de que os critérios foram propostos para resolução de um problema específico ou de uma situação específica, mas não se tem conhecimento em quais situações esses critérios podem ser utilizados. Ou seja, não existem evidências ou

indicações de que determinado critério funcione e em que situações sua utilização é recomendada.

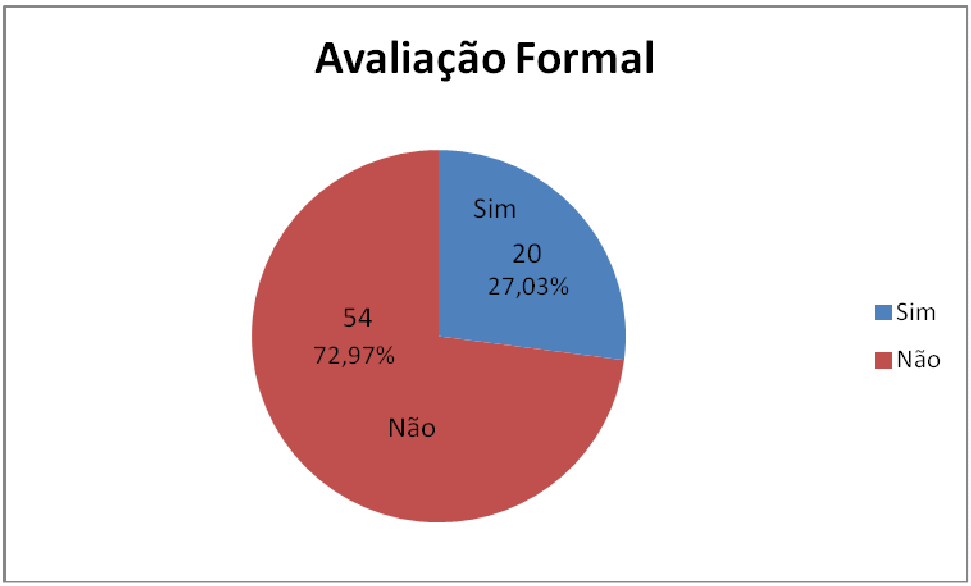


Figura 3-7 – Critérios de parada por avaliação formal

Além disto, através da quasi-revisão sistemática foi possível perceber que os critérios de parada para testes de software podem ser divididos em dois grupos distintos: critérios de parada apriorísticos e critérios de parada não apriorísticos.

Os critérios de parada apriorísticos utilizam metas ou restrições gerenciais definidas a priori para determinar o momento de parada. Essas metas podem ser confiabilidade, orçamento ou tempo para testes. Nesses casos, o critério de parada propõe a identificação da melhor forma de conduzir os testes com base na meta pré-estabelecida e também fornecem subsídios para verificar se a meta foi atingida. Tais metas geralmente são expressas através de requisitos não funcionais e podem ser, por exemplo, requisição do cliente para obter software de maior qualidade, uma exigência legal de algum órgão regulamentador por questões de segurança, uma restrição de qualidade da própria empresa desenvolvedora com a finalidade de produzir softwares que falhem menos e assim obtenham vantagens competitivas sobre concorrentes.

Por exemplo, em LITTLEWOOD e WRIGHT (1995) um órgão regulamentador define que os testes devem parar quando o software atingir, com confiança de 99%, a probabilidade de falha de 10^{-3}. Portanto, em casos como esses, já existe uma meta definida a priori e o critério de parada passa a ter a função de auxiliar a equipe de testes a atingir esta meta e de avaliar se a mesma foi atingida.

Os critérios de parada não apriorísticos devem ser utilizados quando não há metas ou restrições gerenciais pré-estabelecidas. Dessa forma, é preciso fornecer à

equipe de testes um critério de parada que estabeleça um ponto em que os testes devem parar sem levar em consideração nenhuma meta ou restrição gerencial pré-estabelecida. Em DALAL e MALLOWS (1990), é apresentado um exemplo de critério de parada não apriorístico que considera o custo em dar continuidade aos testes, o custo de correção de falhas com o software em produção e no descontentamento do cliente caso a falha ocorra. O critério proposto encontra o melhor momento para parar os testes que equilibre esses custos.

Portanto, as categorias dos critérios de parada podem ser definidas como na Figura 3-8 e a descrição a seguir.

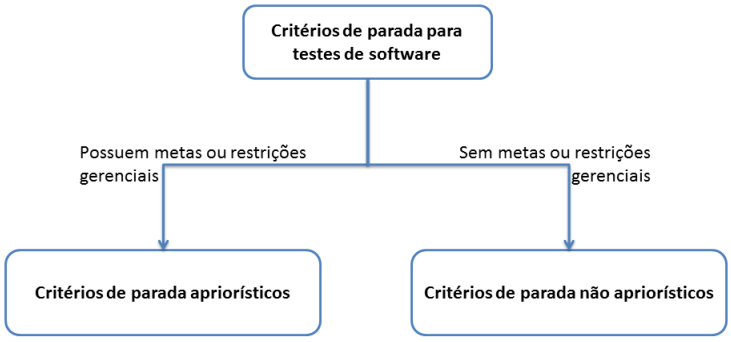


Figura 3-8 – Classificação dos critérios de parada

- **Critérios de parada apriorísticos**: auxiliam a atingir uma meta para a fase de testes determinada a priori e/ou avaliar se esta meta foi atingida.
- **Critérios de parada não apriorísticos**: determinam o melhor momento para parar os testes sem considerar metas definidas a priori.

Assim, com base na análise desses dados é possível perceber em que estado se encontra as pesquisas na área de critérios de parada para testes de software. Os critérios de parada extraídos a partir da *quasi*-revisão sistemática podem ser visualizados através dos formulários de extração contidos no Apêndice C.

3.3 Conclusão

Algumas conclusões podem ser inferidas a partir da interpretação dos dados extraídos da *quasi*-revisão sistemática executada.

O fato de não haver caracterização do software para aplicação da maioria dos critérios de parada, leva-se a acreditar que os critérios de parada propostos poderiam ser aplicados a qualquer tipo de software. Entretanto, como a maioria dos critérios também não apresenta avaliação formal de sua utilização e nenhuma forte evidência de seu funcionamento, a aplicação ampla dos critérios de parada encontrados pode

ser considerada arriscada. Assim, persiste a dúvida de qual critério de parada pode ser utilizado.

Além disto, muitos critérios não consideram o tempo de correção de defeitos e que defeitos podem ser inseridos durante o processo de correção, ou seja, consideram que o processo de depuração é perfeito. Como já é de conhecimento da área, isto não é verdade. Portanto, deve-se ter em mente os riscos de se utilizar um critério de parada que considera depuração perfeita.

Outro resultado encontrado diz respeito às pré-suposições dos critérios de parada. Esse ponto pode ser considerado crítico. Isto porque muitos critérios não explicitam as pré-condições necessárias para sua aplicação ou possuem pré-condições matemáticas de difícil constatação. Por exemplo, alguns critérios possuem como pré-condição que as falhas estejam distribuídas de acordo com alguma distribuição probabilística. Para determinar como as falhas estão distribuídas, é necessário ter uma amostragem dos dados de falhas e realizar testes estatísticos sobre elas. Assim, é necessária a utilização de ferramentas estatísticas e, além disto, esse tipo de pré-condições é possível de ser constatada apenas quando o software já está nas fases de testes ou em produção.

Outro dado importante que se pôde obter, é que a maioria dos critérios de parada baseia-se em modelos de confiabilidade. Esses modelos vêm o software como uma caixa preta e não se importam com a maneira que os testes estão sendo conduzidos e nem a fase de teste em que o software se encontra. Além disto, os modelos necessitam de uma quantidade considerável (não especificada) de dados de falhas para que sua aplicação seja eficiente. O que pode acontecer, é que ao se conseguir dados suficientes para aplicação eficiente do modelo, o momento ótimo para a parada dos testes pode já ter sido ultrapassado.

Ainda se tratando de modelos de confiabilidade, outro problema encontrado está relacionado à forma como esses modelos são implementados. Esses modelos são representados através de fórmulas matemáticas complexas o que pode dificultar sua aplicação prática.

Também foi possível constatar que a maioria dos critérios de parada tem como objetivo minimizar o custo da qualidade. Nesses casos, modelos de custos são utilizados para determinar o custo com as atividades relacionadas aos testes. O principal problema dessa abordagem está relacionado à quantificação dos parâmetros necessários para aplicação dos modelos. Um exemplo de parâmetro difícil de estimar é a quantidade em dinheiro que se perderia por não entregar o software no tempo planejado.

Além das limitações relacionadas anteriormente, através da *quasi*-revisão sistemática foi possível constatar que os critérios de parada podem ser categorizados em dois grandes grupos. O primeiro grupo de critério de parada utiliza metas gerencias definida a priori para determinar o momento de parada. Essas metas ou restrições gerenciais podem ser: confiabilidade desejada, orçamento para testes ou prazo para testes. O segundo grupo propõe a identificação de um ponto onde os testes devem parar sem necessitarem de metas a priori. Baseando-se nesta característica, os critérios de parada foram classificados como **apriorísticos** e **não apriorísticos**.

Em síntese, é possível concluir que não existe um critério de parada único que possa ser utilizado em qualquer tipo de software e em diversas situações diferentes. Talvez nunca exista um critério com estas características, pois o desenvolvimento de software é uma atividade complexa que varia de acordo com o contexto. Entretanto, decidir o melhor momento de parada levando em consideração apenas a experiência da equipe de desenvolvimento ou de forma totalmente aleatória provavelmente não é a melhor opção. O que pode ser feito é tentar encontrar qual o critério de parada que mais se adequa ao software em desenvolvimento.

4 Procedimento para Apoio à Seleção de Critérios de Parada para Testes de Software

Neste capítulo é descrito o procedimento utilizado para apoio à seleção de critérios de parada para testes de software. Através da quasi-revisão sistemática descrita no capítulo anterior, um corpo de conhecimento sobre critério de parada para testes de software e atributos para caracterização de projetos de software foram identificados possibilitando a implementação do procedimento de seleção.

4.1 Introdução

Diversas atividades do processo de desenvolvimento de software precisam ser apoiadas por soluções tecnológicas. A escolha das tecnologias utilizadas nas atividades do processo de desenvolvimento pode considerar fatores técnicos, tecnológicos, sociais, políticos e econômicos (DIAS NETO, 2009). Este Capítulo descreve um procedimento para apoiar a seleção de critérios de parada para testes de software que considera fatores que dizem respeito à forma como a tecnologia funciona na prática (técnico) e o contexto tecnológico que pode ser aplicado (tecnológico).

O procedimento descrito foi batizado de *Porantim CP* (*Porantim* para Critérios de Parada), pois é inspirado em um procedimento para seleção de técnicas de testes baseado em modelos (TTBM) proposto por DIAS NETO (2009) que foi batizado de *Porantim*. Por sua vez, o procedimento *Porantim* é uma evolução da abordagem de apoio à seleção de técnicas de testes chamada de *Esquema de Caracterização* (VEGAS E BASILI, 2005).

O procedimento *Porantim* CP se baseia em dois elementos principais: corpo de conhecimento sobre critérios de parada e o procedimento para apoio a seleção de critérios de parada. A Figura 4-1 demonstra tal processo de apoio à seleção de critérios de parada.

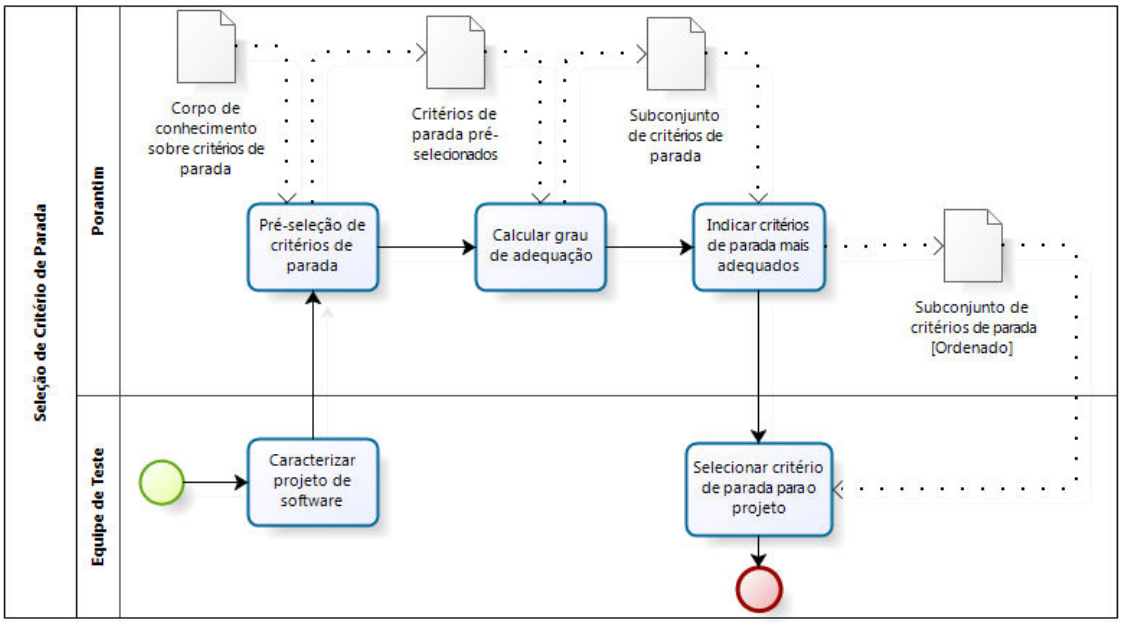


Figura 4-1 – Processo de apoio à seleção de critérios de parada

O corpo de conhecimento sobre critérios de parada para testes de software é composto pelos critérios de parada extraídos através da *quasi*-revisão sistemática descrita no Capítulo 3. O procedimento para seleção dos critérios de parada para teses de software será descrito nas próximas seções.

4.2 Apoio à Seleção de Critérios de Parada

Apenas o corpo de conhecimento sobre critérios de parada para testes de software não seria suficiente para sua utilização. Isto porque, apesar de representar um vasto conhecimento técnico sobre o assunto, o corpo de conhecimento sozinho não é capaz de realizar um redirecionamento a respeito de como o conhecimento contido deve ser utilizado. Portanto, é preciso que se tenha um procedimento bem definido para auxiliar a utilização do corpo de conhecimento.

O procedimento proposto neste trabalho é divido em cinco atividades diferentes. A Figura 4-1 bem como as próximas seções, demonstra cada uma destas atividades.

4.2.1 Caracterizar Projeto de Software

A caracterização do projeto de software é a primeira atividade que compõe o processo *Porantim* CP. A finalidade desta atividade é conhecer as principais características do software no qual o critério de parada será aplicado. Segundo DIAS NETO (2009), esta caracterização deve ser realizada pela equipe de testes, a qual

provê informações sobre atributos relacionados ao software que são importantes para a escolha de critérios de parada.

Os atributos utilizados para caracterizar os projetos de software neste trabalho foram baseados na *quasi*-revisão sistemática da literatura descrita no Capítulo 3, na consulta de opinião de especialistas da área de testes e são descritos a seguir. Além disto, é apresentada também uma justificativa do motivo da escolha de cada atributo.

- **Confiabilidade desejada**: este atributo informa, caso exista, o a confiabilidade que se deseja atingir com a execução dos testes. Caso na caracterização do projeto seja informado esse atributo, é recomendado que seja sugeridos critérios de parada que possuam esta informação como parâmetro de entrada. Nesse caso, o critério de parada fornece informações de como chegar à confiabilidade desejada e também se a confiabilidade foi atingida.
- **Orçamento esperado para testes**: este atributo informa, caso exista, o orçamento destinado à fase de testes do projeto de software. Se esse atributo for informado na caracterização do projeto, é recomendado que critérios de parada que recebam o orçamento pré-determinado para testes como parâmetro de entrada sejam sugeridos. Dessa forma, o critério de parada fornece informações de como conduzir os testes, respeitando o orçamento esperado.
- **Duração estimada da fase de testes**: este atributo informa, caso exista, a duração estimada da fase de testes. Caso o projeto de software caracterizado possua um prazo específico para a fase de testes, é aconselhável que seja sugerido critérios de parada que recebam esta informação como parâmetro de entrada. Nesse caso, o critério de parada fornece informações de como executar os testes da melhor maneira respeitando a duração pré-estabelecida da fase de testes.
- **Perfil de utilização do software**: a finalidade comercial do software, aqui chamada de perfil de utilização, influencia na escolha do critério de parada. Isto porque os parâmetros utilizados para o cálculo do critério de parada podem mudar de acordo com o perfil de utilização. Por exemplo, alguns critérios de parada são propostos para softwares desenvolvidos para utilização própria, desenvolvidos para comercialização para um único cliente, desenvolvidos para comercialização em larga escala.
- **Plataforma de execução**: este atributo informa qual é a plataforma de execução do software em desenvolvimento. Alguns autores consideram que há uma maior dificuldade em testar software embarcado, pois nem sempre

se tem acesso ao hardware específico em que as falhas ocorrem (KIM e WU, 1994). Dessa forma, nem sempre é possível simular as falhas acontecidas para detectar e corrigir a falta que levou a falha. Portanto, é recomendado que projetos de software embarcados utilizem critérios de parada propostos especificamente para software embarcado.

- **Software de missão crítica**: softwares de missão crítica são caracterizados por causar grandes perdas quando falham. Nesse caso, os testes precisam ser conduzidos de maneira que façam com que o sistema atinja um a confiabilidade adequado e que se tenha um grande confiança no resultado encontrado. Para esse tipo de software existem critérios de parada específicos que garantem a confiabilidade do software com um nível de confiança adequado.
- **Objetivo da fase de testes do software**: este atributo informa qual é o objetivo da fase e testes do projeto. Portanto, é preciso escolher um critério de parada que tenha o mesmo objetivo que a fase de testes do software. Por exemplo, se na caracterização do software for definido que o objetivo da fase de testes é minimizar a confiabilidade, um critério de parada que tenha esse mesmo objetivo é mais recomendado para este projeto.
- **Nível(is) de testes desejado(s) para o projeto**: este atributo define quais níveis de testes serão contemplados pelo projeto. A observação desse atributo é importante, pois não é recomendado utilizar um critério de parada para testes do software que não contemple uma fase de testes especificada no projeto. Por exemplo, caso na caracterização do projeto fique definido que haverá a fase de testes de unidade, um critério de parada que contemple tal fase é recomendado.
- **Modelo comportamental/estrutural de software provido pelo projeto**: este atributo é importante para que critérios de parada que exijam determinados modelos não sejam sugeridos para software que não contemplem esses modelos. Por exemplo, CHANG e JENG (2007) utilizam Cadeia de Markov para modelar o perfil operacional e usam esta informação para definir um critério de parada. A utilização desse critério de parada seria recomendada para projetos que já previssem esse modelo no processo de desenvolvimento.
- **Habilidade provida pela equipe de testes alocada para o projeto**: este atributo define quais são as habilidades providas pela equipe de testes. Alguns critérios de parada exigem que a equipe de testes tenha conhecimento sobre alguma técnica ou tecnologia específica. Caso a equipe

que irá conduzir os testes já tenha conhecimento sobre as técnicas previstas como requisitos do critério de parada, esses critérios podem ser utilizados mais facilmente para o projeto em questão.

4.2.2 Pré-seleção de critérios de Parada

Como mencionado na seção 3.2.8, os critérios de parada podem ser divididos em dois grandes grupos: apriorísticos e não apriorísticos.

A partir do corpo de conhecimento sobre critérios de parada para testes de software e da caracterização do software, é possível definir se o mais adequado para o projeto de software é um critério de parada apriorístico ou não apriorístico. Esta decisão é realizada através da verificação de três atributos que caracterizam o software com os respectivos atributos que caracterizam o critério de parada.

A Tabela 4-1 demonstra o relacionamento entre esses atributos.

Tabela 4-1 – Atributos definidores do tipo de critério de parada

Atributo do projeto de software	Atributo do critério de parada
Confiabilidade desejada	Confiabilidade pré-determinada
Orçamento esperado para testes	Orçamento para testes pré-determinado
Duração estimada da fase de testes	Tempo para testes pré-determinado

Dessa forma, caso o atributo “Confiabilidade desejada” seja informado na caracterização do projeto, critérios de parada que possuem o atributo “confiabilidade pré-determinada” devem ser pré-selecionados. Da mesma maneira, caso o atributo “orçamento esperado para testes” seja informado na caracterização do projeto, critérios de parada que possuem o atributo “orçamento para testes pré-determinado” devem ser selecionados. Por último, caso na caracterização do software seja informado o atributo “duração estimada da fase de testes”, critérios de parada que possuem o atributo “tempo para testes pré-determinados” devem ser pré-selecionados. Entretanto, caso nenhum desses atributos sejam informados durante a caracterização do software, critérios de parada não apriorísticos devem ser pré-selecionados.

Assim, é possível afirmar que esses atributos definem se o critério de parada utilizado deve ser um critério apriorístico ou não apriorístico.

4.2.3 Cálculo do Grau de Adequação

Após a atividade de pré-seleção de critérios de parada, independentemente se um conjunto de critérios de parada apriorístico ou não apriorístico foi selecionado, a

próxima atividade a ser executada é ao cálculo do *Grau de Adequação* dos critérios de parada pré-selecionados com o projeto de software caracterizado.

O *Grau de Adequação* pode ser entendido como um indicador do quanto um critério de parada se adere ao projeto de software, ou seja, dado um projeto de software caracterizado e um critério de parada, o quão adequado é o critério de parada para ser aplicado ao projeto. Este indicador é calculado a partir da realização da comparação entre os atributos de caracterização do projeto de software com os atributos que caracterizam o critério de parada e é um valor numérico que pode variar de 0% a 100%.

Além disto, cada atributo que caracteriza o projeto pode receber um peso que indica a importância do atributo na atividade do cálculo de adequação. Portanto, caso seja necessário estabelecer que um atributo deva influenciar mais fortemente na escolha dos critérios de parada, o peso deste atributo deve ser aumentado. A Tabela 4-2 demonstra cada atributo de caracterização do software que é comparado com os atributos do critério de parada bem como os pesos relacionados a cada atributo. É possível observar que os pesos dos atributos estão com valores iguais. Isto significa que os atributos têm a mesma importância no processo de cálculo do *Grau de Adequação*. Também importante mencionar que a soma dos pesos dos atributos deve ser igual a 100%. Isto significa que, caso no cruzamento dos atributos do projeto de software com um critério de parada todos os atributos forem correspondentes, o *Grau de Adequação* será de 100%.

Tabela 4-2 – Atributos utilizados para cálculo do Grau de Adequação

	Atributo do projeto de software	**Atributo do critério de parada**	**Peso**
1	Perfil de utilização do software	Contexto de aplicação	14,2857%
2	Plataforma de execução	Caracterização do software	14,2857%
3	Software de missão crítica	Caracterização do software	14,2857%
4	Objetivo da fase de testes do software	Objetivo do critério	14,2857%
5	Nível(is) de testes desejado(s) para o projeto	Níveis de testes	14,2857%
6	Modelo(s) comportamental/estrutural de software provido pelo projeto	Modelo utilizado	14,2857%
7	Habilidade provida pela equipe de testes alocada para o projeto	Habilidades Necessárias	14,2857%

Definido os atributos que serão utilizados para calcular o *Grau de Adequação* e seus respectivos pesos, é utilizado o conceito matemático de distância euclidiana (BOLDRINI *et al.*, 1980), que também foi utilizado por XAVIER *et al.* (2002) para apoiar

a seleção de padrões arquiteturais em projetos de software e por DIAS NETO (2009) para apoiar a seleção de técnicas de testes baseada em modelo. O mecanismo utilizado explora a possibilidade de avaliação de distâncias conceituais através da comparação de elementos em um espaço vetorial multidimensional (XAVIER et. al, 2002). Matematicamente, a distância conceitual é realizada pela norma da diferença entre dois vetores *v1* e *v2*.

Este conceito foi traduzido para o contexto deste trabalho de forma que as características do projeto e de cada critério de parada são transformadas em valores e na sequência são representadas através de vetores. Quanto menor é a distância entre os vetores, mais adequado o critério de parada para o projeto de software.

- ***v1***: vetor que representa as características do projeto de software.
- ***v2***: vetor que representa as características dos critérios de parada.

Entretanto, antes de calcular a distância, é preciso realizar a normalização dos vetores. Isto porque o interesse é apenas na direção que cada vetor apresenta no espaço vetorial. Dessa forma, minimiza-se também a influência da dimensão dos vetores. Normalizando os vetores, obtém-se a equalização da importância das dimensões vetoriais e o enfoque passa a ser as direções que os vetores podem assumir (KONTIO, 1995, apud DIAS NETO, 2009).

Estes conceitos foram traduzidos para o contexto deste trabalho de forma que cada atributo de caracterização do projeto de software e cada atributo dos critérios de parada passem a ser representados como uma dimensão no espaço vetorial. Portanto, como são sete atributos (Tabela 4-2), o total de dimensões é sete.

$$V = R^7 = (X1 \mid X2 \mid X3 \mid X4 \mid X5 \mid X6 \mid X7)$$

Onde V é um vetor que representa um projeto de software ou critério de parada e X_i é a representação em número da característica do projeto ou critério de parada referente ao atributo de índice i, sendo os atributos de caracterização de índices 1 a 7 identificados na Tabela 4-2

Definido a representação vetorial, é preciso transformar as características do projeto de software e dos critérios de parada em valores numéricos para que o cálculo da distância vetorial possa ser calculado. Como proposto por DIAS NETO (2009), o vetor que representa a caracterização do software é composto pelos pesos de cada atributo que caracteriza o software. Já a representação do vetor referente à caracterização dos critérios de parada é calculada através da comparação entre as características do projeto e as características do critério de parada. Caso o valor do

atributo que caracteriza o projeto seja igual ao valor do atributo que caracteriza o critério de parada, o valor para esse atributo é 1 (indicando adequação) multiplicado por seu peso. Caso contrário, o valor é 0 (indicação não adequação). A regra a seguir foi reescrita com base em DIAS NETO (2009).

SE (atributo do projeto de software = atributo do critério de parada) ENTÃO
atributo do critério de parada é transformado em 1 x seu peso
CASO CONTRÁRIO
atributo do critério de parada recebe 0

Após definir a regra de transformação da caracterização do projeto e da caracterização dos critérios de parada em vetores, o passo seguinte é a formalização da representação vetorial para o projeto de software e critério de parada e o cálculo da distância entre os vetores. Este passo é realizado através da fórmula representada na Figura 4-2.

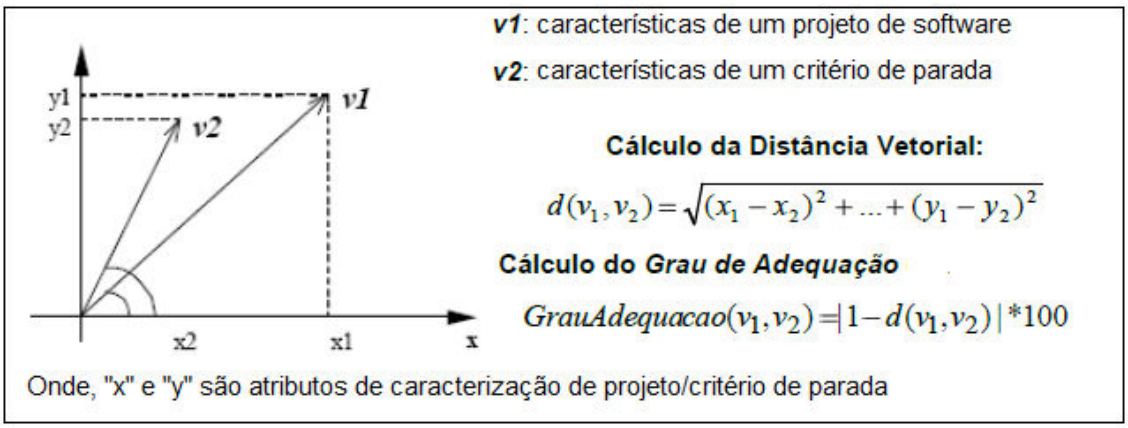


Figura 4-2 – Fórmula para cálculo da distância entre vetores (Adaptado de DIAS NETO, 2009)

4.2.3.1 Exemplo de Cálculo de Grau de Adequação

Nesta seção é apresentado um exemplo de cálculo de *Grau de Adequação* entre em um projeto fictício já caracterizado e dois critérios de parada. Todo o passo-a-passo referente à transformação do projeto e dos critérios de parada em vetores é demonstrado detalhadamente.

A Tabela 4-3 representa a caracterização do projeto de software e dos critérios de parada.

Tabela 4-3 – Exemplo de caracterização de projeto de software e critério de parada

Atributo projeto de software/critério de parada	Projeto de software	Critério de parada A	Critério de parada B
Confiabilidade desejada/Confiabilidade pré-determinada	1 falha/hora	Sim	Não
Orçamento esperado para testes/Orçamento para testes pré-determinado	Nenhum	Não	Não
Duração estimada da fase de testes/Tempo para testes pré-determinado	Nenhum	Não	Não
Perfil de utilização do software/Contexto de aplicação	Comercialização em larga escala	Comercialização em larga escala	Comercialização para único cliente
Plataforma de execução/ Caracterização do software	Embarcado	Embarcado	Embarcado
Software de missão crítica/Caracterização do software	Sim	Software de missão crítica	Software de missão crítica
Objetivo da fase de testes do software/ Objetivo do critério	Minimizar custos com testes	Minimizar custos com testes	Maximizar confiabilidade
Nível(is) de testes desejado(s) para o projeto/Níveis de testes	Testes de unidade	Testes funcionais	Testes de unidade
Modelo(s) comportamental/estrutural de software provido pelo projeto/Modelo utilizado	Nenhum	Nenhum	Cadeia de Markov
Habilidade provida pela equipe de testes alocada para o projeto/Habilidades Necessárias	Conhecimentos em UML	Não Informado	Conhecimentos em UML

Como mencionado na seção 4.2.2, depois da caracterização do projeto de software, a próxima atividade é a decisão pela utilização de critérios apriorísticos ou não apriorísticos. Com base na tabela anterior é possível perceber, através da caracterização do projeto de software, que é especificado a confiabilidade desejada para o software. Portanto, na atividade de pré-seleção, critérios de parada apriorísticos devem ser pré-selecionados. Assim, esta atividade descartaria o critério de parada B.

A próxima atividade consiste em calcular o *Grau de Adequação* do critério de parada ao projeto de software. Para realizar esta atividade, é preciso representar o

projeto de software e o critério de parada em valores vetoriais. Como descrito na seção 4.2.3 a representação do projeto de software em vetores é realizada representando cada atributo de caracterização como um elemento do vetor no qual o valor deste elemento é o peso do atributo (Tabela 4-2). Consequentemente, o vetor de que representa o projeto caracterizado é mostrado abaixo. O índice corresponde aos atributos de caracterização demonstrados na Tabela 4-2 foram utilizados.

	1	2	3	4	5	6	7
v1	0,142858	0,142858	0,142858	0,142858	0,142858	0,142858	0,142858

Depois de representar o projeto caracterizado como um vetor é preciso representar o critério de parada. Nesse caso, o procedimento para esta representação está descrito na seção 4.2.3. O vetor de caracterização do critério de parada é representado a seguir bem como a descrição do cálculo de cada valor.

	1	2	3	4	5	6	7
v2	0,142858	0,142858	0,142858	0,142858	0	0,142858	0

Como na comparação entre o projeto de software caracterizado e o critério de parada os atributos 1, 2, 3, 4 e 6 se corresponderam, o valor referente a cada um desses atributos recebeu 1 * 0,142858 = 0,142858. Por outro lado, os atributos 5 e 7 não corresponderam à comparação. Portanto, receberam o valor 0 * 0,142858 = 0.

O próximo passo é calcular a distância entre a representação vetorial do projeto (**v1**) e a representação vetorial do critério de parada (**v2**) bem como o *Grau de Adequação*.

Cálculo da distância e nível de adequação

$$d(Pr, CP.A) = \sqrt{([0])^2 + ([0])^2 + ([0])^2 + ([0])^2 + ([0{,}142858])^2 + ([0])^2 + ([0{,}142858])^2}$$

$$d(Pr, CP.A) = \sqrt{(0) + (0) + (0) + (0) + (0{,}020408) + (0) + (0{,}020408)}$$

$$d(Pr, CP.A) = 0{,}202031 \quad \textbf{(Distância)}$$

Logo, Grau de Adequação é: 1 – 0,202029 = 0,7979 ou **79,79% de adequação**

Através do procedimento descrito anteriormente, foi constatado que a distância entre os vetores **v1** e **v2** é de 0,202031 e que o Grau de Adequação é de 79,79%. Isto

significa que o critério de parada A é 79,79% adequado ao projeto de software se considerar os atributos escolhidos para a comparação.

4.2.4 Indicação e Seleção de Critérios de Parada

Para que o procedimento tenha utilidade na escolha do critério de parada mais adequado para determinado projeto de software é preciso calcular o *Grau de Adequação* de cada critério de parada.

Portanto, descrita a atividade para o cáculo do *Grau de Adequação* entre o projeto de software caracterizado e um critério de parada, o próximo passo do processo proposto nesta dissertação, é o *Calculo do Grau* de equação para todos os critérios de parada.

Depois disto, os critérios mais adequados podem ser disponibilizados à equipe de testes que por sua vez pode selecionar o critério que desejar.

Dessa forma, o procedimento se encerrada entregando à equipe de testes o critério de parada mais adequado para o projeto de software em desenvolvimento.

4.3 Conclusão

Este capítulo demonstrou o procedimento *Porantim CP* de apoio à seleção de critérios de parada para testes de software. Tal procedimento foi baseado em *Porantim*, um procedimento proposto por DIAS NETO (2009) com a finalidade de seleção de técnicas de testes baseados em modelos.

O procedimento foi dividido em cinco atividades diferentes. A primeira atividade consiste na caracterização do projeto através de atributos. Esses atributos foram levantados através de uma *quasi*-revisão sistemática da literatura descrita no Capítulo 3 e com a ajuda de um especialista da área de teste de software. Como os atributos que caracterizam os projetos podem ser cruzados com os atributos que caracterizam os critérios de parada contidos na base de conhecimento é possível realizar o cálculo do grau de adequação descrito neste capítulo. Os atributos de caracterização do projeto são: Confiabilidade desejada, Orçamento esperado para testes, Duração estimada da fase de teste, Perfil de utilização do software, Plataforma de execução, Software de missão crítica, Objetivo da fase de testes, Nível(is) de testes desejado(s), Modelo(s) comportamental/estrutural de software provido pelo projeto, Habilidade provida pela equipe de testes alocada para o projeto. A descrição detalhada de cada atributo pode ser encontrada na seção 4.2.1.

A segunda atividade do procedimento foi denominada pré-seleção de critérios de parada. Nesta atividade são selecionados do corpo de conhecimento critérios

apriorísticos ou não apriorísticos dependendo da caracterização do projeto realizada na atividade anterior.

Em seguida, foi descrito um passo-a-passo para calcular o *Grau de Adequação* dos critérios de parada selecionados na atividade de pré-seleção de critérios de parada. Este passo-a-passo se baseia no conceito de distância euclidiana. Para calcular esta distância, é preciso transformar a caracterização do projeto de software e os critérios de parada em vetores e posteriormente realizar o cálculo para determinar a distância entre os dois vetores. Como o *Grau de Adequação* pode ser calculado subtraindo a distância euclidiana de 1 (*1 – distância*). Um exemplo do cálculo da distância euclidiana e *Grau de Adequação* também foram demonstrados.

As duas últimas atividades são indicação dos critérios de parada mais adequados e seleção do critério de parada. Na primeira, os critérios de parada que tiveram mais aderência (maior *Grau de Adequação* ou menor distância) são sugeridos à equipe de testes. Na segunda, a equipe de testes pode selecionar o critério de parada dentre os apresentados que desejar.

O procedimento apresentado é importante, pois durante a *quasi*-revisão sistemática executada foi encontrada uma grande quantidade de critérios de parada propostos em momentos diferentes e em contextos diferentes. Assim, com o apoio do *Porantim CP* a equipe de testes pode selecionar o critério de parada mais adequado ao software que está sendo desenvolvido.

Entretanto, realizar estas atividades sem apoio computacional pode ser uma tarefa que exige bastante esforço. Por esse motivo, o procedimento *Porantim CP* foi implementado em uma ferramenta de gerência e acompanhamento de testes chamada Maraká. A implementação desse procedimento é descrita no Capítulo 5.

5 Apoio computacional para Seleção de Critérios de Parada para Testes de Software

Este capítulo apresenta a extensão de uma ferramenta chamada Maraká para gerenciamento e acompanhamento de testes de software no qual o procedimento para seleção de critérios de parada para testes de software (Porantim CP) foi implementado. Através da ferramenta Maraká, foi possível integrar o procedimento para apoio à seleção de critérios de parada para testes de software em um processo de teste bem definido.

5.1 Introdução

O procedimento *Porantim CP* possui alguns passos onde cálculos matemáticos devem ser realizados. Esses passos são passíveis de falhas quando executados por humanos e podem consumir grande esforço devido à quantidade de critérios de parada existentes na base de conhecimento e aos diferentes atributos de caracterização de projeto e dos critérios de parada. Baseado nessas observações foi constatado que seria importante que alguns passos do procedimento fossem executados com apoio computacional. Além disto, o apoio computacional faz com que o corpo de conhecimento seja armazenado de forma mais estruturada. Isto permite maior agilidade e segurança na consulta aos critérios de parada que integram o corpo de conhecimento.

O apoio computacional para o procedimento *Porantim CP* foi implementado como uma extensão da infraestrutura computacional chama Maraká apresentada em (DIAS NETO e TRAVASSOS, 2006) e (DIAS NETO, 2006). A infraestrutura foi expandida através da criação de novos componentes e funcionalidades que fornecem apoio à execução do procedimento de seleção de critérios de parada para testes de software.

5.2 Infraestrutura Maraká

Originalmente Maraká foi desenvolvida por DIAS NETO e TRAVASSOS (2006) e pode ser definida como uma infraestrutura computacional que apoia o planejamento e controle do processo de teste. Esta infraestrutura é desenvolvida utilizando como

base o framework de acesso livre Joomla!.(http://www.joomla.org). Esse framework utiliza PHP (http://www.php.net/) como linguagem de programação e MySQL (http://www.mysql.com/) como banco de dados.

5.2.1 Arquitetura de Maraká

Segundo DIAS NETO e TRAVASSOS (2006), Maraká é composto por quatro componentes principais. Esses componentes podem ser visualizados na Figura 5-1e são descritos a seguir.

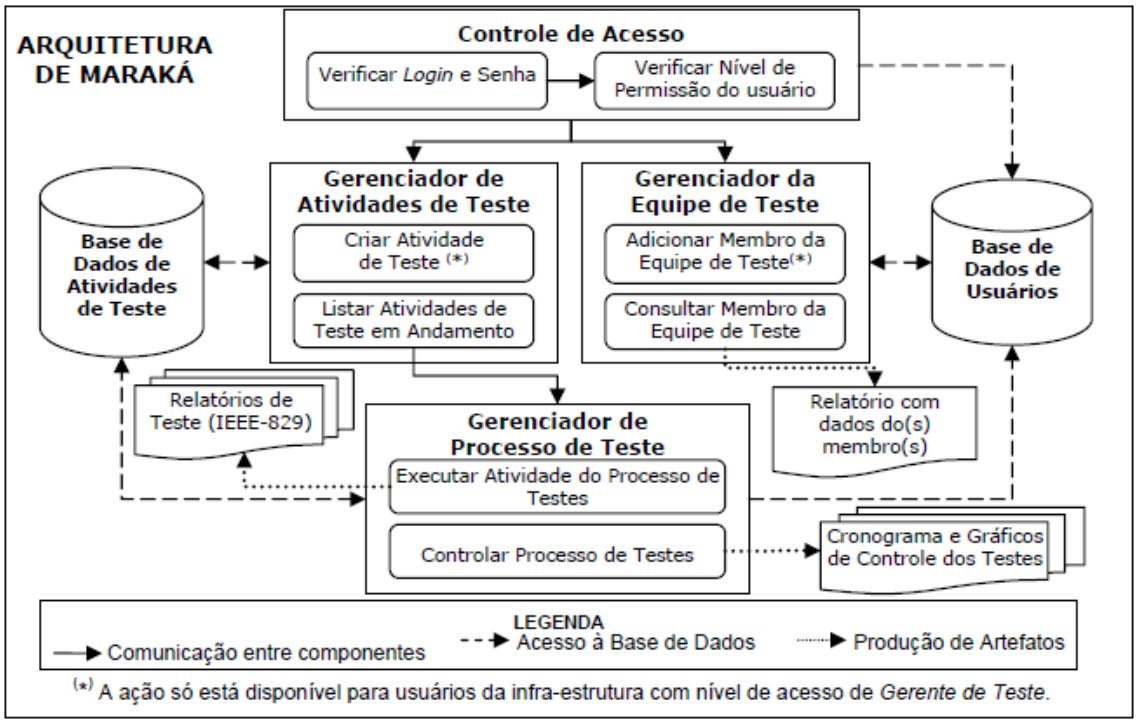


Figura 5-1 – Arquitetura Maraká

- **Controle de Acesso:** permite o acesso à infraestrutura Maraká e verificação do papel desempenhado pelo usuário para conceder o nível de permissão adequado. Após a validação de usuário e senha executados por esse componente é que o usuário tem acesso aos outros componentes
- **Gerenciador da equipe de Teste:** tem a finalidade de manter os usuários que possuem acesso à ferramenta, bem como o controle dos papeis de cada usuário e gerenciamento das permissões de acesso às funcionalidades de acordo com o papel desempenhado
- **Gerenciador de Atividades de Teste:** tem a função de criar e consultar atividades de teste na *Base de Dados de Atividades de Teste*. Através

desse componente é possível verificar quais atividades de testes ainda não foram concluídas e também realizar consultas em atividades de testes já existentes

- **Gerenciador de Processo de Teste:** gerencia o processo de teste a ser seguido pela atividade de testes cadastrada no componente anterior. Através desse componente é possível consultar artefatos do processo, o cronograma da atividade de teste, as informações sobre os membros da equipe de testes alocados para a atividade de teste e aos gráficos de acompanhamento dos testes. Dessa forma, esse componente documenta e sistematiza os testes

5.2.2 Características de Maraká

As características mais importantes da infraestrutura computacional Maraká segundo DIAS NETO (2009) são listadas a seguir.

- Execução e acompanhamento do processo de testes e suas atividades, como demostrado na Figura 5-2

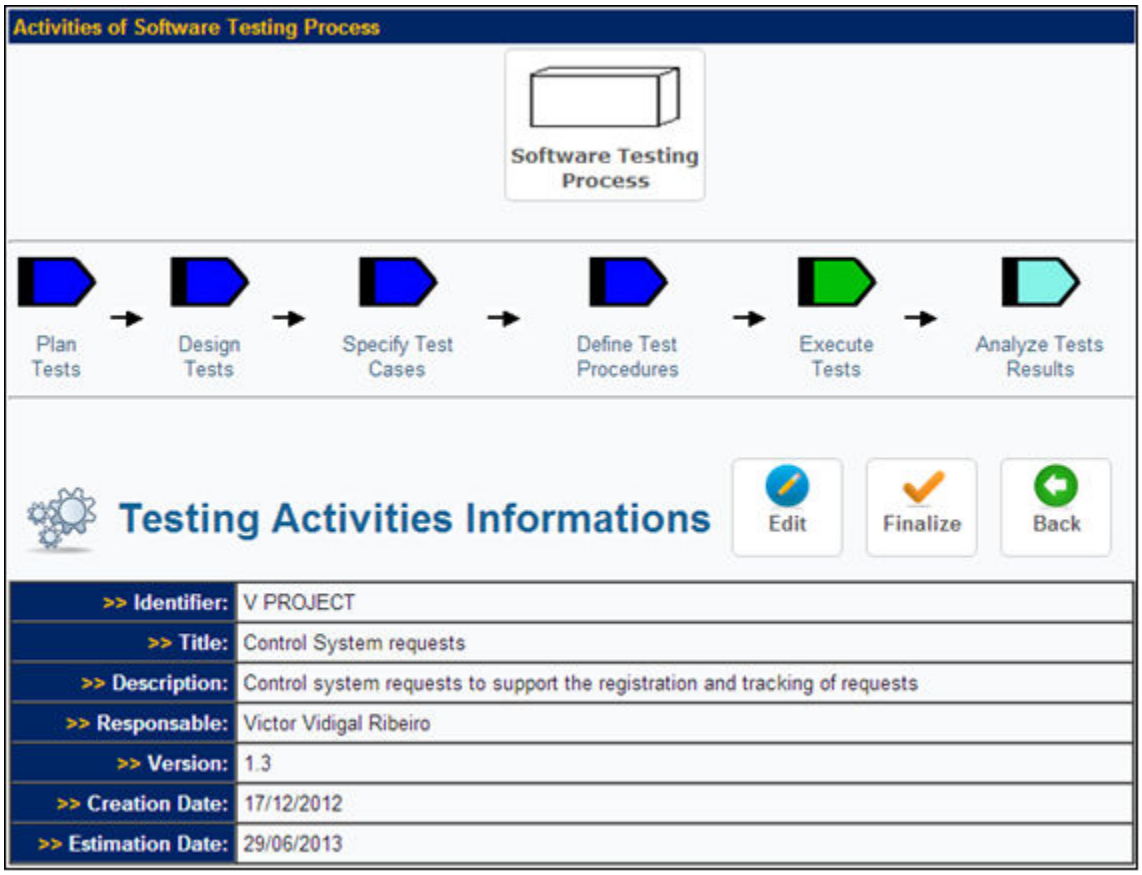


Figura 5-2 – Maraká – Execução e acompanhamento de testes e suas atividades

- Geração semiautomática de artefatos do processo de testes seguindo o padrão IEEE-829, como demonstrado na Figura 5-3

Plano de Testes

Gerado por Maraká - 01/02/2013

Projeto: V Project
Título: PLANO DE TESTE DE SISTEMA
Autor: Victor Vidigal Ribeiro
Data de Criação: 17/12/2012

Identificador do Plano: PLANO 01
Versão: 1.3
Status: Finalizado
Data de Conclusão 29/06/2013

1. Introdução

1.1. Objetivo:

O objetivo dos testes é avaliar as funcionalidades definidas para o software.

1.2. Escopo:

Somente testes funcionais serão realizados

1.3. Visão Geral:

O sistema de controle de solicitações é um dos módulos do sistema de gereciamento V Project. Este módulo tem a finalidade de controlar as solicitações geradas através sistema.

Figura 5-3 – Maraká – Extrato do plano de teste

- Controle do cronograma e resultado dos testes por meio de informações visuais e organizadas em tabelas, como demonstrado na Figura 5-4

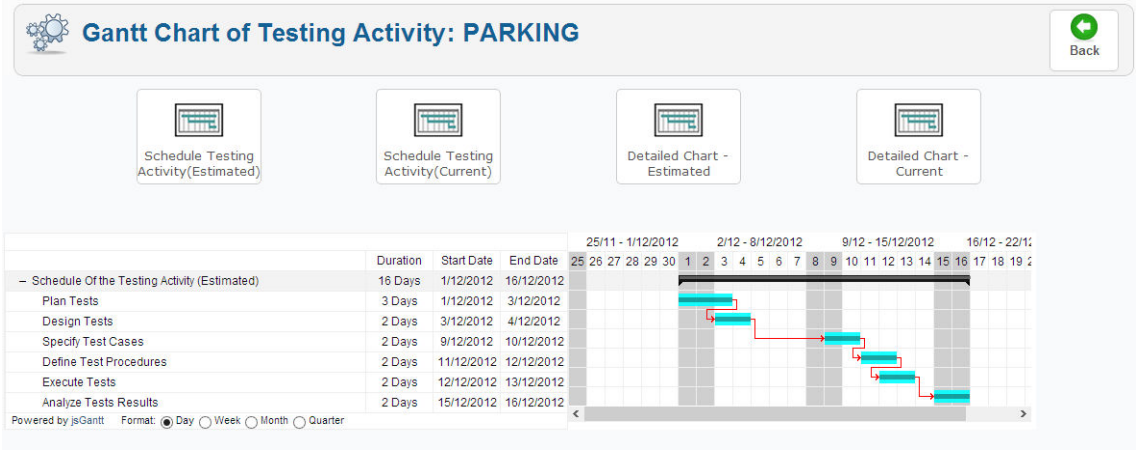


Figura 5-4 – Maraká – Acompanhamento visual do cronograma e resultados dos testes

- Gerenciamento dos papeis de “Gerente de testes”, “Projetista de testes” e “Testador”, bem como o controle de acesso às funcionalidades e projetos de acordo com o papel, como demonstrado na Figura 5-5

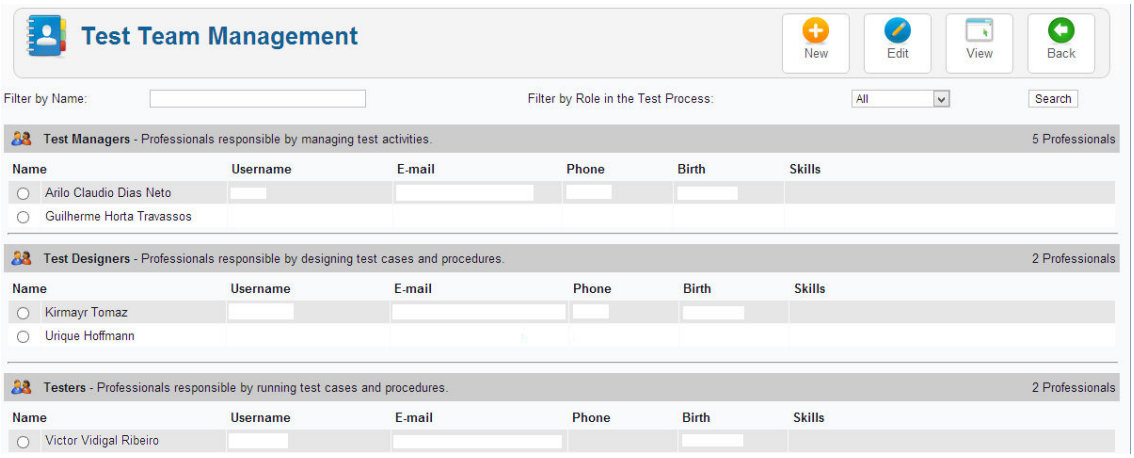


Figura 5-5 – Maraká – Gerenciamento dos papeis da esquipe de testes

Além dessas funcionalidades apontadas como principais, DIAS NETO e TRAVASSOS (2006) explicitam outras funcionalidades presentes em Maraká.

- Permitir criação de novas atividades de teste a serem gerenciadas
- Permitir a construção, visualização e impressão de artefatos do processo de testes de software
- Possibilita a visualização de incidentes em cada item de teste com a finalidade de itens com maior probabilidade de falhas
- Acesso realizado através da internet possibilitando acesso remoto e por múltiplos usuários

Outras funcionalidades também foram implementadas de forma que expandissem a infraestrutura Maraká. Uma dessas funcionalidades foi desenvolvida por SANTA ISABEL (2011) e tem por finalidade a inclusão do procedimento Porantim-WAT. Esse procedimento apoia a seleção de técnicas de testes para projetos de software web. O procedimento se baseia em um corpo de conhecimento onde as técnicas de testes web são armazenadas e caracterizadas através de atributos. Durante a execução do procedimento, o projeto de software para o qual se deseja escolher a técnica de teste é caracterizado através de atributos. Posteriormente, o grau de adequação entre o projeto de software e cada técnica de teste web contida no corpo de conhecimento é calculado. Dessa forma, a infraestrutura Maraká é capaz de sugerir à equipe de teste as técnicas de testes web mais adequadas para o projeto.

5.3 Evolução da Infraestrutura Maraká

Neste trabalho a infraestrutura Maraká foi evoluída para contemplar o procedimento de apoio à seleção de critérios de parada para testes de software. Como destacado na Figura 5-6, esta evolução foi realizada através da inclusão do *Corpo de Conhecimento* sobre critérios de parada e do componente *Porantim CP*. Este

componente foi incluído na ferramenta de forma que se integre ao processo de testes implementado por Maraká. Isto significa que o procedimento para seleção de critérios de parada não é isolado dos outros componentes. O *Porantim CP* é acessado durante a execução das atividades que compõem o processo de teste. Mais especificamente, o componente pode ser acessado através da atividade: *Planejar Testes / Riscos e Critérios de Teste*.

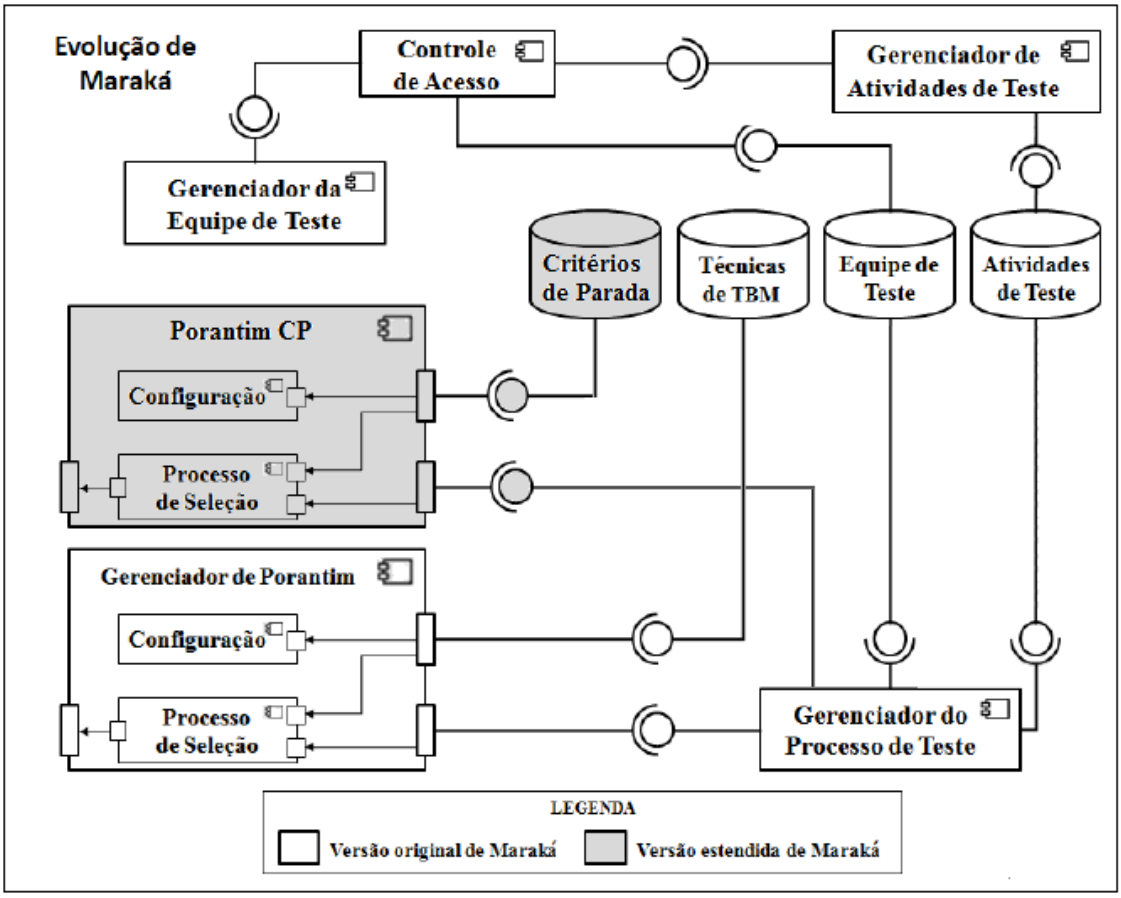


Figura 5-6 – Evolução da arquitetura de Maraká

O repositório *Critérios de Parada* implementa o corpo de conhecimento extraído a partir da *quasi*-revisão sistemática descrita no capítulo 3. De forma semelhante, o componente *Porantim CP* implementa o procedimento de apoio à seleção de critérios de parada descrito no capítulo 4. As funcionalidades deste componente são descritas a seguir.

- **Configuração do corpo de conhecimento**: permite que os usuários com perfil de *Administrador* incluam, excluam ou alterem os critérios de parada contidos no corpo de conhecimento
- **Configuração de parâmetros *Porantim CP***: possibilita aos usuários com perfil de *Administrador* definir quais os pesos dos atributos que são

utilizados para calcular o *Grau de Adequação* entre o projeto de software e os critérios de parada. Além disto, através desta funcionalidade também é possível determinar quantos critérios de parada serão exibidos para que possam ser selecionados durante o processo *Porantim CP*

- **Execução do procedimento *Porantim CP***: permite aos usuários com perfil de *Gerente de Testes* executarem o procedimento para selecionar o critério de parada mais adequado ao projeto

A versão de Maraká que implementa o procedimento de apoio à seleção de critérios de parada para testes de software pode ser acessada através da URL: http://lens-ese.cos.ufrj.br/maraka.

5.4 Funcionalidades da Maraká Referentes ao Porantim CP

As funcionalidades referentes ao procedimento *Porantim CP* e mencionados na seção anterior são apresentadas nesta seção.

5.4.1 Configuração do Corpo de Conhecimento

O corpo de conhecimento implementado no *Porantim CP* disponibiliza os critérios de parada para testes de software. Através desta funcionalidade é possível gerir os critérios de parada e seus atributos realizando operações de inclusão, alteração e exclusão dos critérios de parada. Esta funcionalidade pode ser acessada através da opção *Configuração / Critério de Parada*.

Após acessar esta opção, uma tela com a listagem dos critérios de parada e seus principais atributos é exibida como pode ser visualizado na Figura 5-7. Nesta tela, também é possível baixar o artigo de onde foi retirado o critério de parada e o formulário de extração com todos os dados retirados da *quasi*-revisão sistemática.

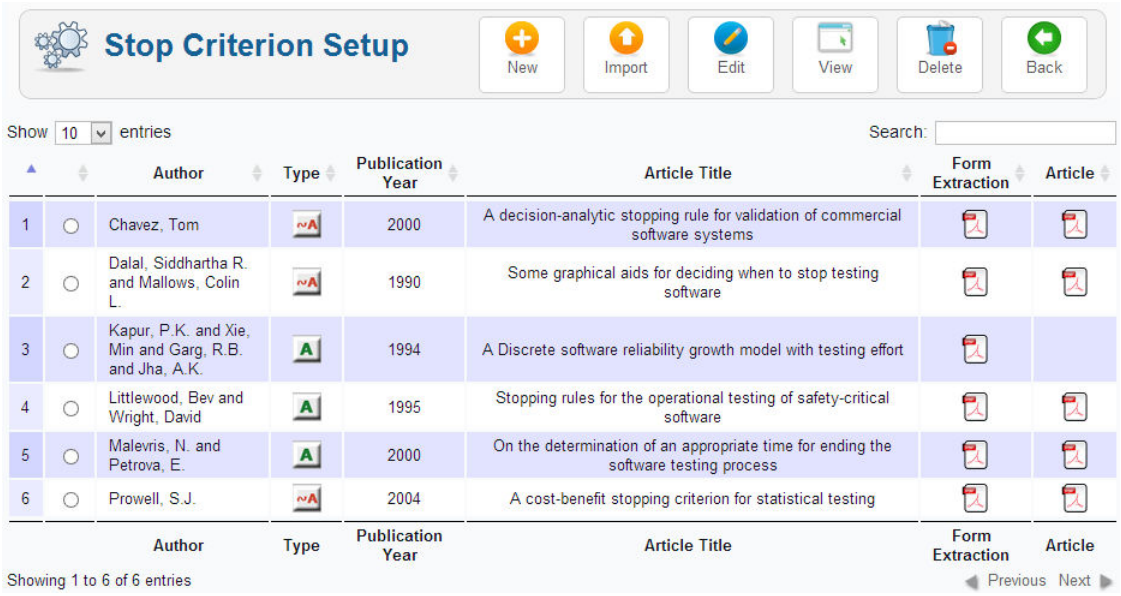


Figura 5-7 – Maraká – Tela de listagem de critérios de parada

A partir da tela anterior é possível acessar as funcionalidades referentes à manutenção dos critérios de parada através dos ícones situados na parte superior direita. Através destas funcionalidades é possível incluir, alterar e visualizar informações detalhadas sobre algum critério de parada específico.

A Figura 5-8 demonstra parte desta tela. É importante comentar que nem todos os atributos contidos nesta tela são utilizados para calcular o *Grau de Adequação* entre o critério de parada e o projeto de software. Alguns dos atributos podem ser utilizados pela equipe de testes como uma forma de decisão de qual critério de parada escolher caso o *Grau de Adequação* seja igual para critérios diferentes. Ou seja, como uma forma de desempate.

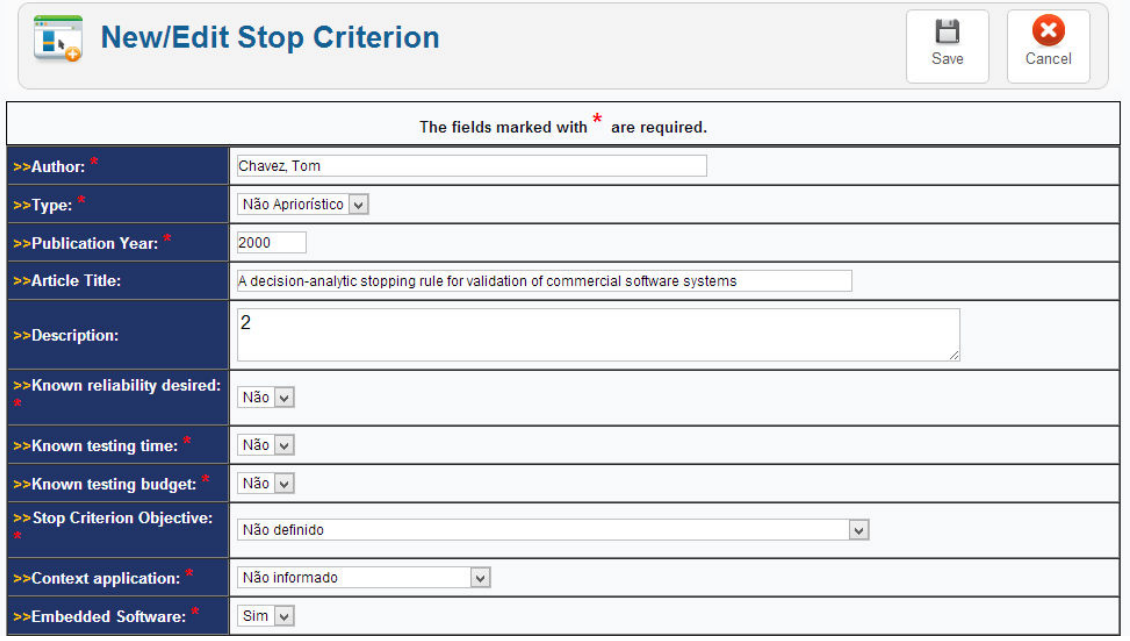


Figura 5-8 – Maraká – Tela de detalhamento dos critérios de parada

5.4.2 Configuração dos parâmetros Porantim CP

Como descrito no capítulo 4, o cálculo do *Grau de Adequação* entre o projeto de software e os critérios de parada é realizado através de atributos que os caracterizam. Além disto, cada um desses atributos pode possuir um peso diferente, apesar de no contexto deste trabalho todos os pesos estarem iguais. O peso dos atributos pode ser configurado pela equipe de testes de acordo com a suas necessidades. A configuração desse peso que define o quanto um atributo é importante no cálculo do *Grau de Adequação* é configurado através da opção *Configurações / Configurar Critérios de Parada*.

Através desta opção também é possível configurar a quantidade de critérios de parada que serão exibidos para a equipe de testes durante o procedimento de seleção de critérios de parada. Isto evita o excesso de informação no momento da decisão de qual critério de parada utilizar. A Figura 5-9 demostra a tela de configuração dos pesos dos atributos.

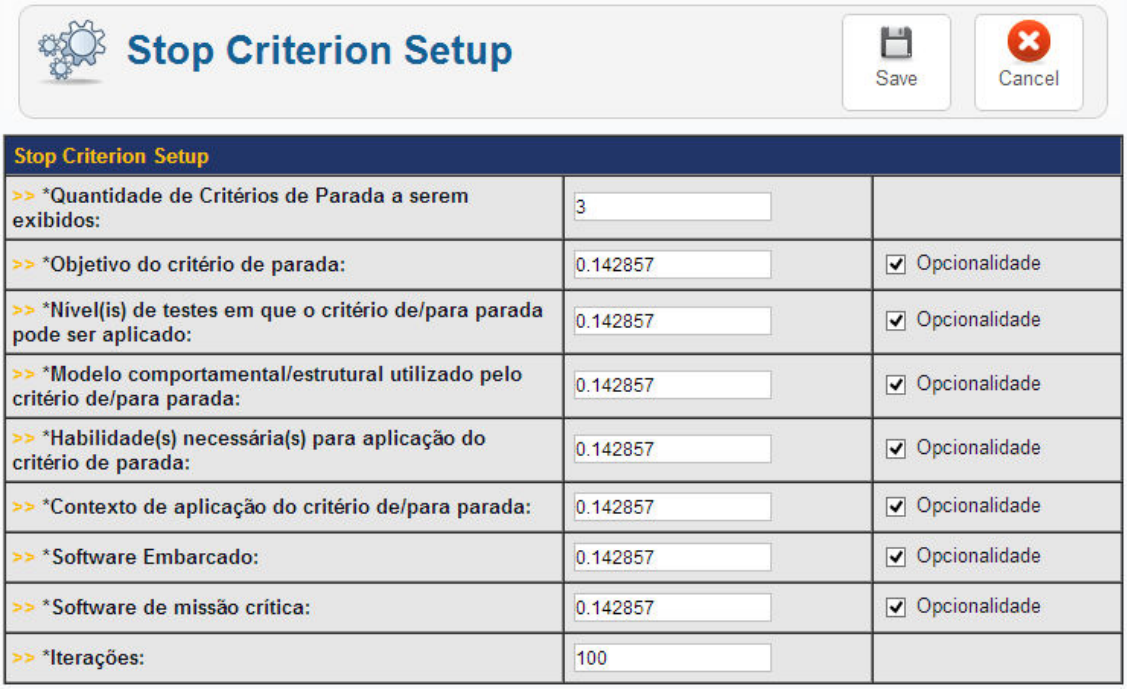


Figura 5-9 – Maraká – Tela de configuração de pesos dos atributos

5.4.3 Apoio ao Procedimento de Seleção de Critérios de Parada

O procedimento *Porantim CP* foi implementado em Maraká de modo que se integrasse ao processo de testes proposto pela infraestrutura. Assim, esse procedimento pode ser acessado através da subatividade *Riscos e Critérios de Testes* que por sua vez compõe a atividade *Planejar Testes*.

Na versão original de Maraká esta subatividade possuía apenas a funcionalidade de incluir riscos e critérios manualmente como um cadastro comum. Já a versão que contempla o procedimento *Porantim CP*, os critérios de parada podem ser selecionados através da execução do procedimento *Porantim CP* e, ao final, são vinculados ao projeto de software.

O procedimento para seleção de critérios de parada pode ser executado a partir da tela inicial acessando a funcionalidade *Testes em Andamento / <<Selecionar Projeto>> / Planejar Testes / Riscos e Critérios de Testes / Porantim CP*. A Figura 5-10 mostra o ícone *Porantim CP* a partir do qual o procedimento é disparado.

Figura 5-10 – Maraká – Ícone para execução do *Porantim CP*

Ao iniciar o procedimento para seleção de critérios de parada para testes de software, a primeira tela exibida é dividida em dois passos.

- **Passo 1**: permite ao usuário caracterizar o projeto de software. É importante compreender que os atributos que compreendem a caracterização do software já podem ter sido preenchidos por outras atividades do processo de software. Entretanto, como mostrado na Figura 5-11, é possível visualizar e alterar esses atributos. Dessa forma, a ferramenta fornece à equipe de testes um resumo sobre os atributos que são importantes no cálculo do *Grau de Adequação*. Assim, é possível que esses atributos sejam alterados caso tenham sido definidos erroneamente ou caso queiram verificar qual seria o resultado da execução do procedimento se algum atributo fosse alterado. Depois de terminar a caracterização do software, o botão *Próximo Passo* pode ser acionado. Com isto, o *Porantim CP* pré-seleciona os critérios de parada apriorísticos ou não apriorísticos de acordo com a caracterização do software. Depois disto, é calculado o *Grau de Adequação* para cada critério de parada e os critérios de parada que mais se adaptam ao software são exibidos no *Passo 2*.

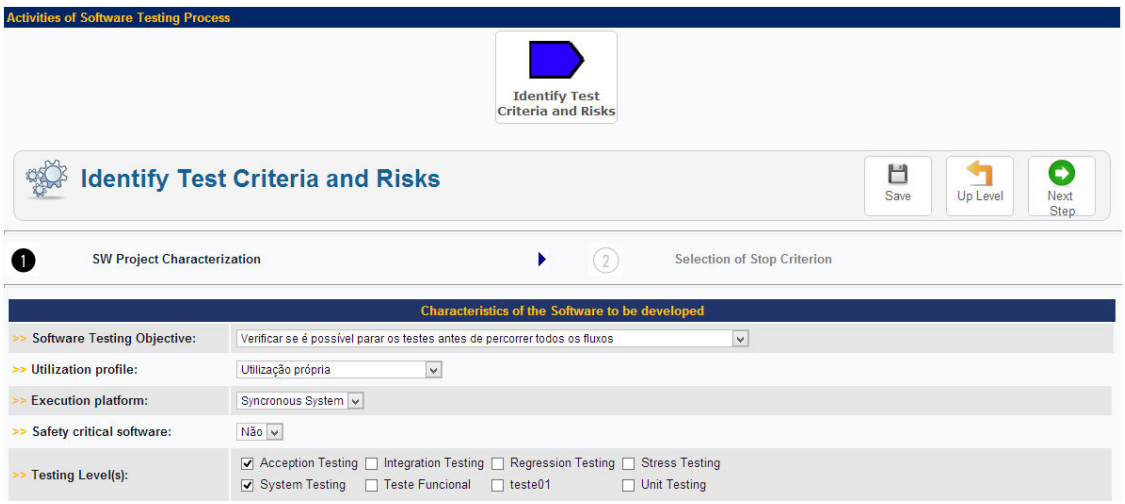


Figura 5-11 – Maraká – Tela de caracterização do projeto de software

- **Passo 2**: são exibidos os critérios de parada que mais se adequam ao projeto caracterizado (Figura 5-12) bem como o valor do *Grau de Adequação*. Como mostrado na seção 5.4.2, a quantidade de critérios de parada exibida pode ser configurada no *Porantim CP*. Neste momento, a equipe de teste pode selecionar o critério de parada que mais se adequa ao projeto. Ao selecionar a opção Salvar, o critério de parada é vinculado ao

projeto de software. Assim, a equipe de testes passa saber qual critério de parada deve ser utilizado e o procedimento é finalizado.

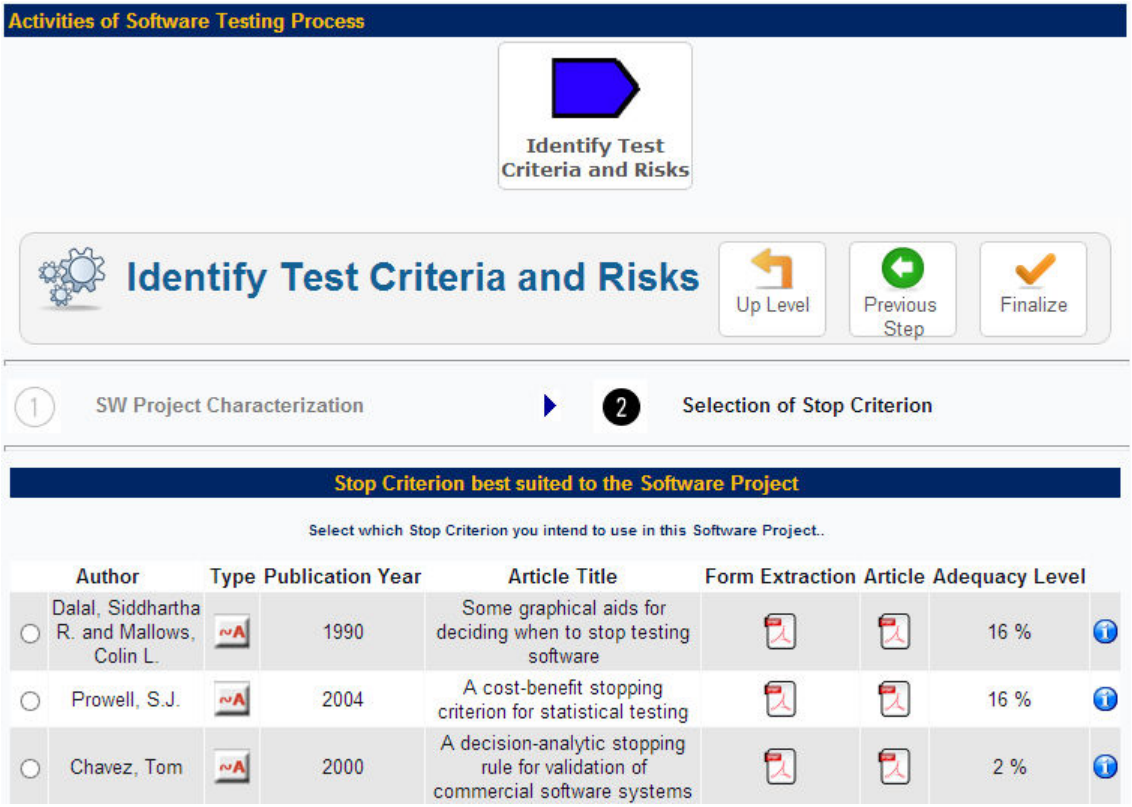
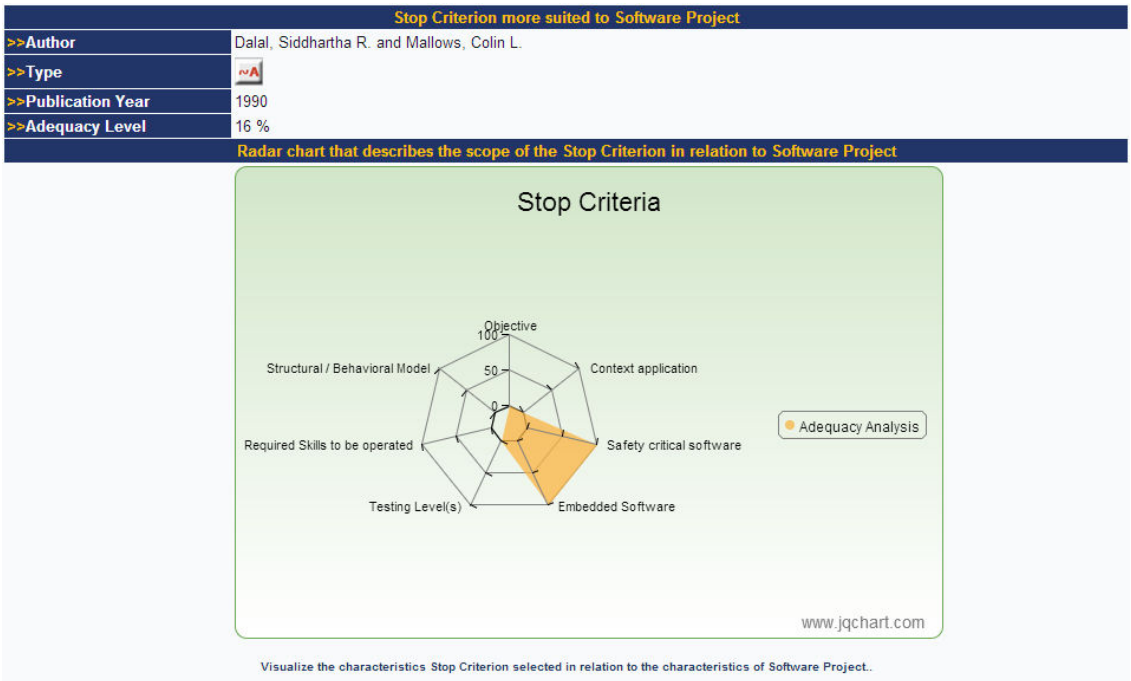


Figura 5-12 – Maraká – Tela de seleção de critérios de parada

Na tela exibida no *Passo 2* existem duas funcionalidades que merecem destaque. A primeira funcionalidade permite que a equipe de testes visualize (PDF) o artigo de onde o critério de parada foi retirado e o formulário de extração (PDF) com todos os dados extraídos na *quasi*-revisão sistemática descrita no capítulo 3. Com o acesso a este formulário é possível ter informações mais detalhadas sobre o critério de parada selecionado. Além disto, é possível verificar também quais atributos levaram ao *Grau de Adequação* exibido, ou seja, quais atributos da caracterização do projeto são compatíveis com os atributos de caracterização do critério de parada. Para visualizar esta funcionalidade o ícone de "informação" deve ser acionado. Assim, uma tela é exibida onde detalhes sobre a comparação entre o projeto de software e o critério de parada podem ser visualizados.

A Figura 5-13 demonstra a tela de comparação do projeto de software com o critério de parada. Na parte superior da janela as informações relativas à identificação do critério são exibidas. A segunda parte da tela consiste em um gráfico composto por cada um dos atributos utilizados para o cálculo do *Grau de Adequação*. Por fim, são apresentados os valores dos atributos tanto para o projeto quanto para o critério de

parada. Com base no gráfico ou na tabela exibida ao final desta tela, é possível verificar quais são os atributos que influenciaram para o *Grau de Adequação* calculado. Dessa forma, caso haja vários critérios de parada com alto *Grau de Adequação* a equipe de testes pode recorrer a esses recursos para sanar as dúvidas referentes à escolha do critério de parada.



Attributes Characterization	Software Project	Stop Criterion	Adequacy Level
>>Objective	Verificar se é possível parar os testes antes de percorrer todos os fluxos	Equilíbrio entre o custo com testes e custo de correção de falhas com o software em produção	0%
>>Utilization Context	Utilização própria	Comercialização para cliente único	0%
>>Safety critical software	Não	Não	100%
>>Embedded	Não	Não	100%
>>Testing Level(s)	Acception Testing System Testing		0%
>>Required Skills			0%
>>Structural / Behavioral Model			0%

Figura 5-13 – Maraká – Tela de comparação projeto de software x critério de parada

5.5 Conclusão

Este capítulo apresentou a construção *Porantim CP* para seleção de critérios de parada para testes de software. O apoio computacional foi considerado importante, pois algumas atividades do procedimento exigem cálculos matemáticos passíveis de erro quando executado por humanos. Além disto, devido ao grande número de critérios de parada esses cálculos podem demandar muito esforço. O *Porantim CP* foi implementado como um componente da infraestrutura computacional de apoio e planejamento de testes: a Maraká.

O *Porantim CP* permite que o corpo de conhecimento sobre critérios de parada para testes de software seja organizado de forma mais estruturada facilitando as operações de inclusão, alteração, exclusão bem como a consulta de critérios de parada.

Além disto, foi implementado todo o procedimento descrito no capítulo 4 que apoia a seleção de critérios de parada. Esse procedimento foi implementado como parte integrante do processo de teste e incluído na subatividade *Riscos e Critérios de Testes* da atividade *Planejar Testes*. O procedimento foi dividido em dois passos:

- **Passo 1, caracterização do projeto**: onde o projeto é caracterizado e em seguida são selecionados critérios de parada apriorísticos ou não apriorísticos de acordo com a caracterização do projeto. O cálculo do *Grau de Adequação* é realizado em seguida para todos os critérios de parada pré-selecionados
- **Passo 2, seleção do critério de parada**: é exibida para a equipe de testes um conjunto de critérios de parada que mais se adequaram ao projeto de software. É possível visualizar informações referentes à comparação dos atributos do projeto de software com os critérios de parada através de uma tabela e de um gráfico de radar. Por fim, a equipe de testes deve escolher o critério de parada

Outra funcionalidade da ferramenta é possibilidade de acesso a informações mais detalhadas sobre os critérios de parada. Por exemplo, a ferramenta pode exibir diversos atributos que caracterizam o critério de parada bem como o artigo de onde o critério de parada foi retirado.

Dessa forma, todo o procedimento referente à seleção de critérios de parada para testes de software é disponibilizado através do componente *Porantim CP* implementado na infraestrutura Maraká.

6 Prova de Conceito

Este capítulo apresenta a descrição de uma prova de conceito sobre a aplicação do procedimento Porantim CP implementado como um componente na ferramenta Maraká em seis casos de projetos de software criados em contextos diferentes. O primeiro caso demonstra a aplicação do procedimento em um projeto fictício. Em seguida, são utilizados quatro projetos obtidos de um estudo experimental no qual diferentes cenários de software são considerados. Por fim, é apresentada a aplicação do procedimento em um projeto de software real desenvolvido para gestão de informações de uma fundação que atua no apoio a projetos de ciência e tecnologia.

6.1 Introdução

Neste capítulo uma prova de conceito é executada com a finalidade de observar a viabilidade de utilização do componente que implementa o procedimento *Porantim CP* na infraestrutura Maraká. Para fazer uso do componente em diferentes contextos, seis projetos de software distintos são utilizados.

O primeiro projeto, usado como exemplo inicial, trata-se de um caso fictício (Projeto F) onde um software com características definidas pelo autor desse trabalho é utilizado para observação.

Em seguida, quatro casos de observação adicionais foram obtidos do estudo experimental conduzido por Vegas e Basili (2005). Nesse estudo softwares desenvolvidos para diferentes cenários são caracterizados. Segundo os autores, o estudo não cobre todas as possíveis categorias de projetos de software, entretanto há uma tentativa de representar diferentes cenários de desenvolvimento de software.

Por fim, o último caso de observação apresentado descreve a aplicação do procedimento *Porantim CP* em um projeto de software real construído para o gerenciamento das atividades de uma fundação de fomento à pesquisa e tecnologia.

O objetivo da prova de conceito é realizar uma avaliação preliminar da possibilidade de aplicação do procedimento para escolha de critério de parada bem como avaliação do funcionamento do componente desenvolvido. Como não foi definido nenhum critério de parada para os cinco projetos utilizados, não foi possível

comparar se o critério de parada sugerido por *Porantim CP* traz alguma vantagem em relação ao critério originalmente utilizado. Portanto, se procura avaliar questões como a correção do cálculo do grau de adequação, problemas de usabilidade que impeçam a execução de alguma etapa do processo, como por exemplo, impossibilidade de escolha de algum valor para um atributo de caracterização do projeto. Além disso, procura-se verificar se pelo menos um critério de parada é sugerido para cada um dos projetos de software e se esta sugestão de critério de parada é minimamente coerente com o software para o qual ele foi proposto, de acordo com a perspectiva do pesquisador.

Cada um dos casos de observação da prova de conceito seguiu os passos descritos abaixo. Cabe ressaltar que para evitar repetição, os passos 3 e 4 são omitidos a partir da descrição do segundo caso de observação da prova de conceito.

7. Descrição do projeto de software;
8. Definição dos valores dos atributos relativos à caracterização do software;
9. Inclusão do projeto na infraestrutura Maraká;
10. Preenchimento dos atributos de caracterização na infraestrutura Maraká;
11. Execução do procedimento *Porantim* CP;
12. Seleção do critério de parada;
13. Análise dos resultados.

6.2 Exemplo de Aplicação – Projeto F

Esta seção descreve a aplicação do procedimento *Porantim CP* em um projeto de software fictício denominado Projeto F. A descrição do projeto é apresentada e em seguida os atributos de caracterização são extraídos dessa descrição. Posteriormente, as informações relacionadas à caracterização do projeto são inseridas no componente *Porantim CP* e o procedimento para sugestão de critério de parada é executado. Por fim, é realizada uma análise dos resultados obtidos.

6.2.1 Descrição do Projeto F

O Projeto F é um projeto fictício que tem por objetivo o desenvolvimento de uma aplicação Web que exibe imagens de um conjunto de câmeras. O sistema possui apenas as opções: “Incluir Câmera” (inclui uma nova câmera no sistema), “Câmera Anterior” (exibe as imagens da câmera anterior a que estava sendo exibida), “Próxima Câmera” (exibe as imagens da câmera posterior a que estava sendo exibida) e “Descartar Câmera” (retira a câmera da sequência de exibição). Esse software não é considerado de missão crítica, pois apenas exibe imagens de câmeras. Além disso, o

software está sendo desenvolvido com a linguagem de programação PHP e será comercializado para diversos fabricantes de câmeras.

A equipe de desenvolvimento e de testes é considerada experiente nesse tipo de sistemas e sempre desenvolve um artefato que representa o perfil de utilização do software através de uma Cadeia de Markov.

Em relação aos testes, espera-se que o software tenha maior confiabilidade após sua execução, ou seja, o objetivo dos testes é maximizar a confiabilidade. Assim, foi utilizado teste funcional e de unidade e nenhuma restrição de orçamento, prazo ou confiabilidade foi definida previamente.

6.2.2 Caracterização do Projeto F

Com base na descrição do sistema apresentada anteriormente, é possível caracterizar o Projeto F utilizando os atributos de caracterização de projetos definidos no Capítulo 4. Assim, a Tabela 6-1 apresenta cada atributo de caracterização bem como seu valor referente ao Projeto F.

Tabela 6-1 – Caracterização do Projeto F

Atributo do projeto de software	Projeto de software
Confiabilidade desejada	Nenhum
Orçamento esperado para testes	Nenhum
Duração estimada da fase de testes	Nenhum
Perfil de utilização do software	Comercialização em larga escala
Plataforma de execução	Web
Software de missão crítica	Não
Objetivo da fase de testes do software	Maximizar confiabilidade
Nível(is) de testes desejado(s) para o projeto	Teste Funcional Teste de Unidade
Modelo(s) comportamental	Cadeia de Markov
Habilidade provida pela equipe de testes alocada para o projeto	Criação de cadeias de Markov com o perfil operacional do software

6.2.3 Inclusão do Projeto F em Maraká

Para realizar a inclusão de um projeto na infraestrutura Maraká, o usuário deve estar autenticado e selecionar a opção “Testing Activities / New Testing Activity”. Após isso, os dados básicos do projeto devem ser preenchidos como demonstrado na Figura 6-1. A partir desse momento, as atividades de testes, incluindo a escolha de critérios de parada, podem ser executadas.

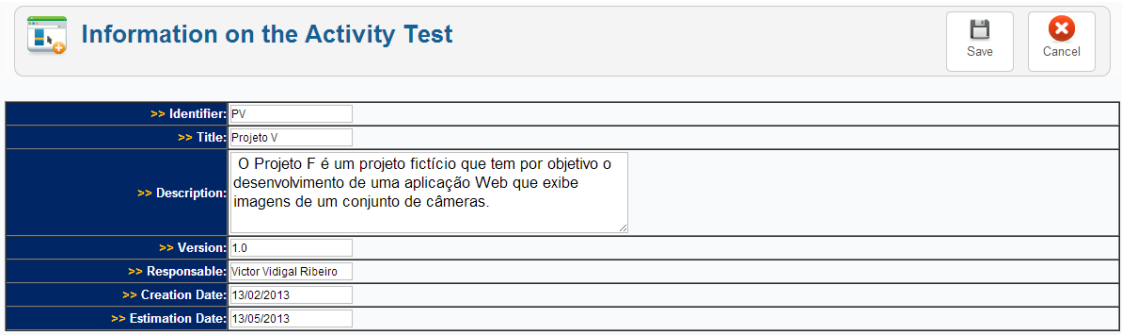


Figura 6-1 – Inclusão do Projeto F em Maraká

6.2.4 Preenchimento dos atributos de caracterização – Projeto F

A partir da definição dos valores de cada atributo referente à classificação do Projeto F é possível fornece-los à infraestrutura Maraká para que o procedimento de sugestão de critérios de parada seja executado. Para isso, o Projeto F deve ser selecionado em Maraká e a opção "*Software Testing Process*" deve ser acessada. Após acessar essa opção, as atividades do processo de teste são exibidas e, como descrito no Capítulo 5, o procedimento para sugestão de critério de parada é definido na subatividade "*Identify Test Criteria and Risks"* que compõe a atividade "*Plan Tests*", portanto, essa subatividade deve ser selecionada. Nesse ponto, a opção "*Porantim CP*" deve ser selecionada para que o procedimento seja iniciado.

Com a realização do procedimento descrito anteriormente, Maraká exibe a tela de caracterização do projeto. Parte dessa tela é apresentada na Figura 6-2 na qual o projeto é caracterizado através dos atributos de caracterização.

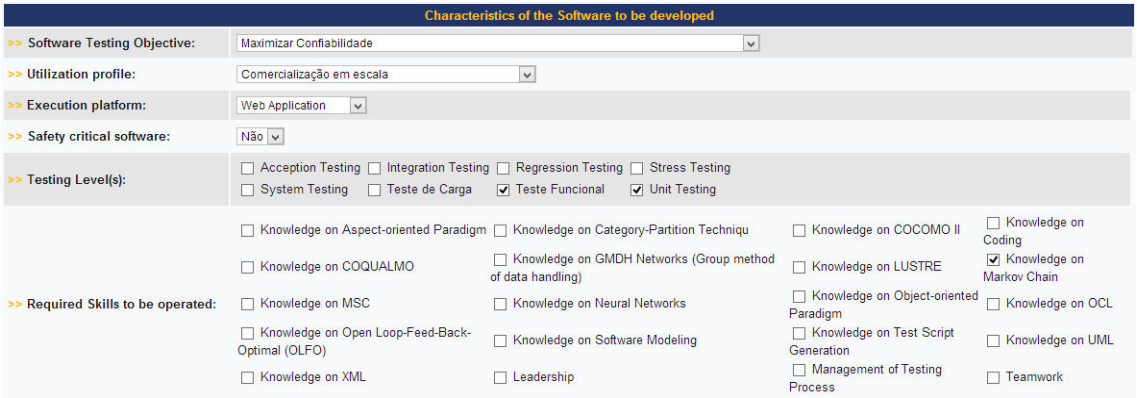


Figura 6-2 – Caracterização Projeto F

6.2.5 Execução do *Porantim CP* – Projeto F

Depois da caracterização do projeto, a opção "*Next Step*" pode ser selecionada. Com isso, o componente *Porantim CP* calcula o grau de adequação do projeto em relação a cada um dos critérios de parada cadastrados na base de

conhecimento de acordo com a fórmula exibida no Capítulo 4. Após o cálculo, os três (quantidade definida como um parâmetro configurável do componente *Porantim CP*) critérios de parada que possuem maior grau de adequação com o projeto Projeto F são disponibilizados para escolha pelo engenheiro de software como pode ser visualizado na Figura 6-3.

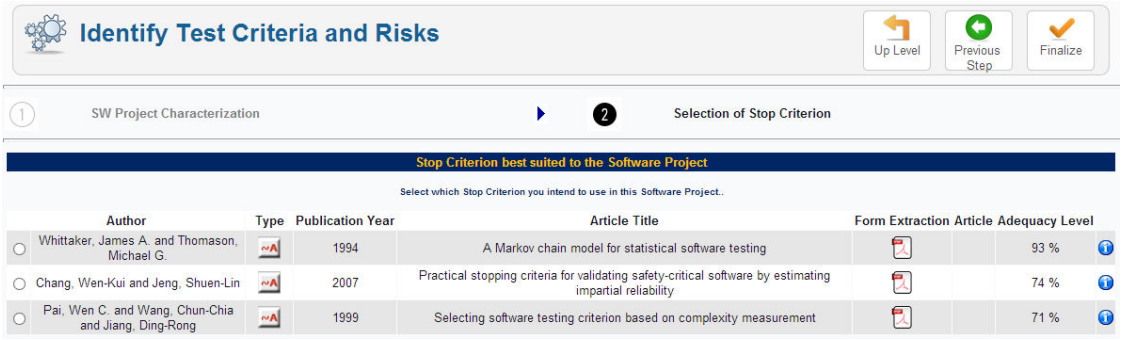


Figura 6-3 – Critérios de parada sugeridos para Projeto F

Nessa tela, é possível visualizar as informações dos critérios de parada sugeridos como autor, tipo do critério, ano de publicação, título do artigo no qual o critério foi proposto. Além disso, é possível obter o formulário com os dados extraídos a partir da *quasi*-revisão sistemática, obter o artigo onde o critério foi publicado, visualizar o grau de adequação do projeto e visualizar informações mais detalhadas sobre a adequação do critério ao projeto através da opção (*View Details of Stop Criterion*).

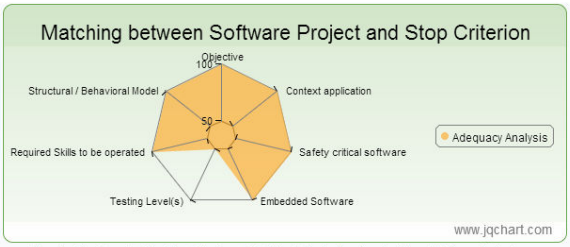
Como pode ser visto na Figura 6-17, o critério de parada mais adequado é proposto por Whittaker e Thomason (1994) e possui adequação de 93% com o Projeto F.

6.2.6 Seleção do critério de parada – Projeto F

Para verificação dos atributos que levaram o critério de parada ser mais adequado ao projeto a opção (*View Details of Stop Criterion*) pode ser selecionada. Com isso, uma tela semelhante à Figura 6-4 é exibida. Nesta tela é possível visualizar um gráfico radar e uma tabela que mostram quais atributos são equivalentes entre o projeto de software e o critério de parada. Portanto, pode-se perceber que em relação aos atributos “Objetivo”, “Contexto de Aplicação”, “Software de missão Crítica”, “Software Embarcado” e “Habilidades Necessárias” o critério de parada está 100% aderente ao projeto. Ou seja, o critério de parada foi proposto exatamente para projetos em que a atividade de teste possui o objetivo de maximizar a confiabilidade, softwares que serão comercializados em larga escala, software que não são de

segurança crítica e nem embarcados e, para utilização do critério de parada é necessário o conhecimento sobre Cadeias de Markov.

Apesar dos atributos mencionados anteriormente serem equivalentes entre projeto e critério de parada, o atributo “Níveis de teste” atinge apenas 50% de adequação. Isso acontece, pois o projeto de software está planejado para possuir dois níveis de testes (Funcional e Unidade) e o critério de parada foi proposto apenas para contemplar o nível de teste Funcional.



Visualize the characteristics Stop Criterion selected in relation to the characteristics of Software Project..

Attributes Characterization	Software Project	Stop Criterion	Adequacy Level
>>Objective	Maximizar Confiabilidade	Maximizar Confiabilidade	100%
>>Utilization Context	Comercialização em escala	Comercialização em escala	100%
>>Safety critical software	Não	Não	100%
>>Embedded	Não	Não	100%
>>Testing Level(s)	Teste Funcional Unit Testing	Teste Funcional	50%
>>Required Skills	Knowledge on Markov Chain	Knowledge on Markov Chain	100%
>>Structural / Behavioral Model	Markov chain	Markov chain	100%

Figura 6-4 – Comparação entre o Projeto F e Critério de parada

Através das explicações anteriores, é possível observar que existem dois tipos de atributos, aqueles que podem ser considerados binários, ou seja, são totalmente aderentes ao projeto ou não são aderentes ao projeto (“Objetivo”, “Contexto de aplicação”, “Software de missão crítica” e “Software embarcado” se encaixam nessa classificação), ou os atributos que podem representar equivalência parcial ao projeto. Ou seja, não são propostos para contemplar todas as necessidades do projeto, mas contemplam algumas dessas necessidades. Os atributos “Modelo estrutural ou comportamental”, “Habilidades necessárias” e “Níveis de teste” são exemplos desse tipo. O Projeto F define que tem dois níveis de testes: funcional e unidade. Entretanto, o critério de parada selecionado foi proposto apenas para testes funcionais. Portanto, é apenas 50% aderente em relação a esse projeto.

Além disso, os atributos “Software de missão crítica” e “Software embarcado” possuem uma particularidade que pode levar a uma interpretação equivocada ao analisar apenas o gráfico radar. O fato desses atributos estarem preenchidos não significa que o critério de parada é proposto para um software de missão crítica ou embarcado, mas sim que em relação a esse atributo o critério de parada se adequa ao software. Por exemplo, no caso demonstrado, o software não é embarcado e o critério de parada não foi proposto para um sistema embarcado. Portanto, esse atributo é equivalente entre software e critério e, portanto, preenchido.

O grau de adequação (93%) entre o Projeto F e o critério de parada proposto por Whittaker e Thomason (1994) é bastante alto, portanto satisfatório. Assim, esse critério de parada, de acordo com o procedimento *Porantim CP*, deveria ser utilizado para esse projeto. Para isso, a janela de detalhes de critérios de parada deve ser fechada e, na janela representada na Figura 6-3, o critério de parada deve ser selecionado. Para que a operação seja salva, a opção “Finalize” deve ser acionada.

6.2.7 Analise dos resultados – Projeto F

A execução desse passo do estudo de caso mostra que a implementação de *Porantim CP* pode ser utilizada. Além disso, o resultado de 93% de adequação entre o Projeto F e o critério de parada pode ser considerado satisfatório, principalmente se levamos em consideração que apenas um atributo (níveis de teste) utilizado no cálculo do grau de adequação não está completamente aderente ao projeto. Entretanto, é preciso analisar se o tipo de critério sugerido está apropriado e se o cálculo do grau de adequação está correto.

Como para o *Projeto F* não foi definida nenhuma restrição através dos atributos “Confiabilidade desejada”, “Orçamento esperado para testes” e “Duração estimada da fase de testes”, então o critério sugerido deve ser um critério não-apriorístico. Através da coluna “Type” da Figura 6-3 pode ser verificado que todos os três critérios de parada sugeridos pelo procedimento são não-apriorísticos. Portanto, o primeiro passo do procedimento foi realizado com sucesso.

Como descrito no Capítulo 4, para se encontrar o grau de adequação utiliza-se o conceito de distância euclidiana que é calculado representando o projeto de software e o critério de parado através de vetores. Para cada um dos elementos do vetor que representa o projeto é atribuído o valor do peso dos atributos (0,142858) e cada um dos elementos do vetor que representa o critério de parada recebe a porcentagem (variando de 0 a 1) que o atributo está aderente ao projeto multiplicado pelo seu peso. Por exemplo, caso um atributo do critério de parada seja totalmente equivalente ao projeto ele recebe o valor 1*0,142858, caso esteja apenas 50% aderente ao projeto ele recebe 0,5*0,142858.

A seguir, é demonstrado o vetor que representa o projeto de software (v1) e o vetor que representa o critério de parada (v2) sendo os índices de 1 a 7 mapeados de acordo com a Tabela 4-2 demonstrada no Capítulo 4.

	1	2	3	4	5	6	7
v1	0,142858	0,142858	0,142858	0,142858	0,142858	0,142858	0,142858

	1	2	3	4	5	6	7
v2	0,142858	0,142858	0,142858	0,142858	0,071429	0,142858	0,142858

Depois disso, a realização do cálculo da distância euclidiana e posteriormente do grau de adequação como descrito anteriormente na Figura 4-2 do Capítulo 4 é realizado a seguir.

Cálculo da distância e nível de adequação

$$d(Pr, CP.A) = \sqrt{(0)^2 + (0)^2 + (0)^2 + (0)^2 + (0{,}071429)^2 + (0)^2 + (0)^2}$$

$$d(Pr, CP.A) = \sqrt{0{,}005102102041}$$

$$d(Pr, CP.A) = 0{,}071429 \text{ (Distância)}$$

Logo, Grau de Adequação é: $1 - 0{,}071429 = 0{,}928571$ ou **93% de adequação**

Assim, pode-se afirmar que o grau de adequação foi calculado corretamente para esse projeto.

Uma avaliação mais completa para o procedimento *Porantim CP* seria calcular o grau de adequação para todos os critérios de parada em relação ao Projeto F e verificar se o critério sugerido realmente é o mais adequado. Entretanto, devido à quantidade de critérios de parada cadastrados na base de conhecimento seria inviável realizar esse tipo de operação em prazo curto.

Além disso, cabe ressaltar que é possível visualizar as informações detalhadas de todos os critérios de parada sugeridos e que a escolha pelo critério de parada com maior grau de adequação não é obrigatória. Apesar disso, para esse caso de observação será selecionado o critério de parada com maior adequação.

Em uma análise subjetiva da usabilidade realizada durante a execução desse exemplo inicial percebeu-se que poderia haver uma melhoria futura para a implementação do procedimento. No momento da escolha do atributo “Modelos estruturais ou comportamentais” é exibida uma lista contendo um grande número de modelos cadastrados. A procura dos modelos nessa lista não se demonstrou agradável ou produtiva. A mudança na forma como esse atributo é escolhido poderia ser aprimorada na implementação do procedimento.

Utilizando a base de conhecimento para analisar o critério de parada sugerido pelo procedimento, é possível perceber que o critério sugerido é um critério não-apriorístico, proposto para ser utilizado em testes funcionais, e utilizado Cadeia de Markov para descrever o perfil operacional do software. Além disso, esse critério foi proposto para ser aplicado em um contexto onde o software testado seria

comercializado em larga escala. Portanto, com base nessas características e na descrição do projeto pode-se inferir que realmente esse pode ser um critério de parada adequado para o Projeto F.

6.3 Caso de Observação 1: Vídeo Locadora

Esta seção descreve a caracterização de um projeto para gerenciamento de uma vídeo locadora (Vegas e Basili, 2005) a inserção dessa caracterização na infraestrutura Maraká e a execução do procedimento *Porantim CP* para o projeto.

6.3.1 Descrição do projeto – Vídeo locadora

O Sistema caracterizado nesta seção é um sistema de informação web baseado em banco de dados que tem o objetivo de gerenciar uma vídeo locadora. Esse sistema foi desenvolvido por uma equipe composta por desenvolvedores inexperientes nas tarefas de especificação de requisitos e projeto de sistema, entretanto a equipe de teste é experiente em testar este tipo de sistema. Tanto a complexidade quanto a frequência de mudança dos requisitos podem ser consideradas baixa. Além disso, o sistema foi desenvolvido utilizando o paradigma de desenvolvimento orientado a objetos, linguagem de programação Java e os seguintes diagramas da UML para modelagem: Diagrama de Casos de Uso, Diagrama de Atividades, Diagrama de Estado, Diagrama de Classes.

Em relação aos testes de software, a equipe possuía um prazo de 2 meses (suficiente) para realizar os testes na aplicação e teste de sistema foi o único nível de teste utilizado.

6.3.2 Caracterização do projeto – Vídeo locadora

A Tabela 6-2 demonstra os atributos de caracterização desse sistema e seus respectivos valores.

Tabela 6-2 – Caracterização do projeto Vídeo Locadora

Atributo do projeto de software	Projeto de software
Confiabilidade desejada	Nenhum
Orçamento esperado para testes	Nenhum
Duração estimada da fase de testes	2 meses
Perfil de utilização do software	Comercialização em escala
Plataforma de execução	Cliente/Servidor
Software de missão crítica	Não
Objetivo da fase de testes do software	Não definido
Nível(is) de testes desejado(s) para o projeto	Teste de sistema

Modelo(s) comportamental	Diagrama de Casos de Uso Diagrama de Atividades Diagrama de Estado Diagrama de Classes
Habilidade provida pela equipe de testes alocada para o projeto	Não definido

6.3.3 Execução do *Porantim CP* – Vídeo locadora

Definido os atributos de caracterização do projeto, esses podem ser inseridos na infraestrutura Maraká para execução do procedimento *Porantim CP*. Após sua execução, os critérios de parada demonstrados na Figura 6-5 foram sugeridos. Pode-se perceber que os dois primeiros critérios de parada possuem alto grau de adequação com o projeto (75% e 71%), portanto esses critérios devem ser analisados.

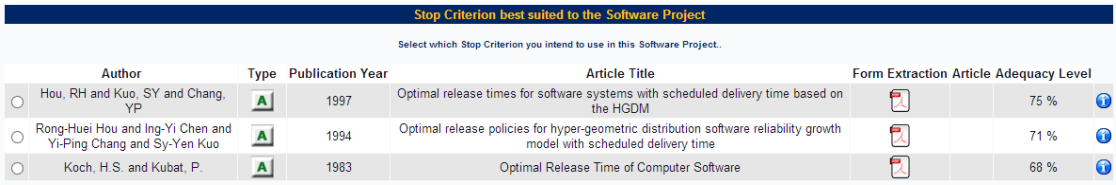
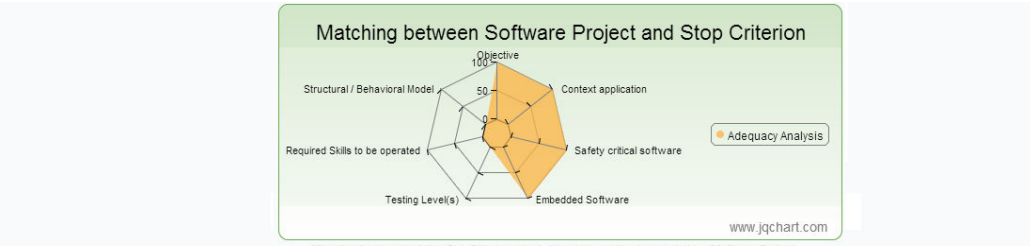


Figura 6-5 – Critérios sugeridos para sistema de Vídeo Locadora

6.3.4 Seleção do critério de parada – Vídeo locadora

Para visualizar maiores detalhes sobre a comparação do projeto com os dois critérios mais adequados, o ícone (*View Details of Stop Criterion*) deve ser acionado para cada um dos projetos. A **Erro! Fonte de referência não encontrada.** demonstra o gráfico radar e a tabela de comparação de atributos do projeto com o critério de parada proposto por Hou *et al* (1997). Pode-se perceber que os atributos “Objetivo”, “Contexto de aplicação”, “Software de missão crítica” e “Software embarcado” são equivalentes entre o projeto e esse critério de parada.

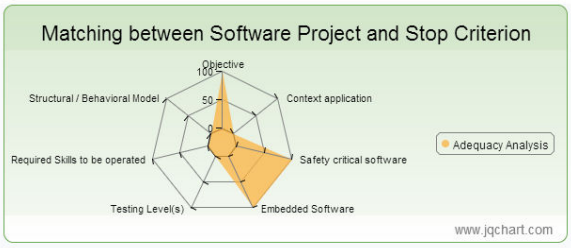


Visualize the characteristics Stop Criterion selected in relation to the characteristics of Software Project..

Attributes Characterization	Software Project	Stop Criterion	Adequacy Level
>>Objective	Minimizar custo restrito ao prazo	Minimizar custo restrito ao prazo	100%
>>Utilization Context	Comercialização em escala	Comercialização em escala	100%
>>Safety critical software	Não	Não	100%
>>Embedded	Não	Não	100%
>>Testing Level(s)			0%
>>Required Skills			0%
>>Structural / Behavioral Model	UML Activity Diagram UML Class Diagram UML Statechart Diagram UML Use Cases Diagram		0%

Figura 6-6 – Comparação projeto e critério de parada Hou *et al.*

Essa mesma análise pode ser realizada para o critério de parada proposto por Hou *et al.* (1994). Assim, a Figura 6-7 demonstra o gráfico radar e a tabela de comparação dos atributos do projeto e critério de parada. Como pode ser observado, os atributos equivalentes entre projeto e critério de parada são “Objetivo”, “Software de missão crítica” e “Software embarcado”.



Attributes Characterization	Software Project	Stop Criterion	Adequacy Level
>>Objective	Minimizar custo restrito ao prazo	Minimizar custo restrito ao prazo	100%
>>Utilization Context	Comercialização em escala	Não informado	0%
>>Safety critical software	Não	Não	100%
>>Embedded	Não	Não	100%
>>Testing Level(s)			0%
>>Required Skills			0%
>>Structural / Behavioral Model	UML Activity Diagram UML Class Diagram UML Statechart Diagram UML Use Cases Diagram		0%

Figura 6-7 – Comparação projeto e critério de parada Hou *et al.*

A análise dos resultados permite identificar o primeiro critério de parada difere do segundo, em relação ao projeto, por apresentar o mesmo contexto de aplicação do projeto e no segundo o contexto de aplicação é desconhecido.

Dessa forma, o atributo “Contexto de aplicação” pode ser utilizado para tomada de decisão em relação a qual critério de parada pode ser utilizado para o projeto de vídeo locadora sendo, neste caso, o critério de Hou *et al.* (1997) o mais adequado. Esse critério sugerido pelo procedimento é um critério apriorístico, utilizado para software que possui tempo de entrega programado.

6.3.5 Análise dos resultados – Vídeo locadora

Primeiramente o cálculo do grau de adequação deve ser verificado para se ter certeza de que foi realizado corretamente. Assim, é demonstrado o vetor que representa o projeto de software (v1) e o vetor que representa o critério de parada (v2) sendo os índices de 1 a 7 mapeados de acordo com a Tabela 4-2 demonstrada no Capítulo 4.

	1	2	3	4	5	6	7
v1	0,142858	0,142858	0,142858	0,142858	0,142858	0,142858	0,142858

	1	2	3	4	5	6	7
v2	0,142858	0,142858	0,142858	0,142858	0	0	0

Depois disso, a realização do cálculo da distância euclidiana e posteriormente do grau de adequação como descrito anteriormente na Figura 4-2 do Capítulo 4 é demonstrado a seguir.

Cálculo da distância e nível de adequação

$$d(Pr, Cp.A) = \sqrt{(0)^2 + (0)^2 + (0)^2 + (0)^2 + (0{,}142858)^2 + (0{,}142858)^2 + (0{,}142858)^2}$$

$$d(Pr, Cp.A) = \sqrt{0{,}061225224}$$

$$d(Pr, Cp.A) = 0{,}247437314 \textbf{ (Distância)}$$

Logo, Grau de Adequação é: $1 - 0{,}247437314 = 0{,}752562686$ ou **75% de adequação**

Assim, pode-se afirmar que o grau de adequação foi calculado corretamente para esse projeto.

Analisando os dados contidos no corpo de conhecimento a respeito do critério selecionado é possível perceber que esse critério é um critério apriorístico que define a duração da fase de testes e, da mesma forma, o projeto de software possui uma restrição de prazo para a fase de teste. Portanto, analisando esse atributo separadamente o critério de parada se demonstra adequado ao projeto.

O critério de parada possui o objetivo de minimizar o custo restrito ao prazo que é o mesmo objetivo da fase de testes do projeto. O contexto de aplicação também é equivalente entre projeto e critério de parada, pois o projeto é desenvolvido para comercialização em larga escala assim como o critério é proposto para software que serão comercializados em larga escala. Além disso, o critério de parada foi proposto para softwares que não são embarcados e não são de missão crítica e o software em questão atende a essas características.

Portanto, pode-se dizer que a utilização do critério de parada sugerido pelo procedimento *Porantim CP* é viável para este projeto.

6.4 Caso de Observação 2: Controle de Empréstimo Bancário

Esta seção descreve a caracterização de um projeto para gerenciamento empréstimos bancário (Vegas e Basili, 2005), a inserção dessa caracterização na infraestrutura Maraká e a execução do procedimento *Porantim CP* para o projeto.

6.4.1 Descrição do projeto – Controle de Empréstimo Bancário

Nesta seção, um sistema *batch* para controle de empréstimo bancário é utilizado. O Objetivo do sistema é controlar a aprovação ou reprovação de empréstimos bancários, além de calcular os valores a serem pagos no empréstimo.

Uma característica enfatizada na descrição do software é que as regras de negócio são constantemente alteradas. Além disso, o domínio do problema possui complexidade elevada, pois o sistema possui muitas fórmulas a serem calculadas nos processos internos.

Em relação à equipe alocada para construção do software, os integrantes da equipe de desenvolvimento possuem larga experiência enquanto os integrantes da equipe de teste são inexperientes com critérios de geração funcional e estrutural de testes. Para a construção do sistema foi utilizado o diagrama de estados da UML e optou-se pela linguagem de programação C.

Em relação aos testes de software foram executados testes de unidade.

6.4.2 Caracterização do projeto – Controle de Empréstimo Bancário

A Tabela 6-3 demonstra a caracterização desse projeto de software.

Tabela 6-3 – Caracterização do projeto Controle de Empréstimo Bancário

Atributo do projeto de software	Projeto de software
Confiabilidade desejada	Nenhum
Orçamento esperado para testes	Nenhum
Duração estimada da fase de testes	Nenhum
Perfil de utilização do software	Comercialização para cliente único
Plataforma de execução	Batch - Concorrente
Software de missão crítica	Não
Objetivo da fase de testes do software	Maximizar confiabilidade
Nível(is) de testes desejado(s) para o projeto	Testes de unidade
Modelo(s) comportamental	Diagrama de estado
Habilidade provida pela equipe de testes alocada para o projeto	Não informado

6.4.3 Execução do *Porantim CP* – Controle de Empréstimo Bancário

Após a descrição e caracterização do projeto as informações são inseridas na infraestrutura Maraká para uso do procedimento *Porantim CP*. Depois da execução do procedimento, os critérios de parada sugeridos para o projeto de software para controle de Empréstimo Bancário são apresentados na Figura 6-8.

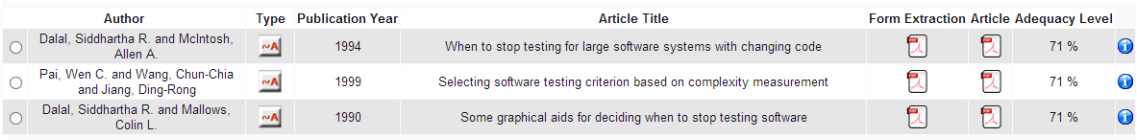
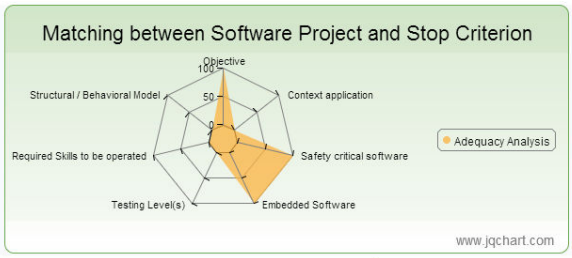


Figura 6-8 – Critérios sugeridos para sistema de Empréstimo Bancário

Diferente do caso anterior, os critérios de parada sugeridos apresentam o mesmo grau de adequação (71%). Nesse caso, a tomada de decisão pelo engenheiro de software deve ser realizada de forma mais atenta, pois é preciso analisar cada um dos critérios sugerido, os atributos que influenciaram no grau de adequação e talvez até características que não estejam explicitadas através dos atributos de caracterização.

6.4.4 Seleção do critério de parada – Controle de Empréstimo Bancário

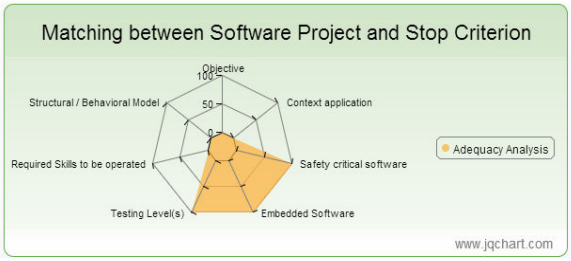
Para seleção do critério de parada mais adequado para o projeto cada um dos critérios de parada deve ser analisado em detalhe. A Figura 6-9 demonstra os detalhes da comparação entre o critério de parada proposto por Dalal e McIntosh (1994) e o projeto. Como pode ser percebido, os atributos equivalentes entre o projeto e o critério de parada são o “Objetivo”, “Software de missão crítica” e “Software Embarcado”.



Attributes Characterization	Software Project	Stop Criterion	Adequacy Level
>>Objective	Não definido	Não definido	100%
>>Utilization Context	Comercialização para cliente único	Sistemas com alta volatilidade de requisitos	0%
>>Safety critical software	Não	Não	100%
>>Embedded	Não	Não	100%
>>Testing Level(s)	Unit Testing		0%
>>Required Skills			0%
>>Structural / Behavioral Model	UML Statechart Diagram		0%

Figura 6-9 – Comparação projeto e critério Dalal e McIntosh (1994)

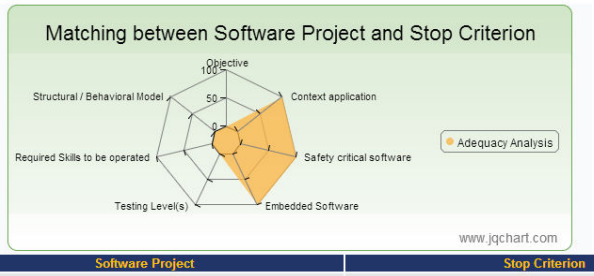
Já o critério proposto por Pai *et al.* (1999) não possui o atributo “Objetivo” definido e, portanto, não se identifica como aderente ao projeto. Entretanto, apresenta os atributos “Software de missão crítica”, “Software embarcado” e “Níveis de testes” aderentes ao projeto de empréstimo bancário.



Attributes Characterization	Software Project	Stop Criterion	Adequacy Level
>>Objective	Não definido	Verificar se é possível parar os testes antes de percorrer todos os fluxos	0%
>>Utilization Context	Comercialização para cliente único	Não informado	0%
>>Safety critical software	Não	Não	100%
>>Embedded	Não	Não	100%
>>Testing Level(s)	Unit Testing	Unit Testing	100%
>>Required Skills			0%
>>Structural / Behavioral Model	UML Statechart Diagram		0%

Figura 6-10 – Comparação projeto e critério Pai *et al.* (1999)

Em complemento, o critério proposto por Dalal e Mallows (1990) apresenta os atributos “Contexto de aplicação”, “Software de missão crítica” e “Software embarcado” aderentes ao projeto.



Attributes Characterization	Software Project	Stop Criterion	Adequacy Level
>>Objective	Não definido	Equilíbrio entre o custo com testes e custo de correção de falhas com o software em produção	0%
>>Utilization Context	Comercialização para cliente único	Comercialização para cliente único	100%
>>Safety critical software	Não	Não	100%
>>Embedded	Não	Não	100%
>>Testing Level(s)	Unit Testing		0%
>>Required Skills			0%
>>Structural / Behavioral Model	UML Statechart Diagram		0%

Figura 6-11 – Comparação projeto e critério Dalal e Mallows (1990)

Como pode ser percebido, apenas através dos atributos de caracterização não é possível identificar um critério de parada mais apropriado para esse projeto. Isso porque, os atributos possuem o mesmo peso (importância) no cálculo do grau de adequação e como a quantidade de atributos equivalentes entre projeto e critério de parada é sempre igual para cada um dos três critérios de parada, o grau de adequação também é sempre igual.

Dessa forma, é preciso analisar mais cuidadosamente os critérios de parada sugeridos, buscando no corpo de conhecimento informações detalhadas sobre cada um deles. Essas informações podem ser obtidas através de Maraká acessando o ícone “Form Extraction”.

Com a leitura da descrição de cada um dos critérios de parada pode-se perceber que o critério proposto por Dalal e McIntosh (1994) foi proposto para ser aplicado em projetos que possuem constantes mudanças no código. Como na

descrição do projeto foi mencionado que as regras de negócio desse sistema apresentam volatilidade alta pode-se inferir que o código desse sistema passará por diversas mudanças. Portanto, o critério de parada proposto por Dalal e McIntosh (1994), além de atender a outras características do projeto, atende também a essa característica e assim permite uma melhor aplicação no projeto. Portanto, esse deve ser o critério de parada selecionado.

6.4.5 Análise dos resultados – Controle de Empréstimo Bancário

Assim como realizado para o projeto anterior, a corretude do grau de adequação também deve ser verificada para o critério de parada selecionado. Dessa forma, os vetores que representam o projeto de software (v1) e o critério de parada selecionado (v2) são demonstrados a seguir.

	1	2	3	4	5	6	7
v1	0,142858	0,142858	0,142858	0,142858	0,142858	0,142858	0,142858

	1	2	3	4	5	6	7
v2	0	0,142858	0,142858	0,142858	0	0	0

Depois disso, é demonstrado o cálculo da distância euclidiana e, em seguida, o cálculo do grau de adequação de acordo com a Figura 4-2 do Capítulo 4.

Cálculo da distância e nível de adequação

$$d(Pr, Cp.A) = \sqrt{(0)^2 + (0{,}142858)^2 + (0{,}142858)^2 + (0{,}142858)^2 + (0)^2 + (0)^2 + (0)^2}$$

$$d(Pr, Cp.A) = \sqrt{0{,}081633633}$$

$$d(Pr, Cp.A) = 0{,}285716 \text{ (Distância)}$$

Logo, Grau de Adequação é: $1 - 0{,}285716 = 0{,}714284$ ou **71% de adequação**

Como demonstrado anteriormente, o grau de adequação foi calculado corretamente para esse projeto.

A execução do *Porantim CP* no projeto de controle de empréstimo bancário gerou uma situação diferente das analisadas anteriormente. Isso porque o grau de adequação para os critérios de parada sugeridos foram iguais. Nesse caso foi preciso utilizar o corpo de conhecimento para decidir qual critério de parada seria o mais adequado para o projeto.

Como demonstrado, apesar da igualdade do grau de adequação, alguns atributos que levaram ao cálculo são distintos. Esse tipo de situação poderia ser

evitada atribuindo pesos diferentes para cada um dos atributos. Dessa forma, os atributos mais importantes para a seleção do critério de parada receberiam pesos maiores que os atributos menos importantes. Para realizar essa classificação dos atributos sem a utilização de suposições que possam enviesar os resultado, um *survey* com especialistas da área de testes de software poderia ser executado. Entretanto, este estudo esta fora do escopo dessa dissertação.

Apesar dos impasses encontrados, o critério de parada pode ser considerado utilizável para o projeto em questão. Isso porque, o objetivo da fase de testes do projeto é maximizar a confiabilidade e o critério foi proposto para softwares que possuem esse mesmo objetivo. Além disso, o critério também foi proposto para ser aplicado em softwares que não são de missão crítica, não são embarcados e o projeto possui essas características.

Apesar da possibilidade de aplicação do critério de parada selecionado, é recomendável que a condução dos testes e execução do critério de parada deve ser monitorada para avaliar se o critério escolhido realmente atende as expectativas.

6.5 Caso de Observação 3: Estacionamento

Esta seção descreve a caracterização de um projeto de software para gerenciamento de um estacionamento (Vegas e Basili, 2005), a inserção dessa caracterização na infraestrutura Maraká e a execução do procedimento *Porantim CP* para o projeto.

6.5.1 Descrição do projeto – Estacionamento

O Sistema caracterizado nesta seção é um sistema síncrono que tem por objetivo controlar a disponibilidade de vagas em estacionamento de veículos e deve ser comercializado para diversos estacionamentos. É um sistema de baixa complexidade e que possui requisitos estáveis. Além disso, foi utilizada a tecnologia de modelagem LUSTRE e a linguagem de programação ADA.

A equipe de desenvolvimento bem como a equipe de testes foi composta por pessoas experientes no desenvolvimento e testes respectivamente.

Em relação aos testes havia um orçamento restrito para essas atividades e os níveis de testes executados foram testes de sistema e testes de integração.

6.5.2 Caracterização do projeto – Estacionamento

A Tabela 6-4 demonstra os atributos que caracterizam esse sistema.

Tabela 6-4 – Caracterização do projeto Controle de Estacionamento

Atributo do projeto de software	Projeto de software
Confiabilidade desejada	Nenhum
Orçamento esperado para testes	R$ 10.000,00
Duração estimada da fase de testes	Nenhuma
Perfil de utilização do software	Comercialização em larga escala
Plataforma de execução	Síncrono
Software de missão crítica	Não
Objetivo da fase de testes do software	Minimizar custo restrito ao prazo
Nível(is) de testes desejado(s) para o projeto	Teste de Sistema Teste de Integração
Modelo(s) comportamental	Descrição do Ambiente
Habilidade provida pela equipe de testes alocada para o projeto	Não definido

6.5.3 Execução do *Porantim CP* – Estacionamento

Repetindo o procedimento realizado para os projetos anteriores, os atributos de caracterização foram inseridos na infraestrutura Maraká e o procedimento *Porantim CP* sugeriu os três critérios mais adequados a esse projeto. Na Figura 6-12 pode ser observado que dos três critérios sugeridos o mais adequado (75% de adequação) é o critério proposto por Hou *et al.* (1997).

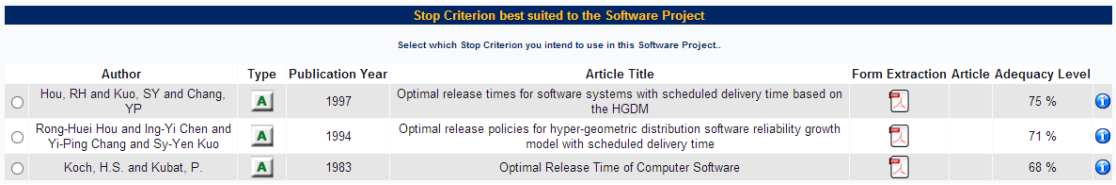
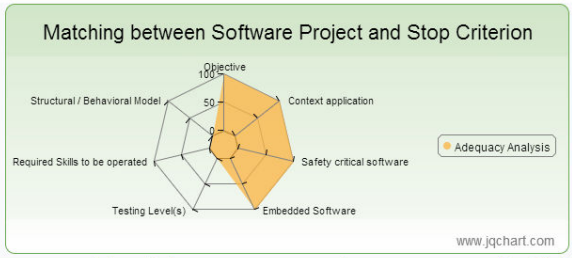


Figura 6-12 – Critérios sugeridos para sistema de Estacionamento

Pode ser observado que, além de ter sido sugerido para esse projeto, o critério proposto por Hou et al. (1997) também foi sugerido para o projeto de Vídeo Locadora. Isto ocorreu porque esses projetos possuem características comuns. Por exemplo, o contexto de aplicação para os dois projetos é comercialização em escala, os projetos não são de missão crítica e também não são embarcados. Além disso, os dois projetos necessitariam de um critério de parada apriorístico, pois definem restrições gerenciais de prazo ou orçamento para a realização dos testes. O gráfico e a tabela apresentados na Figura 6-13 demonstram os atributos que levaram ao cálculo desse grau de adequação.



Attributes Characterization	Software Project	Stop Criterion	Adequacy Level
>>Objective	Minimizar custo restrito ao prazo	Minimizar custo restrito ao prazo	100%
>>Utilization Context	Comercialização em escala	Comercialização em escala	100%
>>Safety critical software	Não	Não	100%
>>Embedded	Não	Não	100%
>>Testing Level(s)	Integration Testing System Testing		0%
>>Required Skills			0%
>>Structural / Behavioral Model	Environment Description		0%

Figura 6-13 – Comparação entre projeto vídeo locadora e critério de parada

Através da comparação dos atributos do projeto com os atributos do critério de parada pode-se perceber que as atividades de testes do projeto têm o objetivo de minimizar o custo restrito ao prazo e que o projeto está sendo desenvolvido para comercialização em larga escala e o critério de parada está aderente tanto com o objetivo quanto com o contexto de aplicação. Além disso, o software não é de missão crítica e também não é embarcado e o critério de parada foi proposto para esse tipo de software. Portanto, com base nos atributos, a escolha desse critério de parada seria viável.

6.5.4 Análise dos resultados – Estacionamento

O primeiro passo realizado na análise dos resultados é a verificação da corretude do grau de adequação. Para isso, o projeto de software (v1) e o critério de parada (v2) são representados através de vetores como demonstrado a seguir.

	1	2	3	4	5	6	7
v1	0,142858	0,142858	0,142858	0,142858	0,142858	0,142858	0,142858

	1	2	3	4	5	6	7
v2	0,142858	0,142858	0,142858	0,142858	0	0	0

Depois disso, é demonstrado o cálculo da distância euclidiana e do grau de adequação como descrito anteriormente na Figura 4-2 do Capítulo 4.

Cálculo da distância e nível de adequação

$$d(Pr, Cp.A) = \sqrt{(0{,}142858)^2 + (0{,}142858)^2 + (0{,}142858)^2 + (0{,}142858)^2 + (0)^2 + (0)^2 + (0)^2}$$

$$d(Pr, Cp.A) = \sqrt{0{,}061225224}$$

$$d(Pr, Cp.A) = 0{,}247437314 \text{ (Distância)}$$

Logo, Grau de Adequação é: $1 - 0{,}247437314 = 0{,}752562686$ ou **75% de adequação**

Baseado na demonstração anterior infere-se que o grau de adequação foi calculado corretamente.

Como o projeto de software e o critério de parada possuem mesmo objetivo (minimizar custo restrito ao prazo) e contexto de aplicação (comercialização em escala) e, além disso, ambos são para software que não são de missão crítica e não são embarcados, o critério de parada pode ser coerente com o projeto.

Entretanto, a partir da execução desse caso da prova de conceito, percebeu-se que os atributos “Modelos comportamentais e estruturais”, “Habilidades necessárias” e “Níveis de testes” são subutilizados. Ou seja, apenas um projeto utilizou esses atributos. Isso pode ser um problema na caracterização do projeto, na caracterização do software ou na escolha dos atributos de caracterização. Uma discussão mais detalhada sobre esse problema é realizada ao final desse capítulo.

6.6 Caso de Observação 4: Controle do Nível de Água em Represa

Esta seção descreve a caracterização de um projeto de software que controla o nível de água em uma represa (Vegas e Basili, 2005), a inserção dessa caracterização na infraestrutura Maraká e a execução do procedimento *Porantim CP* para o projeto.

6.6.1 Descrição do projeto – Controle de Nível de Água em Represa

O Sistema caracterizado nessa seção é um sistema embarcado de tempo real que tem por objetivo monitorar o nível de água em uma represa. Esse sistema foi desenvolvido especificamente para um único cliente. Além disso, uma especificação do software através de modelos formais, diagrama de estado, diagrama de sequência e colaboração foram utilizados para modelar o sistema.

A equipe de desenvolvimento bem como a equipe de teste foi considerada experiente em desenvolvimento e testes de sistemas de tempo real.

Em relação aos níveis de testes os testes de unidade foram os únicos executados. Além disso, havia um requisito de confiabilidade mínima para o software.

6.6.2 Caracterização do projeto – Controle de Nível de Água em Represa

A Tabela 6-5 exibe os atributos de caracterização desse projeto bem como o valor de cada um dos atributos.

Tabela 6-5 – Caracterização do projeto Controle de Nível de Água em Represa

Atributo do projeto de software	Projeto de software
Confiabilidade desejada	1 falha/dia
Orçamento esperado para testes	Nenhum
Duração estimada da fase de testes	Nenhum
Perfil de utilização do software	Comercialização para cliente único
Plataforma de execução	Embarcado
Software de missão crítica	Sim
Objetivo da fase de testes do software	Maximizar confiabilidade
Nível(is) de testes desejado(s) para o projeto	Teste de unidade
Modelo(s) comportamental	Especificação do software através de modelos Formais Diagrama de Estado Diagrama de Sequência/Colaboração
Habilidade provida pela equipe de testes alocada para o projeto	Não informado

6.6.3 *Execução do Porantim CP* – Controle de Nível de Água em Represa

Repetindo o procedimento descrito para os outros casos de observação, os critérios de parada sugeridos para o sistema de controle de nível de água em represa são demonstrados através da Figura 6-14.

Como pode ser observado, dois critérios de parada possuem o mesmo grau de adequação (71%) com o projeto. Dessa forma, é preciso realizar uma analise semelhante à realizada para o sistema de controle de empréstimo bancário.

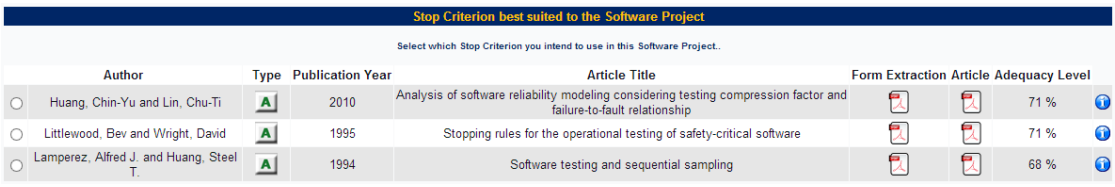
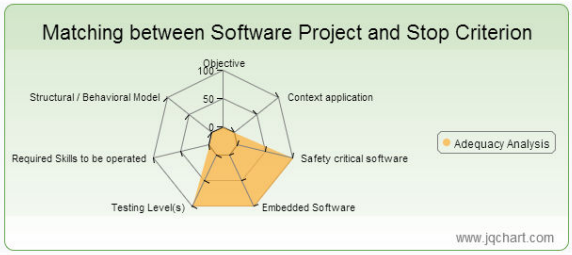


Figura 6-14 – Critérios sugeridos para sistema de Nível de Água em Represa

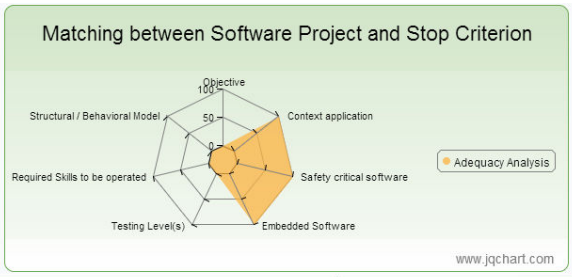
O primeiro critério a ser analisado foi proposto por Huang e Lin (2010) no qual os atributos que levaram à adequação de 71% são “Software de missão crítica”, “Software embarcado” e “Níveis de testes”. Ou seja, esses atributos são equivalentes entre o projeto e o critério de parada. A Figura 6-15 demonstra a comparação entre eles.



Attributes Characterization	Software Project	Stop Criterion	Adequacy Level
>>Objective	Maximizar Confiabilidade	Atingir confiabilidad pré-determinada	0%
>>Utilization Context	Comercialização para cliente único	Não informado	0%
>>Safety critical software	Sim	Sim	100%
>>Embedded	Sim	Sim	100%
>>Testing Level(s)	Unit Testing	Acception Testing Integration Testing Regression Testing Stress Testing System Testing Teste Funcional Unit Testing	100%
>>Required Skills			0%
>>Structural / Behavioral Model	Customizable Formal Model UML Collaboration Diagram UML Sequence Diagram UML Statechart Diagram		0%

Figura 6-15 – Comparação projeto controle do nível de água e o critério proposto por Huang e Lin (2010)

Como demonstrado na Figura 6-16, o segundo critério analisado foi proposto por Littlewood e Wright (1995) no qual os atributos que levaram à adequação de 71% são “Contexto de aplicação”, “Software de missão crítica” e “Software embarcado”. Ou seja, esses atributos são equivalentes entre projeto e esse critério de parada.



Attributes Characterization	Software Project	Stop Criterion	Adequacy Level
>>Objective	Maximizar Confiabilidade	Atingir confiabilidad pré-determinada	0%
>>Utilization Context	Comercialização para cliente único	Comercialização para cliente único	100%
>>Safety critical software	Sim	Sim	100%
>>Embedded	Sim	Sim	100%
>>Testing Level(s)	Unit Testing		0%
>>Required Skills			0%
>>Structural / Behavioral Model	Customizable Formal Model UML Collaboration Diagram UML Sequence Diagram UML Statechart Diagram		0%

Figura 6-16 – Comparação projeto controle nível de água e critério proposto por Littlewood e Wright (1995)

Como pode ser visualizado, os atributos que diferenciam o primeiro critério de parada do segundo são “Níveis de testes” e “Contexto de aplicação”. O primeiro critério de parada contempla testes de unidade e o segundo não. Entretanto, o segundo critério foi proposto para software comercializado para cliente único e o primeiro não informa o contexto de aplicação.

Dessa forma, existe um conflito na escolha do critério de parada mais adequado, pois sem um estudo preliminar poderia ser arriscado afirmar que um

atributo é mais importante que o outro na tomada de decisão sobre que critério utilizar. Assim, a base de conhecimento pode ser consultada na tentativa de conseguir alguma informação adicional que ajude na decisão.

Com a análise do corpo de conhecimento foi possível constatar que o critério de parada proposto por Littlewood e Wright (1995) foi proposto para ser aplicado especificamente em um software de controle de sensores de um reator nuclear. Devido à semelhança de contexto do software para o qual o critério foi proposto e o software descrito nessa seção esse critério de parada seria uma alternativa viável e poderia ser escolhido.

6.6.4 *Análise dos resultados* – Controle de Nível de Água em Represa

Para verificar a corretude do grau de adequação o projeto de software (v1) e o critério de parada (v2) devem ser representados através de vetores. Esses vetores são representados a seguir.

	1	2	3	4	5	6	7
v1	0,142858	0,142858	0,142858	0,142858	0,142858	0,142858	0,142858

	1	2	3	4	5	6	7
v2	0,142858	0,142858	0,142858	0	0	0	0

Depois disso, o cálculo da distância euclidiana e do grau de adequação é realizado e demonstrado a seguir. Esse cálculo é demonstrado em detalhes na Figura 4-2 do Capítulo 4 é realizado a seguir.

Cálculo da distância e nível de adequação

$$d(Pr, Cp.A) = \sqrt{(0{,}142858)^2 + (0{,}142858)^2 + (0{,}142858)^2 + (0)^2 + (0)^2 + (0)^2 + (0)^2}$$

$$d(Pr, Cp.A) = \sqrt{0{,}081633633}$$

$$d(Pr, Cp.A) = 0{,}285716 \text{ (Distância)}$$

Logo, Grau de Adequação é: $1 - 0{,}285716 = 0{,}714284$ ou **71% de adequação**

Como demonstrado, o resultado (71%) do cálculo realizado pela implementação do procedimento e o cálculo demonstrado anteriormente são equivalentes. Portanto, infere-se que o grau de adequação foi calculado corretamente.

Nesse caso de observação uma situação diferente pode ser percebida ao analisar os resultados de comparação entre o projeto de software e o critério sugerido por Huang e Lin (2010). Através da Figura 6-16 pode ser visualizado que o atributo

“Níveis de teste” para o projeto prevê apenas que testes de unidade devem ser executados e esse mesmo atributo do critério de parada está prevendo diversos níveis de teste, incluindo os testes de unidade. Contudo, a adequação referente a esse atributo é 100%, ou seja, o critério de parada é 100% adequado ao projeto se considerado apenas esse atributo. Isso acontece, pois o critério de parada contempla o nível de teste de unidade previsto no projeto e os outros níveis contemplados pelo critério de parada ficariam ociosos. Caso os valores dos atributos fossem invertidos (O projeto com diversos níveis de teste e o critério apenas com testes de unidade) a adequação em relação a esse atributo não seria 100% e sim cerca de 14%, pois apenas um nível de teste previsto para o projeto estaria sendo contemplado pelo critério de parada.

Nesse caso, também foi observado que os atributos “Habilidades necessárias” e “Modelos estruturais e comportamentais” não foram preenchidos pelos critérios de parada analisado. Isso reforça a hipótese de que pode haver um problema na caracterização dos projetos, do critério de parada ou na escolha desses atributos.

A solução para esse problema não é trivial e deve passar por diversas etapas que fogem ao escopo desse trabalho. Um *survey* com especialistas em testes de software poderia ajudar a definir se esses atributos realmente são úteis na seleção de critérios de parada. Caso não sejam úteis, esses atributos podem ser substituídos por atributos mais adequados. Caso sejam realmente úteis, seria preciso alterar a forma como os critérios de parada são divulgados na literatura técnica de forma que explicitem o valor para esses atributos.

6.7 Prova de Conceito – Sistema de Informação WEB

Esta seção descreve a aplicação do procedimento *Porantim CP* em um projeto de software denominado SiGIC (SANTOS e TRAVASSOS, 2010). A descrição do projeto é apresenta e em seguida os atributos de caracterização são extraídos dessa descrição. Por fim, as informações relacionadas à caracterização do projeto são inseridas no componente *Porantim CP* e o procedimento para sugestão de critério de parada é executado.

6.7.1 Descrição do projeto SiGIC

O Sistema de Gestão de Informações (SiGIC) é um sistema de informação Web para gerenciamento das atividades dessa fundação que atua no apoio a projetos de ciência e tecnologia. SiGIC é um software em larga escala que envolve diferentes setores da fundação, como: recursos humanos, financeiros, contabilidade,

gerenciamento de projetos e protocolo. O sistema foi desenvolvido utilizando tecnologia Java+JavaServer Faces (JSF) e MySQL como banco de dados. Além disso, o processo para desenvolvimento do sistema seguiu um modelo de referência de processo baseado nos níveis G e C do MPS.br (SOFTEX, 2007). A Tabela 6-6 ilustra o tamanho do sistema demonstrando a quantidade de casos de uso, regras de negócio, linhas de código SQL e linhas de código em Java+JSF.

Tabela 6-6 – Métricas SiGIC

Métrica	Valor
Casos de Uso	180
Regras de Negócio	906
Linhas de Código SQL	2500
Classes	2034
Linhas de Código	315329

Devido ao tamanho e complexidade da aplicação o sistema foi dividido em seis módulos: Módulo de Gestão de Usuários (MGU), Módulo de Solicitações (MSL), Módulo de Protocolo (MPT), Módulo de Relatórios (MRT), Módulo de Acompanhamento de Projetos (MAP) e Módulo Financeiro (MFI).

Cada um dos módulos apresentados anteriormente passou pelas etapas de análise, projeto, codificação e testes (DIAS NETO *et al.*, 2007). A UML (*Unified Modeling Language*) foi utilizada como linguagem para modelagem dos artefatos do sistema da qual os seguintes diagramas foram utilizados: Casos de Uso, Classe, Sequências e Estado.

Em relação aos testes pode-se dizer que o objetivo era atingir o equilíbrio entre o custo com testes e custo de correção de falhas com o software em produção. Além disso, diferentes níveis de testes foram utilizados (DIAS NETO *et al.*, 2007):

- **Teste de estrutura**: consiste na execução, sem utilização de ferramentas (manual) e sem uso de técnicas (forma *ad hoc*), dos casos de uso utilizados para especificar o sistema. Esse nível de teste foi utilizado de forma a garantir que todos os fluxos de atividades foram implementados e que não existe nenhuma falha na navegação durante a execução dos casos de uso. Além disso, foi utilizado um formulário pré-definido que contém questionamentos sobre possíveis falhas que podem ocorrer. Essa estratégia foi escolhida para que não houvesse qualquer preocupação com o planejamento que pudesse resultar em esforço adicional ao processo de testes. Este nível de teste foi denominado teste de estrutura, pois os casos de uso são vistos como grafos e a preocupação não é com a cobertura dos

testes, mas em seguir os fluxos e revelar falhas associadas à ausência de passos no fluxo de atividades ou processamento incorreto de determinada operação.

- **Teste funcional:** consiste na execução de testes de forma mais formal, seguindo o processo descrito em Dias Neto e Travassos (2006). Essa abordagem é composta por um processo de teste, um conjunto de roteiros de documentos pré-definidos de acordo com o padrão IEEE 829 (1998) e a própria infraestrutura Maraká. Seis macros atividades compõe essa estratégia de teste: planejar testes, projetar testes, especificar casos de testes, definir procedimentos de teste, executar teste e analisar os resultados dos testes. O Objetivo dessa fase é a avaliação funcional dos casos de uso, fluxos principais e alternativos, suas exceções, regras de negócio e restrições.
- **Teste beta:** consiste na disponibilização de um módulo do sistema para um conjunto restrito de usuários para que possam utilizar, avaliar, testar e/ou sugerir alterações no sistema antes de sua entrega definitiva.

A equipe responsável pela qualidade do sistema era composta por membros altamente qualificados que possuem conhecimento e experiência em gerenciamento de processos de teste, execução das atividades do processo de teste além de conhecimentos gerais de desenvolvimento de software, como modelagem de software utilizando UML e programação orientada a objetos.

6.7.2 Caracterização do Projeto SiGIC

Com base na descrição do sistema apresentada anteriormente, é possível caracterizar o projeto SiGIC utilizando os atributos de caracterização de projetos definidos no Capítulo 4. Assim, a Tabela 6-7 apresenta cada atributo de caracterização bem como seu valor referente ao projeto SiGIC.

Cabe ressaltar que os atributos "Confiabilidade desejada", "*Orçamento esperado para testes*" e "Duração estimada da fase de testes" não foram preenchidos, pois não haviam decisões gerenciais que levassem à definição dos mesmos.

Tabela 6-7 – Caracterização do projeto SiGIC

Atributo do projeto de software	Projeto de software
Confiabilidade desejada	Nenhum
Orçamento esperado para testes	Nenhum
Duração estimada da fase de testes	Nenhum
Perfil de utilização do software	Comercialização para único cliente
Plataforma de execução	Web

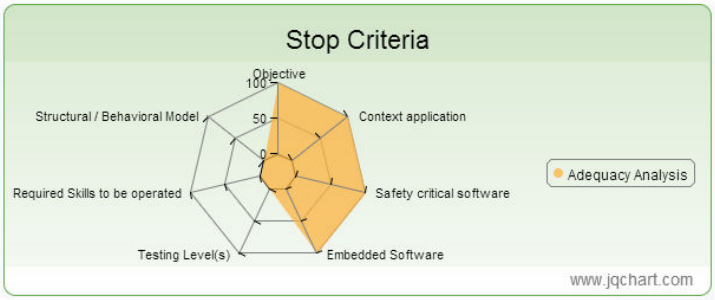
Software de missão crítica	Não
Objetivo da fase de testes do software	Equilíbrio entre o custo com testes e custo de correção de falhas com o software em produção
Nível(is) de testes desejado(s) para o projeto	Teste de estrutura, teste funcional e teste beta
Modelo(s) comportamental	Diagramas da UML
Habilidade provida pela equipe de testes alocada para o projeto	Conhecimentos em modelagem de software, programação orientada a objetos, gerenciamento de processos de testes e em UML

6.7.3 Execução Prova de Conceito SiGIC

Assim como realizado para os outros projetos, os atributos de caracterização podem ser inseridos na infraestrutura Maraká. Com isso, o procedimento *Porantim CP* pode ser executado e assim Maraká exibe os critérios de parada mais adequados ao projeto SiGIC de acordo com a fórmula exibida no Capítulo 4. Após o cálculo, os três critérios de parada que possuem maior grau de adequação com o projeto SiGIC são disponibilizados para escolha como pode ser visto na Figura 6-17.

Figura 6-17 – Critérios de parada sugeridos para SiGIC

Assim, o critério de parada mais adequado é proposto por Dalal e Mallows (1990) e possui adequação de 75% com o projeto SiGIC. Para verificação dos atributos que levaram o critério de parada ser mais adequado ao projeto a opção (*View Details of Stop Criterion*) pode ser selecionada. Com isso, uma tela semelhante à Figura 6-18 é exibida. Nesta tela é possível visualizar um gráfico radar e uma tabela que mostram quais atributos levaram o critério de parada estar adequado ao projeto SiGIC. Assim, é possível visualizar que os atributos “Objetivo”, “Contexto de utilização”, “Software de missão crítica” e “Software embarcado” do projeto são equivalentes a esses mesmos atributos do critério de parada.



Visualize the characteristics Stop Criterion selected in relation to the characteristics of Software Project..

Attributes Characterization	Software Project	Stop Criterion	Adequacy Level
>>Objective	Equilíbrio entre o custo com testes e custo de correção de falhas com o software em produção	Equilíbrio entre o custo com testes e custo de correção de falhas com o software em produção	100%
>>Utilization Context	Comercialização para cliente único	Comercialização para cliente único	100%
>>Safety critical software	Não	Não	100%
>>Embedded	Não	Não	100%
>>Testing Level(s)	Acception Testing System Testing Teste Funcional		0%
>>Required Skills	Knowledge on Object-oriented Paradigm Knowledge on Software Modeling Knowledge on UML Management of Testing Process		0%
>>Structural / Behavioral Model	UML 2.0 Models		0%

Figura 6-18 – Comparação de atributos entre SiGIC e critério de parada

6.7.4 Análise dos resultados - SiGIC

De forma semelhante à realizada para os outros projetos a corretude do grau de adequação deve ser verificada. Dessa forma, o projeto de software (v1) e o critério de parada selecionado (v2) são representados através de vetores sendo os índices de 1 a 7 mapeados de acordo com a Tabela 4-2 demonstrada no Capítulo 4.

	1	2	3	4	5	6	7
v1	0,142858	0,142858	0,142858	0,142858	0,142858	0,142858	0,142858

	1	2	3	4	5	6	7
v2	0,142858	0,142858	0,142858	0,142858	0	0	0

O cálculo da distância euclidiana e do grau de adequação é demonstrado a seguir.

Cálculo da distância e nível de adequação

$$d(Pr, Cp.A) = \sqrt{(0{,}142858)^2 + (0{,}142858)^2 + (0{,}142858)^2 + (0{,}142858)^2 + (0)^2 + (0)^2 + (0)^2}$$

$$d(Pr, Cp.A) = \sqrt{0{,}061225224}$$

$$d(Pr, Cp.A) = 0{,}247437314$$ **(Distância)**

Logo, Grau de Adequação é: $1 - 0{,}247437314 = 0{,}752562686$ ou **75% de adequação**

Assim, a equivalência entre o grau de adequação calculado anteriormente (75%) e o grau de adequação calculado pela implementação do procedimento (75%) demonstram que o cálculo foi realizado corretamente.

Pode-se considerar o critério aplicável ao projeto SiGIC, primeiramente porque o critério é não apriorístico e o projeto não define confiabilidade requerida, orçamento ou prazo para as atividades de testes. Além disso, o projeto está sendo desenvolvido para ser comercializado para apenas um cliente e o critério de parada foi proposto para ser aplicado exatamente nesse contexto.

O software não ser de missão crítica e não ser embarcado também são características que contribuíram para que esse critério de parada fosse escolhido, pois o critério de parada é proposto para softwares com essas características.

Uma característica importante do projeto diz respeito aos níveis de testes planejados. Como pode ser observado através da Figura 6-18 nenhum dos níveis de testes planejado pelo projeto é contemplado pelo critério de parada. Isso porque, o critério de parada não especifica os níveis de testes que ele pode ser executado. Nesse caso, há um risco na utilização do critério de parada, pois pode estar sendo aplicado para um nível de teste no qual não foi planejado.

Através desse caso de observação foi possível perceber também que os atributos “Modelo comportamental ou estrutural”, “Habilidades necessárias” e “Níveis de testes” também não foram preenchidos na caracterização do critério de parada apenar de terem sido preenchidos na caracterização do projeto. Assim, reforçando a hipótese de que pode haver um problema na escolha desses atributos para o cálculo do grau de adequação.

6.8 Conclusão

Esse capítulo apresentou uma prova de conceito (1 exemplo, 4 casos de observação e uma aplicação em software real) na qual o procedimento para sugestão de critérios de parada *Porantim CP* foi aplicado. A finalidade de executar a prova de conceito em diferentes cenários foi avaliar como o *Porantim* CP se comporta em diferentes contextos de software. Além disso, foi possível concluir que o procedimento é viável de ser utilizado e em todos os casos apresentados sugeriu critérios de parada que poderiam ser utilizados nos projetos de software.

Para cada um dos casos de observação a conferência do cálculo do grau de adequação foi realizada e em todos eles foi constatado que o cálculo foi realizado de maneira correta. Uma avaliação mais adequada para o procedimento seria calcular o grau de adequação de todos os critérios de parada contidos na base de conhecimento

em relação a cada projeto utilizado e posteriormente verificar se os critérios de parada sugeridos realmente eram os mais adequados para o projeto. Entretanto, essa opção se demonstrou inviável tendo em vista o grande número de critérios de parada contidos na base de conhecimento e os prazos disponíveis para execução do trabalho.

Ainda em relação ao grau de adequação, percebeu-se que sempre resultavam valores altos, mesmo quando poucos atributos entre projeto e critério de parada eram correspondentes. Por exemplo, em casos em que apenas 3 dos 7 atributos são correspondentes o grau de adequação é de 71%. Isso se deve a forma como é calculado o grau de adequação e também é influenciado pelo fato todos os atributos estarem com o mesmo peso. Sabendo disso, é preciso ser cuidadoso ao utilizar um critério de parada sugerido pelo procedimento e ao longo da utilização desse critério, avaliar sua real eficiência.

Outro fato verificado foi que, de acordo com a perspectiva do pesquisador, o procedimento sempre sugeriu critérios de parada adequados ao perfil do projeto. Ou seja, para projetos que definiam um orçamento ou prazo para as atividades de teste ou projetos que definiam a confiabilidade a ser atingida pelo software sempre tiveram critérios apriorísticos sugeridos com essa mesma característica. Por exemplo, no caso de observação do projeto de software de vídeo locadora definiu-se que o prazo para as atividades de testes seria de 2 meses e o critério sugerido para esse projeto era um critério apriorístico que define justamente o prazo para as atividades de testes como parâmetro de entrada. O mesmo aconteceu para os projetos que não definiram nenhuma dessas informações. Nesses casos, critérios de parada não apriorísticos foram sugeridos.

Um dos casos de observação sugeriu três critérios de parada com o mesmo grau de adequação. Nesse caso, foi preciso analisar informações detalhadas no corpo de conhecimento para decidir qual critério era mais apropriado para o software em questão. Essa análise acaba sendo subjetiva, pois depende da forma como o responsável pela decisão consegue assimilar as informações disponíveis no corpo de conhecimento e também dos detalhes disponíveis sobre o projeto de software.

Também é importante ressaltar que alguns atributos possuem características particulares. Por exemplo, os atributos “Software embarcado” e “Software de missão crítica”, quando estão preenchidos e analisados a partir do gráfico radar , podem dar a falsa impressão de que ambos o projeto e o critério de parada atendem satisfatoriamente estas propriedades. Entretanto, o que esse resultado representa quando está preenchido é que o critério de parada e o projeto possuem o mesmo valor, ou seja, o resultado indica o percentual de concordância entre os atributos e não especificamente a caracterização do projeto e/ou critério de parada. Por exemplo, no

caso de observação do Projeto F tanto o projeto quando os critérios de parada escolhidos não são para sistemas embarcados, portanto esse atributo é preenchido no gráfico radar (Figura 6-4) e atingem o nível de 100% (apresentam valores semelhantes).

Outro fato relacionado aos atributos é que alguns podem ser considerados binários, ou seja, são totalmente adequados ao projeto ou não são nada adequados. Os atributos “Objetivo”, “Contexto de aplicação”, “Software de missão crítica” e “Software embarcado” se enquadram nessa classificação. Entretanto, existem atributos que podem ser parcialmente adequados ao projeto. Por exemplo, o projeto pode estar planejado para ter duas fases de testes e o critério de parada só contempla uma dessas fases. Nesse caso, o atributo é apenas 50% adequado ao projeto. Os atributos “Modelos estruturas e comportamentais”, “Habilidades requeridas” e “Níveis de teste” se enquadram nessa classificação.

Ainda em relação aos atributos, constatou-se que para os projetos reais os atributos “Modelos estruturais e comportamentais”, “Habilidades necessárias” e “Níveis de testes” são sempre subutilizados. Analisando o motivo da não utilização desses atributos na maioria dos casos, percebeu-se que o problema está na caracterização dos critérios de parada. Isso porque, a maioria dos critérios não informa valores para esses atributos. Portanto, a utilização desses atributos pode ser avaliada em trabalhos futuros.

Apesar da execução da prova de conceito ter sido realizada pelo próprio desenvolvedor do software, percebeu-se oportunidades de melhoria em relação à usabilidade do software. No momento da escolha do atributo “Modelos comportamentais e estruturais” na caracterização do projeto, a quantidade de modelos exibida é excessiva. Isso leva à dificuldade de encontrar e selecionar os modelos desejados. Uma mudança nessa parte da implementação poderia aumentar a facilidade de interação com o usuário.

Por fim, pode-se dizer que o procedimento *Porantim CP* oferece auxílio na escolha de critérios de parada de testes de software, porém a decisão final cabe ao engenheiro de software, tendo em vista as avaliações adicionais eventualmente necessárias e de difícil implementação computacional. Por isso, trabalhos futuros no sentido de melhor definir os atributos de comparação entre projeto e critério de parada e o peso (importância) desses atributos no processo também devem ser considerados.

7 Conclusão e Trabalhos Futuros

Neste capítulo as conclusões desta dissertação são apresentadas, resumindo sua motivação e proposta, e destacando suas contribuições. Os trabalhos futuros indicam direções que podem ser tomadas no sentido de dar continuidade à pesquisa aqui apresentada.

7.1 Considerações Finais

A inviabilidade de execução de testes exaustivos que comprovem a corretude de um software leva ao problema de indecisão do melhor momento para encerrar essa atividade. Dessa forma, a definição de um critério de parada para as atividades de teste deve ser considerada para que essa decisão não seja tomada de forma empírica. A execução de uma *quasi*-revisão sistemática da literatura resultou na identificação de diversos critérios de parada que podem ser utilizados nesse contexto.

Dessa forma, esse trabalho apresentou o planejamento, execução e os resultados de uma pesquisa sobre critérios de parada para testes de software. Devido à quantidade expressiva de critérios de parada encontrados na literatura técnica, foi definido um procedimento para auxiliar na seleção de um critério de parada para determinado projeto. Além disso, o procedimento foi implementado como um componente da infraestrutura computacional para planejamento e acompanhamento de testes Maraká.

Para que a implementação do procedimento fosse possível, um corpo de conhecimento contendo 74 critérios de parada idenificados na literatura técnica de engenharia de software foi organizado. Cada um dos critérios de parada que integram o corpo de conhecimento são caracterizados através de atributos como “tipo do critério”, “objetivo do critério de parada”, “contexto de aplicação”, “critério para software embarcado”, “critério para software de missão crítica”, “níveis de testes”, “modelos estruturais/comportamentais requeridos”, “Habilidades requeridas”.

Assim, após a organização do corpo de conhecimento foi apresentado o procedimento para auxílio à seleção de critérios de parada denominado *Poramtim CP*. O procedimento recebe como entrada um projeto caracterizado através de atributos como “objetivo da fase de testes”, “perfil de utilização”, “plataforma de execução”, “software de missão crítica”, “software embarcado”, “níveis de testes contemplados

pelo projeto", "habilidades disponíveis pelos integrantes da equipe de desenvolvimento", "modelos estruturais/comportamentais criados para o projeto". Com o projeto caracterizado, o procedimento realiza uma consulta ao corpo de conhecimento que contém todos os critérios de parada caracterizados e utiliza um mecanismo que avalia o grau de adequação entre o projeto de software e cada um dos critério de parada. Por fim, o procedimento sugere à equipe de testes os critérios de parada que são mais adequados ao projeto.

Ao final, esse trabalho apresenta uma prova de conceito do uso de *Porantim CP* em cinco casos distintos. A prova de conceito foi executada com a finalidade de realizar uma avaliação prévia sobre a possibilidade de utilização do procedimento.

Apesar da grande quantidade de critérios de parada encontrados, nenhum deles responde definitivamente a questão de quando parar os testes de software. Os critérios de parada para testes de software herdaram características de critérios de parada utilizados em testes de produtos tais como no desenvolvimento de hardware. Entretanto, as características intrínsecas ao software (não sofrer desgaste, por exemplo) nos fazem acreditar que as abordagens utilizadas para a criação dos critérios de parada não são adequadas. Critérios de parada para testes de software poderiam ser mais eficientes ao considerar dentre outros fatores a cobertura dos testes, a forma de condução dos testes, e as características do processo de desenvolvimento ou do ecosistema no qual o software está inserido.

7.2 Contribuições da Pesquisa

As principais contribuições dessa dissertação são apresentadas a seguir.

- **Definição de um protocolo de pesquisa (*quasi*-revisão sistemática) com a finalidade de identificar e caracterizar critérios de parada para testes de software:**

Foi executada uma *quasi*-revisão sistemática da literatura com a finalidade de identificação e caracterização de critérios de parada para testes de software. Para a realização dessa revisão um protocolo foi criado. Esse protocolo pode ser utilizado ou estendido em novas pesquisas sobre o tema e também para a atualização da pesquisa apresentada nessa dissertação.

- **Criação de um corpo de conhecimento sobre critérios de parada para testes de software:**

Após a quasi-revisão sistemática da literatura foi organizado um corpo de conhecimento com 74 critérios de parada encontrados. Os critérios de parada

armazenados nesse corpo de conhecimento são caracterizados através de atributos. Esses atributos foram identificados de maneira que auxiliem na execução do procedimento proposto nesse trabalho.

Esse corpo de conhecimento é representado através de um banco de dados relacional para que a implementação do procedimento *Porantim CP* fosse possível. Além disso, informação adicionais sobre os critérios de parada também podem ser acessadas através dos formulários de extração da revisão. Essas informações podem ser acessadas através da própria implementação do procedimento e também estão disponibilizadas como anexo (Apêndice C) nesse trabalho.

Além disso, o corpo de conhecimento pode ser considerado extendivel. Ou seja, em caso de identificação de novos critérios de parada, é possível inseri-los no corpo de conhecimento. Esses novos critérios de parada podem ser encontrados, por exemplo, pela reexecução da quasi-revisão sistemática.

- **Definição do procedimento *Porantim CP*:**

Foi proposto um procedimento para auxiliar a seleção de critérios de parada para testes de software: *Porantim CP*. Esse procedimento foi baseado no procedimento proposto por DIAS NETO (2009) que tem a finalidade de apoiar a seleção de técnicas de teste baseado em modelos.

Porantim CP consiste em um procedimento que recebe como entrada um projeto caracterizado através de atributos, realiza a comparação desse projeto com diversos critérios de parada contidos no corpo de conhecimento e, após isso, sugere os critérios de parada mais adequados ao projeto em questão.

Além disso, inserindo um critério de parada novo no corpo de conhecimento o procedimento passa a considera-lo automaticamente.

- **Implementação do procedimento Porantim CP na infraestrutura Maraká:**

O procedimento *Porantim CP* foi implementado como um componente da infraestrutura Maraká. Essa implementação viabiliza a execução do procedimento com um corpo de conhecimento contendo uma quantidade expressiva de critérios de parada.

A implementação do procedimento foi avaliada através de uma prova de conceito.

7.3 Limitações

Diversos aspectos podem influenciar na seleção de uma tecnologia. Nesse trabalho, escolhemos um conjunto de características para representar critérios de parada e calcular o grau de adequação com projetos de software. Esse conjunto de características foi proposto e, em seguida, avaliado por um especialista da área de testes. Entretanto, esse conjunto de características, na perspectiva de outros pesquisadores, pode não ser o mais adequado para caracterizar os critérios de parada e serem utilizados no cálculo de adequação.

Além disso, os atributos de caracterização foram preenchidos a partir de informações retiradas dos artigos retornados pela *quasi*-revisão sistemática. Eventualmente, esses artigos podem não explicitar valores para todos os atributos necessários à caracterização.

Outra limitação está relacionada à avaliação do procedimento e da implementação do procedimento. Por questões de prazo, não foi possível executar estudos experimentais para avaliação da viabilidade do procedimento proposto bem como da ferramenta de apoio, embora tenha ocorrido uma avaliação inicial através de prova de conceito.

7.4 Trabalhos Futuros

Este trabalho resultou na criação de um procedimento para auxílio à seleção de critérios de parada para testes de software, além de um componente responsável pela implementação desse procedimento que se integra a uma infraestrutura computacional para gerenciamento e acompanhamento de testes.

O corpo de conhecimento, constituído pelos critérios de parada encontrados através de uma *quasi*-revisão sistemática é utilizado como fonte de dados para o procedimento de seleção. Os critérios de parada contidos no corpo de conhecimento são caracterizados através de atributos. Esses atributos foram escolhidos pelo autor desse trabalho com base no conhecimento adquirido através da revisão executada e dos estudos realizados sobre o tema. Além disso, os atributos foram avaliados por um especialista em testes de software. Entretanto, uma avaliação mais formal dos atributos poderia ser realizada. Para isso, um survey com especialistas na área de teste de software poderia ser executado. Nesse survey, poderia ser avaliado se os atributos escolhidos estão adequados e se algum atributo deve ser incluído.

Além disso, o survey mencionado anteriormente poderia servir também para atribuir pesos para cada um dos atributos. Esses pesos fariam os atributos terem mais ou menos força no procedimento de escolha dos critérios de parada.

Outro trabalho que pode ser executado é a evolução e reexecução do protocolo de pesquisa. Com isso, novas fontes de pesquisa podem ser incluídas de forma que o corpo de conhecimento possa ser atualizado.

Essa pesquisa se fundamentou em uma metodologia baseada em evidências (SHULL *et al.* 2001; MAFRA *et al.* 2006; SPÍNOLA 2008) que se divide em concepção e avalição da abordagem proposta. Nesse trabalho, as atividades referentes à etapa de concepção foram executadas. Entretanto, apesar da execução de uma prova de conceito demonstrando a execução do procedimento proposto, não foi possível executar estudos experimentais para avaliar a viabilidade do procedimento *Porantim CP* bem como da ferramenta que implementa o procedimento.

Assim, consideramos necessária a realização de estudos experimentais em diferentes contextos para avaliar o procedimento proposto e reduzir os riscos para sua introdução na indústria.

REFERÊNCIAS BIBLIOGRÁFICAS

BAUER, FRIEDRINCH, (1969), “Software Engineering: Report on a conference sponsored by the NATO SCIENCE COMMITTEE”. Garmisch, Alemanha, Outubro.

BINDER, R. V., (2006), “Testing Object-Oriented Systems – Models, Patters, and Tools”. Canada: Addison-Wesley

BIOLCHINI, J. C. D. A., MIAN, P. G., NATALI, A. C. C., CONTE, T., TRAVASSOS, G. H. (2007). "Scientific research ontology to support systematic review in software engineering" Advanced Engineering Informatics, v. 21, n. 2, pp. 133-151.

BOLAND, P. J., CHUIV, N. N., (2007), “Optimal Times for Software Release When Repair is Imperfect”, Statistics & Probability Letters, vol. 77, 1176-1185

BOLDRINI, J. L., COSTA, S. R., FIGUEIREDO, V. L., et al. (1980), —Álgebra Linear‖, 3 ed., capítulo 8, Harper & Row do Brasil.

CAMUFFO, M., MAIOCCHI, M., MORSELLI, M., (1990), “Automatic software test generation”. Information Software. Technology. 32 (5), 337–346.

CHANG, W.-K. E JENG, S.-L., (2007) “Practical stopping criteria for validating safety-critical software by estimating impartial reliability”, Applied Mathematical Modelling, 31, 1411 – 1424.

CHÁVEZ, T., (2000) “A Decision-Analytic Stopping Rule for Validation of Commercial Software Systems”, IEEE Transactions on Software Engineering.

CHO, B. C., PARK, K. S., (1994), “An Optimal Time for Software Testing Under the User’s Requirement of Failure-Free Demonstration Before Release”, IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences, vol. E77-A, 563-570

DALAL, S. R., MCINTOSH, A. A, (1994), “When to Stop Testing for Large Software Systems with Changing Code”, *IEEE Transactions on Software Engineering*, vol. 20, 318-323

DALAL, SIDDHARTHA R., MALLOWS, COLIN L., (1990) “Some Graphical Aids for Deciding When to Stop Testing Software”, IEEE Journal on Selected Areas in Communications.

DIAS NETO, A. C., (2006) "Uma Infra-estrutura Computacional para Apoiar o Planejamento e Controle de Testes de Software", Dissertação de M.Sc., COPPE/UFRJ, Abril.

DIAS-NETO, A. C.; TRAVASSOS, G. H. (2006), —Maraká: Uma Infra-estrutura Computacional para Apoiar o Planejamento e Controle de Testes de Softwarell. In: Simpósio Brasileiro de Qualidade de Software, Vila Velha-ES, Junho.

DIAS NETO, A. C., SPÍNOLA, R. O., BOTT, A., TRAVASSOS, G. H. (2007). "Estratégia de Teste de Software no Desenvolvimento Incremental de um Sistema de Informação". In: Workshop on Systematic and Automated Software Testing, João Pessoa – PB, Brasil.

DIAS NETO, A. C., (2009) "Seleção de Técnicas de Teste Baseado em Modelos", Tese de D.Sc., COPPE/UFRJ, Novembro.

DI LUCCA, G.A., FASOLINO, A.R. (2006) "Testing Web Applications: The state of the art and future trends", J. Inf. Softw. Technol. 48(2), p. 1172–1186.

DIJKSTRA, E. W. (1972). "The Humble Programmer". Communications of the ACM 15 (10): 859–866. doi:10.1145/355604.361591. (EWD340) PDF, ACM Turing Award lecture

GARG, M., LAI, R., HUANG JEN, S., (2010), "When to Stop Testing: A Study from the Perspective of Software Reliability Models", IET Software, Vol. 5, pp. 263-273

GOEL, A. L., (1978), "A Software Error Detection Model with Applications", 2nd Software Life Cycle Management Workshop, sponsored by U.S. Army, Atlanta, Georgia

GOEL, A. L., (1985) "Software Reliability Models: Assumptions, Limitations, and Applicability," IEEE Trans. Software Eng., vol. 11, pp. 1411-1423.

GOKHALE, S., (2003). "Optimal Software Release Time Incorporating Fault Correction", Software Engineering Workshop. 28th Annual NASA Goddard, 175-184

GOKHALE, S., LYU, M. e TRIVEDI, K., (2006), "Incorporating Fault Debugging Activities into Software Reliability Models: a Simulation Approach", IEEE Transactions on Reliability, vol. 55, 281-292

HOU, R., KUO, S., CHANG, Y., (1997), "Optimal Release Times for Software Systems with Scheduled Delivery Time Based on the HGDM", *IEEE TRANSACTIONS ON COMPUTERS*, vol. 46, 216-221

HUANG, C. Y. e LIN, C. T., (2006), “Software Reliability Analysis by Considering Fault Dependency and Debugging Time Lag”, IEEE Transactions on Reliability, vol. 55, 436–450

HUANG, C. Y., LIN, C.-T. (2010). “Analysis of software reliability modeling considering testing compression factor and failure-to-fault relationship”, IEEE Transactions on Computers, 59, 283 - 288.

HSU, C. J., HUANG, C. Y., CHANG, J. R., (2011), “Enhancing Software Reliability Modeling and Prediction Through the Introduction of Time-Variable Fault Reduction Factor”, Applied Mathematical Modelling, vol. 35, 506-521

IEEE Standard 610-1990: IEEE Standard Glossary of Software Engineering Terminology, IEEE Press

JELINSKI, Z., MORANDA, P., (1972), “Software Reliability Research”, Statistical Computer Performance Evaluation, 465-484

KAPUR, P., XIE, M., GARG, R. e JHA, A., (1994), “A Discrete Software Reliability Growth Model with Testing Effort”, International Conference on Software Testing, Reliability and Quality Assurance, STRQA, 16-20

KIM, K. e WU, C. A., (1994) “software reliability model in the embedded system”, Software Testing, Reliability and Quality Assurance, 1994. Conference Proceedings., First International Conference on, 59 -63

KITCHENHAM, B., (2004). “Procedures for Performing Systematic Reviews”, Joint Technical Report Keele University TR/SE-0401 and NICTA Technical Report 0400011T.1, Keele University and NICTA.

KOCH, H. S., KUBAT, P., (1983), “Optimal Release Time of Computer Software”, IEEE Transactions on Software Engineering, vol. SE-9, 323-327

LEUNG, Y., (1992), “Optimum Software Release Time with a Given Cost Budget”, Journal of Systems and Software, vol. 17, 233-242

LITTLEWOOD, B., WRIGHT, D., (1995) “Stopping Rules for the Operational Testing of Safaety-Critical Software”.

LYU, M. R., (1996). “Handbook of Software Reliability Engineering”, McGraw-Hill.

MAFRA, S.N., BARCELOS, R.F., TRAVASSOS, G.H. (2006), “Aplicando uma Metodologia Baseada em Evidência na Definição de Novas Tecnologias de Software”, XX SBES, pp. 239-254.

MALDONADO, E. D.; DELAMARO, J.C.; JINO, M., (2007), “Introdução ao Teste de Software”. Rio de Janeiro: Campus

MALEVRIS, N. AND PETROVA, E., (2000) “On the Determination of an Appropriate Time for Ending the Software Testing Process”, First Asia-Pacific Conference on Quality Software.

MATHUR A. P., (2008), “Foundations of Software Testing”, 1ª Edition. Addison-Wesley Professional.

MASUDA, Y., MIYAWAKI, N., SUMITA, U. e YOKOYAMA, S., (1989), “A statistical approach for determining release Time of Software System with Modular Structure Reliability”, IEEE Transactions on, 365 -372

MCCABE, T. J., (1976), “A Complexity Measure”. IEEE Transactions on Software Engineering, Vol. SE-2, No. 4

MUSA, J.D., (1993), “Operational profiles in software reliability engineering”, IEEE Software, 10, pp. 14-32.

MYERS, G.J, (1979), “The Art of Software Testing”. Jonh Wiley & Sons, New York, NY

MYERS, G. J., SANDLER, C., BADGETT, T., THOMAS, T. M., (2004), ”The Art of Software Testing”. 2 ed., John Wiley & Sons, Nova York, NY, EUA

OKUMOTO, K e GOEL, A., (1979a), “ A time Dependent Error Detection Rate Model for Software Reliability and Other Performance Measures”, IEEE Transactions on Reliability, Special Issue on Software Reliability

OKUMOTO, K. e GOEL, A., (1979b), “Optimum Release Time for Software Systems”, Computer Software and Applications Conference. Proceedings. COMPSAC 79. The IEEE Computer Society's Third International, 500 - 503

OHTERA, H., e YAMADA, S., (1990), “Optimum Software-Release Time considering an Error-Detection Phenomenon During Operation”, IEEE Transactions on Reliability

OZEKICI, S., ALTINEL, I., OZCELIKYUREK, S., (2000), “Testing of Software with an Operational Profile”, *Naval Research Logistics,* vol. 47, 620-634

PAI, M. MCCULLOCH, M. GORMAN, J.D. *et al.* (2004) “Systematic Reviews and meta-analyses: An illustrated, step-by-step guide”, The National Medical Journal of India, vol. 17, n.2.

PETERS, J. E., PEDRYCZ, W., (2001), “Engenharia de Software: Teoria e Prática”. Rio de Janeiro: Campus.

PFLEEGER, SHARI; ATLEE, JOANNE, (2006), “Software Engineering”. 3ª Edição. New Jersey: Pearson Prentice Hall.

PHAM, H., ZHANG, X., (1999), “A Software Cost Model with Warranty and Risk Costs”, IEEE Transactions on Computers, vol. 48, 71-75

PHAM, H., (2003). “Handbook of Reliability Engineering”, Springer.

PRESSMAN, ROGER, (2006), “Engenharia de Software”. 6ª Edição. São Paulo: McGraw-Hill.

PROWELL, S.J., (2004), “A Cost-Benefit Stopping Criterion for Statistical Testing”.

ROCHA, A. R. C., MALDONADO, J. C., WEBER, K. C. (2001), “Qualidade de software - Teoria e prática ll”, Prentice Hall, São Paulo.

SANTA ISABEL, S. L., (2011), “Seleção de Abordagens de Teste para Aplicações Web”, Dissertação de M.Sc., COPPE/UFRJ, Julho.

SCHNEIDEWIND, N., (2001), “Modelling the Fault Correction Process”, Proceedings of the International Symposium on Software Reliability Engineering, 185-190

SHOOMAN, M. L., (1983), “Software Engineering: Reliability, Development, and Management”. New York: McGraw-Hill, Inc.

SHULL, F., CARVER, J., TRAVASSOS, G. (2001), “An Empirical Methodology for Introducing Software Processes”, No: Joint 8th ESEC and 9th ACM SIGSOFT FSE-9, pp. 288-296.

SANTOS, P. S. M., TRAVASSOS, G. H. (2010). “Inspeção de Qualidade em Descrições de Casos de Uso: uma Avaliação Experimental em um Projeto Real”, Anais do IX Simpósio Brasileiro de Qualidade de Software, pp. 261-275, Belém, Brasil.

SOFTEX: MPS.BR (2007) “Melhoria de Processo do Software Brasileiro”. Guia Geral Versão 1.2, Campinas, SP.

SOMMERVILLE, IAN, (2003), “Engenharia de Software”. 6ª Edição. São Paulo: Pearson Prentice Hall.

SPÍNOLA, R. O., DIAS-NETO, A. C., TRAVASSOS, G. H. (2008), “Abordagem para Desenvolver Tecnologia de Software com Apoio de Estudos Secundários e Primário”, Experimental Software Engineering Latin American Workshop (ESELAW), Salvador, Novembro.

STAA, A. V. . Instrumentação. In: Rocha, A. R. C.; Maldonado, J. C.; Weber, K. C.;. (Org.). Qualidade de Software - Teoria e Prática. São Paulo: Prentice Hall, 2001, v. , p. 226-237.

TENG, X. E PHAM, H., (2004), “A Software Cost Model for Quantifying the Gain with Considerations of Random Field Environments”, IEEE Transactions on Computers, 380 - 384

TRAVASSOS, G. H., SANTOS, P. M., MIAN, P. G., DIAS NETO, A. C., BIOLCHINI, J. (2008) "An Environment to Support Large Scale Experimentation in Software Engineering," Engineering of Complex Computer Systems, IEEE International Conference on, pp. 193-202, 13th IEEE International Conference on Engineering of Complex Computer Systems (iceccs 2008).

VEGAS, S.; BASILI, V., (2005), A Characterization Schema for Software Testing Techniques, Empirical Software Engineering, Outubro, v.10 n.4, p.437-466

WHITTAKER, J.; THOMASON, M., (1994), “A Markov chain model for statistical software testing Software Engineering”, IEEE Transactions on, 20, 812 -824

WOJCICKI, M. A.; STROOPER, P. (2007), “An Iterative Empirical Strategy for the Systematic Selection of a Combination of Verification and Validation Technologies”. Proceedings of the 5th international Workshop on Software Quality (May 20 - 26). International Conference on Software Engineering. DOI= http://dx.doi.org/10.1109/WOSQ.2007.4

XAVIER, J. R.; WERNER, C.M.L.; TRAVASSOS, G.H.; (2002), “Uma Abordagem para a Seleção de Padrões Arquiteturais Baseada em Características de Qualidade II”, XVI Simpósio Brasileiro de Engenharia de Software, Gramado, RS, Brasil.

XIE, M. e YANG, B., (2003) “A study of the effect of imperfect debugging on software development cost Software Engineering”, IEEE Transactions on, 29, 471-473.

YAMADA, S. e OSAKI, S., (1985) “Cost-reliability optimal release policies for software systems”. IEEE Transactions on Reliability, R-34, 422 – 424.

APÊNDICE A – Protocolo da *quasi*-Revisão Sistemática

Este apêndice apresenta o protocolo utilizado na quasi-revisão sistemática da literatura.

CRITÉRIOS DE PARADA PARA PROCESSOS DE TESTE DE SOFTWARE

Protocolo Versão 1.5 preparado por:

Guilherme Horta Travassos
Victor Vidigal Ribeiro
Grupo de Engenharia de Software Experimental da COPPE/UFRJ
Março, 2011

Histórico de Revisões

Data	Versão	Descrição	Autores
01/03/2011	1.0	Criação do protocolo	Guilherme Horta Travassos /Victor Vidigal Ribeiro
18/07/2011	1.1	Redefinição da *string* de busca e critérios de inclusão e exclusão de artigos	Guilherme Horta Travassos /Victor Vidigal Ribeiro
10/08/2011	1.2	Definição da *string* de busca, redefinição das máquinas de busca, alterações e alteração dos critérios de inclusão e exclusão	Guilherme Horta Travassos /Victor Vidigal Ribeiro
15/08/2011	1.3	Redefinição da *string* de busca e inclusão de máquinas de busca	Guilherme Horta Travassos /Victor Vidigal Ribeiro
17/09/2012	1.4	Fechamento do protocolo	Guilherme Horta Travassos /Victor Vidigal Ribeiro
10/06/2013	1.5	Correção do critério de avaliação de qualidade dos artigos: mudança de estudo de caso para prova de conceito	Victor Vidigal Ribeiro

1 Cenário de Investigação Principal

Com a evolução dos simples programas de computadores para softwares cada vez mais complexos criou-se a necessidade de atividades que permitam o desenvolvimento de software com mais qualidade. Essas atividades têm como principal finalidade assegurar que o software seja desenvolvido de maneira que possa resolver os problemas que foram designados a ele [PRESSMAN, 2006].

Uma dessas atividades são os testes de software que pertencem a um tópico da engenharia de software chamado Verificação, Validação e Testes (VV&T). Os testes podem ser definidos como uma análise dinâmica do produto, ou seja, na sua execução, com o objetivo de revelar falhas e a partir delas se encontrar os defeitos presentes no software. Com isto, os testes contribuem para uma futura detecção de defeitos por meio de um processo de depuração e, posteriormente, a correção deste defeito contribuindo para a qualidade do software [ROCHA *et al.*, 2001].

A importância dos testes de software é enfatizada por diversos autores. Por exemplo, Perry [1995] diz que não é adequado desenvolver software sem testes, pois o processo de desenvolvimento perfeito não existe e a probabilidade de existir, em um futuro previsível, é perto de zero. Enquanto os processos de desenvolvimento e manutenção de software continuarem a inserir defeitos no software, testes continuarão a ser um componente muito importante no processo de desenvolvimento. Pressman [2006] afirma que as atividades de teste possuem um papel fundamental no processo de desenvolvimento de um software e atuam como um mecanismo de apoio à garantia da qualidade do produto, correspondendo ao último recurso para avaliação do produto antes da sua entrega ao usuário final.

Além de ser considerada uma atividade importante pela área acadêmica, a presença de atividades de testes nos modelos de maturidade mostra a importância que a indústria atribui aos testes de software. Por exemplo, o extinto modelo CMM – *Capability Maturity Model* – [SEI, 1993], desenvolvido pelo SEI (*Software Engineering Institute*) como mecanismo de avaliação do processo de desenvolvimento de organizações já previa atividades de testes a partir do nível 2 (Repetível) e o CMMI - *Capability Maturity Model Integration* – uma evolução do modelo CMM, prevê atividades de testes a partir do Nível 3 [CMMI, 2000]. Enquanto o MPS.BR – Melhoria do Processo de Software Brasileiro – que tem como objetivo a definição e disseminação de um modelo de referência para melhoria do processo de software, prevê atividades de testes a partir do nível D (Definido) [MPS.BR, 2005].

Entretanto, por ser uma das atividades mais custosas do processo de desenvolvimento, os testes necessitam de um bom gerenciamento a fim de evitar perda de recursos, atrasos no cronograma, ou pior, não contribuir para avaliação do produto e, com isto, não ajudar na sua qualidade [JURISTO *et al.*, 2004]. Devido a limites tecnológicos, de tempo ou financeiros, também pode ser impossível testar por completo um software que tenha um mínimo de complexidade. Binder [1999] exemplifica o grande número de combinações de parâmetros de entradas e o tempo excessivo que demoraria para testar todas essas combinações através de uma classe que representa um triângulo. Esta classe recebe três parâmetros e cada um deles define duas coordenadas para se formar uma linha. Limitando os valores de entradas a números inteiros que variam de 1 a 10, existem 10^4 possibilidades de combinações possíveis que formam uma linha. Considerando que para formar um triangulo precisa-se de três linhas este número seria $10^4 \times 10^4 \times 10^4 = 10^{12}$. Supondo que seja possível testar 1000

combinações de entrada por segundo, 24 horas por dia, 365 dias por ano os testes demorariam $10^{12} / 10^{3} = 10^{9}$ segundos. Como um ano tem aproximadamente 3171 x 10^{7} segundos os testes demorariam cerca de $10^{9} / 3171 \times 10^{7} = 3171 \times 10^{2}$ segundos que correspondem à aproximadamente 317 anos.

Como é inviável, e na prática impossível, executar todas as combinações possíveis de um software, não é possível provar que um software esteja isento de falhas através de testes de software. Por outro lado, é possível mostrar que um software possui defeitos. Assim, o que é proposto é que seguindo um processo de teste com atividades e critérios bem definidos é possível que o software atinja um nível de confiança adequado e se comporte corretamente para grande parte do seu domínio de entrada [DELAMARO, 2007].

Portanto, não é possível provar que um software está livre de defeitos através de testes de software mesmo utilizando critérios de testes bem definidos, pois sempre é possível existir um cenário de teste que revele algum defeito. É neste contexto que esta *quasi*-revisão sistemática está inserida, objetivando o levantamento dos artifícios que são utilizados para determinar quando o software atinge uma confiabilidade adequada e os testes podem ser parados para que o software seja entregue ao cliente. Em outras palavras, o objetivo desta revisão é identificar os critérios de parada para processos de teste de software.

2 Protocolo de Busca

O protocolo de revisão sistemática inicial é baseado em [Biolchini et al., 2005]. Para organizar e estruturar a string de busca, é explorado a abordagem PICO [Pai et al., 2004]. Esta abordagem separa a questão em quarto partes: População (**P**opulation), Intervenção (**I**ntervention), Comparação (**C**omparison) e Saída (**O**utcome). Devido ao objetivo do estudo não é possível aplicar nenhuma comparação. Portanto, é possível classificar esta revisão como *quasi*-revisão sistemática [Travassos et al, 2008].

2.1.1 Foco de Interesse

O Objetivo deste estudo é identificar critérios que possam ser utilizados para determinar em que momento os testes de um software devem ser encerrados para que o mesmo seja colocado em produção.

2.1.2 Qualidade e Amplitude da Questão

- **Problema:** testes são importantes para avaliar a qualidade do software. Entretanto, sabe-se que na prática não é possível provar a correção de um programa utilizando teste de software ou qualquer outro método. Assim há sempre a possibilidade de haver defeitos no software o que leva ao problema da indecisão do momento de parada dos testes de software.

- **Questões:**
  - **Questão Principal:** Quais critérios têm sido utilizados para determinar o momento de parada dos testes de software?

- **Criação *String* de busca:** Parte das palavras chaves que compõem a string de busca foram escolhidas com a ajuda de um pesquisador especialista na área. Outra parte foi baseada nos artigos de controle que foram encontrados através de uma busca *ad-hoc*. Estes artigos foram importantes para entender os termos utilizados pelos autores e forneceram um parâmetro para a criação da *string* de busca. Ao realizar a extração das palavras chaves dos artigos de controle pôde-se perceber que alguns artigos estavam indexados de forma inconsistente nas máquinas de busca, pois as palavras chaves impressas no artigo não correspondiam com as palavras chaves indexadas pelas máquinas de busca. Portanto, foi realizado um trabalho no sentido de extrair as palavras chaves pelas quais as máquinas de busca estavam indexando os artigos. Os seguintes artigos de controle foram utilizados:
  - Dalal, Siddhartha R., Mallows, Colin L., Some Graphical Aids for Deciding When to Stop Testing Software, IEEE Journal on Selected Areas in Communications, 1990.
  - Littlewood, Bev, Wright, David, Stopping Rules for the Operational Testing of Safaety-Critical Software, 1995.
  - Chávez, T., A Decision-Analytic Stopping Rule for Validation of Commercial Software Systems, IEEE Transactions on Software Engineering, 2000.
  - Malevris, N. and Petrova, E. , On the Determination of an Appropriate Time for Ending the Software Testing Process, First Asia-Pacific Conference on Quality Software, 2000.
  - Prowell, S.J., A Cost-Benefit Stopping Criterion for Statistical Testing, 2004.
- **População:** Projetos e processos de software.

  - **Palavras Chaves**
    - software application, software development, software project, software product, software system, software safety system, Safety-Critical Software, Computer Software, Software Components, project management, software measurement, software industry;

- **Intervenção:**
  - **Questão Principal:** Testes de software
  - **Palavras Chave**
    - software testing, statistical testing, testing requirement, testing process, program testing, reliability requirement, testing the software, testing procedure;

- **Comparação:** Nenhuma

– **Saída:** Conjunto de critérios que podem ser utilizados para a tomada de decisão de quando os testes devem parar.

  - **Palavras Chave**
    - decision analysis, degree of correctness, failure behaviour, probability of failure, reliability state, software assurance, stop testing, stopping analysis, stopping criteria, stopping criterion, stopping rule, testing costs, testing requirement, reliability model, optimal software release, release time;

– **Efeito:** Identificar critérios ou regras que possam ser utilizados para decidir quando parar os testes de software.

– **Aplicação:** Tornar a atividade de teste de software mais eficiente.

– **Projeto Experimental:** Nenhum método estatístico será aplicado.

2.2 Seleção de Fontes

2.2.1 Definição de Critérios para seleção de fontes:

– Artigos disponíveis na web através de máquinas de busca que permitam a pesquisa de strings no *abstract*, título e palavra chave.
– Livros sobre engenharia de software disponíveis na web ou impressos.
– Anais de conferencias brasileiras:
  - SBES: 1993, 1994, 1995, 1997, 1999, 2000, 2002, 2003, 2004, 2005, 2006, 2007, 2008;
  - SBQS: 2003, 2004, 2005, 2006, 2008, 2009, 2010;
  - ESELAW: 2008, 2009, 2010

2.2.2 Idioma das fontes: Inglês e Português.

2.2.3 Identificação das Fontes

– **Método de pesquisa das fontes:** busca no *abstract*, título e palavras chaves através das máquinas de busca disponíveis.
  - **String de busca:** ("software application" OR "software development" OR "software project" OR "software product" OR "software system" OR "software safety system" OR "Safety-Critical Software" OR "Computer Software" OR "Software Components" OR "project management" OR "software measurement" OR "software industry") AND ("software testing" OR "statistical testing" OR "testing requirement" OR "testing

process" OR "program testing" OR "reliability requirement" OR "testing the software" OR "testing procedure") AND ("decision analysis" OR "degree of correctness" OR "failure behaviour" OR "probability of failure" OR "reliability state" OR "software assurance" OR "stop testing" OR "stopping analysis" OR "stopping criteria" OR "stopping criterion" OR "stopping rule" OR "testing costs" OR "testing requirement" OR "reliability model" OR "optimal software release" OR "release time")

- **Máquinas de busca:**

Name	Link
Scopus	http://www.scopus.com/
IeeeXplore	http://ieeexplore.ieee.org/
EI Compendex	http://www.engineeringvillage.org/
Web of Science	http://isiknowledge.com/

  - **String de busca Scopus:** TITLE-ABS-KEY(("software application" OR "software development" OR "software project" OR "software product" OR "software system" OR "software safety system" OR "Safety-Critical Software" OR "Computer Software" OR "Software Components" OR "project management" OR "software measurement" OR "software industry") AND ("software testing" OR "statistical testing" OR "testing requirement" OR "testing process" OR "program testing" OR "reliability requirement" OR "testing the software" OR "testing procedure") AND ("decision analysis" OR "degree of correctness" OR "failure behaviour" OR "probability of failure" OR "reliability state" OR "software assurance" OR "stop testing" OR "stopping analysis" OR "stopping criteria" OR "stopping criterion" OR "stopping rule" OR "testing costs" OR "testing requirement" OR "reliability model" OR "optimal software release" OR "release time"))

  - **String de busca IeeeXplore:** ("software application" OR "software development" OR "software project" OR "software product" OR "software system" OR "software safety system" OR "Safety-Critical Software" OR "Computer Software" OR "Software Components" OR "project management" OR "software measurement" OR "software industry") AND ("software testing" OR "statistical testing" OR "testing requirement" OR "testing process" OR "program testing" OR "reliability requirement" OR "testing the software" OR "testing procedure") AND ("decision analysis" OR "degree of correctness" OR "failure behaviour" OR "probability of failure" OR "reliability state" OR "software assurance" OR "stop testing" OR "stopping analysis" OR "stopping criteria" OR "stopping criterion" OR "stopping rule" OR "testing costs" OR "testing requirement" OR "reliability model" OR "optimal software release" OR "release time")

- **String de busca EI Compendex:** ("software application" OR "software development" OR "software project" OR "software product" OR "software system" OR "software safety system" OR "Safety-Critical Software" OR "Computer Software" OR "Software Components" OR "project management" OR "software measurement" OR "software industry") AND ("software testing" OR "statistical testing" OR "testing requirement" OR "testing process" OR "program testing" OR "reliability requirement" OR "testing the software" OR "testing procedure") AND ("decision analysis" OR "degree of correctness" OR "failure behaviour" OR "probability of failure" OR "reliability state" OR "software assurance" OR "stop testing" OR "stopping analysis" OR "stopping criteria" OR "stopping criterion" OR "stopping rule" OR "testing costs" OR "testing requirement" OR "reliability model" OR "optimal software release" OR "release time")

- **String de busca Web of Science:** TS=(("software* application*" OR "software* development*" OR "software* project*" OR "software* product*" OR "software* system*" OR "software* safety system*" OR "Safety-Critical Software*" OR "Computer* Software*" OR "Software* Component*" OR "project* management*" OR "software* measurement*" OR "software* industr*") AND ("software* testing" OR "statistical testing" OR "testing requirement*" OR "testing process" OR "program testing" OR "reliability requirement*" OR "testing the software*" OR "testing procedure*") AND ("decision* analysis" OR "degree of correctness" OR "failure* behaviour" OR "stopping rule*" OR "probabilit* of failure*" OR "reliability state*" OR "software* assurance" OR "stop* testing" OR "stopping analysis" OR "stopping criteria" OR "stopping criterion*" OR "testing cost*" OR "testing requirement*" OR "reliability model*" OR "optimal software* release*" OR "release time*"))

2.3 Seleção dos Estudos

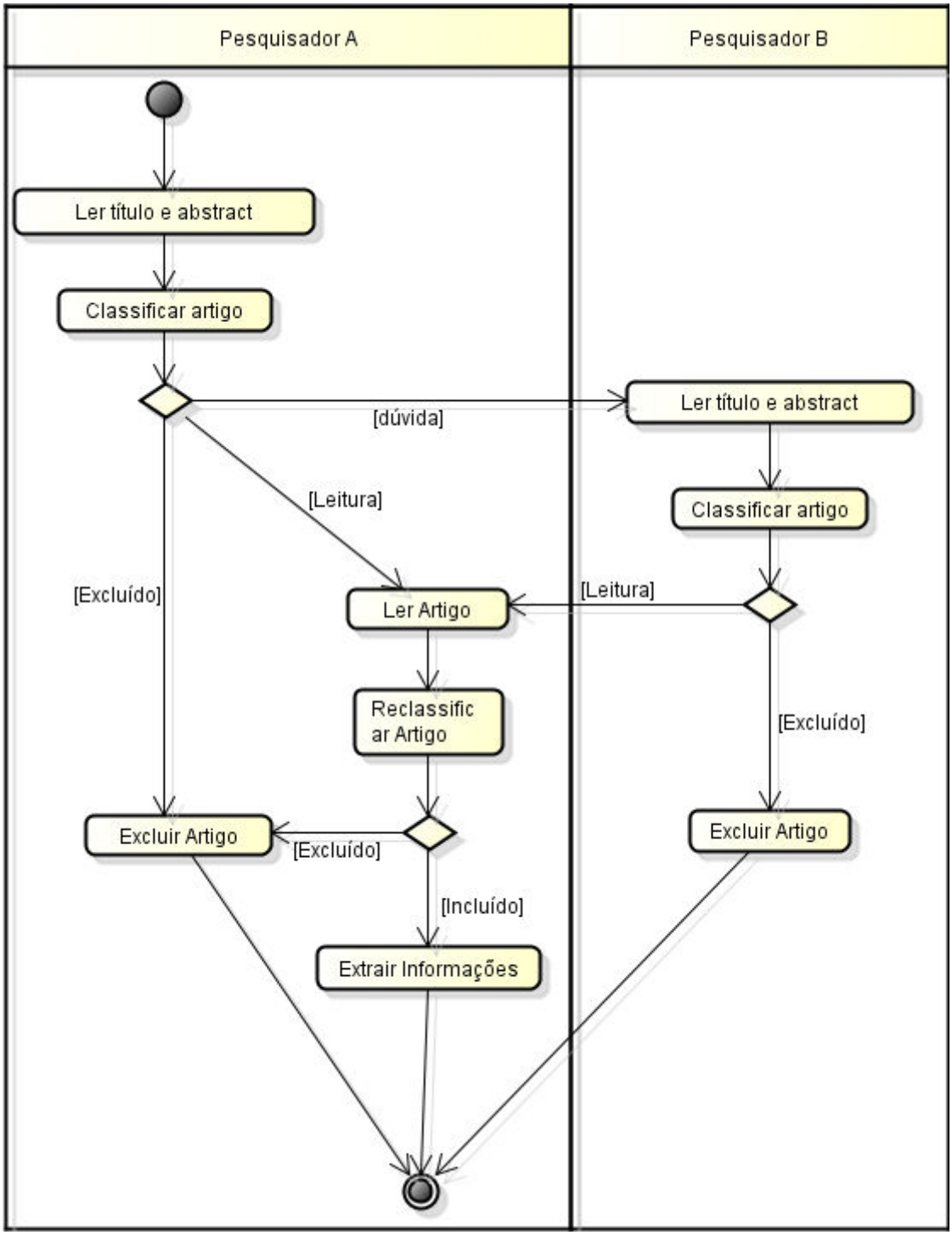
2.3.1 Definição dos estudos

- **Definição dos critérios de inclusão e exclusão:**
  - **Critérios de inclusão:**
    - Tratar de testes de software;
    - Descrever critérios de parada para processos testes de software;
    - Apresentar alguma aplicação dos critérios de parada propostos; e
    - Apresentar referência bibliográfica que caracterize o critério apresentado caso não seja de autoria.

  - **Critérios de exclusão:**

        - Artigos que não tratam critérios de parada para testes de software; ou
        - Artigos que não apresentem alguma forma de avaliação (experimento, prova de conceito, qualquer outro tipo de estudo ou simples demonstração) sobre o critério proposto; ou
        - Artigos que não estejam disponíveis por meio digital ou impresso; ou
        - Artigos publicados em idiomas diferentes do português e inglês;

- **Procedimento para seleção de artigos:**
    - A seleção dos artigos será realizada por dois pesquisadores: **Pesquisador A** e **Pesquisador B** (grande experiência).
    - O **Pesquisador A** realizará a leitura do título e *abstract* de todos os documentos retornados pelas máquinas de busca. Os artigos serão classificados com os seguintes status:
        - **Leitura:** documentos que tratam de alguma forma de critérios de parada para testes de software ;
        - **Excluído:** documentos que não tratam de critérios de parada para testes de software;
        - **Dúvida:** documentos em que o pesquisador teve dúvida se tratam de alguma forma de critérios de parada para testes de software;
    - O **Pesquisador B** irá realizar a leitura do título e *abstract* dos documentos que foram classificados com o status **Dúvida** e irá reclassificar estes documentos como **Leitura** ou **Excluído**;
    - O **Pesquisador A** realiza a leitura de todos os documentos classificados como **Leitura** e os classifica como:
        - **Incluído** – documentos que atendam os critérios de inclusão e não atendam aos critérios de exclusão. Esses artigos vão ter suas informações extraídas;
        - **Excluído**: documentos que não atendam os critérios de inclusão ou que atendam aos critérios de exclusão;
- **O mesmo processo pode ser representado pelo seguinte diagrama:**


Pesquisador A
Pesquisador B
Ler título e abstract
Classificar artigo
[dúvida]
Ler título e abstract
[Leitura]
Classificar artigo
[Excluído]
Ler Artigo
[Leitura]
Reclassific
ar Artigo
[Excluído]
Excluir Artigo
[Excluído]
Excluir Artigo
[Incluído]
Extrair Informações

2.4 Resultados da Pré-execução

Máquina de Busca	Número de artigos encontrados	Controles
Scopus	378	5
IeeeXplore	129	5
EI Compendex	191	5
Web of Science	103	1
Total Bruto	801	-
Duplicados	314	-
Total	**487**	**5**

2.5 Estratégia para Extração da Informação

Para cada artigo selecionado as informações contidas no **Anexo 1 – Modelo de Formulário de Extração** devem ser extraídas com a ajuda da ferramenta JabRef (http://jabref.sourceforge.net/):

2.6 Critérios para Avaliação da Qualidade dos Artigos

Os critérios a seguir serão utilizados para avaliar a qualidade dos artigos selecionados. O objetivo é identificar os artigos que possuem uma relação mais estreita com o tema que está sendo investigado e, com isto, terão maior confiabilidade no resultado final.

A. O artigo apresenta algum tipo de estudo experimental ou avaliação formal do critério proposto? (2 pt)
B. O artigo apresenta alguma prova de conceito? (1 pt)
C. O artigo caracteriza o software em que o critério pode ser aplicado? (2 pt)
D. O artigo utiliza metodologia e linguagem que facilita o entendimento (2 pt)
E. O artigo utiliza terminologia adequada? (1 pt)
F. O artigo deixa explícitas as condições e restrições de aplicação do critério? (1 pt)

3 Execução

- Data de Execução: 14/10/2011

Após o primeiro pesquisador avaliar os artigos seguindo os critérios de inclusão e exclusão o seguinte resultado foi obtido:

Status	Quantidade
Leitura	110
Dúvidas	12

Excluídos	360
Controles	5
Total	**487**

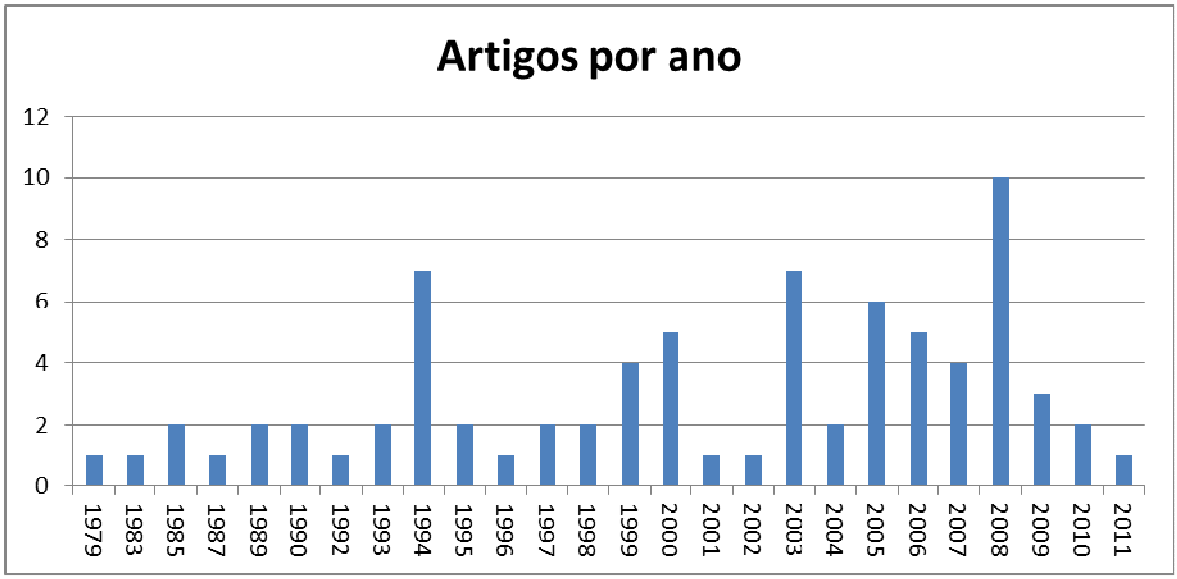
Como houveram 12 artigos classificados com o status **Dúvida** um segundo pesquisador foi envolvido. Após a avaliação do segundo pesquisador o seguinte resultado foi obtido:

Status	Quantidade
Leitura	116
Excluídos	366
Controles	5
Total	**487**

Ao longo da do processo de leitura dos artigos, alguns artigos foram desconsiderados, pois não atendiam aos critérios de inclusão/exclusão. O resultado final é representado a seguir.

Status	Quantidade
Leitura+Controle	121
Não Encontrados	16
Não apresenta critério de parada	23
Não apresenta demonstração	3
Artigo em chinês	2
Total	**77**

Uma divisão por ano dos artigos que foram incluídos também foi realizada para verificar em que período ouve mais interesse em pesquisa por critérios de parada para testes de software. O gráfico que representa tabela também pode ser visto a seguir.



Em cada um dos artigos encontrados, um critério era proposto. Portanto, foram encontrados 77 critérios de parada. Destes 77 critérios, 3 eram critérios repetidos. Assim, **foi possível extrair 74 critérios de parada para testes de software**.

4 Refências

Binder, R., "Testing Object-Oriented Systems: Models, Patterns, and Tools", Bostom: Addison-Wesley Longman Publishing Co., 1999.

Biolchini, J., Mian, P.G., Natali, A.C., Travassos, G.H. (2005). "Systematic Review in Software Engineering: Relevance and Utility". Technical Report. PESC - COPPE/UFRJ. Brazil. http://www.cos.ufrj.br/uploadfiles/es67905.pdf.

Delamaro, E., Maldonado, J. C., Jino, M., "Introdução ao Teste de Software", Rio de Janeiro: Elsevier, 2007.

CMMI PRODUCT DEVELOPMENT TEAM (2000): "CMMI-SE/SW: Capability Maturity Model – Integrated for Systems Engineering/Software Engineering", version 1.0 staged representation. Techinical Report 2000-TR-012, Software Engineering Institute, Carnegie Mellon University, USA.

Denaro, G., Polini, A. and Emmerich W. (2004) "Early performance testing of distributed software applications". In WOSP'04: Proceedings of the 4th international workshop on Software and performance, New York, NY, USA, p. 94-103.

Juristo, N., Moreno, A. M., Vegas, S., "Reviewing 25 years of testing of technique experiments". Empirical Software Engineering: An International Journal, 9(1), p. 7-44, 2004.

MPS.BR – Melhoria de Processo de Software Brasileiro, Guia Geral, 2005.

Pai, M. McCulloch, M. Gorman, J.D. et al. (2004) "Systematic Reviews and meta-analyses: An illustrated, step-by-step guide", The National Medical Journal of India, vol. 17, n.2.

Perry, W., "Effective Methods For software Testing", New York: John Wiley, 1995.

Pressman, R.S. (2006). Software Engineering: A Practtioner´s Approach. McGraw Hill.

Rocha, A. R. C., Maldonado, J. C., Weber, K. C., "Qualidade de software – Teoria e prática", Prentice Hall, São Paulo, 2001.

SEI, SOFTWARE ENGINEERING INSTITUTE, "CMM for Software", SEI-93-TR-24 and -25, 1993.

Travassos, G. H.; Santos, P. M.; Mian, P. G.; Dias Neto, A. C.; Biolchini, J. (2008) "An Environment to Support Large Scale Experimentation in Software Engineering,"

Engineering of Complex Computer Systems, IEEE International Conference on, pp. 193-202, 13th IEEE International Conference on Engineering of Complex Computer Systems (iceccs 2008).
DOI: http://doi.ieeecomputersociety.org/10.1109/ICECCS.2008.30

APÊNDICE B – Modelo de Formulário de Extração

Este apêndice apresenta o modelo de formulário de extração utilização na execução da quasi-revisão sistemática.

Campos extraídos diretamente	
Título	
Autores	
Ano de publicação	
Fonte de publicação	
Abstract	
Máquina de Busca	
Campos extraídos a partir do entendimento	
Tipo de critério	
Caracterização do software	
Níveis de testes	
Modelo Utilizado	
Habilidades Necessárias	
Contexto de aplicação	
Princípio do critério	
Objetivo do critério	
Considera depuração perfeita	
Confiabilidade pré-determinada	
Orçamento para testes pré-determinado	
Tempo para testes pré-determinado	
Número de faltas pré-determinado	
Taxa de identificação de falhas pré-determinada	
Considera qualidade dos casos de testes	
Considera atividade de testes isolada	
Diferencia taxa de identificação de falhas	
Variações do critério	
Modelo de confiabilidade base	
Modelo de Custo base	
Distribuição de Falhas	
Unidade de Medida de Falhas	
Tipo de modelo de confiabilidade	
Nível(is) de testes	
Modelo estrutural utilizado	
Habilidade(s) necessária(s)	
Contexto de aplicação	

Duração da fase de testes pré-determinada													
Restrições do critério													
Parâmetros do Critério													
Critério													
Critérios de Avaliação													
A		**B**		**C**		**D**		**E**		**F**		**Total**	

Legenda

1. **Campos extraídos diretamente das máquinas de busca**

   1.1. **Título:** o título do artigo

   1.2. **Autores:** autores do artigo

   1.3. **Ano de publicação:** ano de publicação do artigo

   1.4. **Fonte de publicação:** conferência/journal onde o artigo foi publicado

   1.5. **Abstract:** resumo do artigo

   1.6. **Máquina de busca:** nome da máquina de busca onde foi encontrado o artigo. Ordem de prioridade: IEEE, SCOPUS, EI Compendex, Web Of Science.

2. **Campos extraídos a partir do entendimento do artigo**

   2.1. **Tipo de Critério:** define se é um critério de parada apriorístico ou não apriorístico.

   2.2. **Caracterização do software:** descrição do tipo de software em que o autor sugere que o critério possa ser aplicado ou que o critério foi aplicado para avaliação.

   2.3. **Níveis de testes:** define quais níveis/fases de testes o critério de parada está preparado para funcionar. Exemplo de níveis de testes: Testes de aceitação, Testes de integração, Testes de regressão, Testes de stress, Testes de sistemas, testes funcionais, testes de unidade

   2.4. **Modelo Utilizado:** Identifica os modelos que o processo de desenvolvimento do software deve prover para que o critério possa ser aplicado. Por exemplo, diagramas da UML, Diagrama de fluxo de dados, Cadeia de Markov.

   2.5. **Habilidades Necessárias:** Descreve as habilidades que devem ser providas pela equipe de testes para a aplicação do critério de parada.

2.6. **Contexto de Aplicação:** Identificada em qual contexto o critério de parada foi proposto e aplicado. Exemplos de contexto são comercialização em escala, comercialização para cliente único, utilização própria.

2.7. **Princípio do critério:** princípio em que se baseia o critério de parada. Por exemplo, modelo de confiabilidade, fluxo de execução do sistema, cadeia de Markov.

2.8. **Objetivo do critério:** o principal objetivo do critério. Por exemplo, maximizar a confiabilidade, minimizar o custo com testes.

2.9. **Considera depuração perfeita:** [Sim] quando uma falha é encontrada, o defeito que provocou essa falha é corrigido imediatamente, instantaneamente e sem provocar novos defeitos; [Não] quando uma falha é encontrada, o defeito que provocou esta falha não é corrigido imediatamente, nem instantaneamente ou pode provocar novos defeitos.

2.10. **Confiabilidade pré-determinada:** [Sim] Confiabilidade é uma entrada fixa para o critério de parada. [Não] Caso contrário.

2.11. **Orçamento para testes pré-determinado:** [Sim] Custo com os testes é uma entrada fixa para o critério de parada. [Não] Caso contrário.

2.12. **Tempo para testes pré-determinado**: [Sim] Quantidade de tempo que deve ser utilizada com os testes de software é uma entrada fixa para o critério de parada. [Não] Caso contrário.

2.13. **Número de faltas é pré-determinado:** [Sim] caso o número de faltas deva ser conhecido ou estimado e serve como entrada para o critério. [Não] Caso contrário

2.14. **Taxa de identificação de falhas é pré-determinada:** [Sim] caso a taxa de identificação de falhas deva ser conhecida ou estimada e serve como entrada para o critério. [Não] Caso contrário.

2.15. **Considera a qualidade dos casos de testes:** [Sim] caso a qualidade dos casos de testes sejam relevantes para o resultado do critério de parda. [Não] Caso contrário.

2.16. **Considera atividade de teste isolada:** [Sim] caso o critério considere que a atividade de teste aconteça isoladamente ao final do processo de codificação. [Não] Caso contrário.

2.17. **Diferencia taxa de falhas** [Sim] caso a taxa de falhas seja diferenciada na fase de testes e na fase de produção. [Não] Caso contrário.

2.18. **Variações do critério:** Quantidade de variações do critério que são apresentadas no artigo. Por exemplo, um mesmo modelo de confiabilidade pode

resultar em dois critérios de parada. Um que minimiza o custo e outro que maximiza a confiabilidade.

2.19. **Modelo de confiabilidade base:** é o modelo de confiabilidade utilizado para definir o critério de parada. Geralmente os modelos de confiabilidade não são nomeados. Portanto, é referenciado o autor que propôs o modelo. Quando há uma evolução do modelo, a referência é feita ao modelo de origem e é especificado que houve uma evolução.

2.20. **Modelo de custo base:** é o modelo de custo no qual o critério é baseado. Geralmente os modelos de custo não são nomeados. Portanto, é referenciado o autor que propôs o modelo. Quando há uma evolução do modelo, a referência é feita ao modelo de origem e é especificado que houve uma evolução.

2.21. **Distribuição de falhas:** é a forma como presume-se que as falhas estão distribuídas no software. Os modelos de confiabilidade utilizam distribuições diferentes. Por exemplo, distribuição exponencial, Distribuição de Poisson, etc.

2.22. **Unidade de medida de falhas:** define em qual unidade as falhas são medidas. Os valores possíveis são: Falhas Acumuladas (função de valor médio), Intensidade de falhas, taxa de ocorrência de falhas e tempo médio entre falhas.

2.23. **Tipo de modelo de confiabilidade:** define o tipo de modelo de confiabilidade que pode ser modelo de **estimativa**, **modelo de predição** e **modelo de predição em fases iniciais**. Os modelos de estimativa determinam a confiabilidade atual do software. Os modelos de predição determinam a confiabilidade futura do software e são utilizados quando o software se encontra nas fases de testes ou operacional, pois nestas fases é possível obter dados sobre as falhas do software. Os modelos de predição em fases iniciais são utilizados quando ainda não se tem dados sobre as falhas.

2.24. **Nível(is) de testes:** define em quais níveis/fases de testes o critério de parada pode ser aplicado. Os valores podem ser testes de unidade, testes de integração, testes de sistema, testes de aceitação.

2.25. **Modelo estrutural utilizado:** informa se é preciso utilizar algum modelo comportamental/estrutural para a aplicação do critério.

2.26. **Habilidade(s) necessária(s):** informa se é preciso ter conhecimento de alguma técnica, procedimento, ferramenta, tecnologia ou qualquer habilidade específica para a aplicação do critério de parada.

2.27. **Contexto da aplicação:** informa a situação ideal para a aplicação do critério em termos dos objetivos propostos para o software. Por exemplo, o critério de parada pode ter sido construído para ser aplicado em um software que será utilizado pela própria instituição desenvolvedora, para ser vendido para um único cliente, para ser vendido em larga escala.

2.28. **Duração da fase de testes pré-determinada:** informa se o critério recebe como entrada a duração da fase de testes.

2.29. **Restrições do critério:** restrições que possam ser impostas ao critério ou ao modelo que dê origem ao critério. Por exemplo, a identificação de falhas é modelada por um NHPP.

2.30. **Parâmetros do Critério:** são as parâmetros de entrada necessários para a aplicação do critério

2.31. **Critério:** Uma descrição textual do critério e sua representação através de fórmulas e/ou gráficos.

3. **Critérios de Avaliação**

3.1. **A:** O artigo apresenta algum tipo de estudo experimental ou avaliação formal do critério proposto? (2 pontos)

Um estudo experimental pode ser considerado uma forma mais formal de avaliação. Com estudos experimentais a parcialidade na avaliação dos critérios pode ser mitigada.

3.2. **B:** O artigo apresenta alguma prova de conceito? (1 ponto)

Apensar de não ser uma avaliação tão formal quanto estudos experimentais, provas de conceitos podem ser um critério de avaliação.

3.3. **C:** O artigo caracteriza o software em que o critério pode ser aplicado? (2 pontos)

A caracterização do software é importante, pois um bom critério para um tipo de software pode ser ruim para outros tipos.

3.4. **D:** O artigo utiliza metodologia e linguagem que facilita o entendimento? (2 pontos)

O artigo deve ser escrito com linguagem clara que facilite a interpretação e extração do critério apresentado.

3.5. **E:** O artigo utiliza terminologia adequada? (1 ponto)

É importante que o artigo utilize os termos conceitualmente corretos para evitar danos a interpretação

3.6. **F:** O artigo deixa explícitas as condições e restrições de aplicação do critério? (1 ponto)

Sem a devida avaliação, não é possível afirmar que um mesmo critério seja capaz de ter a mesma eficiência em qualquer ambiente, com qualquer equipe e em qualquer tipo de software. Por este motivo, é preciso que os artigos deixem explícitos as restrições do critério apresentado.

APÊNDICE C – Extração dos Dados dos Critérios de Parada

Este apêndice apresenta cada um dos critérios de parada encontrados através da quasi-revisão sistemática. Esse apêndice representa o corpo de conhecimento utilizado nesse trabalho.

Campos extraídos diretamente	
Título	A cost-benefit stopping criterion for statistical testing
Autores	Prowell, S.J.
Ano de publicação	2004
Fonte de publicação	System Sciences, 2004. Proceedings of the 37th Annual Hawaii International Conference on
Abstract	
Determining when to stop a statistical test is an important management decision. Several stopping criteria have been proposed, including criteria based on statistical similarity, the probability that the system has a desired reliability, and the expected cost of remaining faults. This paper proposes a new stopping criterion based on a cost-benefit analysis using the expected reliability of the system (as opposed to an estimate of the remaining faults). The expected reliability is used, along with other factors such as units deployed and expected use, to anticipate the number of failures in the field and the resulting anticipated cost of failures. Reductions in this number generated by increasing the reliability are balanced against the cost of further testing to determine when testing should be stopped.	
Máquina de Busca	IEEE
Campos extraídos a partir do entendimento	
Tipo de Critério	Não apriorístico. Determina a confiabilidade e o custo mínimo.
Caracterização do software	Sistemas compostos por software e hardware
Níveis de testes	Testes funcionais
Modelo Utilizado	Nenhum
Habilidades Necessárias	Nenhum
Contexto de aplicação	Não informado
Princípio do critério	Modelo de confiabilidade
Objetivo do critério	Minimizar o custo de desenvolvimento
Considera depuração perfeita	*Não informado*
Confiabilidade pré-determinada	Não
Orçamento para testes é pré-determinado	Não
Tempo para testes pré-determinado	Não
Número de faltas pré-determinado	Não
Taxa de identificação de falhas pré-determinada	Não
Considera qualidade dos casos de testes	Não
Considera atividade de testes isolada	Sim

Diferencia taxa de detecção de falhas	Não
Variações do critério	Não
Modelo de confiabilidade base	Prowell (2004) (Próprio)
Modelo de Custo	Prowell (2004) (Próprio)
Distribuição de Falhas	Não aplicável (Modelo não trabalha com falhas e sim com a probabilidade de execução de demandas com sucesso)
Unidade de Medida de Falhas	Não aplicável (Modelo não trabalha com falhas e sim com a probabilidade de execução de demandas com sucesso)
Tipo de modelo de confiabilidade	Modelo de predição

Suposições

Não informado

Parâmetros do critério

- C_0 – Todos os custos iniciais de teste
- C_t – Custo médio por rodar um teste

Critério

O Autor propõe um critério baseado na expectativa de confiabilidade do sistema que está sendo testado e no custo de falhas encontradas depois que o software está em produção.
Enquanto o custo dos testes for menor que o ganho por aumentar a confiabilidade o sistema deve ser testado.
Seguindo o raciocínio de que quanto mais rodadas de testes o software é submetido mais confiável ele fica. Entretanto, cada uma dessas rodadas tem um custo e quando este custo ultrapassar o benefício que ele adiciona ao sistema o teste não deve ser executado.
A seguinte fórmula é proposta:

$$\frac{C_0}{C_t} + n \leq \text{ØÑ}e(r_n - r_0)$$

Onde a parte esquerda da equação corresponde ao custo dos testes e a parte direita ao ganho por aumentar a confiabilidade e:

- C_0 – Todos os custos iniciais de teste
- C_t – Custo médio por rodar um teste
- n – número de testes que devem ser executados
- Ø – C_f/C_t
- Ñ – número de unidades descontadas em campo ao longo da vida da versão lançada
- e – taxa de uso esperada de uma unidade
- r_n – confiabilidade estimada depois de executar n testes
- r_0 – confiabilidade estimada antes de executar n testes

Critérios de Avaliação

A	0	**B**	1	**C**	1	**D**	2	**E**	1	**F**	0	**Total**	5

Campos extraídos diretamente	
Título	A decision-analytic stopping rule for validation of commercial software systems

Autores	Chavez, Tom
Ano de publicação	2000
Fonte de publicação	IEEE Transactions on Software Engineering

Abstract

The decision about when to release a software product commercially is not a question of when the software has attained some objectively justifiable degree of correctness. It is, rather, a question of whether the software achieves a reasonable balance among engineering objectives, market demand, customer requirements, and marketing directives of the software organization. In this paper, we present a rigorous framework for addressing this important decision. Conjugate distributions from statistical decision theory provide an attractive means of modeling the cost and rate of bugs given information acquired during software testing, as well as prior information provided by software engineers about the fidelity of the software before testing begins. In contrast to methods such as [1] and [15], the stopping analysis presented here yields a computationally simple rule for deciding when to release a commercial software product based on information revealed to engineers during software testing - complicated numerical procedures are not needed. Our method has the added benefits that it is sequential: It measures explicitly the costs of customer dissatisfaction associated with bugs as well as the costs of declining market position while the testing process continues; and it incorporates a practical framework for cost-criticality assessment that makes sense to professional software developers. A probabilistic model of catastrophic bugs provides another useful way of characterizing and measuring the software's expected performance after commercial release. Taken together, these tools provide a software organization with a clearer basis for making decisions about when to release a commercial software product.

Máquina de Busca	IEEE
Campos extraídos a partir do entendimento	
Tipo de Critério	Não apriorístico. Determina até quando os testes devem continuar.
Caracterização do software	Softwares comerciais no geral
Níveis de testes	*Não informado*
Modelo Utilizado	Nenhum
Habilidades Necessárias	Nenhuma
Contexto de aplicação	Comercialização em escala
Princípio do critério	Modelo probabilístico
Objetivo do critério	Minimizar custo de desenvolvimento
Considera depuração perfeita	*Não Informado*
Confiabilidade pré-determinada	Não
Orçamento para testes é pré-determinado	Não
Tempo para testes pré-determinado	Não
Número de faltas pré-determinado	Não
Taxa de identificação de falhas pré-determinada	Não
Considera qualidade dos casos de testes	Não
Considera atividade de testes isolada	Sim
Diferencia taxa de detecção de falhas	Não
Variações do critério	Não
Modelo de confiabilidade base	Chavez, Tom (2000) (Próprio)
Modelo de Custo	Chavez, Tom (2000) (Próprio)
Distribuição de Falhas	Não aplicável
Unidade de Medida de Falhas	Não aplicável
Tipo de modelo de confiabilidade	Modelo de Estimativa

Suposições

- O Framework é independente de plataforma (Windows, Linux, mac)
- O Framework pode ser utilizado também para hardware
- Não é necessária nenhuma informação sobre o conjunto de fluxos lógicos do software
- Um bug é uma falta ou uma falha no software
- Teste é um procedimento que tem a finalidade de encontrar bugs
- Os bugs são categorizados em 4 níveis de criticidade e custo de correção
- Os bugs que são menos críticos e os que possuem criticidade baixa e alto custo de correção são ignorados

Parâmetros do critério

- Tempo de execução dos testes
- Custo de corrigir um bug antes do lançamento
- Custo de corrigir um bug após o lançamento
- Custos de perda de posição no mercado
- Custo de descontentamento com cliente

Critério

O critério apresentado pelo artigo pode ser utilizado para ajudar na decisão de quando parar os testes de sistemas comerciais. Para isto, o critério utiliza de informações que podem estar disponíveis aos engenheiros durante a fase de teste.
O modelo faz um balanceamento entre os custos de identificar e corrigir bugs contra os custos de se perder posições no mercado por não disponibilizar o software em determinado momento.
O autor assume que a taxa de bug é única para todo o sistema. Apesar disto, ela pode ser recalculada ao longo do processo de testes.
O autor diz também que os bugs que possuem baixa criticidade podem e devem ser ignorados, apresentando a figura a seguir e dizendo que os quadrantes II e IV podem ser ignorados.

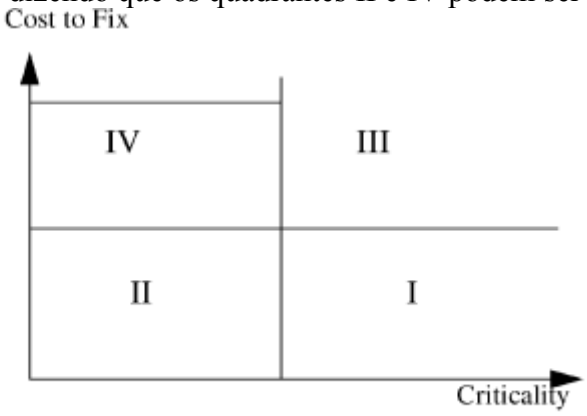


Fig. 1. Cost/Criticality trade-off for bugs.

Os testes devem continuar enquanto:

$$\lambda(t)\, C_{BN}(t) + C_{MP}(t) < \lambda(t)[\, C_{BL}(t) + C_{CD}]$$

Onde:

- t=0 – Início do ciclo de testes
- T_f – Tempo de desenvolvimento do software até a sua entrega
- t* - Tempo em que o teste termina o produto é entregue
- $C_{BN}(t)$ – O custo de se corrigir um bug antes do lançamento do software no mercado
- $C_{BL}(C_{BN}(t))$ – O custo de corrigir um bug depois do lançamento do software no mercado
- $C_{MP}(t)$ – O custo acumulado da perda de posição no mercado até o tempo t
- C_{CD} – O custo de descontentamento dos clientes por bug encontrado.

• λ(t) – Estimativa da probabilidade de encontrar um bug no próximo período.												
Critérios de Avaliação												
A	0	**B**	1	**C**	2	**D**	2	**E**	1	**F**	1	**Total**	7

Campos extraídos diretamente	
Título	On the determination of an appropriate time for ending the software testing process
Autores	Malevris, N. and Petrova, E.
Ano de publicação	2000
Fonte de publicação	Quality Software, 2000. Proceedings. First Asia-Pacific Conference on
Abstract	
Software testing is widely used as a means of increasing software reliability. The prohibitive nature of exhaustive testing has given rise to the problem of determining when a system has reached an acceptable reliability slate and can be released. This has probably become the hardest problem facing a project manager. In this paper, a stopping rule that indicates the appropriate time at which to stop testing is presented. The rule automatically adapts to modifications in the assumptions, since it can be applied under any software error-counting model. An investigation of the properties of the rule is described and the results obtained after applying it to a set of real data in conjunction with two statistical models are presented.	
Máquina de Busca	IEEE
Campos extraídos a partir do entendimento	
Tipo De Critério	Apriorístico. É definido um tempo entre falhas considerado crítico que é o Critério De Parada.
Caracterização do software	Não Informado
Níveis de testes	Não informado
Modelo Utilizado	Nenhum
Habilidades Necessárias	Nenhuma
Contexto de aplicação	Não especificado
Princípio do critério	Modelo de confiabilidade
Objetivo do critério	Atingir Confiabilidade adequada (tempo entre falhas)
Considera depuração perfeita	Não aplicável
Confiabilidade pré-determinada	Sim
Orçamento para testes é pré-determinado	Não
Tempo para testes pré-determinado	Não
Número de faltas pré-determinado	Não
Taxa de identificação de falhas pré-determinada	Não
Considera qualidade dos casos de testes	Sim- garante a cobertura das características estruturais do software
Considera atividade de testes isolada	Sim
Diferencia taxa de detecção de falhas	Não
Variações do critério	2- Uma variação assume que há dependência no tempo entre falhas e outra variação assume que não há
Modelo de confiabilidade base	Multi modelo
Modelo de Custo	Não aplicável

Distribuição de Falhas	Não aplicável (Depende do modelo de confiabilidade utilizado)
Unidade de Medida de Falhas	Tempo médio entre falhas
Tipo de modelo de confiabilidade	Não aplicável (Podem ser utilizados diversos modelos)

Suposições

Como o critério pode utilizar diversas modelos de aprendizagem cronológicos, as restrições estão vinculadas a estes modelos.

Parâmetros do critério

- Falhas encontradas durante o processo de teste
- Tempo entre as falhas encontradas durante o processo de teste

Critério

O critério de parada proposto no artigo observa os intervalos de tempo entre falhas. Um intervalo de falhas considerado crítico é pré-determinado. Quando o intervalo de tempo que está sendo observado ultrapassa o intervalo crítico, então sugere-se que os testes devem parar.
O critério de parada pode ser utilizado em qualquer modelo de confiabilidade existente.
O autor exemplifica o uso em dois modelos de confiabilidade,

- J-M Model:

$$\Pr(T_i \geq T_c \text{ for the first time}) = \left(1 - F_i(T_c)\right)$$

$$E(Y_i) = \sum_{j=1}^{i-1} \int_0^{T_c} t g_j(t) dt + T_c, i = 1, \ldots, N$$

G-P Model:

$$\Pr(T_i \geq T_c \text{ for the first time}) = \int_0^{T_c} \ldots \int_0^{T_c} \int_{T_c}^{\infty} f_{T_1 \ldots T_i}(t_1, \ldots, t_i) dt_1 \ldots dt_i$$

$$E(Y_i) = \sum_{j=1}^{i-1} \int_0^{T_c} t g_j(t) dt + T_c, i = 1, \ldots, N$$

Onde,

- T_i - O "Iézimo" intervalo de tempo
- T_c – Intervalo crítico pré-determinado
- Y_i – O tempo total decorrido do início dos testes até o "Iézimo" intervalo de tempo que excede T_c pela primeira vez.
- Medidas estatísticas como a probabilidade (P_i) de T_i ultrapassar T_c pela primeira vez e a probabilidade de Y_i possa ser calculado e utilizado para encontrar T_c ($E(Y_i)$).
- g_j – Função de probabilidade condicional de densidade dado que $T_1, \ldots, T_{i-1} <= T_c$ and $T_i >= T_c$, $1<=i<=N$, $1<=j<=N$
- F*i* – Função de distribuição cumulativa de T*i*
- $F_{T1 \ldots Ti}(t_1, \ldots, t_i)$ – densidade de probabilidade conjunta de T_i, i = 1, ..., N

Critérios de Avaliação

A	0	**B**	1	**C**	0	**D**	2	**E**	1	**F**	1	**Total**	5

Campos extraídos diretamente	
Título	Some graphical aids for deciding when to stop testing software
Autores	Dalal, Siddhartha R. and Mallows, Colin L.
Ano de publicação	1990
Fonte de publicação	IEEE Journal on Selected Areas in Communications
Abstract	
It is noted that the developers of large software systems must decide how much software should be tested before releasing it. An explicit tradeoff between the costs of testing and releasing is considered. The former may include the opportunity cost of continued testing, and the latter may include the cost of customer dissatisfaction and of fixing faults found in the field. Exact stopping rules were obtained by the authors in a previous paper (1988), under the assumption that the distribution of the fault finding rate is known. Here, two important variants where the fault finding distribution is not completely known are considered. They are i) the distribution is exponential with unknown mean and ii) the distribution is locally exponential with the rate changing smoothly over time. New procedures for both cases are presented. In case (i) it is shown how to incorporate information from related projects and subjective inputs. Several novel graphical procedures which are easy to implement are proposed, and these are illustrated for data from a large telecommunications software system.	
Máquina de Busca	IEEE
Campos extraídos a partir do entendimento	
Tipo de Critério	Não apriorístico. Mostra o ponto exato onde os testes devem parar.
Caracterização do software	Grandes Sistemas de Software. O artigo demonstrou o critério utilizando um sistema de telecomunicação.
Níveis de testes	Testes funcionais
Modelo Utilizado	Nenhum
Habilidades Necessárias	Nenhuma
Contexto de aplicação	Comercialização para cliente único
Princípio do critério	Modelo de confiabilidade
Objetivo do critério	Equilíbrio entre o custo com testes e custo de correção de falhas com o software em produção
Considera depuração perfeita	*Não informado*
Confiabilidade pré-determinada	Não
Orçamento para testes é pré-determinado	Não
Tempo para testes pré-determinado	Não
Número de faltas pré-determinado	Não
Taxa de identificação de falhas pré-determinada	Não
Considera qualidade dos casos de testes	Não
Considera atividade de testes isolada	Sim
Diferencia taxa de detecção de falhas	Não
Variações do critério	Não há
Modelo de confiabilidade base	Extensão de Dalal e Mallows (1988) (próprio)
Modelo de Custo	Dalal e Mallows (1988) (próprio)
Distribuição de Falhas	Distribuição Exponencial
Unidade de Medida de Falhas	Número acumulado de falhas
Tipo de modelo de confiabilidade	Modelo de estimativa de confiabilidade
Suposições	

- A distribuição do tempo entre falhas é conhecida. (dada uma falta, a quantidade de tempo gasta para encontrá-la é dada por uma distribuição conhecida G(t))
- Tempo entre falhas é independente

Parâmetros do critério

- Tempo dos testes em alguma unidade padrão (ex.: tempo de calendário/tempo do CPU)
- Número total de falhas observadas até determinado tempo
- Custo de se resolver uma falha em teste
- Custo de se resolver uma falha em produção

Critério

O artigo propõe critérios baseados no custo de dar continuidade aos testes, o custo, no custo de se corrigir falhas com o software já em produção e no descontentamento do cliente em caso de falhas. Segundo os autores os critérios propostos são fáceis de demonstrar, intuitivos para se aprender e precisos.

1) Economic model: modelo utilizado pelo autor para calcular o custo total dos testes até um determinado tempo t

   L(t, K(t), N) = f(t) – cK(t) +bN; //Calcula o custo total dos testes até o tempo t

   Onde:
   - N = número total de falhas no módulo (desconheido)
   - K(t) = número total de falhas observadas até o tempo t
   - a = custo de resolver uma falha em teste
   - b = custo de resolver uma falha em produção
   - c = (b – a) = custo liquido de se resolver uma falha em produção
   - f(t) = soma dos custos de testes até o tempo t mais o custo da probabilidade de não entregar o software até o tempo t
     - f(t) pode ser linear. Isto acontece quando o custo dos testes é constante para cada unidade
     - f(t) é linear até t_{max} e depois se torna infinito
     - f(t) é linear inicialmente e depois se torna exponencial

2) Stochastic model: modelo utilizado para ajudar na decisão de quando os testes devem parar

   f'(t)G(t) / cg(t) >= K(t)

   Onde:
   - G(t) = função que define a distribuição da quantidade de tempo que é preciso para encontrar uma falha
   - g = densidade de G

No artigo este modelo é estendido para situações em que o G é exponencial com uma taxa fixa μ:

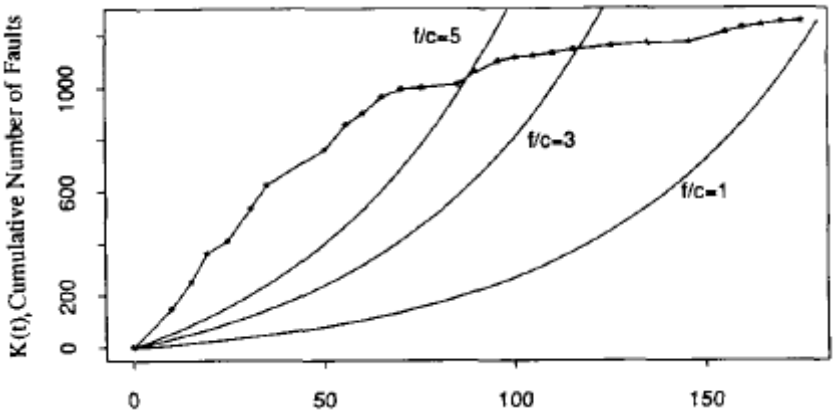
(f/ μc)(e $^{\mu t}$ – 1) >= K(t)

Onde:
- μ = taxa de variação de G

No artigo o autor demonstra um gráfico onde 3 possibilidades de parada são apresentadas. Para isto

utiliza-se dos seguintes parâmetros:

- f(t) = f . t //custo linear
- f = 6000 dolares
- c = 2000 dolares
- Com estes valores f/c = 3. Entretanto, como esta medida não é precisa, o autor preferiu atribuir mais dois valores para ela para que ele pudesse analisar o resultado e também traçou gráficos para f/c= 5 e f/c=1



Os testes devem parar nos pontos de interseção.

Critérios de Avaliação													
A	0	**B**	1	**C**	2	**D**	2	**E**	1	**F**	1	**Total**	7

Campos extraídos diretamente	
Título	Stopping rules for the operational testing of safety-critical software
Autores	Littlewood, Bev and Wright, David
Ano de publicação	1995
Fonte de publicação	IEEE Transactions on Software Engineering
Abstract	
It has been proposed to conduct a test of a software safety system for a nuclear reactor by subjecting it to demands that are statistically representative of those it will meet in operational use. The intention behind the test is to acquire a high confidence (99%) that the probability of failure on demand is smaller than 10^{-3}. To this end the test takes the form of executing about 5000 demands and requiring that all of these are successful. In practice it is necessary to consider what happens if the software fails the test and is repaired. We argue that the earlier failure information needs to be taken into account in devising the form of the test that the modified software will need to pass - essentially that after such failure the testing requirement might need to be more stringent (i.e. the number of tests that must be executed failure-free should increase). We examine a Bayesian approach to the problem, for this stopping rule based upon a required bound for the probability of failure on demand, as above, and also for a requirement based upon a prediction of future failure behaviour. We show that the first approach seems to be less conservative than the second, and argue that the second should be preferred for practical application.	
Máquina de Busca	IEEE
Campos extraídos a partir do entendimento	
Tipo de critério	Apriorístico. Encontra o tempo de teste necessário para atingir

	uma probabilidade de falha de 10^{-3}.
Caracterização do software	Software de Missão Crítica. O critério foi aplicado em um sistema de proteção primária de um reator nuclear)
Níveis de testes	Testes funcionais
Modelo Utilizado	Nenhum
Habilidades Necessárias	Nenhuma
Contexto de aplicação	Comercialização para único cliente
Princípio do critério	Abordagem Bayesiana
Objetivo do critério	Atingir uma confiabilidade pré-estabelecida
Considera depuração perfeita	Sim
Confiabilidade pré-determinada	Sim
Orçamento para testes é pré-determinado	Não
Tempo para testes pré-determinado	Não
Número de faltas pré-determinado	Não
Taxa de identificação de falhas pré-determinada	Não
Considera qualidade dos casos de testes	Não
Considera atividade de testes isolada	Sim
Diferencia taxa de detecção de falhas	Não
Variações do critério	Não há
Modelo de confiabilidade base	Não aplicável
Modelo de Custo	Não aplicável
Distribuição de Falhas	Não aplicável
Unidade de Medida de Falhas	Não aplicável
Tipo de modelo de confiabilidade	Não aplicável

Suposições

- O Critério é criado especificamente para um software de controle de um reator nuclear

Parâmetros do critério

- Número da rodada de teste
  - No artigo, uma rodada de testes é chamada de “demanda”. Uma demanda é uma seqüência de leitura dos sinais dos sensores.
- Quantidade de falhas encontradas

Critério

(Mesmo critério de "Some conservative stopping rules for the operational testing of safety-critical software")

O artigo propõe um critério de parada para um software específico de controle de um reator nuclear. Este software é responsável receber sinais de alguns sensores e avaliar o estado do reator através destes sinais. Caso seja detectado alguma anomalia o sistema é deve desligar o reator de forma segura e reiniciá-lo em modo de segurança.

A finalidade do artigo é criar e avaliar um critério de parada para este software que ofereça uma confiabilidade adequada para este tipo de sistema. Para avaliar a confiabilidade é considerado quantas “demandas” o software realizou sem que ocorresse uma falha.

Quatro passos compõem o algoritmo que define o critério de parada:

1. É computado o número de demandas que precisam ser executadas para que o software seja considerado seguro e que os testes possam parar (n_1).
2. O sistema é colocado em teste até que n_1 demandas sejam executadas. Caso todas elas sejam executadas sem erro o sistema é considerado seguro e os testes podem ser encerrados. Caso uma falha seja encontrada na demanda s_1 a rodada atual de testes é parada e o próximo passo

é executado.

3. Como uma falha foi encontrada, é computado um novo número de demandas que devem acontecer sem falhas para que o sistema seja considerado seguro (n_2).
4. O sistema é colocado em teste e caso execute n_2 demandas sem erros é considerado seguro. Caso uma falha aconteça na demanda s_1+s_2 os testes são parados.

Para realizar os cálculos de n_1 a fórmula seguinte é utilizada:

$$\int_0^{p0} \frac{(1-p)^n dp}{B(1,1+n)} \geq 1-\alpha$$

Para realizar os cálculos de n_2 a seguinte fórmula é utilizada:

$$\int_0^{p0} \frac{p^j (1-p)^{\sum_1^j si+n-j} dp}{B(j+1, \sum_1^j si+n-j+1)} \geq 1-\alpha$$

Critérios de Avaliação													
A	0	**B**	1	**C**	2	**D**	2	**E**	1	**F**	1	**Total**	7

Campos extraídos diretamente	
Título	A Discrete software reliability growth model with testing effort
Autores	Kapur, P.K. and Xie, Min and Garg, R.B. and Jha, A.K.
Ano de publicação	1994
Fonte de publicação	Software Testing, Reliability and Quality Assurance, 1994. Conference Proceedings., First International Conference on
Abstract	
In this paper we propose a discrete Software Reliability Growth Model with testing effort. The behaviour of the testing effort is described by a discrete Rayleigh curve. Assuming that the discrete failure intensity to the amount of current testing effort is proportional to the remaining error content, we formulate the model as a Non-Homogeneous Poisson Process. Parameters of the model are estimated. We then discuss a release policy based on cost and failure intensity criteria. Numerical results are also presented.	
Máquina de Busca	IEEE
Campos extraídos a partir do entendimento	
Critério Para Parada	Apriorístico. Descobre o menor custo para um critério de parada baseado na confiabilidade.
Caracterização do software	*Não informado*
Níveis de testes	Não informado
Modelo Utilizado	Nenhum
Habilidades Necessárias	Nenhuma
Contexto de aplicação	*Não informado*
Princípio do critério	Modelo de Confiabilidade
Objetivo do critério	Minimizar o custo restringido a um intensidade específica de falhas
Considera depuração perfeita	Sim
Confiabilidade pré-determinada	Sim
Orçamento para testes é pré-determinado	Não
Tempo para testes pré-determinado	Não
Número de faltas pré-determinado	Sim
Taxa de identificação de falhas pré-	Sim

determinada	
Considera qualidade dos casos de testes	Não
Considera atividade de testes isolada	Sim
Diferencia taxa de detecção de falhas	Não
Variações do critério	Não há
Modelo de confiabilidade base	Kapur (1994) (Próprio)
Modelo de Custo	Kapur (1994) (Próprio)
Distribuição de Falhas	NHPP
Unidade de Medida de Falhas	Intensidade de falhas
Tipo de modelo de confiabilidade	Modelo de estimativa de confiabilidade

Suposições

1. Software is subject to failures at random test runs caused by errors remaining in the software.
2. When a failure occurs, the error causing that failure is immediately removed and the error removing process does not introduce any new errors in the software.
3. The error detection phenomenon is modelled by NHPP.
4. Testing effort expenditures are described by a discrete Rayleigh curve.
5. Software life cycle is assumed to be more than the optimal number of test cases run before release of the software.
6. The expected discrete failure intensity to the current testing effort expenditure is proportional to the current remaining error content.
7. Corresponding to the error detection phenomenon at the manufacturer/user end, there exists an equivalent error detection phenomenon at the user/ manufacturer end.
8. Software is never released without testing.

Parâmetros do critério

- C_1 – Custo de correção de um erro antes da entrega do software
- C_2 – Custo de correção de um erro depois da entrega do software
- C_3 – Custo pode unidade de teste
- n_{ic} – Duração do ciclo de vida do software expressa em número de casos de teste
- λ_0 – Intensidade de falha desejada

Critério

O autor propõe um modelo de confiabilidade baseado no esforço de teste. O modelo de confiabilidade descreve o comportamento do esforço de teste através da curva discreta Rayleigh.
A partir do modelo proposto o autor sugere o critério de parada dado pela minimização do custo e intensidade de falhas.
O Critério é descrito pelo seguinte teorema:

(i) if LH > $\frac{C_3}{C_2-C_1}$ > RH, $\lambda(n_x) > \lambda_0$ and there exist n_2 and n_3 satisfying (19) and (20) then
if $n_1 \le n_2$ or $n_1 \ge n_3$, $n^* = n_1$
else if $n_2 < n_1 < n_3$, then if
$C(n_2) < C(n_3)$, $n^* = n_2$
$C(n_2) > C(n_3)$, $n^* = n_3$
$C(n_2) = C(n_3)$, $n^* = n_2$ or n_3
while if there exists n_3 satisfying (20) then
if $n_1 \ge n_3$, $n^* = n_1$ else $n^* = n_3$
else if $\lambda(n_x) \le \lambda_0$, $n^* = n_1$

(ii) if LH > $\frac{C_3}{C_2-C_1}$ = RH, $\lambda(n_x) > \lambda_0$ and there exist n_2 and n_3 satisfying (19) and (20) then if $n_1 \le n_2$, $n^* = n_1$
if $n_1 \ge n_3 - 1$, $n^* = n_1 + 1$
else if $n_2 < n_1 < n_3 - 1$, then if
$C(n_2) < C(n_3)$, $n^* = n_2$
$C(n_2) > C(n_3)$, $n^* = n_3$
$C(n_2) = C(n_3)$, $n^* = n_2$ or n_3
while if there exists n_3 satisfying (20) then
if $n_1 \ge n_3 - 1$, $n^* = n_1 + 1$ else $n^* = n_3$
else if $\lambda(n_x) \le \lambda_0$ and $n_1 < n_x$, $n^* = n_1$
else if $\lambda(n_x) \le \lambda_0$ and $n_1 \ge n_x$, $n^* = n_1 + 1$

(iii) if $ab \le \frac{C_3}{C_2-C_1}$, $\lambda(n_x) > \lambda_0$ and there exist n_2 and n_3 satisfying (19) and (20) then $n^* = n_2$ else if there exists n_3 satisfying (20) then $n^* = n_3$,
while if $ab \le \frac{C_3}{C_2-C_1}$ and $\lambda(n_x) \le \lambda_0$, $n^* = 1$.

Onde:

a :	expected initial error content, $a > 0$
b :	error detection rate per unit testing effort (proportionality constant), $0 < b < 1$
α , β :	parameters in the testing effort function, $\alpha > 0$, $\beta > 0$
m(n) :	mean value function in the NHPP model, $m(0) = 0$
λ (n)	failure intensity for m(n), $\lambda(0) = 0$

$w(n)$: testing effort expenditure at the n^{th} test run, $w(0) = 0$

$W(n)$: $\sum_{i=0}^{n} w(i)$, $W(0) = 0, W(\infty) = \alpha$

C_1 (C_2) : cost of fixing an error before (after) release of the software, $C_2 > C_1 > 0$

C_3 : cost per unit testing effort expenditure

$C(n)$: total expected software cost incurred during software life cycle, when software is released after n test runs

n_{lc} : software life cycle length expressed in terms of number of test cases

λ_0 : desired failure intensity (> 0)

n^* : optimal number of test runs executed before releasing the software

LH : $ab \prod_{i=0}^{n_1-1} [1-bw(i)]$

RH : $ab \prod_{i=0}^{n_1} [1-bw(i)]$

Critérios de Avaliação

A	0	**B**	1	**C**	0	**D**	2	**E**	1	**F**	0	**Total**	4

Campos extraídos diretamente	
Título	A dynamic software release model
Autores	Süleyman Özekici and Nese A. Çatkan
Ano de publicação	1993
Fonte de publicação	Computational Economics

Abstract

The determination of optimal software release times constitutes an interesting decision making problem which involves the stochastic structure of the underlying software reliability model, as well as various cost parameters. There is an apparent tradeoff between testing the software further to improve its reliability, and releasing it for operational use to decrease the costs. We propose and analyze in depth a new dynamic model with sufficient generality. After each failure, a debugging activity, possibly imperfect, is undertaken and a decision is made regarding the duration of additional testing. If no failure is observed during this time, then the software is released. Otherwise, the failure is debugged and the decision process is repeated in a dynamic fashion. The problem is formulated using dynamic programming and interesting characterizations of the optimal release policy are presented. The dynamic solution procedure is demonstrated by some numerical illustrations. Â© 1993 Kluwer Academic Publishers.

Máquina de Busca	SCOPUS
Campos extraídos a partir do entendimento	
Tipo de Critério	Não apriorístico. Determina a duração dos testes depois que a última falha é corrigida.
Caracterização do software	*Não Informado*
Níveis de testes	Testes funcionais
Modelo Utilizado	Nenhum

Habilidades Necessárias	Nenhuma
Contexto de aplicação	*Não informado*
Princípio do critério	Programação Dinâmica
Objetivo do critério	Minimizar Custo de desenvolvimento
Considera depuração perfeita	Não
Confiabilidade pré-determinada	Não
Orçamento para testes é pré-determinado	Não
Tempo para testes pré-determinado	Não
Número de faltas pré-determinado	Não
Taxa de identificação de falhas pré-determinada	Sim
Considera qualidade dos casos de testes	Não
Considera atividade de testes isolada	Sim
Diferencia taxa de detecção de falhas	Não
Variações do critério	Não há
Modelo de confiabilidade base	A-zekici (1993) (Próprio)
Modelo de Custo	A-zekici (1993) (Próprio)
Distribuição de Falhas	*Não informado*
Unidade de Medida de Falhas	*Não informado*
Tipo de modelo de confiabilidade	Modelo de estimativa de confiabilidade

Suposições

- O tempo entre falhas diminui de acordo com que o software é testado
- Devido a atividade de debug, a confiabilidade do software aumenta. Ou seja, o número de falhas diminui
- O número da faltas restantes diminui de acordo com que o número de falhas experimentadas aumenta.
- $c_1 + c_2 > 0$ e $c_3 > 0$
- $c_1 + [c_3 - c_2 h(+\infty)]r_\infty(0) > 0$

Parâmetros do critério

- c_1 – Custo por tempo de unidade de CPU para teste
- c_2 – Custo de uma falha depois do lançamento durante a fase operacional
- c_3 – Custo de depuração durante a fase de testes

Critério

O artigo apresenta um novo procedimento dinâmico para determinar uma política de liberação de software ótima.
O Procedimento é baseado em um modelo de custo construído que considera os custos dos testes atuais, do processo de depuração e futuros custos devido a falhas na fase de produção.
O critério propõe que o software seja testado por um tempo adicional depois que uma falha foi encontrada e a atividade de depuração seguinte.
A quantidade de tempo dos testes adicionais é determinada pela minimização da função de custo total esperado usando programação dinâmica.
Se nenhuma falha é observada durante o tempo adicional de testes sugerido, o software é entregue para operação. Caso uma falha seja observada, uma nova atividade de depuração é iniciada e os testes continuam por um período de tempo determinado utilizando informações atualizadas do número de faltas esperadas no software.
O critério de parada é expresso pela seguinte formulação em programação dinâmica:

$$v(n) = \min_{u \geq 0} \{c_1 E[X_n \wedge u] + c_2 h(n) \bar{F}_n(u) + [c_3 + v(n+1)] F_n(u)\}$$

where

$$E[X_n \wedge u] = u\bar{F}_n(u) + \int_{[0,u)} sF_n(ds)$$

E a seguinte notação é utilizada:

T_n: time of the nth failure,
X_n: T_{n+1}-T_n,
$F_n(t)$: cumulative distribution function (cdf) of the time to the next failure of the software system after the nth failure (i.e., the cdf of X_n),
$\bar{F}_n(t)$: $1 - F_n(t)$,
$r_n(t)$: failure rate function corresponding to $F_n(t)$,
$h(n)$: expected number of faults remaining after the nth failure,
$\Delta h(n)$: $h(n) - h(n+1)$,
u: decision variable representing software release time,
$X_n \wedge u$: $\min\{X_n, u\}$,
c_1: cost of unit CPU time for testing,
c_2: cost of failure after release during the operational phase,
c_3: cost of debugging during the testing phase,
$v(n)$: the minimum expected total cost using the optimal policy after the nth failure.

Além disto, o critério exige que as seguintes suposições sejam atendias:

ASSUMPTION 1.

1. $r_n(t)$ is continuous and decreasing in $t \geq 0$ for all $n \geq 0$,
2. $r_n(t)$ is decreasing in n for all $t \geq 0$,
3. $h(n)$ is convex decreasing in $n \geq 0$, i.e., $h(n)$ and $\Delta h(n)$ are both decreasing in $n \geq 0$.

ASSUMPTION 2.

1. $c_1 + c_3 > 0$ and $c_2 > 0$,
2. $c_1 + [c_3 - c_2 h(+\infty)] r_\infty(0) > 0$.

Critérios de Avaliação													
A	0	**B**	1	**C**	0	**D**	2	**E**	1	**F**	1	**Total**	5

Campos extraídos diretamente	
Título	A Markov chain model for statistical software testing
Autores	Whittaker, James A. and Thomason, Michael G.
Ano de publicação	1994
Fonte de publicação	IEEE Transactions on Software Engineering

Abstract

Statistical testing of software establishes a basis for statistical inference about a software system's expected field quality. This paper describes a method for statistical testing based on a Markov chain model of software usage. The significance of the Markov chain is twofold. First, it allows test input sequences to be generated from multiple probability distributions, making it more general than many existing techniques. Analytical results associated with Markov chains facilitate informative analysis

of the sequences before they are generated, indicating how the test is likely to unfold. Second, the test input sequences generated from the chain and applied to the software are themselves a stochastic model and are used to create a second Markov chain to encapsulate the history of the test, including any observed failure information. The influence of the failures is assessed through analytical computations on this chain. We also derive a stopping criterion for the testing process based on a comparison of the sequence generating properties of the two chains.

Máquina de Busca	IEEE
Campos extraídos a partir do entendimento	
Tipo de Critério	Não apriorístico. Determina quando parar os testes.
Caracterização do software	*Não informado*
Níveis de testes	Testes funcionais
Modelo Utilizado	Cadeia de Markov. Descrevendo o perfil operacional do software.
Habilidades Necessárias	Modelar perfil de utilização com cadeia de Markov
Contexto de aplicação	Comercialização em larga escala
Princípio do critério	Modelagem do uso do software em uma cadeia de Markov
Objetivo do critério	Maximizar confiabilidade Atingir uma semelhança ideal entre a cadeia de Markov ideal e a cadeia real
Considera depuração perfeita	*Não Informado*
Confiabilidade pré-determinada	*Não Informado*
Orçamento para testes é pré-determinado	*Não Informado*
Tempo para testes pré-determinado	*Não informado*
Número de faltas pré-determinado	*Não Informado*
Taxa de identificação de falhas pré-determinada	*Não Informado*
Considera qualidade dos casos de testes	*Não Informado*
Considera atividade de testes isolada	Sim
Diferencia taxa de detecção de falhas	*Não Informado*
Variações do critério	Não há
Modelo de confiabilidade base	*Não aplicável*
Modelo de Custo	*Não aplicável*
Distribuição de Falhas	*Não aplicável*
Unidade de Medida de Falhas	*Não aplicável*
Tipo de modelo de confiabilidade	*Não aplicável*
Suposições	
Não Informado	
Parâmetros do critério	
Não Informado	
Critério	
O Autor propõe um critério de parada baseado a comparação de duas cadeias de Markov. O uso do sistema é representado por uma cadeia de Markov (Cadeia U) que consiste em estados em que o sistema pode estar, suas transições, as entradas do sistema de cada transição (por exemplo o pressionamento de uma tecla) e a probabilidade daquela transição. As probabilidades de transição de cada estado são retiradas do uso do protótipo do sistema ou de versões anteriores. Em caso de não haver fontes que revelem a probabilidade de uma transição ela é	

estimada utilizando padrões de uso.
Um exemplo de cadeia de uso pode ser visto a seguir:

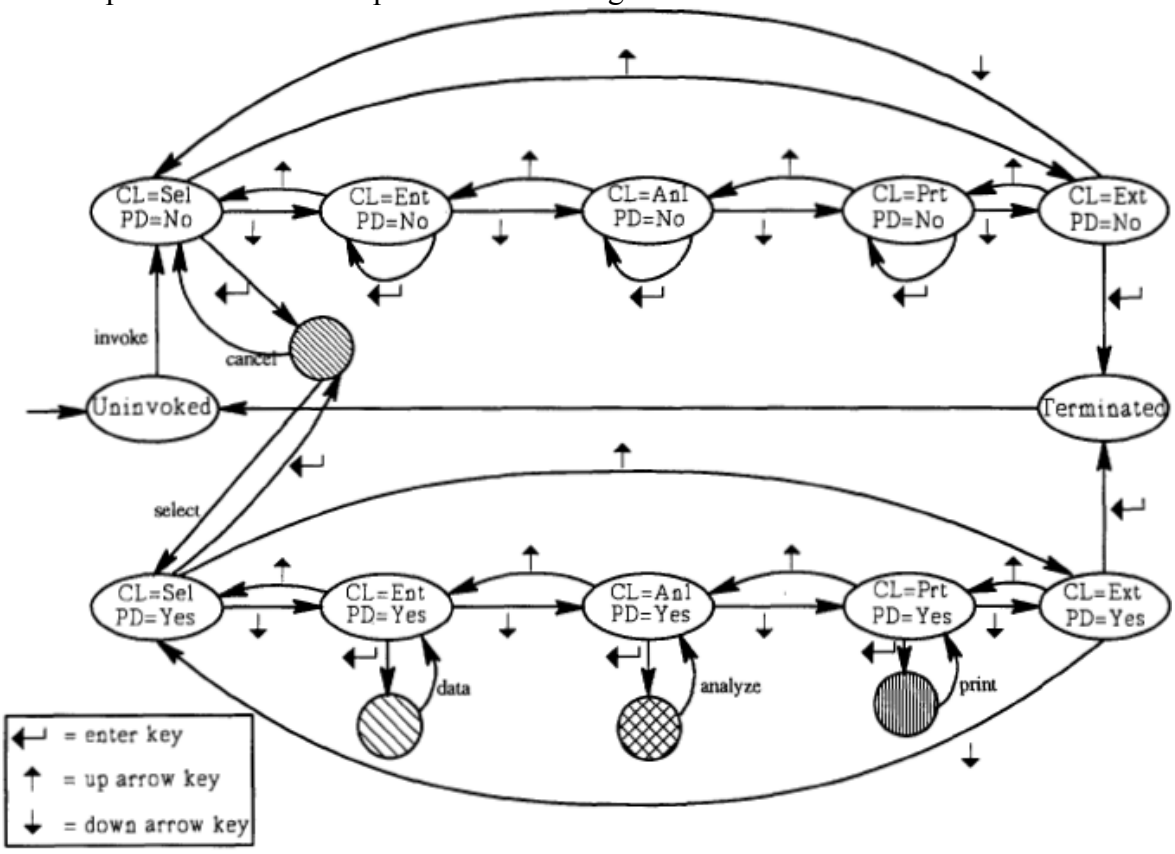


Fig. 2. Usage chain (structure) for the example.

Depois de construir a cadeia de uso é construído uma cadeia de testes. Esta cadeia de teste é construída por fases:

1) Primeiro é copiado a cadeia de uso e todas as probabilidades são zeradas (T0 = U)
2) Uma sequência de teste é rodada e, ao passar por um arco, sua probabilidade é incrementada, criando T1
3) Repete-se o procedimento obtendo T2 a partir de T1 e assim por diante sempre obtendo Ti através de Ti-1
4) Quando uma correção ocorre, a freqüência dos arcos são reiniciadas

Com isto o Autor tem duas cadeias:

- Cadeia de uso (U) – Representa o que ocorreria na ausência de falhas. O cenário ideal dado pelas probabilidades de uso
- Cadeia de teste (T) – Representa o que realmente ocorreu

Assim, o autor utiliza a seguinte fórmula para verificar a semelhança entre as duas cadeias:

$$D(U,T) = \sum_{ij} \pi_i p_{ij} \log_2 \frac{p_{ij}}{\hat{p}_{ij}},$$

Os testes devem parar quando uma semelhança aceitável entre a cadeia U e a cadeia T for atingida.

Onde:

- π – distribuição estacionária de U

• P*ij* –probabilidade de transição de i para j em U ^p_{ij} –probabilidade de transição de i para j em T

Critérios de Avaliação												
A	0	**B**	1	**C**	0	**D**	2	**E**	1	**F**	0	**Total**	4

Campos extraídos diretamente	
Título	A New Criterion for the Optimal Software Release Problems Moving Average Quality Control Chart with Bootstrap Sampling
Autores	Kimura, Mitsuhiro and Fujiwara, Takaji
Ano de publicação	2009
Fonte de publicação	ADVANCES IN SOFTWARE ENGINEERING, PROCEEDINGS
Abstract	
This paper proposes a new practical method for determining when to stop software testing. This issue has been widely known as the optimal release problem of software product, and many researchers have been developing mathematical models for finding the solution. We try to develop a new quality control charting to help making the right decision for it, by employing the moving average model and bootstrap scheme. After discussing the modeling, we show all example of the statistical decision making of the optimal software release time.	
Máquina de Busca	Scopus
Campos extraídos a partir do entendimento	
Tipo de Critério	Não apriorístico. Avalia se a confiabilidade desejada foi atingida.
Caracterização do software	*Não informado*
Níveis de testes	Não informado
Modelo Utilizado	Nenhum
Habilidades Necessárias	Nenhum
Contexto de aplicação	*Não informado*
Princípio do critério	Probabilidade e estatística
Objetivo do critério	Atingir confiabilidade desejada
Considera depuração perfeita	*Não informado*
Confiabilidade pré-determinada	Sim
Orçamento com testes pré-determinado	Não
Tempo para testes pré-determinado	Não
Número de faltas pré-determinado	*Não informado*
Taxa de identificação de falhas pré-determinada	*Não informado*
Considera qualidade dos casos de testes	Não
Considera atividade de testes isolada	Sim
Diferencia taxa de detecção de falhas	Não
Variações do critério	Não há
Modelo de confiabilidade base	*Não aplicável*
Modelo de Custo base	*Não aplicável*
Distribuição de Falhas	*Não aplicável*
Unidade de Medida de Falhas	*Não aplicável*
Tipo de modelo de confiabilidade	*Não aplicável*
Suposições	
Não informado	
Parâmetros do critério	

- j – ponto onde a qualidade começa a ser avaliada
- k – quantidade de bootstrap
- C_1 - Constante arbitraria

Critério

O artigo apresenta um critério de parada para testes de software que se baseia na análise da quantidade de falhas restantes no software.
É considerado que exista uma série temporal de valores estimados de avaliações da confiabilidade resultados da avaliação de uma fase de testes (esses resultados podem ser obtidos por modelos de confiabilidade de software ou por métodos estatísticos).
A partir desta série, a quantidade de falhas no software é estimada.
O critério de parada é representado pela seguinte equação:

$$t^* = \inf_{t_i (i=j,\ldots,n)} \left\{ \Pr\left[M_{i,3} \leqq a_r \times \hat{H}(\infty) \right] \geqq \alpha \right\}.$$

Onde,

- X_i - Variável aleatória que representa a medida da i-ésima estimativa da avaliação da confiabilidade do software
- t_i - i-ésimo teste
- μ_i - Médiade Xi
- s_2 - Variância de Xi
- x_i - Realização/Instanciação de Xi (número restantes de falhas no software)
- H(t) – Descreve algum modelo de curva de crescimento (exponencial, delayed-S-shaped)

Critérios de Avaliação

A	0	**B**	1	**C**	0	**D**	2	**E**	1	**F**	0	**Total**	4

Campos extraídos diretamente	
Título	A software cost model for quantifying the gain with considerations of random field environments
Autores	Teng, Xiaolin and Pham, Hoang
Ano de publicação	2004
Fonte de publicação	IEEE Transactions on Computers

Abstract

In Ihis paper, we present a software gain model under random field environment with consideration of not only time to remove faults during in-house testing, cost of removing faults during beta testing, risk cost due to software failure; but also the benefits from reliable executions of the software during the beta testing and field operation. To our knowledge, this is the first study that incorporates the random field environmental factor into the cost model. We also provide an optimal release policy in which the net gain of the software development process is maximized. This gain model can help managers and developers to determine when to stop testing the software and release it to beta testing users-and to end-users.

Máquina de Busca	IEEE
Campos extraídos a partir do entendimento	
Tipo de critério	Não apriorístico. Encontra o tempo que maximiza o ganho liquido do software.

Caracterização do software	*Não informado*. O critério foi testado em um software de telecomunicação
Níveis de testes	Testes internos (pela empresa que desenvolve o software) Testes beta
Modelo Utilizado	Nenhum
Habilidades Necessárias	Nenhuma
Contexto de aplicação	*Não informado*
Princípio do critério	Modelo de confiabilidade
Objetivo do critério	Maximizar o ganho líquido de desenvolvimento do software
Considera depuração perfeita	Não
Confiabilidade pré-determinada	Não
Orçamento para testes é pré-determinado	Não
Tempo para testes pré-determinado	Não
Número de faltas pré-determinado	Sim
Taxa de identificação de falhas pré-determinada	Sim
Considera qualidade dos casos de testes	Não
Considera atividade de testes isolada	Sim
Diferencia taxa de detecção de falhas	Não
Variações do critério	Não há
Modelo de confiabilidade base	Teng and Pham (2003)
Modelo de Custo	Pham and Zhang (1999) (Evolução deste modelo)
Distribuição de Falhas	NHPP
Unidade de Medida de Falhas	Número acumulado de falhas
Tipo de modelo de confiabilidade	Modelo de estimativa de confiabilidade

Suposições

1. *Software beta testing cost*. After in-house testing, the software will be released to some users (called the beta-tester) to conduct field test (beta testing) for some time period. Software faults can still be removed from the software during beta testing. After beta testing, the software product will be finally released to all end-users.
2. *Benefits from reliable executions of the software product*. The more reliable the software product is during beta testing and final operation, the more benefits for software organization can be obtained.
3. *The environmental factor on the software cost*. Different from the in-house testing environment, the beta testing environment is closed to the end-user environment, which is a randomly distributed environment.

Parâmetros do critério

- C_0 – Custo de configuração dos testes
- C_1 – Testes de software in-house (no ambiente de desenvolvimento) por unidade de tempo
- C_2 – Custo por remoção de faltas por unidade de tempo durante teste in-house
- C_3 – Custo por remoção de faltas por unidade de tempo durante beta teste
- C_4 – Custo de penalidade devido a falha do software
- C_5 –Benefícios devido ao software não falhar durante o beta teste

- C_6 – Benefícios devido ao software não falhar durante período de operação em campo

Critério

O artigo apresenta um modelo de ganho de softwares que funcionam em ambientes aleatórios. Ou seja, software que rodam em diferentes locais.
O modelo considera o tempo de remoção de falhas, o custo de remoção de falhas durante testes beta, custo devido ao risco de uma falha, benefícios de execuções confiáveis do software durante os testes beta e durante produção.
A equação abaixo expressa o modelo:

$$E(T) = C_5 \cdot R(T_w|T) + (C_4 + C_6) \cdot R(x|T + T_w) - (C_0 + C_4) - C_1 \cdot T - C_2 \cdot m_1(T) \cdot \mu_y - C_3 \cdot \mu_w \cdot (m_2(T + T_w) - m_2(T))$$

$$m_1(t) = \frac{a}{p-\beta}\left(1 - e^{-b(p-\beta)t}\right) \qquad t \le T$$

$$m_2(t) = \frac{a}{p-\beta}\left(1 - e^{-b(p-\beta)T}\left(\frac{\theta}{\theta + b(p-\beta)(t-T)}\right)^{\gamma}\right) \qquad t \ge T. \tag{8}$$

A partir do estudo do modelo o tempo de entrega ótimo do software (T*) é calculado maximizando o E(T) (ganho líquido no desenvolvimento do software).
O seguinte teorema representa o critério de parada:

Theorem. *Given values of C_0, C_1, C_2, C_3, C_4, C_5, C_6, x, μ_y, μ_w, T_w, the optimal value of T, say T^*, which maximizes the expected net gain $E(T)$ is as follows:*

1. *If $u(0) \le C$ and*
   a. *If $y(0) \le 0$, then $T^* = 0$;*
   b. *If $y(\infty) > 0$, then $T^* = \infty$;*
   c. *If $y(0) \ge 0$, $y(T) \ge 0$ for $T \in (0, T']$ and $y(T) < 0$ for $T \in (T', \infty]$, then $T^* = T'$, where $T' = y^{-1}(0)$*
2. *If $u(\infty) > C$ and*
   a. *If $y(0) \ge 0$, then $T^* = \infty$;*
   b. *If $y(\infty) < 0$, then $T^* = 0$;*
   c. *If $y(0) < 0$, $y(T) \le 0$ for $T \in (0, T'']$ and $y(T) > 0$ for $T \in (T'', \infty]$, where $T'' = y^{-1}(0)$, then:*

   $T^* = \infty$ if $E(0) < E(\infty)$; $T^* = 0$ if $E(0) \ge E(\infty)$.

3. *If $u(0) > C$, $u(T) \ge C$ for $T \in (0, T^0]$ and $u(T) < C$ for $T \in (T^0, \infty]$, $T^0 = u^{-1}(C)$, then:*
   a. *If $y(0) < 0$ and*
      i. *If $y(T^0) \le 0$, then $T^* = 0$*
      ii. *If $y(T^0) > 0$, then*
         - *$T^* = 0$ if $E(0) \ge E(T_b)$*
         - *$T^* = T_b$ if $E(0) < E(T_b)$*

      where $T_b = y^{-1}(0)$ and $T_b \ge T^0$
   b. *If $y(0) \ge 0$, then $T^* = T_c$, where $T_c = y^{-1}(0)$.*

Onde:

T The time to stop testing and release the software for field operations

$G(\eta)$ Cumulative distribution function of random environmental factor

γ Shape parameter of field environmental factor

θ Scale parameter of field environmental factor

$N(t)$ The number of software failures discovered by time t

$m(t)$ Expected number of software failures detected by time t, $m(t) = E[N(t)]$

$m_1(t)$ Expected number of software failures detected during in-house testing by time t, $t \leq T$

$m_2(t)$ Expected number of software failures detected, since the beginning of the beta testing, by time $t \geq T$

$m_F(t|\eta)$ Expected number of software failures detected in field by time t

C_0 Set-up cost for software testing

C_1 Software in-house testing per unit time

C_2 (C_3) Cost of removing a fault per unit time during in-house (beta) testing

C_4 Penalty cost due to software failure

C_5 (C_6) Benefits if software does not fail, during beta testing (in field operation)

μ_y (μ_w) Expected time to remove a fault during in-house (beta) testing phase

a Number of initial software faults at the beginning of testing

a_F Number of initial software faults at the beginning of the field operations

t_r Actual time to stop testing and release the software

b Fault detection rate per fault

T_w Time length of the beta testing

x Time length that the software is going to be used

p Probability that a fault is successfully removed from the software

β Probability that a fault is introduced into the software during debugging and $\beta << p$

Critérios de Avaliação													
A	0	**B**	1	**C**	2	**D**	2	**E**	1	**F**	1	**Total**	7

Campos extraídos diretamente	
Título	A software cost model with warranty and risk costs
Autores	Hoang Pham and Xuemei Zhang
Ano de publicação	1999
Fonte de publicação	A software cost model with warranty and risk costs
Abstract	
In this paper, a cost model with warranty cost, time to remove each error detected in the software system, and risk cost due to software failure is developed. A software reliability model based on	

non-homogeneous Poisson process is used. The optimal release policies to minimize the expected total software cost are discussed. A software tool is also developed using Excel and Visual Basic to facilitate the task of determining the optimal software release time. Numerical examples are provided to illustrate the results	
Máquina de Busca	IEEE
Campos extraídos a partir do entendimento	
Tipo de Critério	Não apriorístico. Critério para minimizar o custo da qualidade.
Caracterização do software	*Não informado.* Mas o critério foi testado em um software comercial com 350 mil linhas de código e 2 mil erros de programação
Níveis de testes	*Não informado*
Modelo Utilizado	Nenhum
Habilidades Necessárias	Nenhuma
Contexto de aplicação	*Não informado*
Princípio do critério	Modelo de Confiabilidade
Objetivo do critério	Minimizar custo total de desenvolvimento
Considera depuração perfeita	Não
Confiabilidade pré-determinada	Não
Orçamento para testes é pré-determinado	Não
Tempo para testes pré-determinado	Não
Número de faltas pré-determinado	Sim
Taxa de identificação de falhas pré-determinada	Sim
Considera qualidade dos casos de testes	Não
Considera atividade de testes isolada	Sim
Diferencia taxa de detecção de falhas	Não
Variações do critério	Não há
Modelo de confiabilidade base	Okumoto and Goel (1979)
Modelo de Custo	Pham and Zhang (1999) (Próprio)
Distribuição de Falhas	NHPP
Unidade de Medida de Falhas	Número acumulado de falhas
Tipo de modelo de confiabilidade	Modelo de estimativa de confiabilidade
Suposições	
• Existe um custo de configuração no inicio do processo de desenvolvimento do software • No início dos testes, o custo aumenta com uma grande inclinação, depois o crescimento perde aceleração • O custo para remoção de um erro durante o período de depuração é proporcional ao tempo total de remoção de todos os erros deste período • O curto para remoção de um erro durante o período de garantia é proporcional ao tempo de remoção de todos os erros detectados no intervalo de tempo [T, $T+T_w$]	
Parâmetros do critério	
• C_0 – custo de configuração dos testes de software • C_1 – custo de teste por unidade de tempo • C_2 – custo por unidade de tempo para remover um erro durante a fase de testes • C_3 – custo por unidade de tempo para remover um erro durante o período de garantia • C_4 – perda devido a falha do software • T_w – período de garantia	

Critério

O artigo apresenta um critério de parada baseado no critério de Okumoto and Goel (1979).
O critério apresentado se baseia em um modelo que considera também o custo de teste, o custo de remoção de erros detectados durante a fase de teste, custo de remoção de erros detectados durante o período de garantia e o custo do risco devido a uma falha no software.
O Critério de parada é dado pela minimização do custo total do sistema e é definido pelo seguinte teorema:

THEOREM. *Given* $C_0, C_1, C_2, C_3, C_4, x, \mu_y, \mu_w, T_w$, *the optimal value of* T, *say* T^*, *which minimizes the expected total cost of the software is as follows:*

1) *If* $u(0) \geq C$ *and*
   a) *If* $y(0) \geq 0$, *then* $T^* = 0$;
   b) *If* $y(\infty) < 0$, *then* $T^* = \infty$;
   c) *If* $y(0) < 0$ $y(T) < 0$ *for* $T \in (0, T']$ *and* $y(T) > 0$ *for* $T \in (T', \infty)$, *then* $T^* = T'$ *where* $T' = y^{-1}(0)$.
2) *If* $u(\infty) < C$ *and*
   a) *If* $y(0) \leq 0$, *then* $T^* = \infty$;
   b) *If* $y(\infty) > 0$, *then* $T^* = 0$;
   c) *If* $y(0) > 0$ $y(T) > 0$ *for* $T \in (0, T'']$ *and* $y(T) < 0$ *for* $T \in (T'', \infty)$, *then:*

      $T^* = 0$ if $E(0) \leq E(\infty)$

      $T^* = \infty$ if $E(0) > E(\infty)$,

      where $T'' = y^{-1}(0)$.
3) *If* $u(0) < C$, $u(T) \leq C$ *for* $T \in (0, T^0]$, *and* $u(T) > C$ *for* $T \in (T^0, \infty)$, *where* $T^0 = u^{-1}(C)$, *then:*
   a) *If* $y(0) \geq 0$, *then*

      $T^* = 0$ *if* $E(0) \leq E(T_b)$;

      $T^* = T_b$ *if* $E(0) > E(T_b)$;

      where $T_b = \inf\{T > T_a : y(T) > 0\}$;
   b) *If* $y(0) < 0$, *then* $T^* = T_b''$, *where* $T_b'' = y^{-1}(0)$.

Onde:

$R(x/T)$	reliability function of software by time T for a mission time x
T	software testing time
T^*	optimal software release time
C_0	set-up cost for software testing
C_1	software test cost per unit time
C_2	cost of removing an error per unit time during testing period
C_3	cost of removing an error per unit time during warranty period
C_4	loss due to software failure
$E(T)$	expected cost of software systems at time T
Y	variable of time to remove an error during testing phase
μ_y	expected time to remove an error during testing phase
W	variable of time to remove an error during warranty period in operation phase
μ_w	expected time to remove an error during warranty period in operation phase, which is E(W)
T_w	period of warranty time
α	the discount rate of the testing cost

Critérios de Avaliação													
A	0	**B**	1	**C**	1	**D**	2	**E**	1	**F**	1	**Total**	6

Campos extraídos diretamente	
Título	A software reliability model in the embedded system
Autores	Kapsu Kim and Chisu Wu
Ano de publicação	1994
Fonte de publicação	Software Testing, Reliability and Quality Assurance, 1994. Conference Proceedings., First International Conference on
Abstract	
In this paper, we propose a software reliability model for estimating, measuring, and controlling software reliability of embedded system, and a software test stopping equation for determining software testing time. It is not easy to correct errors occurred in embedded system on site. The proposed model can be applied to the embedded system.	
Máquina de Busca	IEEE
Campos extraídos a partir do entendimento	
Tipo de Critério	Não apriorístico. Determina o momento de parada que satisfaz uma confiabilidade pré-estabelecida pelo usuário.
Caracterização do software	Softwares Embarcados. O Critério foi aplicado ao software de uma impressora de feixe de laser.
Níveis de testes	*Não informado*
Modelo Utilizado	Nenhum
Habilidades Necessárias	Nenhuma
Contexto de aplicação	Software embarcado para ser comercializado em escala
Princípio do critério	Modelo de Confiabilidade
Objetivo do critério	Atingir confiabilidade especificada (Número de faltas aceitável pelo usuário)
Considera depuração perfeita	Não. (O defeito não é corrigido imediatamente, mas não diz se é corrigido instantaneamente)

Confiabilidade pré-determinada	Sim
Orçamento para testes é pré-determinado	Não
Tempo para testes pré-determinado	Não
Número de faltas pré-determinado	Não
Taxa de identificação de falhas pré-determinada	Não
Considera qualidade dos casos de testes	Não
Considera atividade de testes isolada	Sim
Diferencia taxa de detecção de falhas	Não
Variações do critério	Não há
Modelo de confiabilidade base	Kim and Wu (1994) (Próprio)
Modelo de Custo	*Não aplicável*
Distribuição de Falhas	Exponencial
Unidade de Medida de Falhas	Tempo médio entre falhas
Tipo de modelo de confiabilidade	Modelo de estimativa de confiabilidade

Suposições

- Erros devem ser corrigidos em um intervalo de tempo específico e erros detectados durante a atividade de testes são acumulados até um intervalo de tempo especificado
- A taxa de detecção de erros é igual a taxa de correção de erros
- Os erros são classificados em níveis de severidade de acordo com que causam faltas no software (ele usa o termo falta, mas dá para perceber que esta frase quer dizer que as faltas são classificadas de acordo com a severidade das falhas que elas causam)

Parâmetros do critério

- Quantidade de falhas ocorridas
- Momento da ocorrência de falhas

Critério

O autor propõe um critério de parada que pode ser aplicado em softwares embarcados. Este critério é baseado em um modelo de confiabilidade também proposto pelo autor.
A seguinte equação define o critério de parada:

$$t > -\frac{1}{C}\log\left[\ 1 - \frac{(K^2+2\times A)+K\times(K^2+4\times A)}{2(N_0 - \sum_{k=1}^{k-1}\sum_{i=1}^{n} N_{ki} + \sum_{k=1}^{k-1}\sum_{i=1}^{n} M_{ki})}\right.$$

Onde:

- C – é uma constante
- N_0 – Número inicial de erros
- N – Número de que já foram corrigidos
- M – Número de erros que foram inseridos quando os erros foram corrigidos
- A – Número de erros aceitável pelos usuários especificado nos requisitos
- K – Constante de nível de confidencia (k = 1 então 68,23%; k = 2 – 95,44%; k = 3 – 99,74%)

Critérios de Avaliação

A	0	**B**	1	**C**	2	**D**	1	**E**	0	**F**	1	**Total**	5

Campos extraídos diretamente	
Título	A statistical approach for determining release time of software system with modular structure

Autores	Masuda, Y. and Miyawaki, N. and Sumita, U. and Yokoyama, S.
Ano de publicação	1989
Fonte de publicação	Reliability, IEEE Transactions on

Abstract

An algorithmic procedure is developed for determining the release time of a software system with multiple modules where the underlying module structure is explicitly incorporated. Depending on how much the module is used during exception, the impact of software bugs from one module is distinguished from the impact of software bugs from another module. It is assumed that software bugs in one module have i.i.d. lifetimes but lifetime distributions can vary from one module to another. For the two cases of exponential and Weibull lifetimes, statistical procedures are developed for estimating distribution parameters based on failure data during the test period for individual modules. In the exponential case, the number of software bugs can also be estimated following H. Joe and N. Reid (J. Amer. Statis. Assoc., vol.80, p.222-6, 1985). These estimates enable one to evaluate the average cost due to undetected software bugs. By introducing an objective function incorporating this average cost as well as the time-dependent value of the software system and the cumulative running cost of the software testing, a decision criterion is given for determining whether the software system should be released or the test should be continued further for a certain period Delta;. The validity of this procedure is examined through extensive Monte-Carlo simulation.

Máquina de Busca	IEEE
Campos extraídos a partir do entendimento	
Tipo de Critério	Não apriorístico. Determina se os testes devem parar ou se devem continuar por um período de tempo.
Caracterização do software	Software composto por múltiplos módulos
Níveis de testes	*Não informado*
Modelo Utilizado	Nenhum
Habilidades Necessárias	Nenhuma
Contexto de aplicação	*Não informado*
Princípio do critério	Modelo de confiabilidade
Objetivo do critério	Determinar se a próxima rodada de testes é necessária
Considera depuração perfeita	Sim
Confiabilidade pré-determinada	Não
Orçamento para testes é pré-determinado	Não
Tempo para testes pré-determinado	Não
Número de faltas pré-determinado	Sim
Taxa de identificação de falhas pré-determinada	Sim
Considera qualidade dos casos de testes	Não
Considera atividade de testes isolada	Sim
Diferencia taxa de detecção de falhas	Não
Variações do critério	Não há
Modelo de confiabilidade base	Massuda (1989) (Próprio)
Modelo de Custo	Massuda (1989) (Próprio)
Distribuição de Falhas	• Exponencial • Weibull
Unidade de Medida de Falhas	Taxa de ocorrência falhas
Tipo de modelo de confiabilidade	Modelo de estimativa de confiabilidade
Suposições	

Assumptions

1. The β_j's remain constant even after the delivery of the software system, provided that testing conditions simulate real conditions and the length of the period in which the software system is in use is sufficiently large.
2. Each of N_j bugs in module j has a random lifetime X_j, that is, when the accumulation of the CPU time spent for processing module j amounts to $X_j(w)$, this bug is detected.
3. The X_j has a Weibull distribution with positive parameters (a_j, b_j):

$$F_j(x) = P[X_j \leq x] = 1 - e^{-a_j x^{b_j}}. \quad (2\text{-}1)$$

When the test conditions simulate real conditions, parameters (a_j, b_j) remain constant. The exponential distribution is included as a special case with $b_j = 1$.

4. All underlying random variables are mutually statistically independent of each other.
5. When a bug is detected, it is removed.
6. The CPU time spent for processing module j during the test period is the average value $\beta_j \alpha_T T$. □

Parâmetros do critério

- Número de módulos do software
- Tempo em que o software permanecerá em teste
- Tempo em que o software será executado durante seu ciclo de vida
- Tempo em que cada módulo do software ficará em execução

Critério

O Artigo apresenta um algoritmo que define um critério de parada que pode ser aplicado em software desenvolvido em múltiplos módulos. Este critério é baseado na frequência de utilização dos módulos.
O critério consiste em observar os resultados dos testes em determinado intervalo de tempo e aplicar uma equação para determinar se a próxima rodada de testes é necessária.
Depois de observar os resultados de testes no intervalo [0, T], deve-se aplicar a equação abaixo para saber se um teste de tamanho τ é necessário:

$$v(\tau) = V_1(T+\tau) - V_2(\tau) - V_3(T+\tau).$$

Onde:

- V1(t) – valor do sistema do tempo t
- V2(t) – custo médio devido a bugs não detectados
- V3(t) – custo cumulativo do teste de software quando o tempo de entrega é no tempo t

Depois disto, calcula-se também o resultado de v(0) utilizando a mesma equação com valores estimados.
Se v(0) >= v(τ) então os testes podem parar o sistema ser entregado no tempo T.
Caso contrário, os testes devem continuar até o tempo T + τ.
Esta comparação deve ser feita novamente até o tempo de entrega do software ser determinado.

Notation

K	number of modules in the software system.
T	length of testing period.
α_T	portion of time in which the software system is in use during the test period.
α_U	portion of time in which the software system is in use after the release of the software.
β_j	portion of time in which module j is executed, given that the software system is executed, $j=1,2,\ldots,K$, where $\Sigma_{j=1}^{K}\ \beta_j=1$.
N_j	number of bugs in module j.
X_j	cumulative execution time of module j to detect a specific bug in module j.
r_j	number of bugs detected in module j so far.
$t_j(k)$	time at which bug k in module j is detected, $k=1,2,\ldots,r_j$.
$v(\tau)$	profit obtained by releasing the software system after τ time units from T.
$U(t)$	step function at $t=1$: $U(t)\equiv 1$ for $t\geq 1$ and $U(t)\equiv 0$ otherwise.
$u(t)$	$1-U(t)$
t_D	delivery time of the software system.
W	lifetime of the software system.
MLE	maximum likelihood estimator

Critérios de Avaliação

A	2	**B**	0	**C**	2	**D**	2	**E**	1	**F**	1	**Total**	8

Campos extraídos diretamente	
Título	A study of service reliability and availability for distributed systems
Autores	Dai, Y.S. and Xie, M. and Poh, K.L. and Liu, G.Q.
Ano de publicação	2003
Fonte de publicação	Reliability Engineering and System Safety

Abstract

Distributed systems are usually designed and developed to provide certain important services such as in computing and communication systems. In this paper, a general model is presented for a centralized heterogeneous distributed system, which is widely used in distributed system design. Based on this model, the distributed service reliability which is defined as the probability of successfully providing the service in a distributed environment, an important performance measure for this type of systems, is investigated. An application example is used to illustrate the procedure. Furthermore, with the help of the model, various issues such as the release time to achieve a service reliability requirement, and the sensitivity of model parameters are studied. This type of analysis is important in the application of this type of models. © 2002 Elsevier Science Ltd. All rights reserved.

Máquina de Busca	Scopus
Campos extraídos a partir do entendimento	
Tipo de Critério	Apriorístico. Revela o tempo de teste necessário para atingir a confiabilidade desejada.
Caracterização do software	Sistemas distribuído homogêneos centralizados (Trata o software como um serviço)

Níveis de testes	Testes funcionais
Modelo Utilizado	Nenhum
Habilidades Necessárias	Nenhuma
Contexto de aplicação	Não informado
Princípio do critério	Modelo geral para representação do sistema distribuído
Objetivo do critério	Atingir disponibilidade desejada
Considera depuração perfeita	*Não informado*
Confiabilidade pré-determinada	Sim
Orçamento para testes é pré-determinado	Não
Tempo para testes pré-determinado	Não
Número de faltas pré-determinado	Não
Taxa de identificação de falhas pré-determinada	Não
Considera qualidade dos casos de testes	Não
Considera atividade de testes isolada	Sim
Diferencia taxa de detecção de falhas	Não
Variações do critério	Não há
Modelo de confiabilidade base	*Não aplicável*
Modelo de Custo	*Não aplicável*
Distribuição de Falhas	*Não aplicável*
Unidade de Medida de Falhas	*Não aplicável*
Tipo de modelo de confiabilidade	*Não aplicável*

Suposições

- O critério foi proposto especificamente para sistemas distribuídos heterogêneos centralizados

Parâmetros do critério

Não informado

Critério

O artigo apresenta um modelo geral para sistemas distribuídos centralizados heterogêneos (CHDS – *Centralized Heterogeneous distributed System*).
CHDS são definidos no artigo como subsistemas gerenciados por um controle central.
Baseado neste modelo, a confiabilidade do serviço distribuído que é definida como a probabilidade de sucesso do funcionamento do serviço em um ambiente distribuído é investigada.
Com o auxílio deste modelo, o tempo de teste do software necessário para atingir a confiabilidade adequada é estudado.
Segue uma representação gráfica da estrutura de um CHDS:

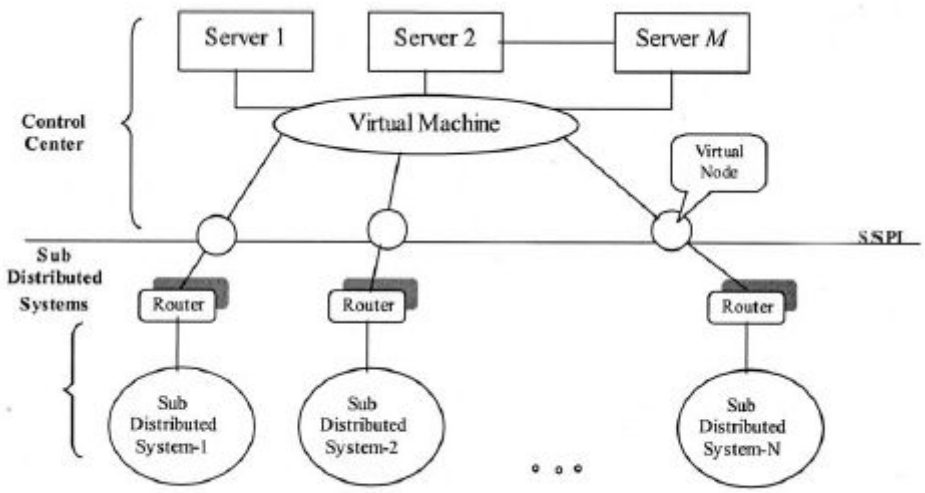


Fig. 1. Structure of the centralized heterogeneous distributed service system.

A confiabilidade do serviço é determinada pela confiabilidade os programas distribuídos em cada subsistema e a disponibilidade do controle central.

O seguinte algoritmo é utilizado para calcular a confiabilidade:

Step 1
Identify the structure of CHDS and relationship between programs and files;

Step 2
Obtain the availability function of the virtual machine with any existing models;

Step 3
Let the virtual machine to be a perfect node in every sub-system and calculate DSR_i $(i = 1, 2, \ldots, N)$;

Step 4
Using the critical path method to determine T_{bf}^j $(j = 1, 2, \ldots, J)$ and T_{bp}^k, T_{ex}^k $(k = 1, 2, \ldots, K)$;

Step 5
Calculate $P_{\mathrm{f}}(j)$ and $P_{\mathrm{pr}}(k)$ as shown in Eqs. (1) and (2).

Step 6
Calculate the distributed service reliability function to the initial time, t_{b}, through Eq. (3).

Onde as equações 1, 2 e 3 são, respectivamente:

$$P_{\mathrm{f}}(j) = A(T_{\mathrm{bf}}^j), \qquad j = 1, 2, \ldots, J. \tag{1}$$

$$P_{\mathrm{pr}}(k) = \int_{T_{\mathrm{bp}}^k}^{T_{\mathrm{bp}}^k + T_{\mathrm{ex}}^k} A(t)\mathrm{d}t / T_{\mathrm{ex}}^k, \qquad k = 1, 2, \ldots, K. \tag{2}$$

$$R_{\mathrm{s}}(t_{\mathrm{b}}) = \prod_{i=1}^{N} \mathrm{DSR}_i \prod_{j=1}^{J} P_{\mathrm{f}}(j) \prod_{k=1}^{K} P_{\mathrm{pr}}(k). \tag{3}$$

A seguinte nomenclatura é utilizada:

Nomenclature

$A(t)$	availability function of VM at time t
DSR_i	distributed system reliability for ith sub-distributed system
$P_0(t)$	probability for the VM in working state at time t
$P_1(t)$	probability for VM in malfunctioning state at time t
$R_s(t_b)$	distributed service reliability function to t_b
t_b	initial time for the service
T_{bf}^j	time point for the jth programs need the files prepared in the VM
T_{bp}^k	beginning time when the kth programs runs in VM
T_{ex}^k	execution time period for those programs in VM
VM	virtual machine
VMi	VM used in sub-system i

Com a fórmula da confiabilidade, é possível calcular a confiabilidade e com isso decidir se ela é suficiente ou não para o sistema específico. Caso seja suficiente os testes podem parar, caso contrário eles devem continuar.

Critérios de Avaliação												
A	0	**B**	1	**C**	2	**D**	2	**E**	1	**F**	0	**Total**	6

Campos extraídos diretamente	
Título	A study of software fault detection and correction process models
Autores	Wu, Y.P. and Hu, Q.P. and Ng, S.H.
Ano de publicação	2006
Fonte de publicação	IEEE International Conference on Management of Innovation and Technology

Abstract

Most of the models for software reliability analysis are based on reliability growth models which deal with the fault detection process only. In this paper, some useful approaches to the modeling of both software fault detection and fault correction processes are discussed. To provide accurate predictions for correct decision-makings, parameters estimation method is critical. Specifically, a new explicit formula for the likelihood function of the combined fault detection and correction process is derived and the maximum likelihood estimates are obtained under various time delay assumptions. As an illustration, actual dataset from a software development project is analyzed. In addition, cost models are discussed in the context of this modeling framework on fault detection and correction. The corresponding effects on optimal release time are analyzed comprehensively. Also, potential benefits of this model on other aspects of software testing management are discussed. © 2006 IEEE.

Máquina de Busca	IEEE
Campos extraídos a partir do entendimento	
Tipo de Critério	Não apriorístico. Encontra momento ótimo de parar os testes minimizando custo.
Caracterização do software	*Não informado*
Níveis de testes	*Não informado*
Modelo Utilizado	Nenhum
Habilidades Necessárias	Nenhuma
Contexto de aplicação	*Não informado*
Princípio do critério	Modelo de confiabilidade (Não só as detecção de faltas é modelada, mas também a correção das faltas)
Objetivo do critério	Minimizar custo de desenvolvimento
Considera depuração perfeita	*Não informado*
Confiabilidade pré-determinada	Não
Orçamento para testes é pré-determinado	Não

Tempo para testes pré-determinado	Não
Número de faltas pré-determinado	Não
Taxa de identificação de falhas pré-determinada	Não
Considera qualidade dos casos de testes	Não
Considera atividade de testes isolada	Sim
Diferencia taxa de detecção de falhas	Não
Variações do critério	Não há
Modelo de confiabilidade base	Okumoto and Goel (1979)
Modelo de Custo base	Xie (1991)
Distribuição de Falhas	NHPP
Unidade de Medida de Falhas	Número acumulado de falhas
Tipo de modelo de confiabilidade	Modelo de estimativa de confiabilidade

Suposições

Não informado

Parâmetros do critério

- c_1 – custo de remoção de uma falta durante fase de teste
- c_2 – custo de remoção de uma falta durante a fase operacional
- c_3 – custo de teste por unidade de tempo

Critério

O artigo aplica o método de estimativa de probabilidade máxima (MLE – *Maximum likelihood estimation*) para a detecção e correção de faltas. O que há de novo na proposta é que os modelos anteriores não modelavam a correção das faltas, apenas a detecção.
A partir daí, o modelo de custo proposto por Xie (1991) é estendido para contemplar, além da correção de faltas, a modelagem de correção de faltas:

$$C = c_1 \cdot m_c(T) + c_2 \cdot \left[m_d(\infty) - m_c(T)\right] + c_3 \cdot T$$

Onde:

- $m_c(T)$ – número total de faltas corrigidas no tempo de entrega T
- $m_d(\infty)$-$m_c(T)$ – número de faltas não corrigidas que inclui dois componentes: faltas não detectadas e faltas detectadas mas não corrigidas

Minimizando este modelo em relação a T o critério de parada T* é calculado.

Critérios de Avaliação

A	0	**B**	1	**C**	0	**D**	2	**E**	1	**F**	0	**Total**	4

Campos extraídos diretamente	
Título	A study of the effect of imperfect debugging on software development cost
Autores	Min Xie and Bo Yang
Ano de publicação	2003
Fonte de publicação	Software Engineering, IEEE Transactions on

Abstract

It is widely recognized that the debugging processes are usually imperfect. Software faults are not completely removed because of the difficulty in locating them or because new faults might be introduced. Hence, it is of great importance to investigate the effect of the imperfect debugging on software development cost, which, in turn, might affect the optimal software release time or operational budget. In this paper, a commonly used cost model is extended to the case of imperfect debugging. Based on this, the effect of imperfect debugging is studied. As the probability of perfect debugging, termed testing level here, is expensive to be increased, but manageable to a certain

extent with additional resources, a model incorporating this situation is presented. Moreover, the problem of determining the optimal testing level is considered. This is useful when the decisions regarding the test team composition, testing strategy, etc., are to be made for more effective testing.

Máquina de Busca	IEEE
Campos extraídos a partir do entendimento	
Tipo de Critério	Não apriorístico. Encontra o tempo de entrega ótimo.
Caracterização do software	*Não informado*
Níveis de testes	*Não informado*
Modelo Utilizado	Nenhum
Habilidades Necessárias	Nenhuma
Contexto de aplicação	*Não informado*
Princípio do critério	Modelo de confiabilidade
Objetivo do critério	Minimizar o custo de desenvolvimento
Considera depuração perfeita	Não
Confiabilidade pré-determinada	*Não informado*
Orçamento para testes é pré-determinado	Não
Tempo para testes pré-determinado	Não
Tempo para testes pré-determinado **Número de faltas pré-determinado**	Não
Taxa de identificação de falhas pré-determinada	Não
Considera qualidade dos casos de testes	Não
Considera atividade de testes isolada	Sim
Diferencia taxa de detecção de falhas	Não
Variações do critério	Não há
Modelo de confiabilidade base	Ohba and Chou (1989)
Modelo de Custo	Xie (1991)
Distribuição de Falhas	NHPP
Unidade de Medida de Falhas	Número acumulado de falhas
Tipo de modelo de confiabilidade	Modelo de estimativa de confiabilidade

Suposições

Não informado

Parâmetros do critério

- C_1 – Custo esperado para remover uma falta durante a fase de testes
- C_2 – Custo esperado para remover uma falta durante a fase de operação
- C_3 – Custo esperado por unidade de teste

Critério

O artigo propõe a extensão do modelo de custo proposto por Xie (1991) para o caso onde há debug imperfeito. Ou seja, para quando a correção de uma falha pode gerar outras falhas.
A alteração no modelo consiste em fazer com que o parâmetro c_3 dependa da composição da equipe de teste e da estratégia de teste utilizada.
O novo modelo de custo proposto é expresso pela seguinte equação:

$$C(T,p) = c_1 m(T) + c_2[m(\infty) - m(T)] + \frac{c}{1-p}T. \qquad (5)$$

Onde:

- m(T) – Função de valor médio da modelagem não homogenia do processo de Poisson (NHPP). Pode ser calculada pela fórmula:

$$m(t) = \frac{a}{p}(1 - e^{-pbt}).$$

A partir do modelo, o seguinte critério de parada é sugerido:

Proposition 1. *The optimal values of p and T, denoted by p^* and T^*, which minimize the expected software cost given by (5) are as follows:*

Case 1. If $K \leq 0$, then $p^* = \inf\{p : h(p) < 0\}$ and $T^* = g(p^*)$.

Case 2. If $K > 0$, then define $p' \equiv \inf\{p : dh/dp < 0\}$ and

1. *If $h(p') > 0$, then $p^* = \min[C(p_1, T_1), C(p_2, T_2)]$ and $T^* = g(p^*)$, where p_1 and p_2 are the solutions to the equation of $h(p) = 0$ and $T_1 = g(p_1)$, $T_2 = g(p_2)$.*
2. *If $h(p') = 0$, then p^* equals the unique solution to the equation of $h(p) = 0$ and $T^* = g(p^*)$.*
3. *If $h(p') < 0$, then p^* and T^* does not exist within $0 < p < 1$ and $T > 0$.*

In the above,

$$K = c\ \ln\left[\frac{ab(c_2 - c_1)}{c}\right] + abc_1 + c,$$

$$h(p) = c(2p-1)\ln\left[\frac{ab(c_2 - c_1)(1-p)}{c}\right] - abc_1(1-p)^2 - c(1-p),$$

and

$$g(p) = \frac{1}{pb}\ln\left[\frac{ab(c_2 - c_1)(1-p)}{c}\right].$$

Onde:

- p – probabilidade de debug imperfeito
- T – tempo
- p* - nível de teste ótimo
- T* - Tempo de entrega ótimo

Critérios de Avaliação													
A	0	**B**	1	**C**	2	**D**	2	**E**	1	**F**	1	**Total**	7

Campos extraídos diretamente	
Título	A study of uncertainty in software cost and its impact on optimal software release time
Autores	Yang, Bo and Hu, Huajun and Jia, Lixin
Ano de publicação	2008
Fonte de publicação	IEEE Transactions on Software Engineering

Abstract

For a development software project, management often faces the dilemma of when to stop testing the software and release it for operation, which requires careful decision-making as it has great impact on both software reliability and project cost. In most existing research on optimal software release problem, the cost considered was the expected cost (EC) of the project. However, what management concerns is the actual cost (AC) of the project rather than the EC. Treatment (such as minimization) of the EC may not ensure a desired low level of the AC, due to the uncertainty (variability) involved in the AC. In this paper, we study the uncertainty in software cost and its impact on optimal software release time in detail. The uncertainty is quantified by the variance of the AC and several risk functions. A risk-control approach to optimal software release problem is proposed. New formulations of the problem which are extensions of current formulations are

developed, and solution procedures are established. Several examples are presented. Results reveal that it seems crucial to take account of the uncertainty in software cost in optimal software release problem, otherwise unsafe decision may be reached which could be a false dawn to management. © 2008 IEEE.	
Máquina de Busca	IEEE
Campos extraídos a partir do entendimento	
Tipo de critério	Apriorístico. Determina momento de entrega que possui custo mínimo ou confiabilidade máxima.
Caracterização do software	*Não informado*
Níveis de testes	*Não informado*
Modelo Utilizado	Nenhum
Habilidades Necessárias	Nenhuma
Contexto de aplicação	*Não informado*
Princípio do critério	Modelo de confiabilidade
Objetivo do critério	• Minimizar o custo (com restrições de risco) • Maximizar a confiabilidade (com restrições de risco)
Considera depuração perfeita	*Não informado*
Confiabilidade pré-determinada	Não
Orçamento com testes pré-determinado	Sim
Tempo para testes pré-determinado	Não
Número de faltas pré-determinado	Não
Taxa de identificação de falhas pré-determinada	Não
Considera qualidade dos casos de testes	Não
Considera atividade de testes isolada	Sim
Diferencia taxa de detecção de falhas	Não
Variações do critério	2. • Minimizar o custo (com restrições de risco) • Maximizar a confiabilidade (com restrições de risco)
Modelo de confiabilidade base	Yang (2008) (Próprio)
Modelo de Custo base	• Pham and Zhang (1999) • Okumoto and Goel (1979)
Distribuição de Falhas	NHPP
Unidade de Medida de Falhas	Número acumulado de falhas
Tipo de modelo de confiabilidade	Modelo de estimativa de confiabilidade
Suposições	
Não informado	
Parâmetros do critério	
• R_0 – Confiabilidade desejada • C_0 – custo permitido para o software (orçamento) • C_1 – custo de remoção de uma falta durante fase de testes • C_2 – custo de remoção de uma falta durante fase de operação • C_3 – custo geral dos testes (pagamento aos membros da equipe de teste)	
Critério	
No artigo é estudado a incerteza (variabilidade) do custo de software e o impacto desta incerteza no	

problema de entrega ótima do software. A discussão se baseia em um NHPP SRM (*Nonhomogeneours Poisson process Software Reliability Model*), no qual o processo de falha do software segue um NHPP.
Baseado no modelo proposto, dois critérios são apresentados:

1. Minimização do custo esperado com restrição de risco

Procedure 1.

i. Obtain T^*_{TBD} $(T^*_{TBD} \geq 0)$ which is the solution to (43)-(44);
ii. If $P_2^{\alpha}(T^*_{TBD}) \leq \beta$, then
$T^* = T^*_{TBD}$; procedure ends.
else
a) Obtain the smallest value of $T(T > T^*_{TBD})$ which makes $P_2^{\alpha}(T) \leq \beta$ satisfied; denote this value by T^1_{P2};
b) $T^* = T^1_{P2}$; procedure ends.

If the cost model used is the generalized software cost model (31), then the solution procedures for Formulation 4 using $P_2^{\alpha}(T)$, $\alpha > 0$ in (45) are given as follows:

Procedure 2.

i. Obtain T^*_{TBD} $(T^*_{TBD} \geq 0)$, which is the solution to (41);
ii. If P_2^{α} $(T^*_{TBD}) \leq \beta$, then
$T^* = T^*_{TBD}$; procedure ends.
else
a) Obtain the smallest value of $T(T > T^*_{TBD})$, which makes $P_2^{\alpha}(T) \leq \beta$ satisfied;
denote this value by T^1_{P2};
b) Obtain the largest value of T $(0 \leq T < T^*_{TBD})$, which makes $P_2^{\alpha}(T) \leq \beta$ satisfied; denote this value by T^2_{P2};
c) If $E[C(T^1_{P2})] < E[C(T^2_{P2})]$, then
$T^* = T^1_{P2}$; procedure ends.
else if $E[C(T^1_{P2})] > E[C(T^2_{P2})]$, then
$T^* = T^2_{P2}$; procedure ends.
else
$T^* = T^1_{P2}$ or $T^* = T^2_{P2}$, i.e., there are two solutions to the optimal software release problem; procedure ends.

2. Maximização da confiabilidade com restrição de custo

Theorem 3. *If the software testing process is modeled by an NHPP SRM, then, under Assumptions 1 and 2, $P_3^{C_0}(T)$ for the basic software cost model (24) is*

$$P_3^{C_0}(T) = 1 - e^{-m_\infty} \sum_{k=0}^{k_{up}} \sum_{j=0}^{j_k} \frac{[m(T)]^k}{k!} \frac{[m_\infty - m(T)]^j}{j!}, \qquad (48)$$

where

$$k_{up} \equiv \langle \gamma_3 \rangle, \quad j_k \equiv \left\langle \frac{c_1(\gamma_3 - k)}{c_2} \right\rangle, \quad \gamma_3 \equiv \frac{C_0 - c_3 T^{\kappa}}{c_1}. \qquad (49)$$

Onde:

- T – tempo de entrega do software (release time)
- C(T) – custo até o tempo de entrega
- E[C(T)] – custo estimado até o tempo de entrega
- R(x|T) – confiabilidade do software se entregue no tempo T

Critérios de Avaliação												
A	2	**B**	0	**C**	0	**D**	2	**E**	1	**F**	0	**Total**	5

Campos extraídos diretamente	
Título	An empirical Bayesian stopping rule in testing and verification of behavioral models
Autores	Sahinoglu, Mehmet
Ano de publicação	2003
Fonte de publicação	IEEE Transactions on Instrumentation and Measurement

Abstract

Software stopping rules are tools to effectively minimize the time and cost involved in software testing. The algorithms serve to guide the testing process such that if a certain level of branch or fault (or failure) coverage is obtained without the expectation of further significant coverage, then the testing strategy can be stopped or changed to accommodate further, more advanced testing strategies. By combining cost analysis with a variety of stopping-rule algorithms, a comparison can be made to determine an optimally cost-effective stopping point. A novel cost-effective stopping rule using empirical Bayesian principles for a nonhomogeneous Poisson counting process compounded with logarithmic-series distribution (LSD) is derived and satisfactorily applied to digital software testing and verification. It is assumed that the software failures or branches covered, whichever the case may be, clustered at the application of a given test-case are positively correlated, i.e., contagious, implying that the occurrence of one software failure (or coverage of a branch) positively influences the occurrence of the next. This phenomenon of clustering of software failures or branch coverage is often observed in software testing practice. The r.v. w<inf>i</inf> of the failure-clump size of the interval is assumed to have LSD(θ) and justified on the data sets by employing a chi-square goodness of fit testing while the distribution of the number of test cases is Poisson (λ). Then, the distribution of the total number of observed failures, or similarly covered branches, X is a compound Poisson LSD, i.e., negative binomial distribution, given that a certain mathematical identity holds. For each checkpoint in time, either the software satisfies a desired reliability attached to an economic criterion, or else the software testing is allowed to continue. By using a one-step-look-ahead formula derived for the model, the proposed stopping rule is applied to five test case-based data sets acquired by testing embedded chips through the complex VHDL models. Further, multistrategy testing is conducted to show its superiority to single-stage testing. Results are satisfactorily interpreted from a practitioner's viewpoint as an innovative alternative to the ubiquitous test-it-to-death approach, which is known to waste billions of test cases in a tedious process of finding more bugs. Moreover, the proposed dynamic stopping-rule algorithm can validly be employed as an alternative paradigm to the existing on-line statistical process control methods static in nature for the manufacturing industry, provided that underlying statistical assumptions hold. A detailed comparative literature survey of stopping-rule methods is also included in terms of pros and cons, and cost effectiveness.

Máquina de Busca	IEEE
Campos extraídos a partir do entendimento	
Tipo de critério	Não apriorístico. Encontra custo mínimo para testes.
Caracterização do software	Software para circuitos digitais implementados em VHDL

Níveis de testes	Teste de unidade
Modelo Utilizado	Nenhum
Habilidades Necessárias	Nenhuma
Contexto de aplicação	*Não informado*
Princípio do critério	Modelo estatístico
Objetivo do critério	Atingir confiabilidade restrito ao custo
Considera depuração perfeita	*Não informado*
Confiabilidade pré-determinada	*Não informado*
Orçamento para testes é pré-determinado	Não
Tempo para testes pré-determinado	Não
Número de faltas pré-determinado	Não
Taxa de identificação de falhas pré-determinada	Não
Considera qualidade dos casos de testes	Sim
Considera atividade de testes isolada	Sim
Diferencia taxa de detecção de falhas	Não
Variações do critério	Não há
Modelo de confiabilidade base	*Não aplicável*
Modelo de Custo	Dalal e Mallows (1988)
Distribuição de Falhas	Poisson geometric distribution
Unidade de Medida de Falhas	Número acumulado de falhas
Tipo de modelo de confiabilidade	*Não aplicável*

Suposições

- Critério criado especificamente para softwares para circuitos digitais implementados em VHDL

Parâmetros do critério

- a – Contante da LSD (*Logarithmic-series distribution*)
- b – custo de corrigir cada falha encontrada durante os testes
- c – custo por teste

Critério

O artigo descreve um modelo estatístico com a finalidade de sugerir um critério de parada para testes randômicos em VHDL (*VHSIC Hardware Description Language*).
O método é baseado em estimativa estatística de cobertura de caminhos e sinaliza o critério para parar o processo de verificação ou mudar para uma estratégia de verificação diferente.
O critério se expressa pela seguinte fórmula:

$$(\mathrm{RF})a \leq (\mathrm{RF})b + (\mathrm{RT})c$$

Onde:

- (RF) – Número de faltas ou de cobertura restantes depois da ação de parada
- (RT) – Número de casos de testes restantes depois da ação de parada

Critérios de Avaliação

A	2	**B**	0	**C**	2	**D**	2	**E**	1	**F**	1	**Total**	8

Campos extraídos diretamente	
Título	An optimal release policy for software testing process
Autores	Cao, Ping and Dong, Zhao and Liu, Ke
Ano de publicação	2010
Fonte de publicação	Proceedings of the 29th Chinese Control Conference

Abstract

In this paper, we discuss the dynamic release problem in software testing processes. If we stop testing too early, there may be too many defects in the software, resulting in too many failures during operation and leading to significant losses due to the failure penalty or user dissatisfaction. If we spend too much time in testing, there may be a high testing cost. Therefore, there is a tradeoff between software testing and releasing. The release time should be dynamically determined by the testing process. The more defects have been detected and removed, the less time will be used for further testing. A continuous time Markov process is proposed to model the testing process. By formulating with dynamic programming we obtain the Hamilton-Jacobi-Bellman equation of the optimal cost function, and derive the threshold structure of the optimal policy. Furthermore, the dynamic optimal release policy is compared with the static optimal release policy by numerical examples, showing that dynamic policy may outperforms static policy very much in some situations.

Máquina de Busca	IEEE
Campos extraídos a partir do entendimento	
Tipo de critério	Não apriorístico. Determina momento de parada que possui custo mínimo.
Caracterização do software	*Não informado*
Níveis de testes	*Não informado*
Modelo Utilizado	Nenhum
Habilidades Necessárias	Nenhuma
Contexto de aplicação	*Não informado*
Princípio do critério	Modelo de confiabilidade
Objetivo do critério	Minimizar o custo
Considera depuração perfeita	Sim
Confiabilidade pré-determinada	Não
Orçamento com testes pré-determinado	Não
Tempo para testes pré-determinado	Não
Número de faltas pré-determinado	Não
Taxa de identificação de falhas pré-determinada	Sim
Considera qualidade dos casos de testes	Não
Considera atividade de testes isolada	Sim
Diferencia taxa de detecção de falhas	Não
Variações do critério	Não há
Modelo de confiabilidade base	Cao et al. (2010) (Próprio)
Modelo de Custo base	Cao et al. (2010) (Próprio)
Distribuição de Falhas	Exponencial
Unidade de Medida de Falhas	Taxa de ocorrência de falhas
Tipo de modelo de confiabilidade	Modelo de estimativa de confiabilidade

Suposições

Não informado

Parâmetros do critério

- $C_1, C_2, ..., C_m$: classes de teste (não podem ser alteradas no percurso dos testes)
- T: Prazo máximo de liberação
- N: Número de defeitos restantes no software
- λ_i - é a taxa de detecção de defeitos da classe de teste Ci
- c_1 - custo por unidade de tempo de CPU
- c_2 - custo de detecção e remoção de um defeito
- T_{LC} - Tamanho do ciclo de vida do software (Fixo)

- c_p - custo de penalidade por [descartar o software]

Critério

O artigo apresente um critério de parada para testes que tem como principal característica a consideração de diversas classes de testes (C_1, ..., C_m).
Além disto, o artigo propõe que a taxa de detecção de falhas varia durante a fase de testes e a fase de operação do software.

O Critério é representado pela seguinte equação dinâmica:

$$\alpha(t)J^*(n,t) + \frac{\partial J^*(n,t)}{\partial t} = c_1(t) + n \min_i \lambda_i(t)(c_2(t) - \Delta J^*(n,t)),$$

if $J^*(n,t) < J_0(n,t)$, and

$$\alpha(t)J^*(n,t) + \frac{\partial J^*(n,t)}{\partial t} \leqslant c_1(t) + n \min_i \lambda_i(t)(c_2(t) - \Delta J^*(n,t)),$$

Critérios de Avaliação

A	2	**B**	0	**C**	2	**D**	2	**E**	1	**F**	0	**Total**	7

Campos extraídos diretamente	
Título	An optimal software release problem under cost rate criterion: Artificial neural network approach
Autores	Shinohara, Yasuhide and Nishio, Yasuhiko and Dohi, Tadashi and Osaki, Shunji
Ano de publicação	1998
Fonte de publicação	Journal of Quality in Maintenance Engineering

Abstract

A method to estimate the optimal software release time which minimizes the expected cost rate, applying artificial neural networks, is developed. The best model and its associated parameters among several existing candidates are selected using the software reliability growth models (SRGM). A simple multi-layer perceptron (MLP) and a recurrent neural network are formulated. Through numerical examples, the neural network approach is better than the classical methods based on SRGM.

Máquina de Busca	Scopus
Campos extraídos a partir do entendimento	
Tipo de Critério	Não apriorístico. Determina o custo mínimo.
Caracterização do software	*Não Informado*
Níveis de testes	Não informado
Modelo Utilizado	Nenhum
Habilidades Necessárias	Redes neurais
Contexto de aplicação	*Não informado*
Princípio do critério	Redes Neurais Artificiais
Objetivo do critério	Minimizar Custo de desenvolvimento
Considera depuração perfeita	*Não Informado*
Confiabilidade pré-determinada	*Não Informado*
Orçamento para testes é pré-determinado	Não
Tempo para testes pré-determinado	Não

Número de faltas pré-determinado	Não
Taxa de identificação de falhas pré-determinada	Não
Considera qualidade dos casos de testes	Não
Considera atividade de testes isolada	Sim
Diferencia taxa de detecção de falhas	Não
Variações do critério	Não há
Modelo de confiabilidade base	*Não aplicável*
Modelo de Custo	Shinohara (1998) (Próprio)
Distribuição de Falhas	Não aplicável
Unidade de Medida de Falhas	Número acumulado de falhas
Tipo de modelo de confiabilidade	*Não aplicável*

Suposições

Não Informado

Parâmetros do critério

- c_1 – Custo de remover uma falha na fase de teste
- c_2 – Custo de remover uma falha na fase de operação
- c_3 – Custo de teste por unidade de tempo
- T_{LC} – Tempo referente ao ciclo de vida do software [Teste de software inicia-se no tempo 0 até o tempo $T(0 <= T <= T_{LC})$]

Critério

O artigo apresenta um critério de parada que utilizada redes neurais e é baseado na minimização da taxa de custo esperado. Dois tipos diferentes de redes neurais são utilizadas como dispositivos de previsão: multi-layer perceptron (MLP) e recurrent neural network.
O algoritmo proposto pelo critério é dividido em quatro passos:

Step 1: Given n fault-detection time interval data $s_1, s_2, \cdots, s_n$, train the neural networks and estimate the future fault-detection time interval $\tilde{s}_j (j = n+1, \cdots, m_n)$.
Step 2: Seek the estimate $\hat{N}(T)$, plotting the points $\{A_1, A_2, \cdots, A_{m_n}\} = \{(1, x_1), \cdots, (n, x_n), (n+1, \tilde{x}_{n+1}), \cdots, (m_n, \tilde{x}_{m_n})\}$ and connecting them by line segments in the two-dimensional plane.
Step 3: Calculate $\hat{N}(T_{LC})$, and search the point $\tilde{x}_{j^*}$ maximizing the slope θ of straight line through the point $(-c_2\hat{N}(T_{LC})/(c_2 - c_1), 0)$ and $A_i (i = 1, \cdots, m_n)$.
Step 4: Calculate the minimum expected cost rate.

As configurações das redes neurais MLP e recurrent são apresentadas respectivamente a seguir.

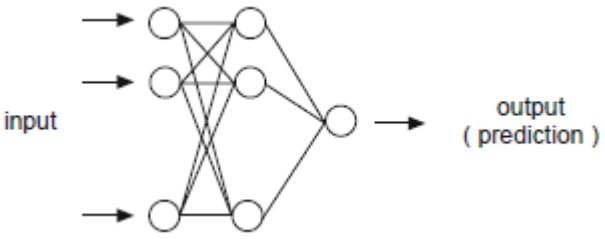


Figura 1 – Configuração da MLP

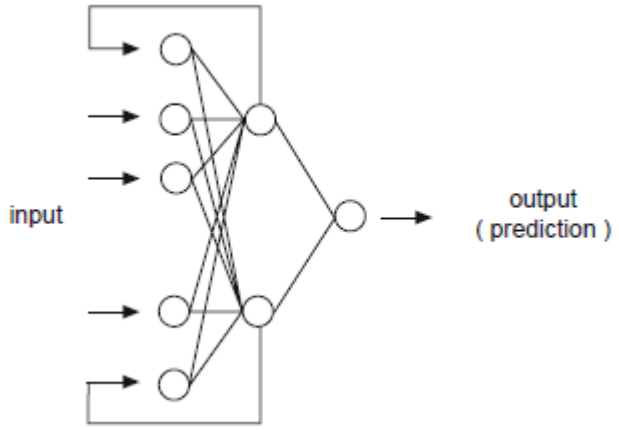


Figura 2 - Configuração da rede neural recorrente

Critérios de Avaliação												
A	0	**B**	1	**C**	0	**D**	2	**E**	1	**F**	0	**Total**	4

Campos extraídos diretamente	
Título	An Optimal time for software testing under the user's requirement of failure-free demonstration before release
Autores	Cho, Byung Chul and Park, Kyung Soo
Ano de publicação	1994
Fonte de publicação	IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences

Abstract
A new approach to the problem of optimal software testing time is described. Most models implicitly assume the testing is terminated at the end of a prescribed period of time without user's approval. It means the release time and the in-service reliability are determined unilaterally by the developer. If software developer uses and maintains it, the assumption is appropriate. But, it may be inappropriate, if a software requiring more stringent reliability is developed by second party on a contract basis. In this case, the time of release is usually determined with the user's approval. To overcome the weaknesses of the assumption, a two stage testing with failure-free release policy is proposed. A software, after being tested by the developer for some time (in-house testing), is transferred to acceptance testing performed jointly with the user. During the acceptance testing, it is released when τ units of time specified by user is observed to be failure-free for the first time. The policy may be attractive to a user because he can determine the time of release, and extend the testing time by increasing τ. A software cost model for the policy is developed. For the software developer, an optimal in-house testing time minimizing software cost, and various quantities of interests, such as expected periods of acceptance testing, are derived based on the Jelinski-Moranda software reliability model. Finally, numerical examples are shown to illustrate the results.

Máquina de Busca	Scopus
Campos extraídos a partir do entendimento	
Tipo de critério	Apriorístico. A quantidade de tempo que o software deve executar livre de falhas é especificada pelo cliente
Caracterização do software	*Não Informado*
Níveis de testes	*In-house* testes e testes de aceitação (beta)
Modelo Utilizado	Nenhum
Habilidades Necessárias	Nenhuma

Contexto de aplicação	*Não informado*
Princípio do critério	Modelo de confiabilidade
Objetivo do critério	Minimizar custo total de desenvolvimento
Considera depuração perfeita	*Não informado*
Confiabilidade pré-determinada	Sim
Orçamento para testes é pré-determinado	Não
Tempo para testes pré-determinado	Sim
Número de faltas pré-determinado	Sim
Taxa de identificação de falhas pré-determinada	Sim
Considera qualidade dos casos de testes	Não
Considera atividade de testes isolada	Sim
Diferencia taxa de detecção de falhas	Não
Variações do critério	Não há
Modelo de confiabilidade base	Jelinski and Moranda (1972)
Modelo de Custo	Cho and Park (1994) (Próprio)
Distribuição de Falhas	Exponencial
Unidade de Medida de Falhas	Tempo médio entre falhas
Tipo de modelo de confiabilidade	Modelo de estimativa de confiabilidade

Suposições

[1] when in-house testing starts, the software contains N errors and during in-house testing and acceptance testing, each time a failure occurs, the error which caused it is immediately fixed with certainty, and no new errors are introduced,

[2] each error contributes the same amount, ϕ, to the overall failure rate of the software, and successive inter-failure times between $(i-1)$-st and i-th failure, X_i, $i=1, \cdots, N$, are statistically independent random variables and exponentially distributed with constant failure rate, $(N-i+1)\phi$, i.e.,

$$\Pr\{X_i < t\} = 1 - \exp\{-(N-i+1)\phi t\}$$

for $t \geq 0, i=1, \cdots, N.$ (1)

Parâmetros do critério

- N – Número inicial de erros estimados
- Φ – Taxa de detecção de erros estimada
- C_1 – Custo por unidade de tempo de teste *in-house*
- C_2 – Custo por unidade de tempo de teste durante a fase de testes de aceitação
- C_3 – Custo para correção de um erro durante a fase de teste *in-house*
- C_4 – Custo para a correção de um erro depois da fase *in-house*
- τ – tempo que o software deve executar livre de falhas especificado pelo usuário
- t - quantidade de tempo dos testes internos

Critério

O artigo propõe um critério de parada para testes de software que incluem o cliente/usuário na decisão de parada.
No artigo, o processo de teste é dividido em duas partes: testes internos e testes de aceitação (testes com o cliente).
Primeiramente, testes internos são realizados e, depois de testado, o software não é encaminhado

para a produção. O software é encaminhado para testes de aceitação que acontecem com o usuário. Durante os testes de aceitação são coletadas o tempo entre falhas. Os testes param quando o tempo entre falha atinge um valor pré-determinado pelo usuário.

O Critério então se baseia em decidir quanto de teste interno é preciso para encontrar menos falhas possíveis nos testes de aceitação.
O seguinte procedimento é utilizado:

Hence, the t^* is summarized as follows. Suppose $C_4 \geqq C_3 \geqq 0$, $C_2 \geqq C_1 \geqq 0$. If Eq. (23a) < 0, then (19) has a positive and finite solution t^* minimizing $C_\tau(t)$ (cases: $N=30$ and $N=15$ in Fig. 6). In this case, t^* is obtained numerically by bisection method. A simple computer program for searching t^* has been written and used to illustrate the results of numerical examples. On the contrary, if Eq. (23a) $\geqq 0$, then (19) has no a positive solution (case: $N=6$ in Fig. 6). In this case, the minimum of $C_\tau(t)$ is at $t=0$, i.e., $t^*=0$ which implies that the software should be immediately transferred to acceptance testing without in-house testing.

Onde:

$$C'_\tau(0) = C_1 + \{C_3 - C_4\} N\phi + C_2 N\phi \{a_1(\tau) - a_0(\tau)\}, \tag{23a}$$

$$C'_\tau(t) = C_1 + C_2 \sum_{i=0}^{N} a_i(\tau) P'_i(t) + \{C_3 - C_4\} N\phi \{1 - G(t)\} = 0, \tag{19}$$

N initial number of errors in software system ($N \geqq 0$, estimated),

ϕ error detection rate per error ($\phi \geqq 0$, estimated),

τ failure-free requirement time specified by the user ($\tau \geqq 0$, given),

t amount of in-house testing time ($t \geqq 0$, decision variable),

t^* optimal t, (throughout this paper, by "time" we mean CPU time),

$M(t)$ cumulative number of errors detected up to time t during in-house testing (random variable),

$m(t)$ expectation of $M(t)$.

And the other random variables for given t and τ ($t, \tau \geqq 0$) under the proposed release policy are:

$N_\tau(t)$ number of errors detected during acceptance testing,

$R_\tau(t)$ $N - M(t) - N_\tau(t)$: number of remaining errors after release,

$A_\tau(t)$ periods of acceptance testing.

The expectations of $N_\tau(t)$, $R_\tau(t)$ and $A_\tau(t)$ are $n_\tau(t)$, $r_\tau(t)$ and $a_\tau(t)$ respectively. Other notations will be explained as needed.

C_1 cost of unit testing time in in-house testing ($C_1 \geqq 0$, known),

C_2 cost of unit testing time in acceptance testing ($C_2 \geqq C_1$, known),

C_3 cost of fixing an error during in-house testing ($C_3 \geqq 0$, known),

C_4 cost of fixing an error after in-house testing ($C_4 \geqq C_3$, known),

$C_\tau(t)$ total expected software cost function of t for given τ ($t, \tau \geqq 0$).

Critérios de Avaliação												
A	0	**B**	1	**C**	0	**D**	2	**E**	1	**F**	0	**Total**	4

Campos extraídos diretamente	
Título	Analysis of incorporating logistic testing-effort function into software reliability modeling
Autores	Huang, CY and Kuo, SY
Ano de publicação	2002
Fonte de publicação	IEEE TRANSACTIONS ON RELIABILITY

Abstract

This paper investigates a SRGM (software reliability growth model) based on the NHPP (nonhomogeneous Poisson process) which incorporates a logistic testing-effort function. SRGM proposed in the literature consider the amount of testing-effort spent on software testing which can be depicted as an exponential curve, a Rayleigh curve, or a Weibull curve. However, it might not be

appropriate to represent the consumption curve for testing-effort by one of those curves in some software development environments. Therefore, this paper shows that a logistic testing-effort function can be expressed as a software-development/test-effort curve and that it gives a good predictive capability based on real failure-data. Parameters are estimated, and experiments performed on actual test/debug data sets. Results from applications to a real data set are analyzed and compared with other existing models to show that the proposed model predicts better. In addition, an optimal software release policy for this model, based on cost-reliability criteria, is proposed.	
Máquina de Busca	IEEE
Campos extraídos a partir do entendimento	
Tipo de Critério	Apriorístico. Determina menor custo restrito à confiabilidade desejada.
Caracterização do software	*Não informado.* O critério foi testando em com os dados do Rome Air Development Center (RADC). O software contém cerca de 21700 objetos de instrução. Levou 21 semanas e 9 programadores para completar os testes
Níveis de testes	*Não informado*
Modelo Utilizado	Nenhum
Habilidades Necessárias	Nenhuma
Contexto de aplicação	*Não informado*
Princípio do critério	Modelo de confiabilidade
Objetivo do critério	Minimizar o custo restringindo a uma confiabilidade
Considera depuração perfeita	Não
Confiabilidade pré-determinada	Sim
Orçamento para testes é pré-determinado	Não
Tempo para testes pré-determinado	Não
Número de faltas pré-determinado	Sim
Taxa de identificação de falhas pré-determinada	Sim
Considera qualidade dos casos de testes	Não
Considera atividade de testes isolada	Sim
Diferencia taxa de detecção de falhas	Não
Variações do critério	Não há
Modelo de confiabilidade base	Huang and Kuo (2002) (Próprio)
Modelo de Custo	Yamada (1986), Yamada (1993) e Huang and Kuo (1997)
Distribuição de Falhas	NHPP
Unidade de Medida de Falhas	Número acumulado de falhas
Tipo de modelo de confiabilidade	Modelo de estimativa de confiabilidade
Suposições	
Não informado	
Parâmetros do critério	
• C_1 – Custo de correção de um erro durante os testes • C_2 – Custo de correção de um erro durante a fase de operação ($C_2 > C_1$) • C_3 – Custo de teste por unidade de teste-esforço	

Critério
O artigo mostra que a curva que representa o esforço de teste no desenvolvimento de software pode ser expressa por uma *Logistic Testing-Effort function*. Assim, o artigo propõe um modelo de confiabilidade que utiliza *Logistic Testing-Effort Function* e o critério de parada é estimado a partir deste modelo. Com isto, o artigo apresenta o critério de parada a seguir: Given: $C_1 > 0, C_2 > C_1, C_3 > 0, \Delta t > 0, 0 < R_0 < 1$: 1) If $\lambda(0)/w(0) > C_3/(C_2 - C_1)$ and $\lambda(T)/w(T) = a \cdot r \cdot \exp(-r \cdot (N \cdot A/(1+A))) \leq C_3/(C_2 - C_1)$, then $T^* = \max[0, T_0, T_1]$ for $R(\Delta t\, 0) < R_0 < 1$, or $T^* = T_0$ for $0 < R_0 < R(\Delta t\, t = 0)$. 2) If $\lambda(0)/w(0) \geq C_3/(C_2 - C_1)$ then $T^* = T_1$ for $R(\Delta t|0) < R_0 < 1$, or $T^* \geq 0$ for $0 < R_0 \leq R(\Delta t|0)$. 3) If $\lambda(0)/w(0) \leq C_3/(C_2 - C_1)$ then $T^* - T_1$ for $R(\Delta t|0) < R_0 < 1$, or $T^* = 0$ for $0 < R_0 \leq R(\Delta t|0)$. Onde: $m(t)$ mean number of faults detected in time $(0, t]$, an MVF $\lambda(t)$ $dm(t)/dt$: failure intensity for $m(t)$ $w(t)$ current testing-effort consumption at time t $W(t)$ cumulative $w(t)$ a mean number of initial faults r fault detection rate per unit testing-effort N total testing-effort eventually consumed α consumption rate of testing-effort expenditures in the logistic TEF A constant parameter in the logistic TEF β scale parameter in the Weibull-type TEF m shape parameter in the Weibull-type TEF $R(x\|t)$ conditional software reliability L likelihood function W_k cumulative testing-effort actually consumed in $(0, t_k]$ m_k cumulative number of faults observed in $(0, t_k]$ T_{LC} software life-cycle length C_1 cost of correcting an error during testing C_2 cost of correcting an error during operation, $C_2 > C_1$ C_3 cost of testing per unit testing-effort expenditures

Critérios de Avaliação												
A	0	**B**	1	**C**	0	**D**	2	**E**	1	**F**	0	**Total**	4

Campos extraídos diretamente	
Título	Analysis of software reliability modeling considering testing compression factor and failure-to-fault relationship
Autores	Huang, Chin-Yu and Lin, Chu-Ti
Ano de publicação	2010
Fonte de publicação	IEEE Transactions on Computers
Abstract	
This paper is an attempt to relax and improve the assumptions regarding software reliability	

modeling. To approximate reality much more closely, we take into account the concepts of testing compression factor and the quantified ratio of faults to failures in the modeling. Numerical examples based on real failure data show that the proposed framework has a fairly good prediction capability. Further, we also address the optimal software release time problem and conduct a detailed sensitivity analysis through the proposed model. © 2010 IEEE.	
Máquina de Busca	IEEE
Campos extraídos a partir do entendimento	
Tipo de Critério	Apriorístico. Minimiza o custo restrito à confiabilidade.
Caracterização do software	*Não informado*
Níveis de testes	Pode ser adaptado para um processo que possui diversos níveis de testes
Modelo Utilizado	Nenhum
Habilidades Necessárias	Nenhuma
Contexto de aplicação	*Não informado*
Princípio do critério	Modelo de confiabilidade
Objetivo do critério	• Atingir confiabilidade desejada • Minimizar o custo com testes • Minimizar o custo restrito a confiabilidade
Considera depuração perfeita	*Não informado*
Confiabilidade pré-determinada	Sim
Orçamento com testes pré-determinado	Não
Tempo para testes pré-determinado	Não
Número de faltas pré-determinado	Não
Taxa de identificação de falhas pré-determinada	Sim
Considera qualidade dos casos de testes	Não
Considera atividade de testes isolada	Sim
Diferencia taxa de detecção de falhas	Sim
Variações do critério	3. • Atingir confiabilidade desejada • Minimizar o custo com testes • Minimizar o custo com testes restrito a confiabilidade
Modelo de confiabilidade base	Huang and Lin (2010) (Próprio)
Modelo de Custo base	Huang and Lin (2010) (Próprio)
Distribuição de Falhas	NHPP
Unidade de Medida de Falhas	Número acumulado de falhas
Tipo de modelo de confiabilidade	Modelo de estimativa de confiabilidade
Suposições	

1. The software system is subject to failures at random times caused by the manifestation of remaining faults in the system.
2. The total number of faults at the beginning of testing is finite. Further, the failures caused by them are also finite.
3. The mean number of expected failures in the time interval $(t, t+\Delta t]$ is proportional to the mean number of remaining faults in the system. It is equally likely that a fault will generate more than one failure and a failure may be caused by a series of dependent faults.
4. The fault detection rate is not always a constant and could change at some time moments. Further, the efficiency of fault detection is affected by the input states.

Parâmetros do critério

- R_0 - confiabilidade aceitável
- T_{LC} - Tamanho do ciclo de vida do software
- C_1 - custo de remoção de uma falta durante os testes
- C_2 - custo de remoção de uma falta durante a fase de operação

C_3 - custo por unidade de tempo dos testes de software

Critério

O artigo propõe um modelo de confiabilidade que tem como características principais a consideração de um fator de compressão de testes e a taxa de relacionamento entre faltas e falhas.
Fator de compressão de testes quer dizer que durante a fase de testes erros são encontrados mais rapidamente porque a variação dos dados de entrada de testes é controlada. Em operação, os dados de entrada são verdadeiramente aleatórios.
O artigo propõe também que diversas falhas podem ser geradas por uma falta e também que diversas faltas podem causar uma única falha. Ou seja, o relacionamento entre falha e falta não é um-para-um.
A partir do modelo de confiabilidade proposto, é apresentado dois critérios de parada. Um critério baseado apenas no custo e outro critério baseado no custo e na confiabilidade.

- O modelo baseado no custo é representado através da seguinte equação:

$$T_m = \begin{cases} \ln\left[\frac{ar}{C_3}(C_2 - C_1\alpha)\right] \Big/ (r\alpha), & 0 \le T_m < \tau, \\ (1-c)\tau + c \times \ln\left[\frac{ar}{C_3 \times c}(C_2 - C_1\alpha)\right] \Big/ (r\alpha), & T_m \ge \tau, \end{cases} \quad (17)$$

which satisfies

$$m'(T_m) = C_3/(\alpha \times (C_2 - C_1)). \quad (18)$$

The optimal release time T_c can be determined by the following three cases. First, if $m'(0) \le C_3/(\alpha \times (C_2 - C_1))$, then we can find that $C(T)$ is monotonously increasing curve for $0 < T < T_{LC}$. In this case, $T_c = 0$. However, when $m'(T_{LC}) \ge C_3/(\alpha \times (C_2 - C_1))$, we have $m'(T) \ge C_3/(\alpha \times (C_2 - C_1))$ for $0 < T < T_{LC}$. Thus, $T_c = T_{LC}$. Finally, if $m'(0) > C_3/(\alpha \times (C_2 - C_1)) > m'(T_{LC})$, there is a unique solution for $C'(T) = 0$, for $0 < T < T_{LC}$, i.e., $T_c = T_m$.

- O modelo baseado no custo e em uma confiabilidade mínima é representado a seguir

$$T^* = max\{T_R, T_c\}.$$

Onde:

- m(t) - função de valor médio do número de falhas no tempo t
- r(t) - função da taxa de detecção de faltas
- R(t) - função de confiabilidade
- T - Momento de entrega do software
- T_R - momento de entrega que satisfaz a confiabilidade
- T_C - momento de entrega que possui menor custo
- T_{LC} - Tamanho do ciclo de vida do software
- C(T) custo total dos testes e de depuração durante o ciclo de desenvolvimento do software
- T_m - tempo de teste que possui custo mínimo

Critérios de Avaliação												
A	2	**B**	0	**C**	0	**D**	2	**E**	1	**F**	1	**Total**	6

Campos extraídos diretamente	
Título	Bayesian updating of optimal release time for software systems
Autores	Chiu, Kuei-Chen and Ho, Jyh-Wen and Huang, Yeu-Shiang
Ano de publicação	2009
Fonte de publicação	SOFTWARE QUALITY JOURNAL

Abstract

In this paper, a Bayesian method dealing with software reliability growth with consideration of the learning effect is proposed to determine an optimal release time for software systems with regard to the testing cost and experts' prior judgments. Such an approach is able to devise an appropriate software-debugging scheme which has the best arrangement of available resources and personnel with a minimal software testing cost when lacking sufficient information for decision making. Past research on software reliability emphasized the estimation of the number of cumulative software errors or the software reliability with respect to a specific time period, yet it neglected the determination of software release time with consideration of the software testing cost, meaning that existing approaches are not entirely practical. Accordingly, the proposed method is concerned with the evaluation of the software testing cost incurred during the testing period based on experts' prior judgments and the software testing data collected within a given duration, and is thus characterized by its practicality as well as meaningfulness with consideration of the learning effect. Finally, a numerical example is given to verify the effectiveness of the proposed approach, and sensitivity and risk analyses are performed on this example.

Máquina de Busca	Scopus
Campos extraídos a partir do entendimento	
Tipo de Critério	Apriorístico. Determina o momento em que o custo é mínimo e com uma confiabilidade pré-determinada.
Caracterização do software	*Não informado*
Níveis de testes	*Não informado*
Modelo Utilizado	Nenhum
Habilidades Necessárias	Nenhuma
Contexto de aplicação	*Não informado*
Princípio do critério	Modelo de confiabilidade (Método *bayesian* criado para lidar com o crescimento da confiabilidade do software)
Objetivo do critério	Minimizar custo restrito a confiabilidade
Considera depuração perfeita	*Não informado*
Confiabilidade pré-determinada	Sim

Orçamento com testes pré-determinado	*Não informado*
Tempo para testes pré-determinado	*Não*
Número de faltas pré-determinado	*Não informado*
Taxa de identificação de falhas pré-determinada	*Não informado*
Considera qualidade dos casos de testes	Não
Considera atividade de testes isolada	Sim
Diferencia taxa de detecção de falhas	Não
Variações do critério	Não há
Modelo de confiabilidade base	Chiu et al. (2008)
Modelo de Custo base	Pham and Zhang (1999)
Distribuição de Falhas	NHPP
Unidade de Medida de Falhas	Número acumulado de falhas
Tipo de modelo de confiabilidade	Modelo de estimativa de confiabilidade

Suposições

Não informado

Parâmetros do critério

- C_0 – custo de configuração dos testes
- C_1 – custo da rotina por unidade de tempo durante os períodos de testes e garantia/operação
- C_2 – custo de remoção de um erro na fase de testes
- C_3 – custo de remoção de um erro no período de garantia/operação
- C_4 – custo da perda quando o software falha

Critério

O artigo propõe um processo de decisão Bayesiana baseado no modelo de Chiu et al. (2008) onde as incertezas sobre os parâmetros do modelo podem ser resolvidas com a opinião de especialistas.
A vantagem do modelo proposto é que ele é capaz de lidar com a falta de dados históricos.
A seguinte equação descreve o custo total dos testes:

$$C(T) = \mathrm{SC} + \mathrm{RC} + \mathrm{EC} + \mathrm{WC} + \mathrm{FC} + \mathrm{OC}$$
$$= C_0 + C_1 \int_0^{T+T_w} \mathrm{e}^{-rt}\mathrm{d}t + C_2 m(T) u_y + C_3(m(T+T_w) - m(T))u_w + C_4(1 - R(x/T))$$
$$+ C_5(v_1 + T)^{v_2}. \qquad (3.7)$$

Onde:

- SC – custo de configuração dos testes
- RC – custo da rotina de teste
- EC – custo de remoção de todos os erros durante o periodo de testes
- WC – custo de remoção de todos os erros durante o período de garantia/operação
- FC – custo do risco do software falhar depois de ser entregue
- OC -

A partir desta equação, os seguintes critérios são propostos:

Case 1: If $E[C(T)]$ (or $E'[C(T)]$) is a strictly increasing function of T through the spectrum analysis, then the solution is the time at which R_0 is met, and which is represented as T_{R_0}, namely, the optimal release time is $T^* = T_{R_0}$, as shown in Fig. 4a. Note that some research may argue that the optimal release time should be at $T^* = 0$, but in practice this violates the responsibility of quality for software providers. Consequently, this case would probably occur when software risk costs are trivial.

Case 2: If $E[C(T)]$ (or $E'[C(T)]$) is a strictly decreasing function of T through the spectrum analysis, then the solution is a specific time after T_{R_0}, as shown in Fig. 4b. Although the software is theoretically more reliable with a longer testing phase, especially when $T^* \to \infty$, such a testing phase is not practical situations, as it is impossible to test the

Case 3: If $E[C(T)]$ (or $E'[C(T)]$) is a convex function of T through the spectrum analysis, then the solution is at either T_{R_0} or T_{S_0}, depending on which one can lead to the minimum $E[C(T)]$ (or $E'[C(T)]$), as shown in Fig. 4c.

Critérios de Avaliação

A	0	**B**	1	**C**	0	**D**	2	**E**	1	**F**	0	**Total**	4

Campos extraídos diretamente	
Título	Cost model for determining the optimal number of software test cases
Autores	Brown, David B. and Maghsoodloo, Saeed and Deason, William H.
Ano de publicação	1989
Fonte de publicação	IEEE Transactions on Software Engineering

Abstract

A probabilistic model is presented that demonstrates the optimal number of software test cases required in situations where the following can be estimated as independent parameters: (1) the cost per test; (2) the cost per error if undetected until field implementation; (3) the number of software executions over its lifetime; (4) the number of possible different executions; and (5) the number of faults embedded in the software. A formula is derived by the use of calculus and is solved by approximation techniques. Tables of the optimal number of tests over a range of parameter values are presented to illustrate the results. The model serves as a basis for further research efforts to improve the accuracy of input variable estimation.

Máquina de Busca	IEEE
Campos extraídos a partir do entendimento	
Tipo de Critério	Não apriorístico. Determina o número de casos de testes que devem ser executados.
Caracterização do software	*Não Informado*
Níveis de testes	*Não informado*
Modelo Utilizado	Nenhum
Habilidades Necessárias	Nenhuma
Contexto de utilização	*Não informado*
Princípio do critério	Modelo Probabilístico
Objetivo do critério	Descobrir quantidade de casos de testes ideal (de certa forma minimizar o custo)
Considera depuração perfeita	Sim
Confiabilidade pré-determinada	Não
Orçamento para testes é pré-determinado	Não
Tempo para testes pré-determinado	Não
Número de faltas pré-determinado	Não
Taxa de identificação de falhas pré-determinada	Sim

Considera qualidade dos casos de testes	Não
Considera atividade de testes isolada	Sim
Diferencia taxa de detecção de falhas	Não
Variações do critério	Não há
Modelo de confiabilidade base	Brown (1989) (Próprio)
Modelo de Custo	Brown (1989) (Próprio)
Distribuição de Falhas	Não aplicável
Unidade de Medida de Falhas	Taxa de ocorrência falhas
Tipo de modelo de confiabilidade	Modelo de estimativa de confiabilidade

Suposições

- Existe um número finito de formas em que um programa pode ser executado em um espaço de tempo

Parâmetros do critério

- Custo por teste
- Custo por erro se detectado até a implementação
- Número de execuções do software durante seu tempo de vida
- Número de possíveis diferentes execuções
- Número de faltas embutidas no software

Critério

O Autor propõe um critério baseado no custo dos testes e o custo de correção de um erro em campo. O seguinte critério é apresentado:

$$t_0 = \frac{\ln K}{\ln (1-p)}, \quad \text{where} \quad K = \frac{c_1}{c_2 p \ln \left[(1-p)^{-1}\right]},$$

$$\text{and } \ln = \log_e.$$

Onde:

- T_0 – número ótimo de testes
- c_1 – custo dos testes
- c_2 – custo de um erro em campo

p – taxa de erro esperada após os testes

Critérios de Avaliação

A	0	**B**	1	**C**	0	**D**	2	**E**	1	**F**	0	**Total**	4

Campos extraídos diretamente	
Título	COST-RELIABILITY OPTIMAL RELEASE POLICIES FOR SOFTWARE SYSTEMS
Autores	Yamada, Shigeru and Osaki, Shunji
Ano de publicação	1985
Fonte de publicação	IEEE Transactions on Reliability

Abstract

The authors extend the Okumoto-Goel (1979) optimal software problem to both cost and reliability requirements by evaluating both criteria simultaneously. The optimum software release time is determined both by minimizing a total average software cost and by satisfying a software reliability requirement. A numerical example illustrates the results.

Máquina de Busca	IEEE

Campos extraídos a partir do entendimento

Tipo de Critério	Apriorístico.

	Determina o menor curso para uma confiabilidade pré-estabelecida.
Caracterização do software	*Não informado*
Níveis de testes	
Modelo Utilizado	
Habilidades Necessárias	
Princípio do critério	Modelo de Confiabilidade
Objetivo do critério	Minimizar o custo total do software restringido a uma confiabilidade
Considera depuração perfeita	*Não Informado*
Confiabilidade pré-determinada	Sim
Orçamento para testes é pré-determinado	Não
Tempo para testes pré-determinado	
Número de faltas pré-determinado	Sim
Taxa de identificação de falhas pré-determinada	Sim
Considera qualidade dos casos de testes	Não
Considera atividade de testes isolada	Sim
Diferencia taxa de detecção de falhas	Não
Variações do critério	Não há
Modelo de confiabilidade base	Goel and Okumoto NHPP model (1979)
Modelo de Custo	Goel and Okumoto (1979)
Distribuição de Falhas	NHPP
Unidade de Medida de Falhas	Número acumulado de falhas
Tipo de modelo de confiabilidade	Modelo de estimativa de confiabilidade

Suposições

- Quantidade de erros iniciais deve ser conhecida ou estimada
- A taxa de detecção de erros deve ser conhecida ou estimada

Parâmetros do critério

- c_1 – custo por corrigir um erro durante os testes
- c_2 – custo por corrigir um erro com o software em produção
- c_3 – custo por unidade de tempo por atrasar na entrega do software
- a – número de erros iniciais
- b – taxa de detecção de erros

Critério

Os autores estendem o critério proposto por Okumoto-Goel (optimum release time for software systems) para criar um critério que lida com dois fatores simultaneamente: custo e confiabilidade. O critério é formulado da seguinte maneira:

$$\left.\begin{array}{l}\text{Minimize } C(T)\\ \text{subject to } R(x \mid T) \geq R_0,\ T \geq 0\\ \text{for } c_2 > c_1 > 0,\ c_3 > 0,\\ \quad x \geq 0,\ 0 < R_0 < 1\end{array}\right\}$$

$$T^* = \max\{T_0, T_1\}$$

$\mathbf{T_0}$ é calculado por:

$$bT_0 = \ln[ab(c_2 - c_1)/c_3]; \text{ Caso } ab(c_2 - c_1) > c_3$$

Ou

$T_0 = 0.$

$\mathbf{T_1}$ é calculado por:

$bT_1 = \ln[m(x)] - \ln[\ln(1/R_0)]$; caso $R(x|0) < R_0$

Ou

$T_1 = 0.$

1. If $ab > c_3/(c_2 - c_1)$ and $R(x|0) < R_0$, then there exist a positive and unique T_0 and T_1 satisfying (5) and (6), respectively, and the optimum software release time is $T^* = \max\{T_0, T_1\}$.
2. If $ab > c_3/(c_2 - c_1)$ and $R(x|0) \geq R_0$, then $T^* = T_0$.
3. If $ab \leq c_3/(c_2 - c_1)$ and $R(x|0) < R_0$, then $T^* = T_1$.
4. If $ab \leq c_3/(c_2 - c_1)$ and $R(x|0) \geq R_0$, then $T^* = 0$.

Onde:

$N(t)$	cumulative number of software errors detected up to time t
a	initial error content, known
b	error detection rate per error, known
$m(t)$	$E[N(t)]$, the mean value function
$R(x\|t)$	software reliability, ie. the probability of no failures in $(t, t + x]$ given that the most recent failure occurred at time $t(x \geq 0)$.
R_0	prespecified software reliability $(0 < R_0 < 1)$
c_1	cost of fixing an error during testing $(c_1 > 0)$, known
c_2	cost of fixing an error during operation $(c_2 > c_1)$, known
c_3	cost of testing per unit time $(c_3 > 0)$, known
$C(T)$	total average software cost
T_{LC}	software life-cycle length
T	software release time, ie, total testing time
T^*	optimum software release time
T_0	unique solution T satisfying $dC(T)/dT = 0$
T_1	unique solution T satisfying $R(x\|T) = R_0$

Critérios de Avaliação												
A	0	**B**	1	**C**	0	**D**	2	**E**	1	**F**	0	**Total**	4

6Campos extraídos diretamente	
Título	Cost-reliability-optimal release policy for software reliability models incorporating improvements in testing efficiency
Autores	Huang, CY
Ano de publicação	2005

Fonte de publicação	JOURNAL OF SYSTEMS AND SOFTWARE
Abstract	
Over the past 30 years, many software reliability growth models (SRGMs) have been proposed for estimation of reliability growth of products during software development processes. One of the most important applications of SRGMs is to determine the software release time. Most software developers and managers always want to know the date on which the desired reliability Goal will be met. In this paper, we first review a SRGM with generalized logistic testing-effort function and the proposed generalized logistic testing-effort function can be used to describe the actual consumption of resources during the software development process. Secondly. if software developers want to detect more faults in practice, it is advisable to introduce new test techniques, tools, or consultants, etc. Consequently, here we propose a software cost model that can be used to formulate realistic total software cost projects and discuss the optimal release policy based on cost and reliability considering testing effort and efficiency. Some theorems and several numerical illustrations are also presented. Based on the proposed models and methods, we can specifically address the problem of how to decide when to stop testing and when to release software for use. (c) 2004 Elsevier Inc. All rights reserved.	
Máquina de Busca	Scopus
Campos extraídos a partir do entendimento	
Tipo de Critério	Apriorístico. Minimiza custo restrito à confiabilidade
Caracterização do software	*Não informado*
Níveis de testes	*Não informado*
Modelo Utilizado	Nenhum
Habilidades Necessárias	Nenhuma
Contexto de utilização	*Não informado*
Princípio do critério	Modelo de confiabilidade
Objetivo do critério	Minimizar o custo total de desenvolvimento restrito a uma confiabilidade pré-determinada
Considera depuração perfeita	Sim
Confiabilidade pré-determinada	Sim
Orçamento para testes é pré-determinado	Não
Tempo para testes pré-determinado	Não
Número de faltas pré-determinado	Não
Taxa de identificação de falhas pré-determinada	Não
Considera qualidade dos casos de testes	Sim
Considera atividade de testes isolada	Sim
Diferencia taxa de detecção de falhas	Não
Modelo de confiabilidade base	Huang (2005) (Próprio)
Modelo de Custo	Yamada (1993) (Modificado)
Distribuição de Falhas	NHPP
Unidade de Medida de Falhas	Taxa de ocorrência de falhas
Tipo de modelo de confiabilidade	Modelo de estimativa de confiabilidade
Variações do critério	3. Sendo que os dois primeiros são citados apenas para chegar ao último • Minimizando custo total de desenvolvimento • Maximizando a confiabilidade • Minimizando custo total de

	desenvolvimento restrito a uma confiabilidade (Principal)

Suposições

(1) The fault removal process follows the NHPP.
(2) The software system is subject to failures at random times caused by the manifestation of remaining faults in the system.
(3) The mean number of faults detected in the time interval $(t, t+\Delta t]$ by the current testing-effort expenditures is proportional to the mean number of remaining faults in the system.
(4) The proportionality is a constant over time.
(5) The consumption curve of testing effort is modeled by a generalized *logistic* TEF.
(6) Each time a failure occurs, the fault that caused it is immediately and perfectly removed and no new faults are introduced. Moreover, correction of faults takes only negligible time and a detected fault is removed with certainty.

Parâmetros do critério

- C_1 – custo de correção de um erro durante a fase de testes
- C_2 – custo de correção de um erro durante a fase de operação
- C_3 – custo por unidade de teste
- T_{LC} – tamanho do ciclo de vida do software
- T_s – momento em que uma nova técnica de teste começou a ser utilizada
- C_{01} – número real não negativo que indica custo básico de adoção de nova ferramenta ou técnica de teste
- C_0 – custo de nova unidade de teste adicionada

Critério

O artigo propõe um novo modelo de custo de software que pode ser utilizado para formular custo total de projeto de software e discutir a política de entrega ótima baseado no custo e na confiabilidade considerando o esforço e eficiência dos testes.
O modelo proposto considera o custo de testes, o custo de debugging durante a fase de teste, e o custo extra devido à introdução de novas técnicas de teste.
Assim, um critério de parada baseado em custo e confiabilidade é proposto. Os teoremas a seguir representam o critério.

Theorem 1. *Assume* $C_0(T) = C_0$ *(constant),* $C_0 > 0$, $C_1 > 0$, $C_2 > 0$, $C_3 > 0$, *and* $C_2 > C_1$, *we have*

(1) *If* $ar(C_2^* - C_1^*)\exp[-rW^*(T_S)] > C_3$ *and* $ar(C_2^* - C_1^*)\exp[-rW^*(T_{LC})] < C_3$, $T^* = \max(T_0, T_1)$ *for* $R(\Delta t|T_S) < R_0 < 1$ *or* $T^* = T_0$ *for* $0 < R_0 \leqslant R(\Delta t|T_S)$.
(2) *If* $ar(C_2^* - C_1^*)\exp[-rW^*(T_S)] < C_3$, $T^* = T_1$ *for* $R(\Delta t|T_S) < R_0 < 1$ *or* $T^* = T_S$ *for* $0 < R_0 \leqslant R(\Delta t|T_S)$.

(3) *If* $ar(C_2^* - C_1^*)\exp[-rW^*(T_{LC})] > C_3$, $T^* \geqslant T_1$ *for* $R(\Delta t|T_S) < R_0 < 1$ *or* $T^* \geqslant T_S$ *for* $0 < R_0 \leqslant R(\Delta t|T_S)$.

Theorem 2. *Assume* $C_0(T) = C_{01} + C_0 \int_{T_S}^{T} w(t)\,dt$, C_{01}, $C_0 > 0$, $C_1 > 0$, $C_2 > 0$, $C_3 > 0$, *and* $C_2 > C_1$, *we have*

(1) *If* $ar(C_2^* - C_1^*)\exp[-rW^*(T_S)] > (C_3 + C_0)$ *and* $ar(C_2^* - C_1^*)\exp[-rW^*(T_{LC})] < (C_3 + C_0)$, $T^* = \max(T_0, T_1)$ *for* $R(\Delta t|T_S) < R_0 < 1$ *or* $T^* = T_0$ *for* $0 < R_0 \leqslant R(\Delta t|T_S)$.
(2) *If* $ar(C_2^* - C_1^*)\exp[-rW^*(T_S)] < (C_3 + C_0)$, $T^* = T_1$ *for* $R(\Delta t|T_S) < R_0 < 1$ *or* $T^* = T_S$ *for* $0 < R_0 \leqslant R(\Delta t|T_S)$.
(3) *If* $ar(C_2^* - C_1^*)\exp[-rW^*(T_{LC})] > (C_3 + C_0)$, $T^* \geqslant T_1$ *for* $R(\Delta t|T_S) < R_0 < 1$ *or* $T^* \geqslant T_S$ *for* $0 < R_0 \leqslant R(\Delta t|T_S)$.

Theorem 3. *Assume* $C_0(T) = C_{01} + C_0 \times \left(\int_{T_S}^{T} w(t)\,dt\right)^m$, $C_{01}, C_0 > 0$, $C_1 > 0$, $C_2 > 0$, $C_3 > 0$, *and* $C_2 > C_1$, *we have*

(1) *If* $ar(C_2^* - C_1^*)\exp[-rW^*(T_S)] > C_3$ *and* $P(T_{LC}) < C_3$, $T^* = \max(T_0, T_1)$ *for* $R(\Delta t|T_S) < R_0 < 1$ *or* $T^* = T_0$ *for* $0 < R_0 \leqslant R(\Delta t|T_S)$.
(2) *If* $ar(C_2^* - C_1^*)\exp[-rW^*(T_S)] < C_3$, $T^* = T_1$ *for* $R(\Delta t|T_S) < R_0 < 1$ *or* $T^* = T_S$ *for* $0 < R_0 \leqslant R(\Delta t|T_S)$.
(3) *If* $P(T_{LC}) > C_3$, $T^* \geqslant T_1$ *for* $R(\Delta t|T_S) < R_0 < 1$ *or* $T^* \geqslant T_S$ *for* $0 < R_0 \leqslant R(\Delta\, t|T_S)$.

Onde:

Nomenclature

$m(t)$	expected mean number of faults detected in time $(0, t)$	$R(\Delta t\|t)$	conditional software reliability
$\lambda(t)$	failure intensity for $m(t)$	T_{LC}	software life-cycle length
$w(t)$	current testing-effort consumption at time t	M_a	actual cumulative number of detected faults after the test
$W(t)$	cumulative testing-effort consumption at time t	C_1	cost of correcting an error during testing
a	expected number of initial faults	C_2	cost of correcting an error during operation, $C_2 > C_1$
r	fault detection rate per unit testing-effort	C_3	cost of testing per unit testing-effort expenditures
N	total amount of testing-effort eventually consumed	T_S	start time of adopting new techniques/ methods
α	consumption rate of testing effort expenditures in the generalized *logistic* testing-effort function	$R(t)$	a measure of software reliability
A	constant parameter in the generalized *logistic* testing effort function	P	additional fraction of faults detected during testing
κ	structuring index	$C_0(t)$	cost function for acquiring or developing the automated test tools or new techniques
β	scale parameter in the *Weibull*-type testing effort function	C_{01}	nonnegative real number that indicates the basic cost of adopting new tools or techniques
m	shape parameter in the *Weibull*-type testing effort function	C_0	unit new-added test cost
		R_0	desired reliability

Critérios de Avaliação

A	2	B	0	C	0	D	2	E	1	F	1	Total	6

Campos extraídos diretamente	
Título	Determining how much software assurance is enough A value-based approach
Autores	Huang, L. and Boehm, B.
Ano de publicação	2005
Fonte de publicação	Empirical Software Engineering, 2005. 2005 International Symposium on
Abstract	
A classical problem facing many software projects is how to determine when to stop testing and release the product for use. On the one hand, we have found that risk analysis helps to address such "how much is enough?" questions, by balancing the risk exposure of doing too little with the risk exposure of doing too much. In some cases, it is difficult to quantify the relative probabilities and sizes of loss in order to provide practical approaches for determining a risk-balanced "sweet spot" operating point. However, we have found some particular project situations in which tradeoff analysis helps to address such questions. In this paper, we provide a quantitative approach based on the COCOMO II cost estimation model and the COQUALMO qualify estimation model. We also provide examples of its use under the differing value profiles characterizing early startups, routine business operations, and high-finance operations in marketplace competition situation. We also show how the model and approach can assess the relative payoff of value-based testing compared to value-neutral testing based on some empirical results. Furthermore, we propose a way to perform cost/schedule/reliability tradeoff analysis using COCOMO II to determine the appropriate software assurance level in order to finish the project on time or within budget.	
Máquina de Busca	IEEE
Campos extraídos a partir do entendimento	
Tipo de Critério	Apriorístico. Provê mecanismos para terminar os testes dentro do orçamento ou tempo.
Caracterização do software	*Não informado*
Princípio do critério	Modelo de custo COCOMO II e COQUALMO
Níveis de testes	*Não informado*
Modelo utilizado	COCOMO II e COQUALMO
Habilidades necessárias	Conhecimentos em COCOMO II e COQUALMO
Contexto de aplicação	Sim
Objetivo do critério	Melhor combinação entre prazo, custo e qualidade
Considera depuração perfeita	*Não informado*
Confiabilidade pré-determinada	Sim
Orçamento para testes é pré-determinado	Sim
Tempo para testes pré-determinado	Sim
Número de faltas pré-determinado	*Não informado*
Taxa de identificação de falhas pré-determinada	*Não informado*
Considera qualidade dos casos de testes	*Não informado*
Considera atividade de testes isolada	Não
Diferencia taxa de detecção de falhas	*Não informado*
Variações do critério	Não há
Modelo de confiabilidade base	*Não aplicável*
Modelo de Custo base	*Não aplicável*
Distribuição de Falhas	*Não aplicável*

Unidade de Medida de Falhas	*Não aplicável*
Tipo de modelo de confiabilidade	*Não aplicável*

Suposições

Não informado

Parâmetros do critério

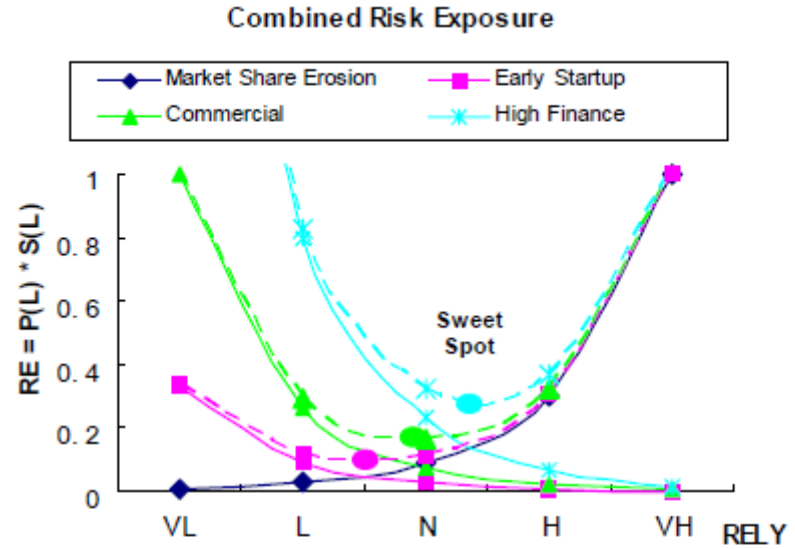
- Métricas necessárias para aplicação do COCOMO II e COQUALMO

Critério

O artigo utiliza os modelos COCOMO II e COQUALMO como modelo de estimativa de custo e sugere pontos de parada para os testes com base nesses modelos.
No artigo, a formula a seguir é apresentada para calcular a exposição ao risco:

$$RE_a = P_a(L) \times S_a(L)$$

A partir desta formula, é traçado os gráficos que demonstram o comportamento de quatro softwares diferentes.



COCOMO II:	0	12	22	34	54	Added % test time
COQUALMO:	1.0	.475	.24	.125	.06	$P_a(L)$
Early Startup:	.33	.19	.11	.06	.03	$S_a(L)$
Commercial:	1.0	.56	.32	.18	.10	$S_a(L)$
High Finance:	3.0	1.68	.96	.54	.30	$S_a(L)$
Market Risk:	.008	.027	.09	.30	1.0	RE_m

Figure 8. Combined Risk Exposures (RE): early startup, commercial and high finance

Segundo artigo, é possível determinar o ponto de parada comparando este gráfico com insuficiência de garantia e perda de fatia de mercado.

Critérios de Avaliação

A	0	**B**	1	**C**	0	**D**	2	**E**	1	**F**	1	**Total**	5

Campos extraídos diretamente	
Título	Economic analysis of software release problems with warranty cost and reliability requirement
Autores	Kimura, M. and Toyota, T. and Yamada, S.
Ano de publicação	1999

Fonte de publicação	Reliability Engineering and System Safety
Abstract	
We discuss optimal software release problems which consider both a present value and a warranty period (in the operational phase) during which the developer has to pay the cost for fixing any faults detected. It is very important with respect to software development management that we solve an optimal software testing time by integrating the total expected testing cost and the reliability requirement. We apply a nonhomogeneous Poisson process model to the formulation of a software cost model and analyze three typical cases of the cost model. Moreover, we derive several optimal release polices. Finally, numerical examples are shown to illustrate the results of the optimal policies.	
Máquina de Busca	Scopus
Campos extraídos a partir do entendimento	
Tipo de Critério	Apriorístico
Caracterização do software	*Não informado*
Níveis de testes	*Não informado*
Modelo Utilizado	Nenhum
Habilidades Necessárias	Nenhuma
Contexto de aplicação	Software para comercialização que possuem tempo de garantia
Princípio do critério	Modelo de confiabilidade
Objetivo do critério	Minimizar o custo total de desenvolvimento restringido a uma confiabilidade
Considera depuração perfeita	*Não informado*
Confiabilidade pré-determinada	Sim
Orçamento para testes é pré-determinado	Não
Tempo para testes pré-determinado	Sim
Número de faltas pré-determinado	Sim
Taxa de identificação de falhas pré-determinada	Sim
Considera qualidade dos casos de testes	Não
Considera atividade de testes isolada	Não
Diferencia taxa de detecção de falhas	Não
Variações do critério	3 variações de acordo com o período de garantia
Modelo de confiabilidade base	Kimura (1999) (Próprio)
Modelo de Custo	Kimura (1999) (Próprio)
Distribuição de Falhas	NHPP
Unidade de Medida de Falhas	Número acumulado de falhas
Tipo de modelo de confiabilidade	Modelo de estimativa de confiabilidade
Suposições	
Não informado	
Parâmetros do critério	
• C_0 – Custo de teste inicial, que é o requisito mínimo indispensável • C_t – Custo de teste por unidade • C_w – Custo de manutenção por falha Durante o período de garantia	
Critério	
O artigo discute algumas políticas de liberação que fornecem o tempo total de teste que minimiza o custo e satisfaz os requisitos de confiabilidade. A principal característica dos critérios apresentados é que eles consideram um tempo de garantia do software depois que ele é colocado em produção. Os problemas que acontecem durante este tempo	

de garantia são de responsabilidade do desenvolvedor.
Três critérios diferentes que atendem três situações foram propostos:

Caso 1 – Quando a duração do período de garantia é constante e não supõe que a confiabilidade crescerá depois da fase de teste.

(4.1) If $h_m(0) > c_t/[c_w T_w(b+\alpha)]$ and $R(x|0) < R_0$, then the optimum release time is $T^* = \max\{T_1, T_R\}$.

(4.2) If $h_m(0) > c_t/[c_w T_w(b+\alpha)]$ and $R(x|0) \geq R_0$, then the optimum release time is $T^* = T_1$.

(4.3) If $h_m(0) \leq c_t/[c_w T_w(b+\alpha)]$ and $R(x|0) < R_0$, then the optimum release time is $T^* = T_R$.

(4.4) If $h_m(0) \leq c_t/[c_w T_w(b+\alpha)]$ and $R(x|0) \geq R_0$, then the optimum release time is $T^* = 0$.

Caso 2 – Quando a duração do período de garantia é constante e supõe o crescimento da confiabilidade do software após os testes.

(5.1) If $h_m(0) > c_t/\{c_w[1 - e^{-(b+\alpha)T_w}]\}$ and $R(x|0) < R_0$, then the optimum release time is $T^* = \max\{T_2, T_R\}$.

(5.2) If $h_m(0) > c_t/\{c_w[1 - e^{-(b+\alpha)T_w}]\}$ and $R(x|0) \geq R_0$, then the optimum release time is $T^* = T_2$.

(5.3) If $h_m(0) \leq c_t/\{c_w[1 - e^{-(b+\alpha)T_w}]\}$ and $R(x|0) < R_0$, then the optimum release time is $T^* = T_R$.

(5.4) If $h_m(0) \leq c_t/\{c_w[1 - e^{-(b+\alpha)T_w}]\}$ and $R(x|0) \geq R_0$, then the optimum release time is $T^* = 0$.

Caso 3 – Quanto a duração do período de garantia obedece a função de distribuição W(t) e o crescimento da confiabilidade do software acontece depois da fase de testes.

(1) If $h_m(0) > c_t/(c_w\gamma)$ and $R(x|0) < R_0$, then the optimum release time is $T^* = \max\{T_3, T_R\}$.

(2) If $h_m(0) > c_t/(c_w\gamma)$ and $R(x|0) \geq R_0$, then the optimum release time is $T^* = T_3$.

(3) If $h_m(0) \leq c_t/(c_w\gamma)$ and $R(x|0) < R_0$, then the optimum release time is $T^* = T_R$.

(4) If $h_m(0) \leq c_t/(c_w\gamma)$ and $R(x|0) \geq R_0$, then the optimum release time is $T^* = 0$.

Critérios de Avaliação

A	0	**B**	1	**C**	0	**D**	2	**E**	1	**F**	1	**Total**	5