Avaliação de Heurísticas para Localização de Defeitos*

Lucas Santos de Oliveira[1], Higor Amario de Souza[2], Danilo Mutti[1], Marcos Lordello Chaim[1]

[1]*Software Analysis and Experimentation Group* (SAEG)
Escola de Artes, Ciências e Humanidades – Universidade de São Paulo (USP), São Paulo, SP, Brasil
[2]Departamento de Ciência da Computação
Instituto de Matemática e Estatística (IME) – Universidade de São Paulo (USP), São Paulo, SP, Brasil
{lucas.santos.oliveira,dmutti,chaim}@usp.br, hamario@ime.usp.br

Abstract. *A program's code coverage, given by structural testing requirements (e.g., nodes, edges, and definition-use associations) exercised by a test set, has been utilized to highlight code excerpts more likely to contain bugs. Coverage-based debugging techniques use heuristics to assign suspiciousness values to testing requirements based on the frequency with which these requirements are executed in failed and passed test cases. Three heuristics for fault localization, namely Tarantula, Ochiai and Wong3, proposed elsewhere were evaluated in this work. The evaluation was performed by comparing the effectiveness of the heuristics. The effectiveness assessment was two-fold: given by the number of localized defects and by the number of lines of code that a programmer has to investigate to locate the fault's site. Experiments performed in medium to large size programs suggest that the heuristic Ochiai localizes more faults in smaller code excerpts.*

Resumo. *A cobertura do código de um programa, dada por requisitos de teste estrutural (e.g., nós, arcos, e associações definição-uso) exercitados por um conjunto de casos de teste, tem sido utilizada para indicar trechos de código mais suspeitos de conter defeitos. Técnicas de depuração baseada em cobertura utilizam heurísticas que atribuem valores de suspeição aos requisitos de teste a partir da frequência com que estes são exercitados em casos de teste de sucesso e de falha. Três heurísticas para localização de defeitos, a saber, Tarantula, Ochiai e Wong3, propostas na literatura, foram avaliadas neste trabalho. A avaliação foi realizada comparando-se a eficácia das heurísticas. A avaliação da eficácia deu-se de duas maneiras: pela quantidade de defeitos encontrados e pelo número de linhas de código que um programador investiga até localizar o defeito. Experimentos realizados com programas de médio e grande porte indicam que a heurística Ochiai localiza mais defeitos em trechos de código menores.*

1. Introdução

Depuração é a atividade de localizar e corrigir defeitos existentes em um

* Este trabalho teve o apoio da Fundação de Amparo à Pesquisa do Estado de São Paulo (processo 2011/03841-0) e do Núcleo de Apoio à Pesquisa em Software Livre (NAPSoL) da USP.

programa. É uma tarefa custosa, pois requer a análise do código fonte. Na prática, os programadores inspecionam o código por meio de comandos que imprimem valores de variáveis ou por meio de depuradores simbólicos que permitem parar a execução do programa e inspecionar o código e os estados das variáveis. Esse processo manual pode ser custoso e *ad-hoc* [Jones et al., 2007].

Diferentes técnicas e ferramentas têm sido propostas para automatizar a tarefa de depuração, principalmente para localização de defeitos [Abreu et al., 2011]. As técnicas de localização de defeitos baseadas em cobertura de código usam heurísticas que associam valores de desconfiança aos requisitos de teste (nós, ramos ou associações definição-uso – adus) para localizar defeitos [Abreu et al., 2011]. A cobertura de código é obtida a partir da execução de um conjunto de teste.

Cada heurística possui um modo de calcular a suspeição dos requisitos, baseando-se na frequência com que os requisitos são executados durante os testes. De forma geral, as heurísticas indicam que requisitos executados com maior frequência em casos de teste que falham são mais suspeitos de conter defeitos. Inversamente, quanto maior a frequência com que um requisito é executado nos casos de teste que passam, menor é a chance desse requisito conter um defeito.

Uma questão importante para o programador que irá utilizar uma técnica de depuração de programas baseada em cobertura de código é: *qual é a heurística mais eficaz para localizar defeitos*? A eficácia diz respeito à capacidade da heurística de localizar defeitos, bem como de indicar o requisito defeituoso com um alto valor de suspeição, o que permite ao programador localizar o defeito investigando uma quantidade menor de requisitos.

A heurística *Tarantula* [Jones et al., 2002] tem sido frequentemente utilizada para experimentos em diversos trabalhos de depuração automatizada. Porém, há outras heurísticas que também apresentam resultados promissores. Entre essas estão as heurísticas *Ochiai* [Abreu et al., 2007] e *Wong3* [Wong et al., 2007]. No entanto, há poucos trabalhos que comparam o desempenho de heurísticas de localização de defeitos. Além disso, esses trabalhos utilizam programas com menos de 150 linhas de código [Naish et al., 2009].

Este trabalho apresenta uma avaliação das heurísticas *Tarantula*, *Ochiai* e *Wong3* para localização de defeitos. Os experimentos foram realizados em programas reais[1] contendo de 1.800 a 80.000 linhas de código. Os defeitos avaliados estão divididos entre defeitos reais (que ocorreram durante o desenvolvimento do sistema) e semeados (introduzidos artificialmente para realizar experimentos).

O restante deste trabalho é organizado da seguinte maneira: na Seção 2 são apresentados conceitos sobre as técnicas de depuração baseada em cobertura de código. Na Seção 3 são descritos o experimento e os resultados obtidos. A Seção 4 apresenta a discussão dos resultados. Na Seção 5 são descritos os trabalhos relacionados. A Seção 6 apresenta as ameaças à validade. Por fim, a Seção 7 traz as conclusões do trabalho.

1Usa-se o termo “programa real” para indicar programas desenvolvidos para uso prático, em oposição a programas simples desenvolvidos para provas de conceito e programas contendo uma quantidade pequena de linhas de código.

2. Depuração baseada em cobertura de código

Cobertura de código pode ser definida como o conjunto dos requisitos cobertos pela execução de casos de teste [Abreu et. al., 2007]. Neste contexto, os requisitos podem ser comandos, predicados, associações definição-uso (adus) ou chamadas de função. As informações de cobertura de código são usadas por heurísticas de localização de defeitos para calcular a suspeição dos requisitos. O resultado é uma lista de requisitos classificados de acordo com seus valores de suspeição em ordem decrescente. A ideia é que o programador siga a ordem de classificação indicada na lista para localizar o defeito.

Assim, a definição da heurística utilizada para atribuir valores de suspeição aos trechos de código é uma questão importante na depuração baseada em cobertura de código. Diversas heurísticas têm sido propostas ou utilizadas para indicar requisitos com maior probabilidade de conter defeitos [Naish et al., 2011]. Em geral, as heurísticas baseiam-se na frequência com que os requisitos são executados durante os casos de teste. O que diferencia as heurísticas é a importância atribuída aos casos de teste de falha e de sucesso. Algumas heurísticas, como a *Tarantula*, foram criadas para localização de defeitos [Jones et al., 2002], enquanto outras, como a *Ochiai*, foram adaptadas de áreas como Biologia Molecular [Abreu et al., 2007].

Tarantula é uma das primeiras heurísticas utilizadas para calcular valores de suspeição em requisitos. Ela calcula a suspeição baseada na frequência de execução de cada requisito nos casos de testes de falha, dividindo-a pela soma da frequência com que o requisito é executado em casos de testes que falham e que passam.

A fórmula da *Tarantula* ($H_{(t)}$) é mostrada na Equação 1. Os valores c_{ef}, c_{nf}, c_{ep}, e c_{np}, conhecidos como *coeficientes de cobertura*, indicam o número de vezes que um requisito assumiu uma das seguintes situações durante a execução dos casos de teste. O coeficiente c_{ef} indica o número de vezes que o requisito *(c)* é executado *(e)* em casos de teste que falham *(f)*, c_{nf} é o número de vezes que um requisito não é *(n)* executado em casos de teste que falham *(f)*, c_{ep} é o número de vezes que o requisito é executado *(e)* por casos de testes que passaram *(p)* e c_{np} representa o número de vezes que o requisito não é *(n)* executado por casos de testes que passaram *(p)*.

$$H(t) = \frac{\frac{Cef}{Cef + Cnf}}{\frac{Cef}{Cef + Cnf} + \frac{Cep}{Cep + Cnp}}$$

Equação 1: Fórmula da heurística Tarantula

Considere o método *max*, mostrado na Listagem 1, que determina o maior elemento num *array* de inteiros. São apresentadas as linhas de código e os nós associados aos comandos do método. Um nó (ou bloco) é um grupo de comandos executados em ordem sequencial, isto é, uma vez que o comando pertence ao nó executado, todos os comandos vão ser executados. O programa tem um defeito localizado na linha 2 ou no nó 1. Esse defeito ocorre porque a variável *i* é incrementada antes que o valor da posição 0 do *array max* seja utilizado.

A Tabela 1 apresenta cinco casos de testes (t1, t2, t3, t4 e t5) para testar o

método *max*. Quando executados, os casos de teste t4 e t5 falham, enquanto t1, t2 e t3 passam. A Tabela 2 apresenta a matriz de cobertura para o método *max*, indicando os comandos e nós cobertos, assim como os coeficientes de cobertura para determinar a suspeição usando a heurística *Tarantula*.

Tabela 1: Casos de teste para o método max

Casos de teste	
t1	{[(1, 2, 3), 3], 3}
t2	{[(5, 5, 5), 3], 5}
t3	{[(2, 10, 1), 3], 10}
t4	{[(4, 2, 3), 3], 4}
t5	{[(4), 1], 4}

Linha	Nó	Comando
-	-	int max (int array [], int length)
-	1	{
1	1	int i = 0;
2	1	int max = array[++i]; // array [i++];
3	2	while(i < lenght)
-	3	{
4	3	if(array[i] > max)
5	4	max = array[i];
6	5	i = i + 1;
-	5	}
7	6	return max;
-	6	}

Listagem 1: Método max

Tabela 2: Comandos e nós cobertos pelo método max.

Linha	Nó	t1	t2	t3	t4	t5	C_{np}	C_{ep}	C_{nf}	C_{ef}	$H_{(t)}$
1	1	x	x	x	x	x	0	3	0	2	0.5
2	1	x	x	x	x	x	0	3	0	2	0.5
3	2	x	x	x	x		0	3	1	1	0.33
4	3	x	x	x	x		0	3	1	1	0.33
5	4	x	x	x	x		0	3	1	1	0.33
6	5	x	x	x	x		0	3	1	1	0.33
7	6	x	x	x	x		0	3	1	1	0.33
-	-	Passa	Passa	Passa	Falha	Falha	-	-	-	-	-

A coluna H(t) da Tabela 2 apresenta os valores de suspeição obtidos usando a heurística *Tarantula*. Para as linhas 1 e 2 é 0,5 e para as outras linhas é 0,33. Portanto, as linhas 1 e 2 (isto é, o nó 1) são os requisitos mais suspeitos de acordo com *Tarantula*.

Outras heurísticas utilizam esses coeficientes de cobertura (e.g., c_{ef}, c_{nf}, c_{ep}, e c_{np}) com diferentes pesos no cálculo de suspeição. A heurística *Ochiai* foi aplicada à

localização de defeitos e os resultados apontaram melhor desempenho em relação à *Tarantula* para os *benchmarks Siemens suite* e *Unix* [Abreu et al., 2007]. A fórmula da heurística *Ochiai* é apresentada na Equação 2.

$$H(o) = \frac{Cef}{\sqrt{(Cef + Cnf) * (Cef + Cep)}}$$

Equação 2: Fórmula da heurística Ochiai

A heurística conhecida como *Wong3* parte da ideia de que um componente de código (nó, arco, associação definição-uso) executado várias vezes por casos de teste de falha e poucas vezes por casos de teste de sucesso fornecem mais informações relevantes para a descoberta de defeitos. Dessa forma, o cálculo da *Wong3* é dado pelo total de vezes que o componente foi executado pelos casos de teste de falha (c_{ef}) menos um valor *p* que depende da quantidade de vezes em que ele foi executado por casos de teste de sucesso (c_{ep}). A Equação 3 descreve a fórmula da heurística *Wong3*.

A heurística *Wong3* obteve um dos melhores resultados em eficácia de localização em experimentos realizados usando o *Siemens suite* [Naish et al., 2011; Wong et al., 2007].

$$H(w) = Cef - p, \text{ onde } p = \begin{cases} Cep & \text{se } Cep \leq 2 \\ 2 + 0.1\ (Cep - 2) & \text{se } 2 \leq Cep \leq 10 \\ 2.8 + 0.001\ (Cep - 10) & \text{se } Cep > 10 \end{cases}$$

Equação 3: Fórmula da heurística Wong3

3. Avaliação de heurísticas para localização de defeitos

Nesta seção é apresentada a avaliação experimental realizada com as heurísticas *Tarantula*, *Ochiai* e *Wong3* em quatro programas reais. A seguir, são descritos os programas usados na avaliação, o processo de condução dos experimentos, a métrica usada na avaliação e os resultados obtidos.

Existem alguns programas que são recorrentemente utilizados pelas técnicas de depuração automatizada. Esses programas, em geral, são pequenos, formados por algumas centenas de linhas executáveis e seus defeitos são semeados. O uso de programas desse porte não possibilita que os experimentos assemelhem-se a ambientes que contenham programas reais, que são o objetivo principal das técnicas desenvolvidas. Neste experimento, são utilizados programas maiores que serão apresentados na Seção 3.2.

3.1. Questões de pesquisa

1. Qual é a heurística mais eficaz para a localizar mais defeitos dentro de um valor fixo limitado de linhas de código?

2. Qual é a heurística mais eficaz para localizar o defeito com o menor número de linhas de código?

3.2. Programas avaliados

Os critérios de seleção de programas foram: programas de tamanho médio a grande, de código aberto e escritos em *Java*; programas reais, ou seja, que são utilizados

para realizar atividades em situações reais, e não apenas para realizar experimentos científicos; e programas com casos de teste automatizados escritos utilizando a biblioteca *JUnit*[2].

Foram selecionados quatro programas para o experimento: *Ant*, *Commons-Math*, *JTopas* e *XML-Security*. O XML-Security[3] implementa assinaturas digitais e padrões de criptografia para XML. O JTopas[4] é um analisador de arquivos de texto para diferentes formatos. Commons-Math[5] é uma biblioteca para realizar cálculos matemáticos e estatísticos. O Ant[6] é um construtor de aplicações utilizado principalmente para aplicações em Java.

Os programas Ant, JTopas e XML-Security foram obtidos do repositório *Software-artifact Infrastructure Repository*[7] (SIR). Os defeitos dos programas retirados do SIR foram inseridos artificialmente. O programa Commons-Math, por sua vez, possui defeitos reais obtidos analisando-se os registros de mudanças do seu repositório. Os defeitos do Commons-Math foram identificados e organizados numa estrutura similar à existente nos programas do SIR [Souza, 2012].

A Tabela 3 mostra as características dos programas selecionados. Na primeira linha são apresentados os dados do programa Ant. Para esse programa, foram utilizadas oito versões diferentes do seu repositório, sendo que a versão de menor tamanho possui 25 KLOC e a de maior tamanho 80KLOC; nas oito diferentes versões foram semeados 20 defeitos. O tamanho dos conjuntos de casos de teste utilizados variaram de 141 a 649 casos de teste, dependendo da versão do programa utilizada. Os conjuntos de testes utilizados na avaliação das heurísticas foram os disponíveis nos repositórios dos sistemas. A cobertura desses conjuntos não foi avaliada porque foi considerado um cenário no qual o programador dispõe apenas dos testes disponíveis no repositório para a depuração dos programas.

Tabela 3: Características dos programas utilizados.

Programa	LOC	Versões	Defeitos	Casos de teste
Ant	25.000-80.000	8	20	141-649
Commons-Math	16.000-39.000	3	20	1167-2006
JTopas	1.800-5.000	3	3	22-31
XML-Security	16.000-18.000	3	13	84-94

3.2 Instrumentação

A cobertura de nós foi obtida utilizando o *framework Instrumentation Strategies Simulator* (InSS). O InSS é uma ferramenta que permite a coleta de cobertura de código relativa a vários requisitos de teste (nós, ramos, associações definição-uso) [Araujo et

2 http://junit.org/ visitado em 24 de agosto de 2013.

3 http://santuario.apache.org/ visitado em 24 de agosto de 2013.

4 http://jtopas.sourceforge.net/ visitado em 24 de agosto de 2013.

5 http://commons.apache.org/proper/commons-math/ visitado em 24 de agosto de 2013.

6 http://ant.apache.org/ visitado em 24 de agosto de 2013.

7 http://sir.unl.edu/portal/index.php visitado em 24 de agosto de 2013.

al., 2011]. Para cada defeito, foi coletada a cobertura de nós de todos os casos de teste, bem como o resultado do caso de teste (falha ou sucesso).

As listas de nós mais suspeitos para todos os defeitos foram obtidas com a ferramenta DCI – *Depuração de programas baseada em Cobertura de Integração* [Souza, 2012]. A DCI recebe como entrada arquivos gerados pelo InSS contendo a cobertura de código dos casos de teste executados, o resultado do caso de teste e a heurística utilizada para o cálculo da suspeição. Como saída, ela fornece uma lista dos requisitos ordenados por valor de suspeição.

Os experimentos foram realizados com um único defeito por versão. Somente os defeitos que geraram falhas na execução dos testes foram incluídos no experimento.

3.3. Tratamentos

O experimento é composto de três tratamentos, a saber, *Tarantula, Ochiai* e *Wong3*. Cada tratamento representa um cenário no qual o programador inspeciona uma lista de requisitos suspeitos, no caso os nós do programa, seguindo a ordem de suspeição. Primeiramente, são inspecionados os nós com o maior valor de suspeição indicado por uma heurística. Caso o defeito não se encontre entre os nós mais suspeitos, retoma-se a investigação a partir dos nós com suspeição imediatamente inferior, e assim por diante, até que o nó com o defeito seja localizado.

3.4. Condução

O número de nós inspecionados para cada heurística foi coletado como descrito na Seção 3.3 utilizando programas escritos especialmente para essa tarefa. Nesse sentido, os resultados diferem do procedimento utilizado por Souza (2012) visto que este autor inspecionou manualmente os programas utilizando a ferramenta CodeForest [Mutti, 2013]. Para cada valor de suspeição, todos os nós com mesmo valor foram somados, inclusive todos os nós que possuíam o mesmo valor do nó que contém o defeito. O número coletado tem por objetivo representar os nós investigados até a localização do defeito no pior caso.

Foi definido arbitrariamente um valor limite de 70 nós a serem inspecionados. A ideia é que se o número de nós a serem analisados for maior do que 70, provavelmente, o programador irá perder o interesse em utilizar a técnica de depuração baseada em cobertura. Estudos empíricos corroboram essa suposição [Parnin e Orso, 2011].

Assim, a eficácia é avaliada pela verificação se o tratamento é capaz de localizar o defeito em menos de 70 nós inspecionados e também pelo número absoluto de nós inspecionados para localizar o defeito – quanto menos nós inspecionados, mais eficaz é a heurística.

3.5. Resultados

A Tabela 4 mostra a quantidade de defeitos localizados de acordo com cada heurística. Em cada célula são apresentados dois valores: o primeiro representa a quantidade de defeitos localizados investigando-se 70 ou menos nós; o segundo representa a quantidade total de defeitos dos programas selecionados para o experimento. A última linha descreve a soma dos valores em cada programa.

Tabela 4: Número de defeitos localizados por heurísticas

Programa	*Tarantula*	*Ochiai*	*Wong3*
Ant	16/20	14/20	11/20
Commons-Math	20/20	20/20	4/20
JTopas	3/3	3/3	2/3
XML-Security	3/13	9/13	8/13
Total	**42/56**	**46/56**	**25/56**

O número absoluto de nós investigados para cada defeito também foi utilizado para avaliar a eficácia das heurísticas. Procedeu-se a uma análise estatística dos dados na qual os defeitos que requereram a investigação de mais de 70 nós receberam o valor de 71 nós investigados. Utilizou-se o teste dos sinais [Sá, 2007] para avaliar a eficácia na localização de defeitos. A escolha desse teste de hipótese deveu-se às características dos dados, que são não-paramétricos com valores numéricos discretos.

A Tabela 5 apresenta os resultados do teste dos sinais aplicado aos tratamentos. A hipótese nula considera que a heurística na linha leva o programador a inspecionar o mesmo número de nós para chegar ao nó defeituoso que o tratamento da coluna. A hipótese alternativa é que o uso da heurística da linha é mais eficaz do que a heurística da coluna, isto é, ela requer uma quantidade menor de nós a serem inspecionados. Cada célula apresenta o *valor p* obtido e o resultado do teste de sinais.

Tabela 5: Análise estatística das heurísticas.

Teste dos sinais	*Tarantula*	*Ochiai*	*Wong3*
Tarantula	--	99,78% Não rejeitada	93,87% Não rejeitada
Ochiai	**0,845%** **Rejeitada**	--	**0,006%** **Rejeitada**
Wong3	95,06% Não rejeitada	100,00% Não rejeitada	--

4. Discussão

Diferentemente de trabalhos realizados por Naish et al. (2009), que indicavam que algumas heurísticas (*Ochiai* e *Wong3*) apresentavam bons resultados e com valores semelhantes, observou-se que a heurística *Ochiai* localiza 46 dos 56 defeitos dentro do limite de 70 nós, isto é, 82%. Já *Wong3* localizou apenas 25 defeitos, ou seja, 45% do total. *Tarantula* teve um desempenho bom, pois localizou 42 defeitos totalizando 75% do total.

A hipótese nula pode ser rejeitada com significância de 5% para *Tarantula* e *Wong3* com relação a *Ochiai* quando foi comparado o número de nós investigados. Portanto, *Ochiai* leva o programador a inspecionar menos código do que essas heurísticas. Assim, para os programas e defeitos analisados nesse experimento, *Ochiai* seria a melhor escolha dentre as três heurísticas.

5. Trabalhos relacionados

Debroy e Wong (2011) realizaram simplificações algébricas em diversas heurísticas para mostrar que existem heurísticas que são equivalentes, ou seja, que geram classificações idênticas. Naish et al. (2011) usaram diversas heurísticas em seu experimento feito com programas de 150 a 9.000 linhas de código, mostrando que algumas heurísticas como *Ochiai* e *Wong3* são mais eficazes para tais programas, porém, os valores obtidos por algumas heurísticas (e.g., *Tarantula*) nesse mesmo artigo são ligeiramente inferiores.

Uma diferença na análise realizada neste trabalho foi mostrar que *Ochiai* foi mais eficaz com significância estatística em relação às outras heurísticas para a métrica de localização adotada. A métrica utilizada considera uma quantidade absoluta de nós a serem inspecionados, enquanto Naish et al. (2011) utilizam um valor percentual.

6. Ameaças à validade

Neste trabalho, foi considerado que o programador é capaz de identificar o defeito ao verificar o comando defeituoso, o que é conhecido como situação de detecção perfeita [Parnin e Orso, 2011]. De fato, não é possível garantir que o programador encontre o defeito ao analisar o requisito defeituoso, já que o código defeituoso pode ser investigado sem que o defeito seja localizado.

Foram analisados e avaliados programas de código-fonte aberto com diferentes propriedades, tais como domínio, funcionalidade, tamanho, e processo de teste utilizados. Apesar da variação de suas características, não se pode afirmar que os padrões observados irão se reproduzir para outros sistemas, especialmente aqueles desenvolvidos em ambientes industriais. Mais estudos serão necessários para comparar os resultados em vários sistemas e domínios. Todos os programas foram depurados considerando a existência de um único defeito, porém, na prática vários defeitos estão presentes no código ao mesmo tempo.

7. Conclusões

Nas técnicas de depuração baseada em cobertura de código, as heurísticas para localização de defeitos são peças fundamentais, pois são elas que indicam os trechos de código mais suspeitos de conter defeitos. Portanto, uma questão relevante para o responsável pela depuração de um programa é saber qual a melhor heurística a ser utilizada.

Diferentemente das avaliações anteriores, esse trabalho avalia as heurísticas em programas de médio e grande porte (2K a 80K linhas de código) e com defeitos semeados (introduzidos artificialmente) e reais. Os trabalhos anteriores utilizavam programas pequenos com somente defeitos semeados. A heurística *Ochiai* localizou mais defeitos com menos código a ser analisado para os programas e defeitos dos experimentos.

No entanto, mais experimentos com programas realísticos fazem-se necessários para que haja mais evidências empíricas que corroborem os resultados obtidos. Além disso, o comportamento das heurísticas precisa ser avaliado na presença de mais de um defeito no programa. Outro ponto a ser avaliado é o nível de cobertura do conjunto de

teste e sua influência no desempenho da heurísticas.

Referências

Abreu, R.; Zoeteweij, P.; Gemund, A. J. C. van. On the accuracy of spectrum-based fault localization. In: Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION, 2007. TAICPART-MUTATION 2007. [s.n.], 2007. p. 89-98.

Abreu, R.; Zoeteweij, P.; Gemund, A. J. C. van. Simultaneous debugging of software faults. J. Syst. Softw., Elsevier Science Inc., New York, NY, USA, v. 84, p. 573-586, 2011.

Araujo, R. A.; Accioly, A.; Alencar, F. A.; Chaim, M. L. Evaluating instrumentation strategies by program simulation," in Proceedings of IADIS International Conference on Applied Computing (2011). Rio de Janeiro, Brazil: IADIS, 2011.

Debroy, V.; Wong, W. On the equivalence of certain fault localization techniques. In: Proceedings of the 2011 ACM Symposium on Applied Computing. New York, NY, USA: ACM, 2011. SAC '11, p. 1457-1463.

Jones, J. A.; Harrold, M. J.; Stasko, J. Visualization of test information to assist fault localization. In: Proceedings of the 24th ACM/IEEE International Conference on Software Engineering. [S.l.: s.n.], 2002.

Jones, J. A.; Bowring, J. F.; Harrold, M. J. Debugging in parallel. In: Proceedings of the 2007 International Symposium on Software Testing and Analysis. New York, NY, USA: ACM, 2007. (ISSTA '07), pp. 16–26.

Mutti, D. Coverage-based Debugging Visualization. Dissertation Proposal. Escola de Artes, Ciências e Humanidades, Universidade de São Paulo, Agosto de 2013. São Paulo, Brasil.

Naish, L.; Lee, H. J.; Ramamohanarao, K. Spectral debugging with weights and incremental ranking. In: Proceedings of the Asia-Pacific Software Engineering Conference, 2009. AP-SEC '09. [s.n.], 2009. p. 168-175.

Naish, L.; Lee, H. J.; Ramamohanarao, K. A model for spectra-based software diagnosis. ACM Trans. Softw. Eng. Methodol., ACM, New York, NY, USA, v. 20, p. 11:1-11:32, August 2011.

Parnin, C.; Orso, A. Are automated debugging techniques actually helping programmers? In: Proceedings of the 2011 International Symposium on Software Testing and Analysis. New York, NY, USA: ACM, 2011. (ISSTA '11), p. 199-209.

Sá, J. P. M. de. Applied Statistics Using SPSS, STATISTICA, MATLAB and R. 2. ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007. 506 p.

Souza, H. A. Depuração de Programas Baseada em Cobertura de Integração. Dissertação de Mestrado. Escola de Artes, Ciências e Humanidades, Universidade de São Paulo, Dezembro de 2012. São Paulo, Brasil.

Wong, W.; Qi, Y.; Zhao, L.; Cai, K.Y. Effective fault localization using code coverage. In: Proceedings of the 31st Annual International Conference on Computer Software and Applications, 2007. COMPSAC 2007.. [s.n.], 2007. v. 1, p. 449-456.