GUSTAVO DA MOTA RAMOS

Seleção entre estratégias de geração automática de dados de teste por meio de métricas estáticas de softwares orientados a objetos

São Paulo

2018

GUSTAVO DA MOTA RAMOS

Seleção entre estratégias de geração automática de dados de teste por meio de métricas estáticas de softwares orientados a objetos

Versão original

Dissertação apresentada à Escola de Artes, Ciências e Humanidades da Universidade de São Paulo para obtenção do título de Mestre em Ciências pelo Programa de Pós-graduação em Sistemas de Informação.

Área de concentração: Metodologia e Técnicas da Computação

Versão corrigida contendo as alterações solicitadas pela comissão julgadora em xx de xxxxxxxxxxxxxxx de xxxx. A versão original encontra-se em acervo reservado na Biblioteca da EACH-USP e na Biblioteca Digital de Teses e Dissertações da USP (BDTD), de acordo com a Resolução CoPGr 6018, de 13 de outubro de 2011.

Orientador: Prof. Dr. Marcelo Medeiros Eler

São Paulo

2018

Ficha catalográfica

Errata

Elemento opcional para versão corrigida, depois de depositada.

Dissertação de autoria de Fulano de Tal, sob o título **"Seleção entre estratégias de geração automática de dados de teste por meio de métricas estáticas de softwares orientados a objetos"**, apresentada à Escola de Artes, Ciências e Humanidades da Universidade de São Paulo, para obtenção do título de Mestre em Ciências pelo Programa de Pós-graduação em Sistemas de Informação, na área de concentração Metodologia e Técnicas da Computação, aprovada em _____ de _____ de _____ pela comissão julgadora constituída pelos doutores:

Prof. Dr. _____
Presidente
Instituição: _____

Prof. Dr. _____
Instituição: _____

Prof. Dr. _____
Instituição: _____

Prof. Dr. _____
Instituição: _____

Lista de figuras

Lista de algoritmos

Lista de tabelas

Sumário

[1] De acordo com a Associação Brasileira de Normas Técnicas. NBR 6023.

1 Introdução

Produtos de software de diferentes tamanhos e complexidades são utilizados todos os dias em atividades profissionais ou de entretenimento. Em qualquer circunstância, entretanto, a falta de qualidade em produtos pode caracterizar uma situação preocupante para seus produtores (HILBURN; TOWHIDNEJAD, 2002) (BINDER, 1994), uma vez que cada vez mais os níveis aceitáveis de qualidade de um software estão aumentando, tanto no que se refere ao seu comportamento observável externamente quanto ao seu processo de desenvolvimento e estrutura interna (NORTHROP, 2006) (BASHIR; BANURI, 2008) (GRAHAM; VEENENDAAL; EVANS, 2008).

O teste de software (TAHIR; MACDONELL; BUCHAN, 2014) é a maneira mais popular de verificar se um software atende às especificações descritas e cumpre o papel desejado pelos interessados (SOMMERVILLE et al., 2008). Ela consiste na execução do programa sob teste para revelar seus defeitos. Para isso, casos de teste são gerados de maneira que satisfaçam a diversos critérios, em geral baseados na especificação e na implementação do software (PEZZÈ; YOUNG, 2008).

Entretanto, gerar casos de teste para testar o software nos mais diversos contextos e dados de entrada é uma das atividades mais custosas no ciclo de vida de software, requisitando muito tempo e esforço para o seu planejamento, execução e manutenção. (TAHIR; MACDONELL; BUCHAN, 2014). Consequentemente, pesquisadores e profissionais da área passaram então a desenvolver abordagens para gerar casos de teste automaticamente e assim reduzir o custo e a complexidade desta tarefa. Diversas técnicas já foram utilizadas com teste objetivo: geração de testes randômicos (PACHECO; ERNST, 2007); execução simbólica (CADAR; SEN, 2013); teste baseado em modelos (DICK; FAIVRE, 1993); e teste baseado em buscas (MCMINN, 2004; HARMAN; MANSOURI; ZHANG, 2012). Diversas ferramentas já foram desenvolvidas utilizando uma ou mais dessas técnicas, como, por exemplo, Randoop (PACHECO; ERNST, 2007), DART (GODEFROID; KLARLUND; SEN, 2005), CUTE (SEN; MARINOV; AGHA, 2005), JCute (SEN; AGHA, 2006), Klee (CADAR; DUNBAR; ENGLER, 2008), JavaPathFinder (VISSER; PăSăREANU; KHURSHID, 2004), PEX (TILLMANN; HALLEUX, 2008) e EvoSuite (FRASER; ARCURI, 2011).

2 Teste de software: Aspectos conceituais e econômicos

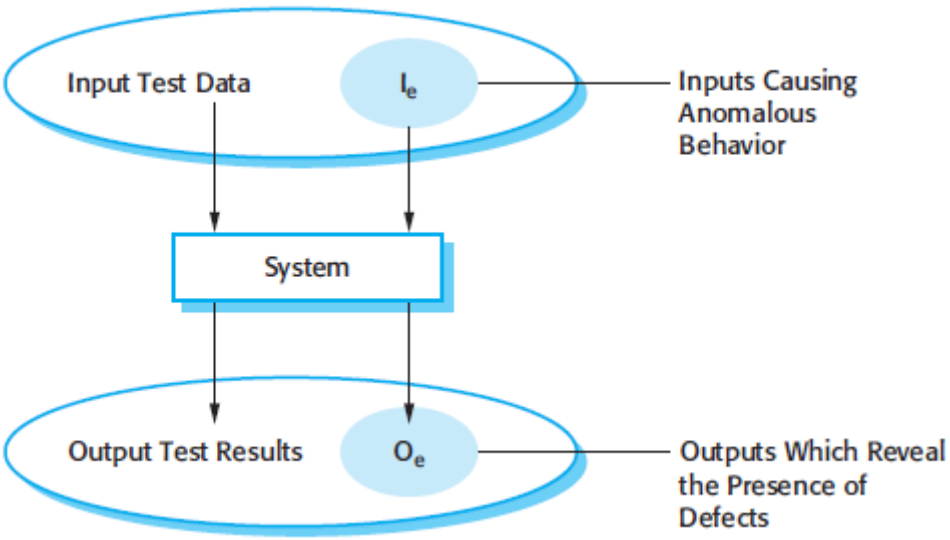
No presente capítulo será apresentado uma síntese dos conceitos base com o objetivo de fundamentação teórica dos capítulos seguintes e auxílio na estrutura do presente documento.

2.1 Teste de software

O teste de software é uma subárea da engenharia de software que compreende as atividades capazes de determinar se um software em análise contém erros (GOODENOUGH; GERHART, 1975) com objetivo encontrar falhas e não de provar a corretitude de software, estes não são capazes de demonstrar se um software em análise está livre de defeitos ou que irá se comportar como esperado em todas as circunstâncias, ou seja, testes são responsáveis somente por evidenciar a presença de possíveis erros e não a falta destes (PFLEEGER; ATLEE, 2010). A partir destes é possível validar se o software cumpre o propósito de desenvolvimento e por fim eliminar possíveis defeitos e assim garantir maior qualidade de software (SOMMERVILLE, 2010).

Para executar testes de software, é necessário simular sua execução e validar, a partir dos resultados extraídos da execução, possíveis anomalias e erros, ou seja, testes de software são executados com dados artificiais (SOMMERVILLE, 2010).

Figura 1 – Modelo de teste de software



Fonte: (SOMMERVILLE, 2010)

De acrodo com (PRESSMAN, 2009) e (DAVIS, 1995) são notórios os princípios de teste:

1. Todos os testes devem ser rastreáveis aos requisitos de usuários;
2. Testes devem ser planjeados antes de sua execução;
3. O princípio de pareto deve ser aplicado ao teste de software;
4. Testes devem começar as menores unidades afim de atingir partes maiores de software;
5. Testes exaustivos não são possíveis.

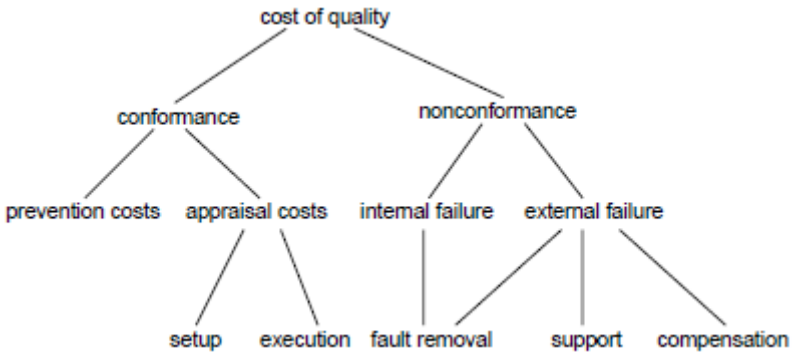
2.2 Qualidade de software e aspectos econômicos

A Garantia de Qualidade ou *Quality Assurance* é considerada uma das partes mais caras do desenvolvimento de software (WAGNER; SEIFERT, 2005), o conceito de garantia de qualidade tornou-se mais evidente após a primeira conferência de engenharia de software com o propósito de estabelecer melhores princípios e estratégias econômicas para atingir um estado viável de software, esta conferência de em 1968 foi responsável inclusive a disciplina de teste e controle de qualidade em diversas fases de desenvolvimento de software desde então (REPASI, 2009).

De fato, os custos de garantia de qualidade são elevados (WAGNER; SEIFERT, 2005) (KOREL, 1990) já que consistem em todo e qualquer valor investido em atividades cujo propósito almeja qualidade de software (PRESSMAN, 2009), porém é notável que fatores econômicos em melhoria de qualidade não são bem aceitos por todos os interessados no desenvolvimento de software e que inclusive exista uma certa confusão sobre o valor de negócio da qualidade de desenvolvimento de software (SLAUGHTER; HARTER; KRISHNAN, 1998) já que teste de software é uma atividade trabalhosa e cara (KOREL, 1990) mas estes investimentos estes são importantes e devem ser planejados visto que cada valor gasto em horas de trabalho e não investido em retrabalho pode ser usado para melhorias rápidas em produtos e processo existentes (SLAUGHTER; HARTER; KRISHNAN, 1998).

Os custos direcionados à qualidade de software são categorizados em *conformance* e *nonconformance* (SLAUGHTER; HARTER; KRISHNAN, 1998)(PRESSMAN, 2009) como demonstrado na figura a seguir:

Figura 2 – Hierarquia da classificação dos tipos de custo presentes em desenvolvimento de software



Fonte: (WAGNER; SEIFERT, 2005)

Os custos de *conformance* caracterizam os valores com o intuito de atingir maior qualidade de produto, este por sua vez é divido em custo de prevenção e custo de avaliação. (WAGNER; SEIFERT, 2005)

Custos de prevenção são aqueles associados com o intuito de prevenir defeitos antes que possam ocorrer, geralmente são compreendidos por treinamentos, equipamentos, , revisões técnicas formais, atividades de planejamento de qualidade e reviews de produto (WAGNER; SEIFERT, 2005) (PRESSMAN, 2009).

Custos de avaliação compreendem custos relacionados a medidas e extração de métricas, avaliação e auditoria de produtos.(WAGNER; SEIFERT, 2005), incluem atividades para ganhar conhecimento da condição do software em análise para cada início de ciclo, incluem: avaliações de processo e entre processos e manutenção (PRESSMAN, 2009).

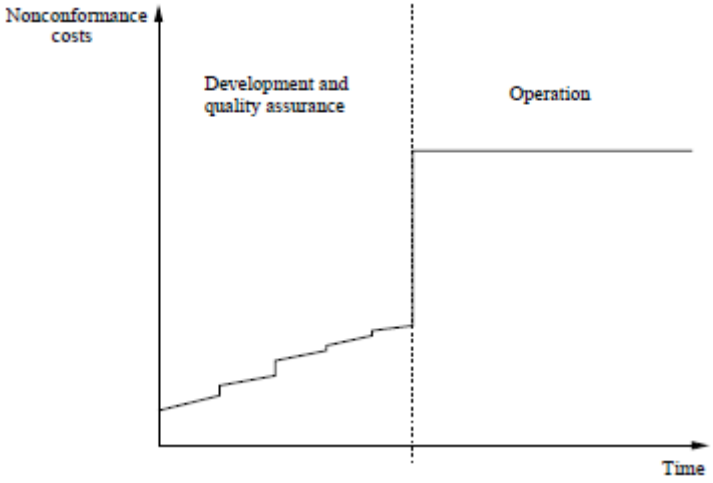
Custos de não conformidade são os custos relacionados aos cenários em que não seguem o planejado e defeitos produzindo um erro e por fim levando à falha (WAGNER; SEIFERT, 2005), nesta categoria estão as subcategorias de custos de falha externa e custos de falha interna (PRESSMAN, 2009).

Os Custos de falha externa caracterizam os custos associados aos defeitos após liberação do produto de software para uso, são exemplos: Resolução de chamados, retorno e substituição de produtos e garantias (PRESSMAN, 2009) enquanto os custos de falha interna são os custos aplicados para remoção de falhas antes de liberação para uso.

O processo de desenvolvimento de um software flui entre os diversos ciclos, e por consequência, diferentes tipos de investimentos são feitos com o intuito de garantiar mainores níveis de qualidade, porém (PRESSMAN, 2009) afirma que o custo relativo para encontrar e reparar um defeito aumenta consideravelmente na linha do tempo entre as fases

iniciais e finais do ciclo de desenvolvimento, sendo assim, custos do tipo *nonconformance* se tornam mais caros ao longo do tempo de projeto, como demonstrado a seguir:

Figura 3 – Crescimento dos custos do tipo *nonconformance* ao longo do tempo de projeto



Fonte: (WAGNER; SEIFERT, 2005)

3 Geração automática de dados de teste

3.1 *Symbolic Execution*

A execução simbólica (*Symbolic Execution*) é um tipo de técnica estrutural para geração de dados de teste que trabalha com base na análise de fluxo de um software para então automaticamente gerar dados de teste (ANAND et al., 2013), o ponto diferencial da *Symbolic Execution* para outras técnicas de geração automática de dados de teste é utilizar valores simbólicos ao invés de valores concretos (KING, 1976) nas entradas do software em análise para que estas possam percorrer seu do fluxo de software e asserções lógicas com a finalidade de geração dos conjuntos de dados de entrada. Valores simbólicos são representações dos valores concretos de variáveis através de expressões simbólicas (ANAND et al., 2013)

Porém a técnica possui alguns problemas fundamentais as quais limitam sua efetividade em casos práticos de desenvolvimento.

Em qualquer passo de execução simbólica, o estado de execução é formado pelos seguintes atributos: *Symbolic Values*, *Path Constraint* e *Program Counter*; *Symbolic Values* representam os valores simbólicos e possíveis de serem atribuídos no passo de execução, o *Path Constraint* é uma relação matemática booleana a qual é o resultado da composição de todas as restrições de entradas dos valores percorridos afim de atingir o determinado passo em análise do software; *Program Counter* é o contador do programa e sempre será responsável por identificar qual será a próxima execução .

A cada passo de análise, o *Path Constraint* é atualizado de maneira que se o seu valor booleano tornar-se infatisfatório, automaticamente o software em análise possui um caminho inatingível, o que torna o fim da análise já que a *Symbolic Execution* não possui caminho para continuar. Se o caminho em análise possuir um *Path Constraint* satisfatório, quaisquer valores que satisfaçam a condição para que o *Path Constraint* são conjuntos possíveis de dados que podem ser utilizados como entrada para validação e atingir o passo em questão.

Uma *Symbolic Execution Tree* é uma representação compacta de todos os possíveis caminhos descobertos por uma análise simbólica durante a execução simbólica de um programa.

Na *Symbolic Execution Tree* os nós representam os estados do software com seus respectivos *Program Counters*, enquanto as arestas representam as transições entre estes estados

Apesar de ser de execução simbólica ser concebida nos anos 70 por (KING, 1976), a técnica tem recebido muita atenção nos últimos anos por dois motivos específicos: Durante a última década, diversas ferramentas de *Constraint solvers* foram desenvolvidos e o uso destes tornou possível a utilização de *Symbolic Execution* em um conjunto mais amplo de softwares.

Um outro ponto é que a *Symbolic Execution* possui um consumo computacional muito maior que todas as outras análises, e por conta das limitações computacionais do *hardware* disponível nos anos 80, ela tornou-se inviável em diversos contextos. Porém no contexto atual, o poder computacional é consideravelmente maior do que o hardware da época, o que atraiu diversos pesquisadores para a utilização desta técnica através de *hardwares* mais poderosos.

Porém a *Symbolic Execution* ainda é uma técnica limita por três problemas distintos: *Path explosion*, *Path divergence* e *Complex constrains*.

3.1.1 Funcionamento do Algoritmo de *Symbolic Execution*

Algoritmo 1 Exemplo de algoritmo demonstrado por King

```
1: procedure PROCEDURE SUM
2:     SUM : PROCEDURE(A, B, C);
3:     X = A + B;
4:     Y = B + C;
5:     Z = X + Y − B
6:     RETURN(Z);
7:     END
```

Fonte: James C. King, 1976

3.1.2 *Path explosion*

É difícil de executar simbolicamente todos os subconjuntos de caminhos de uma *Symbolic Execution Tree* pela ação conjunta de dois fatores: Muitos dos softwares de mercado possuem subconjuntos muito grandes de caminhos a serem testados e a execução simbólica de cada software pode demandar em um uso computacional além do disponível.

Sendo assim, somente um subconjunto limitado de caminhos, muitas vezes um subconjunto não representativo para o teste por completo, pode ser aplicado para execução. Este é um problema com um grau alto de complexidade e necessidade de resolução visto que a razão do número de caminhos inatingíveis em relação aos caminhos atingíveis é alta.

3.1.3 *Path divergence*

Softwares construídos no mundo real possuem diversas implementações em diferentes linguagens, por fim compiladas em um único binário, nestes casos, calcular precisamente as condições requisitam um enorme esforço de desenvolvimento e arquiteturais. Estes casos resultam em divergência de caminhos, ou seja, o caminho que a análise simbólica percorre pode divergir do caminho onde os testes são gerados (ANAND et al., 2013). Isto é dado pelo fato de que cada linguagem possui sua própria semântica de descrição que descrevem como os objetos que representam dados possam ser representados (KING, 1976).

Por conta deste problema, uma execução simbólica de um sistema pode falhar em descobrir um número considerável de caminhos intatingíveis, ou se demandar os modelos pelo usuário, deixar de ser um processo completamente automático.

3.1.4 *Complex constrains*

Resolver todos os *path constraint* pode não ser possível ou trivial, nestes casos o analisador simbólico não consiga resolver conjuntos de *path constraint* que se tornem muito complexos, (ANAND et al., 2013) afirma que isso é pelo fato de *path constraint* possuírem operações lineares como multiplicações, divisões e funções matemáticas como seno e log que se tornam muito complexas para os atuais *Constraint solvers* disponíveis. Por fim, a habilidade de resolver os caminhos diminui e afeta o resultado final.

Estes três problemas devem ser solucionados para que a técnica possa ser usada amplamente em casos de geração de dados de teste e projetos de desenvolvimento reais.

Apesar de de possuir problemas ainda não resolvidos, a técnica de *Symbolic Execution* vem sendo muito usada para geração de dados de teste, porém seu maior uso atualmente é gerar dados de teste para avaliar cobertura de código e expor possíveis falhas nestes.

A técnica de *Symbolic Execution* pode ser efetivamente utilizada em projetos reais desde que estes possuam duas características: eficiência e automação.

Inicialmente a maioria das aplicações atuais utilizam desta técnica para descobrir possíveis caminhos inatingíveis na *Symbolic Execution Tree* e consequentemente, possíveis instruções que nunca serão executadas e por fim, o esforço manual para aplicação da *Symbolic Execution* deve ser aceitável pelo usuário.

A técnica de *Symbolic Execution* difere de outras técnicas por usar valores simbólicos no lugar de valores concretos em variáveis, porém pode ser usada em combinação com outras técnicas de geração.

Symbolic execution tree A *Symbolic Execution Tree* caracteriza os caminhos de execução encontrados durante uma execução simbólica.

3.2 *Model-Based Testing*

O *Model-Based Testing*(MBT) é um método formal que utiliza como insumo para geração de dados de teste. os modelos apresentados no software em análise. Diferente de outras técnicas que procuram fazer a verificação contra modelos formais, o MBT procura reunir conhecimento da corretitude de um software através de abordagens de teste incompletas.

É possível identificar três linhas primárias de estudo em MBT:

- *Axiomatic approaches* (AP);
- *Finite state machines* (FSM);
- *Labeled transition systems* (LTS).

3.2.1 *Axiomatic approaches*

As *Axiomatic approaches* de MBT são bases lógicas para cálculo

3.2.2 *Finite state machines*

A abordagem FSM aplicada em MBT inicialmente foi aplicada para resolver problemas funcionais ocorridos em testes de circuitos de hardware, esta foi depois adaptada

para trabalhar com protocolos de comunicação. Na FSM o modelo é formalizado por um tipo específico de máquina de estados chamado *Mealy machine* onde as entradas e saídas formam tuplas em cada transição. A seleção de testes ocorre pela derivação da sequência de nós percorridos de acordo com o critério de cobertura. A maioria das estratégias de FSM trabalham somente com máquinas de estado determinísticas que é considerada uma restrição já que esta limita trabalhar com casos onde os sistemas são reativos ou pouco especificados.

A seleção de testes em FSM é um assunto vastamente pesquisado nos últimos tempos.

Diversas ferramentas de MBT baseadas em FSM não atingem a completamente a cobertura dos testes, eles utilizam critérios de cobertura como cobertura de transições, cobertura de estados e caminhos.

FSM nãp são expressivas o suficiente para representar modelos reais de software.

3.3 *Combinatorial texting*

Combinatorial Texting tornou-se uma ferramenta muito utilizada pelo profissionais de garantia de qualidade, nesta o foco é selecionar uma amostra das entradas que seja capaz de cobrir primariamente um subconjunto das combinações dos elementos a serem testados.

A representação mais comum destas é a *combinatorial interaction testing* (CIT) onde as N possíveis combinações de valores dos parâmetros estão contidas nas amostras.

Através dos diversos estágios de teste, profissionais de garantia de gualidade confiaram em heurísticas que aproximassem a cobertura dos dados de entrada e saída, a técnica de *Combinatorial Texting* surgiu com base nos conceitos desta época, nesta, os parâmetros e entradas são configurados como fatores e valores, ou seja, para cada fator $f(i)$, é definido um conjunto de N valores, $x1, x2, \cdots xn$. Deste modelo, os casos de teste são gerados a partir do produto cartesiano dos valores de todos os fatores, esta seleção é feita com base em critérios de cobertura.

Um SUT com cinco fatores, sendo que cada fator possua três valores possui 3^5 ou 243 configurações possíveis no total.

CIT se tornou tradicionalmente usada baseada em especificações como uma técnica sistemática para aumentar a performance de outros tipos de teste.

Seu foco é detectar um tipo específico de falha, isso é dado por conta das iterações em cima das combinações das entradas ou das configurações.

3.4 *Adaptive random testing*

A técnica de *Random testing* (RT) é uma das mais populares e fundamentais visto que seu conceito é simples de ser compreendido e implementado.

A TR é a única técnica de geração de dados de teste onde a habilidade de detecção de falhas pode ser analisada teoricamente.

A *Adaptive random testing* (ART) surgiu como uma evolução da RT

Estudos impíricos demonstram que entradas responsáveis por ocorrência de falhas tendem a formar continuas regiões de falhas enquanto entradas não responsáveis por ocorrência de falhas tendem a formar continuas regiões com ausência de falhas

Neste caso, se conjuntos de dados de teste já utilizados não são capazes de revelar uma falha, novos conjuntos de dados de teste devem explorar regiões diferentes dos conjuntos já utilizados.

Por consequência, conjuntos de dados de teste devem ser capazes de explorar proporcionalmente o domínio de valores possíveis de entrada.

O conceito de de explorar proporcionalmente os diferentes subconjuntos de de domínio de entrada é a base intuitiva de formação do ART

Uma contraproposta ao ART surgiu em 1995, o *Anti-random testing*(MALAIYA, 1995), com o mesmo intuito de explorar proporcionalmente os diferentes subconjuntos de de domínio de entrada, porém com uma diferença fundamental: trata-se de um método determinístico, com a exceção do seu primeiro conjunto de dados gerado ser randômico. A essência do ART é ser um método não determinístico, diferindo também *Anti-random testing* em configurações, já que este possui a restrição de começar a geração com um número de casos de teste já gerados pelo profissional de g arantia de qualidade.

O ART possui alguns conceitos que necessitam ser elicitados para sua compreensão:

- *Failure rate*: Caracterizada por um valor numérico que é a relação do número de entradas causadoras de falhas pela quantidade de valores possíveis no domínio de entrada;

- *Failure patterns*: Representa as distribuições e geometria das entradas causadores de falhas;
- *Efficiency*: O tempo computacional total requerido, onde um valor menor representa uma eficiência maior ;
- *Effectiveness*: Refere à capacidade de detectar uma falha a qual pode ser medida a partir de métricas de efetividade como P-measure, E-measure, F-measure e etc; A F-measure é uma métrica que demonstra o número de casos de teste necessários para encontrar a primeira falha; A P-measure é a probabilidade de detectar ao menos uma falha e por fim, a E-measure é o número da falhas esperado.

Diversas abordagens foram propostas para o conceito de explorar proporcionalmente o domínio de valores possíveis de entrada, e como consequência destas abordagens, diversos algoritmos de ART foram propostos.

3.4.1 *Selection of the best candidate*

O conceito desta abordagem é selecionar o melhor candidato como um próximo conjunto de candidatos, sendo assim, diversos conjuntos de entradas são gerados como candidatos e os melhores candidatos passam por um filtro de restrições onde será escolhido o próximo conjunto.

3.4.2 *Exclusion*

A cada iteração da geração, é definida uma zona de exclusão a partir de casos de teste já executados, os próximos candidatos gerados são gerados fora desta zona afim de gerar uma exploração proporcional.

3.4.3 *Partitioning*

A abordagem *partitioning* utiliza informações das localizações já executadas de casos de teste anteriores, um pouco similar a técnica de *exclusion*, com a finalidade de dividir os domínios de entrada em partições e assim definir à qual partição os próximos casos de teste gerados pertencem.

3.4.4 *Test profiles*

3.4.5 *Metric-driven*

4 Métricas de Software orientados a objeto

Projeto e desenvolvimento de softwares orientados a objeto são conceitos populares no cenário atual de desenvolvimento de software (SRIVASTAVA; KUMAR, 2013), neste contexto é notável que classes e métodos são estruturas básicas neste gênero de software (KAN, 1995). O desenvolvimento de software orientado a objetos difere de outros paradigmas pois requer uma abordagem voltada para regras de negócio completamente diferente das tradicionais, dado na maneira como a decomposição funcional e fluxo de dados ocorrem (SRIVASTAVA; KUMAR, 2013).

A análise e projeto de softwares orientados a objeto focam nos objetos como estruturas primárias de computação (KAN, 1995) (SRIVASTAVA; KUMAR, 2013), nesta cada classe é composta por dados e operações realizadas, os quais são representações de uma única entidade real de negócio (SRIVASTAVA; KUMAR, 2013).

As operações realizadas são caracterizadas pelos métodos das classes, a quantidade de funcionalidade, representada por método, agregada ao um software orientado a objeto pode ser estimada com base na quantidade de classe, métodos e variantes destes, utilizados (KAN, 1995). Desta maneira, é comum relacionar métricas de softwares orientados a objeto a valores baseados em atributos de classes e métodos, sejam linhas de código, complexidade, entre outros (KAN, 1995).

As métricas de software tornaram-se elementos primordiais em diversos domínios de Engenharia de software, e principalmente no âmbito de garantia de qualidade, já que toda a informação reúnida por métricas pode ser submetida à diversos tipos de análise e comparação com dados históricos com o objetivo de avaliar e garantir qualidade de software (SRIVASTAVA; KUMAR, 2013). Métricas de software podem ser utilizadas para predizer atributos de qualidade em tempo de execução, inclusive para aplicações cujo requisito é ser uma aplicação de tempo real (SRIVASTAVA; KUMAR, 2013). O real valor de métricas de software vem da sua representação dos atributos de software externos, muitos destes com alto valor de negócio, como confiabilidade, mantenibilidade, reusabilidade, testabilidade e eficiência (SRIVASTAVA; KUMAR, 2013), os quais são requisitos de qualidade descritos pela ISO9126 (ZEISS et al., 2007).

4.1 Medidas diretas

Uma medida direta de software é uma métrica a qual o cálculo de seu valor não depende nenhum outro atributo além do principal utilizado em seu cálculo (SRIVASTAVA; KUMAR, 2013). Exemplos claros de medidas em produtos são: LOC (*Lines of code*), velocidade de execução, tamanho de memória, quantidade de defeitos reportados (SRIVASTAVA; KUMAR, 2013).

4.2 Medidas indiretas

Uma medida indireta de software é uma métrica a qual envolve a medida de um ou mais atributos de software(SRIVASTAVA; KUMAR, 2013). As medidas diretas são, em geral, mais fáceis de serem coletadas.

4.3 O conjunto de métricas CK

Em 1994 Chidamber e Kemerer proporam seis métricas de projeto e complexidade de softwares orientados a objeto, as quais, futuramente, tornaram-se o conjunto de métricas CK (KAN, 1995).

Chidamber e Kemerer aplicaram as seis métricas do conjunto (as quais serão definidas em seções posteriores) em estudos empíricos de duas empresas, uma usando C++ e outra Smalltalk (KAN, 1995), o sumário desta é demonstrado na tabela.

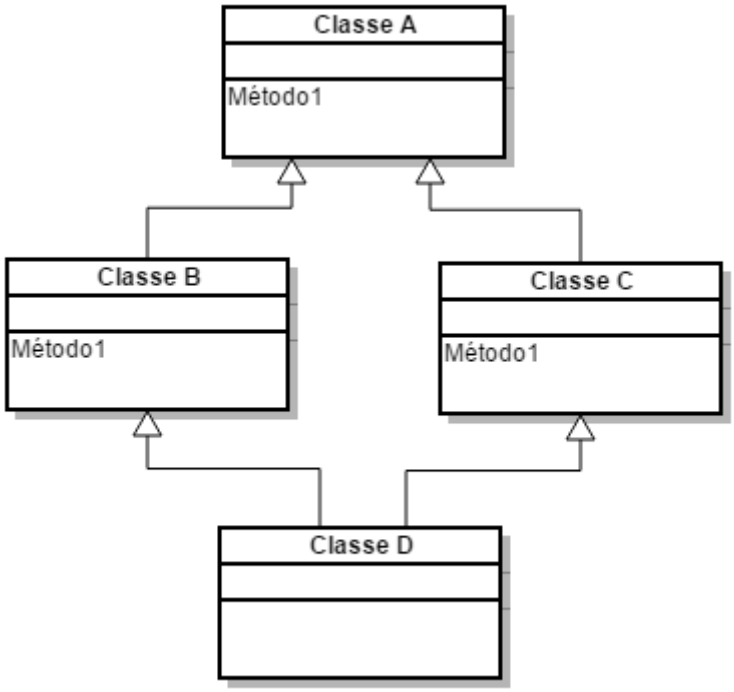
4.3.1 *Weighted methods per class (WMC)*

A WMC é uma métrica contabilizada pela soma das complexidades dos métodos, onde a complexidade é dada pelo *cyclomatic complexity number* de McCabe (CCN)(KAN, 1995) (WATSON; MCCABE; WALLACE, 1996). Medir o *cyclomatic complexity number* não é trivial de implementação dado que nem todos os métodos são acessíveis de acordo com a hierarquia dos objetos, isso é resultado por conta da herança aplicada no projeto do componente (KAN, 1995).

4.3.2 *Depth of inheritance tree (DIT)*

A DIT é uma medida que compreende o valor do comprimento máximo de um caminho em uma árvore de herança, este comprimento é dado pela distância entre o nó em análise até o nó raíz (KAN, 1995). A definição da DIT baseia-se na premissa de que desenvolvedores lidam com linguagens de programação orientadas a objeto as quais permitam que uma classe possua no máximo uma classe pai (BRUNTINK; DEURSEN, 2004), linguagens alheias a este conceito são conhecidas por possuir herança múltipla, presente em linguagens como C++ e Python. Muitas linguagens orientadas a objeto não suportam herança múltipla por conta do problema do diamante, supondo a estrutura a seguir:

Figura 4 – Exemplo de problema do diamante



Fonte: Gustavo Ramos, 2017

Neste caso, um objeto instanciado do tipo D possui acesso para executar o **Método 1**, porém o compilador pode não saber a qual referência de **Método 1** executar já que o objeto do tipo D herda tanto de B e de C quanto de A, todos os quais possuem o **Método 1**. Este caso é conhecido por problema do diamante, por conta deste, linguagens como Java não permitem a herança múltipla.Dada a premissa de que a linguagem do software em análise não suporta herança múltipla, ao calcular o DIT, o número de ancestrais do

nó c em análise, corresponde à profundidade de c na árvore de herança (BRUNTINK; DEURSEN, 2004). A fórmula que descreve o cálculo da DIT para um determinado nó c é dado a seguir:

$$DIT(c) = |Ancestrais(c)| \tag{1}$$

4.3.3 *Number of children of a class (NOC)*

O NOC é uma métrica de simples cálculo, caracterizada pelo número de sucessores imediatos(subclasses) na árvore de hierarquia, calculada com base em uma classe c em análise (KAN, 1995). Seu cálculo é dado pela fórmula a seguir:

$$NOC(c) = |Filhos(c)| \tag{2}$$

4.3.4 *Coupling between object classes (CBO)*

Uma classe A é acoplada a uma classe B se a classe A invoca uma método ou utiliza uma variável de uma instância de B (KAN, 1995), sendo assim, o CBO é dado pelo número de classes a qual uma classe c em análise está acoplada, ou seja, seu cálculo é dado por:

$$CBO(c) = |Quantidadedeclassesquerelaciona(c)| \tag{3}$$

4.3.5 *Response for a class (RFC)*

O RFC é caracterizado pelo número de métodos que podem ser executados na resposta de uma mensagem recebida por uma instância de uma classe (KAN, 1995), ou seja, o RFC é uma contagem do número de métodos de uma classe c em análise e o número de métodos de outras classes que são invocados pelos métodos de c (BRUNTINK; DEURSEN, 2004).

Quanto maior o número de métodos que podem ser invocados indiretamente através de uma chamada de um método, maior a complexidade de uma classe, basicamente o RFC captura o tamanho de conjunto de respostas de uma classe (KAN, 1995). O RFC

é calculado pelo número de métodos locais mais o número de métodos chamados por métodos locais (KAN, 1995).

4.3.6 *Lack Of Cohesion Of Methods (LCOM)*

A independência pode ser medida por critérios qualitativos: coesão e acoplamento, a coesão é a medida relação funcional de um módulo (PRESSMAN, 2009), neste contexto, a coesão de uma classe é dada pela proximidade de métodos locais à instâncias de variáveis na classe, alta coesão indica uma boa subdivisão das classes (KAN, 1995).

A métrica LCOM representa a dissimilaridade dos métodos em uma classe pelo uso de instâncis de variáveis, a alta de coesão aumenta a complexidade e por consequência, oportunidade de ocorrência de erros durante o processo de desenvolvimento (KAN, 1995).

4.4 Métricas de Lorenz e regras de ouro

Baseado nestes conceitos, lorenz(1993) propôs 11 métricas de projetos de softwares orientados a

objeto e diretivas sobre os valores destas

Enquanto a maioria destas métricas são relacionadas à design de código e implementação, as

métricas Agerage number of comment Lines, Number of problem reports per class e numbers od

classes and methods thrown aray se destacam por terem propósitos diferentes.

A tabela proposta por lorenza é definida a seguir:

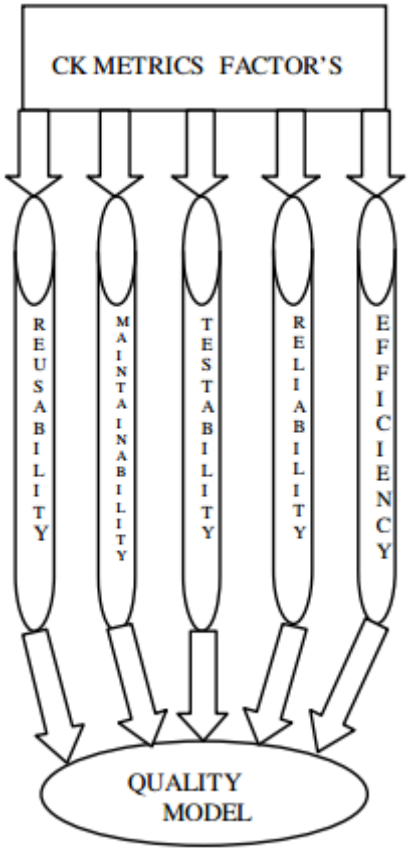
4.5 Modelo proposto por Srivastava e Kumar

Srivastava e Kumar proporam em 2013 um modelo de métricas para validação de qualidade de software com a justificativa de que nenhuma das métricas tratavam acoplamento ou similaridades como relações transacionais, sendo assim, métricas que mapeiam estas características são passíveis de incorporação no modelo. No estudo em questão, diversos dados foram coletados de projetos orientados a objeto nas linguagens Java e C++ e neste constatado que diversas métricas são baseadas em ideias similares e

representam informações redundantes, sendo assim, foi validado que um subconjunto de métricas pode ser utilizado para predição de falhas.

O modelo proposto por (SRIVASTAVA; KUMAR, 2013) atingiu um nível de acurácia superior a 80% na predição de falhas em classes.

Figura 5 – Modelo de qualidade proposto por Srivastava e Kumar



Fonte: (SRIVASTAVA; KUMAR, 2013)

5 Proposta de projeto

Referências[1]

ANAND, S. et al. An orchestrated survey of methodologies for automated software test case generation. *J. Syst. Softw.*, Elsevier Science Inc., New York, NY, USA, v. 86, n. 8, p. 1978–2001, ago. 2013. ISSN 0164-1212. Disponível em: ⟨http://dx.doi.org/10.1016/j.jss.2013.02.061⟩. Citado 2 vezes nas páginas 15 e 17.

BASHIR, M. F.; BANURI, S. H. K. Automated model based software test data generation system. In: *Emerging Technologies, 2008. ICET 2008. 4th International Conference on*. [S.l.: s.n.], 2008. p. 275–279. Citado na página 10.

BINDER, R. V. Design for testability in object-oriented systems. *Commun. ACM*, ACM, New York, NY, USA, v. 37, n. 9, p. 87–101, set. 1994. ISSN 0001-0782. Disponível em: ⟨http://doi.acm.org/10.1145/182987.184077⟩. Citado na página 10.

BRUNTINK, M.; DEURSEN, A. van. Predicting class testability using object-oriented metrics. In: *Source Code Analysis and Manipulation, Fourth IEEE International Workshop on*. [S.l.: s.n.], 2004. p. 136–145. Citado 2 vezes nas páginas 25 e 26.

CADAR, C.; DUNBAR, D.; ENGLER, D. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. Berkeley, CA, USA: USENIX Association, 2008. (OSDI'08), p. 209–224. Disponível em: ⟨http://dl.acm.org/citation.cfm?id=1855741.1855756⟩. Citado na página 10.

CADAR, C.; SEN, K. Symbolic execution for software testing: Three decades later. *Commun. ACM*, ACM, New York, NY, USA, v. 56, n. 2, p. 82–90, fev. 2013. ISSN 0001-0782. Disponível em: ⟨http://doi.acm.org/10.1145/2408776.2408795⟩. Citado na página 10.

DAVIS, M. J. Process and product: Dichotomy or duality? *SIGSOFT Softw. Eng. Notes*, ACM, New York, NY, USA, v. 20, n. 2, p. 17–18, abr. 1995. ISSN 0163-5948. Disponível em: ⟨http://doi.acm.org/10.1145/224155.565634⟩. Citado na página 12.

DICK, J.; FAIVRE, A. Automating the generation and sequencing of test cases from model-based specifications. In: WOODCOCK, J. C. P.; LARSEN, P. G. (Ed.). *FME '93: Industrial-Strength Formal Methods*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993. p. 268–284. Citado na página 10.

FRASER, G.; ARCURI, A. Evosuite: Automatic test suite generation for object-oriented software. In: *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. New York, NY, USA: ACM, 2011. (ESEC/FSE '11), p. 416–419. ISBN 978-1-4503-0443-6. Disponível em: ⟨http://doi.acm.org/10.1145/2025113.2025179⟩. Citado na página 10.

GODEFROID, P.; KLARLUND, N.; SEN, K. Dart: Directed automated random testing. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 40, n. 6, p. 213–223, jun. 2005. ISSN 0362-1340. Disponível em: ⟨http://doi.acm.org/10.1145/1064978.1065036⟩. Citado na página 10.

[1] De acordo com a Associação Brasileira de Normas Técnicas. NBR 6023.

GOODENOUGH, J. B.; GERHART, S. L. Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, SE-1, n. 2, p. 156–173, June 1975. ISSN 0098-5589. Citado na página 11.

GRAHAM, D.; VEENENDAAL, E. V.; EVANS, I. *Foundations of Software Testing: ISTQB Certification*. Cengage Learning, 2008. ISBN 9781844809899. Disponível em: ⟨https://books.google.ie/books?id=Ss62LSqCa1MC⟩. Citado na página 10.

HARMAN, M.; MANSOURI, S. A.; ZHANG, Y. Search-based software engineering: Trends, techniques and applications. *ACM Comput. Surv.*, ACM, New York, NY, USA, v. 45, n. 1, p. 11:1–11:61, dez. 2012. ISSN 0360-0300. Disponível em: ⟨http://doi.acm.org/10.1145/2379776.2379787⟩. Citado na página 10.

HILBURN, Y.; TOWHIDNEJAD, M. Software quality across the curriculum. In: IEEE. *Frontiers in Education, 2002. FIE 2002. 32nd Annual*. [S.l.], 2002. v. 3, p. S1G–18. Citado na página 10.

KAN, S. H. *Metrics and Models in Software Quality Engineering*. Reading, MA: Addison Wesley, 1995. Citado 5 vezes nas páginas 23, 24, 25, 26 e 27.

KING, J. C. Symbolic execution and program testing. *Commun. ACM*, ACM, New York, NY, USA, v. 19, n. 7, p. 385–394, jul. 1976. ISSN 0001-0782. Disponível em: ⟨http://doi.acm.org/10.1145/360248.360252⟩. Citado 3 vezes nas páginas 15, 16 e 17.

KOREL, B. Automated software test data generation. *IEEE Trans. Softw. Eng.*, IEEE Press, Piscataway, NJ, USA, v. 16, n. 8, p. 870–879, ago. 1990. ISSN 0098-5589. Disponível em: ⟨http://dx.doi.org/10.1109/32.57624⟩. Citado na página 12.

MALAIYA, Y. K. Antirandom testing: Getting the most out of black-box testing. In: IEEE. *Software Reliability Engineering, 1995. Proceedings., Sixth International Symposium on*. [S.l.], 1995. p. 86–95. Citado na página 20.

MCMINN, P. Search-based software test data generation: A survey: Research articles. *Softw. Test. Verif. Reliab.*, John Wiley and Sons Ltd., Chichester, UK, v. 14, n. 2, p. 105–156, jun. 2004. ISSN 0960-0833. Disponível em: ⟨http://dx.doi.org/10.1002/stvr.v14:2⟩. Citado na página 10.

NORTHROP, L. M. Let's teach architecting high quality software. In: *19th Conference on Software Engineering Education and Training (CSEE&T 2006), 19-21 April 2006, Turtle Bay, Hawaii, USA*. [s.n.], 2006. p. 5. Disponível em: ⟨http://dx.doi.org/10.1109/CSEET.2006.23⟩. Citado na página 10.

PACHECO, C.; ERNST, M. D. Randoop: Feedback-directed random testing for java. In: *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA*. [S.l.: s.n.], 2007. p. 815–816. Citado na página 10.

PEZZÈ, M.; YOUNG, M. *Software testing and analysis: process, principles, and techniques*. Wiley, 2008. ISBN 9780471455936. Disponível em: ⟨https://books.google.com.br/books?id=mjEiAQAAIAAJ⟩. Citado na página 10.

PFLEEGER, S.; ATLEE, J. *Software Engineering: Theory and Practice*. Prentice Hall, 2010. ISBN 9780136061694. Disponível em: ⟨https://books.google.com.br/books?id=7zbSZ54JG1wC⟩. Citado na página 11.

PRESSMAN, R. *Engenharia de Software - 7.ed.:*. McGraw Hill Brasil, 2009. ISBN 9788580550443. Disponível em: ⟨https://books.google.com.br/books?id=y0rH9wuXe68C⟩. Citado 3 vezes nas páginas 12, 13 e 27.

REPASI, T. Software testing - state of the art and current research challanges. In: *2009 5th International Symposium on Applied Computational Intelligence and Informatics*. [S.l.: s.n.], 2009. p. 47–50. Citado na página 12.

SEN, K.; AGHA, G. Cute and jcute: Concolic unit testing and explicit path model-checking tools. In: BALL, T.; JONES, R. B. (Ed.). *Computer Aided Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006. p. 419–423. ISBN 978-3-540-37411-4. Citado na página 10.

SEN, K.; MARINOV, D.; AGHA, G. Cute: A concolic unit testing engine for c. *SIGSOFT Softw. Eng. Notes*, ACM, New York, NY, USA, v. 30, n. 5, p. 263–272, set. 2005. ISSN 0163-5948. Disponível em: ⟨http://doi.acm.org/10.1145/1095430.1081750⟩. Citado na página 10.

SLAUGHTER, S. A.; HARTER, D. E.; KRISHNAN, M. S. Evaluating the cost of software quality. *Commun. ACM*, ACM, New York, NY, USA, v. 41, n. 8, p. 67–73, ago. 1998. ISSN 0001-0782. Disponível em: ⟨http://doi.acm.org/10.1145/280324.280335⟩. Citado na página 12.

SOMMERVILLE, I. *Software Engineering*. 9th. ed. USA: Addison-Wesley Publishing Company, 2010. ISBN 0137035152, 9780137035151. Citado na página 11.

SOMMERVILLE, I. et al. *Engenharia de software*. ADDISON WESLEY BRA, 2008. ISBN 9788588639287. Disponível em: ⟨https://books.google.com.br/books?id=ifIYOgAACAAJ⟩. Citado na página 10.

SRIVASTAVA, S.; KUMAR, R. Indirect method to measure software quality using ck-oo suite. In: *2013 International Conference on Intelligent Systems and Signal Processing (ISSP)*. [S.l.: s.n.], 2013. p. 47–51. Citado 3 vezes nas páginas 23, 24 e 28.

TAHIR, A.; MACDONELL, S. G.; BUCHAN, J. Understanding class-level testability through dynamic analysis. In: *Evaluation of Novel Approaches to Software Engineering (ENASE), 2014 International Conference on*. [S.l.: s.n.], 2014. p. 1–10. Citado na página 10.

TILLMANN, N.; HALLEUX, J. de. Pex–white box test generation for .net. In: BECKERT, B.; HÄHNLE, R. (Ed.). *Tests and Proofs*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. p. 134–153. ISBN 978-3-540-79124-9. Citado na página 10.

VISSER, W.; PăSăREANU, C. S.; KHURSHID, S. Test input generation with java pathfinder. *SIGSOFT Softw. Eng. Notes*, ACM, New York, NY, USA, v. 29, n. 4, p. 97–107, jul. 2004. ISSN 0163-5948. Disponível em: ⟨http://doi.acm.org/10.1145/1013886.1007526⟩. Citado na página 10.

WAGNER, S.; SEIFERT, T. Software quality economics for defect-detection techniques using failure prediction. *SIGSOFT Softw. Eng. Notes*, ACM, New York, NY, USA, v. 30, n. 4, p. 1–6, maio 2005. ISSN 0163-5948. Disponível em: ⟨http://doi.acm.org/10.1145/1082983.1083296⟩. Citado 3 vezes nas páginas 12, 13 e 14.

WATSON, A. H.; MCCABE, T. J.; WALLACE, D. R. Special publication 500-235, structured testing: A software testing methodology using the cyclomatic complexity metric. In: *U.S. Department of Commerce/National Institute of Standards and Technology*. [S.l.: s.n.], 1996. Citado na página 24.

ZEISS, B. et al. Applying the iso 9126 quality model to test specifications - exemplified for ttcn-3 test specifications. In: *Software Engineering*. [S.l.: s.n.], 2007. Citado na página 23.