Pontifícia Universidade Católica
do Rio de Janeiro

Fischer Jônatas Ferreira

Uma análise da eficácia de assertivas executáveis como observadora de falhas em software

Dissertação de Mestrado

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre pelo Programa de Pós-graduação de Informática da PUC-Rio.

Orientador: Prof. Arndt von Staa

Rio de Janeiro
Abril de 2015

PUC-Rio - Certificação Digital Nº 1221680/CA

Pontifícia Universidade Católica
do Rio de Janeiro

Fischer Jônatas Ferreira

Uma análise da eficácia de assertivas executáveis como observadora de falhas em software

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre pelo Programa de Pós-Graduação em Informática do Centro Técnico Científico da PUC-Rio. Aprovada pela Comissão Examinadora abaixo assinada.

Prof. Arndt von Staa
Orientador
Departamento de Informática - PUC-Rio

Prof. Alberto Barbosa Raposo
Departamento de Informática - PUC-Rio

Prof. Hélio Côrtes Vieira Lopes
Departamento de Informática - PUC-Rio

Profa. Simone Diniz Junqueira Barbosa
Departamento de Informática - PUC-Rio

Prof. José Eugenio Leal
Coordenador (a) Setorial do Centro Técnico Científico - PUC-Rio

Rio de Janeiro, 09 de Abril de 2015

PUC-Rio - Certificação Digital Nº 1221680/CA

Todos os direitos reservados. É proibida a reprodução total ou parcial do trabalho sem autorização da universidade, do autor e do orientador.

Fischer Jônatas Ferreira

Graduou-se em Tecnologia em Análise e Desenvolvimento de Sistema pela Universidade Federal do Rio Grande em Dezembro de 2011. Especializou-se em Análise de Sistema pela Universidade Católica de Petrópolis em outubro de 2013.

Ficha Catalográfica

Ferreira, Fischer Jônatas

Uma análise da eficácia de assertivas executáveis como observadora de falhas em software / Fischer Jônatas Ferreira; orientador: Arndt von Staa – Rio de Janeiro PUC, Departamento de Informática, 2015.

v., 117 f,; il. ; 29,7 cm

1. Dissertação (mestrado) – Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática.

Inclui referências bibliográficas.

1. Informática – Teses. 2. Instrumentação de software 3. Teste de software 4. Software auto verificante 5. Métodos Formais Leves. I. Staa, Arndt von. II. Pontifícia Universidade Católica do Rio de Janeiro. III. Departamento de Informática. IV. Título.

CDD: 004

PUC-Rio - Certificação Digital Nº 1221680/CA

Agradecimentos

Agradeço a DEUS, por tudo ter providenciado com relação a este curso em todos os momentos com intervenções indispensáveis.

Agradeço a minha mãe, Ângela Maria da Silva Ferreira, por todo carinho atenção e cuidado e junto com meu pai Manoel Messias Ferreira ter forjado meu caráter e ter contribuído com tudo que sou. Sem seus cuidados, dedicação e amor incondicional certamente eu não teria persistido e chegado até o término desse curso. Muito obrigado!

Agradeço minha esposa Michele Faria de Oliveira pelo incentivo, por todas as horas sacrificadas do nosso convívio que estive ausente em dedicação a este curso, pelo apoio incentivo e motivação que me prestou em toda minha vida acadêmica. Ainda pelo seu companheirismo principalmente na fase final desse curso.

Agradeço de forma especial ao professor Arndt von Staa por todas as orientações, dedicação e a contribuição significativa e extremamente necessária que me conduziu neste trabalho bem como todo o curso.

Agradeço a Senhora Regina Maria Zanon da Silva pelo apoio dedicação, por todas as horas em que precisei esteve prontamente a ajudar.

Agradeço a professora Maria Carlota Barreto Raimundo por todo apoio nesta dissertação. Por toda atenção a mim dada e por ter contribuído significante com este trabalho.

Agradeço aos professores Edward Hermann Haeusler e Simone Diniz Junqueira Barbosa pela ajuda compreensão e por todas as oportunidades a mim dadas.

Agradeço aos meus amigos e colegas de curso por terem trilhado este caminho junto a mim e dividido momentos de alegria, tensão e conquistas.

Finalmente, gostaria de agradecer à PUC-Rio que me deu a oportunidade de realizar o Mestrado em Informática, por toda a estrutura que me propiciou.

PUC-Rio - Certificação Digital Nº 1221680/CA

Resumo

Ferreira, Fischer Jônatas; Staa, Arndt von. **Uma análise da eficácia de assertivas executáveis como observadora de falhas em software.** Rio de Janeiro, 2015. 117p. Dissertação de Mestrado - Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

A confiabilidade absoluta do software é considerada inatingível, pois mesmo quando confeccionado seguindo regras rígidas de qualidade, o software não está livre da ocorrência de falhas durante a sua vida útil. O nível de confiabilidade do software está relacionado, entre outros, à quantidade de defeitos remanescentes que serão exercitados durante seu uso. Contendo menos defeitos remanescentes, espera-se que o software falhe menos frequentemente, embora muitos desses defeitos sejam exercitados nenhuma vez durante a vida útil do software. Mas desenvolvedores, além de redigir programas, utilizam cada vez mais bibliotecas e serviços remotos que muitas vezes possuem qualidade duvidosa. Na tentativa de tornar o software capaz de observar erros em tempo de execução, surge a hipótese que o uso dos Métodos Formais Leves, por meio do emprego sistemático de assertivas executáveis, pode ser eficaz e economicamente viável para assegurar a confiabilidade do software, tanto em tempo de teste como em tempo de uso. O objetivo principal desta pesquisa é avaliar a eficácia de assertivas executáveis para prevenção e observação de falhas em tempo de execução. As avaliações da eficácia foram feitas por intermédio de uma análise quantitativa utilizando experimentos. Estes, utilizam, implementações de estruturas de dados instrumentadas com assertivas executáveis, submetidas a testes baseados em mutações. Os resultados mostraram que todos os mutantes não equivalentes foram identificados pelas assertivas, embora os testes não foram capazes disso. Também é apresentada uma estimativa do custo computacional relativo ao uso de assertivas executáveis. Com base na infraestrutura criada para realização dos experimentos é proposta uma política de instrumentação de programas utilizando assertivas executáveis a serem mantidas ativas tanto durante os testes como durante o uso produtivo.

Palavras-chave

Instrumentação de software; Teste de software; Software autoverificante; Métodos Formais Leves.

PUC-Rio - Certificação Digital Nº 1221680/CA

Abstract

Ferreira, Fischer Jônatas; Staa, Arndt von (Advisor). **An Effective Analysis of Executable Assertives as Indicators of Software Fails.** Rio de Janeiro, 2015. 117p. MSc. Dissertation - Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Absolute reliability of software is considered unattainable, because even when it is build following strict quality rules, software is not free of failure occurrences during its lifetime. Software's reliability level is related, among others, to the amount of remaining defects that will be exercised during its use. If software contains less remaining defects, it is expected that failures will occur less often, although many of these defects will never be exercised during its useful life. However, libraries and remote services of dubious quality are frequently used. In an attempt to enable software to check mistakes at runtime, hypothetically Lightweight Formal Methods, by means of executable assertions, can be effective and economically viable to ensure software's reliability both at test time as well as at run-time. The main objective of this research is to evaluate the effectiveness of executable assertions for the prevention and observation of run-time failures. Effectiveness was evaluated by means of experiments. We instrumented data structures with executable assertions, and subjected them to tests based on mutations. The results have shown that all non-equivalent mutants were detected by assertions, although several of them were not detected by tests using non-instrumented versions of the programs. Furthermore, estimates of the computational cost for the use of executable assertions are presented. Based on the infrastructure created for the experiments we propose an instrumentation policy using executable assertions to be used for testing and to safeguard run-time.

PUC-Rio - Certificação Digital Nº 1221680/CA

Keywords

Software Instrumentation; Software Testing; Software and self-verificante; Lightweight Formal Methods.

Sumário

PUC-Rio - Certificação Digital Nº 1221680/CA

PUC-Rio - Certificação Digital Nº 1221680/CA

PUC-Rio - Certificação Digital Nº 1221680/CA

PUC-Rio - Certificação Digital Nº 1221680/CA

Lista de Figuras

PUC-Rio - Certificação Digital Nº 1221680/CA

PUC-Rio - Certificação Digital Nº 1221680/CA

Lista de Tabelas

PUC-Rio - Certificação Digital Nº 1221680/CA

1
Introdução

Elevada confiabilidade há muito é desejada em sistemas de computação. Essa necessidade vem se tornando mais intensa, uma vez que o software está presente em inúmeras atividades humanas e de controle de sistemas. Dia a dia, aumenta a nossa dependência de sistemas informatizados (Weber, 2002). Porém, a confiabilidade absoluta é considerada inatingível (Weber, 2002), pois software, mesmo quando confeccionado seguindo regras rígidas de qualidade, não está livre da ocorrência de falhas durante a sua vida útil (Brown, Patterson, 2001). Além disso, desenvolvedores, utilizam cada vez mais bibliotecas e serviços remotos que muitas vezes possuem qualidade duvidosa. O nível de confiabilidade do software está relacionado à quantidade de defeitos remanescentes exercitados durante seu uso. Como o software pode conter defeitos remanescentes e o hardware pode falhar mesmo que muito raras vezes, é impossível prever ou impedir por completo que o software falhe. Assim, é desejado que as falhas possam ser observadas o quanto antes em tempo de execução, a fim de que potenciais danos causados possam ser mantidos sob controle, bem como as causas possam ser removidas ou tratadas (Magalhães et al., 2007).

Uma das soluções utilizadas na tentativa de tornar o software correto por construção é o uso de Métodos Formais (Calinescu, Kikuchi, 2011). Esses se baseiam em modelos e especificações formais detalhadas, e que apoiam todas as fases de desenvolvimento e manutenção de um software. Um dos maiores benefícios da utilização de métodos formais é o entendimento minucioso que se adquire dos requisitos do software, pois os desenvolvedores são forçados a uma análise pormenorizada do sistema. Isso acarreta a detecção prévia de ambiguidades, inconsistências e lacunas que poderiam posteriormente gerar falhas (Easterbrook, 1998). Por outro lado, os altos custos envolvidos no uso dos Métodos Formais e as difíceis tarefas inerentes à sua utilização fazem com que, de modo geral, sejam usados apenas em casos muito específicos (Wikipédia, 2013), onde defeitos remanescentes no sistema possam levar a danos catastróficos, tais como: perdas de vidas, danos para o meio ambiente ou enormes prejuízos. Como o uso de métodos formais ainda requer uma

PUC-Rio - Certificação Digital Nº 1221680/CA

participação substantiva de humanos, erros humanos muitas vezes comprometem as provas da corretude (Yelowitz, 1976).

Infelizmente, uma especificação, mesmo quando formal, pode estar errada, ou, ainda que correta, pode não corresponder ao problema a ser resolvido. Além disso, existe a questão da discrepância entre a prova formal e a implementação realizada (Kneuper, 1997). As práticas modernas de desenvolvimento tornam o software dependente de terceiros, virtualmente impossibilitando uma prova completa (Thomas, 2002). Finalmente, as provas podem estar erradas, dando a falsa impressão que a solução está correta quando, na realidade, contém defeitos remanescentes (Yelowitz, 1976). Disto tudo, a conclusão dos desenvolvedores de software é que o custo do uso de técnicas formais só compensa nos raros casos em que as falhas remanescentes poderiam produzir catástrofes. Mesmo assim, segundo Jackson (2012) as técnicas formais não garantem que o software seja totalmente livre de falhas remanescentes.

Em busca de uma abordagem formal com custos menores são propostos os Métodos Formais Leves (Técnicas Formais Leves). Assertivas executáveis são uma das formas de implementar tais métodos. As Técnicas Formais Leves possuem uma base formal, mas são limitadas quanto ao rigor de uma ou mais das seguintes formas: linguagem, modelagem, análise ou composição (Boyatt, Sinclair, 2008; Easterbrook, 1998).

Construir software livre de falhas ou eliminá-las “a posteriori”, por completo, de um sistema que já esteja em produção é utopia (Berry, 1992). As falhas dos programas acontecem por vários fatores sejam elas internas ou externas ao sistema. Segundo (Magalhães et al., 2007) não são todos os tipos de falhas que podem ser prevenidos, principalmente aquelas externas ao sistema, já que fogem da capacidade de uma medida preventiva inserida no software como, por exemplo, interrupção de transmissão de dados, quedas de energia, ou bibliotecas contendo defeitos, embora, seja possível controlar os danos gerados por esses problemas.

Com relação às falhas internas, podem ser originadas pelos seguintes fatores: Primeiramente, inconsistência nos requisitos, aproximadamente 50% das falhas detectadas na fase de testes são oriundas de problemas no levantamento de requisitos (Blackburn et al., 2001). Erros ocasionados pela falibilidade humana: sejam no desenvolvimento, manutenção, uso do software, ou até mesmo pela interação maliciosa, que visa propositalmente a provocar danos ao sistema (Weber, 2002). Podem ser destacados, também, problemas ocasionados por configurações incorretas ou ausentes (Araújo, 2014) ou até

PUC-Rio - Certificação Digital Nº 1221680/CA

mesmo erros em hardware e plataformas (Magalhães et al., 2007). Por fim, as falhas ocorridas por questões de sincronismo que podem ser observadas quando um determinado fragmento de código apresenta defeito em um dado estado computacional, e esse não apresenta o mesmo defeito quando for executado em outro momento, ainda que sejam aparentemente estados computacionais iguais.

Utilizaremos no restante do texto a seguinte terminologia (Staa, 2014):

- ***Artefatos*** são resultados tangíveis resultantes do desenvolvimento ou manutenção.
- ***Defeito*** é um fragmento de um artefato que, se utilizado, pode levar a um erro.
- ***Erro*** é um desvio entre o que é desejado ou intencionado e o que é gerado ou derivado. Erro é causado por um defeito.
- ***Falha*** é um erro observado.
- ***Latência*** do erro é o tempo decorrido entre o momento em que o erro é gerado e o momento em que é observado.
- ***Dano*** é a consequência externa ao software conhecida (prejuízo) provocada por uma falha.
- ***Lesão*** é a consequência externa ao software desconhecida provocada por um erro não observado

Um aspecto importante a ser destacado em um mecanismo de observação de falhas são os custos relativos à manutenção de software, pois, atualmente a manutenção representa 90% do custo total da vida útil de um software (Erlikh, 2000, apud Araújo, 2011). Sendo que a tarefa mais difícil na manutenção de software, segundo (Souza, 2004), é o entendimento do sistema, correspondendo mais de 50% do trabalho total nesta fase. Durante a depuração de software, e também durante o uso produtivo, uma quantidade significativa de esforço é despendida para identificar a causa raiz de uma falha (Yi et al., 2015)

Como os defeitos não podem ser retirados por completo dos sistemas, é necessário que exista um mecanismo que se possa identificar e tratá-los da melhor forma possível, removendo aqueles que provoquem maiores danos ao software (Staa, 2006). Assim, o ideal é que esse mecanismo possa evitar possíveis defeitos, ou impedir que os erros causados possam se propagar para outros módulos do programa relacionados ao fragmento de código onde foi exercitado o defeito. Contudo, quando não for possível evitá-los que possam ao menos facilitar o processo de sua identificação, proporcionando menores custos para encontrá-los no código.

PUC-Rio - Certificação Digital Nº 1221680/CA

Além disso, é importante que seja pequena a latência entre o momento da geração do erro e o da sua observação, a fim de que a causa real do defeito possa ser mais facilmente descoberta (Magalhães et al., 2007). Pois, com uma latência menor a análise se limitará a um conjunto menor de código, assim, subentende-se que seja menor a interferência de outros dados e fragmentos de código não relacionados ao defeito, dessa forma evita-se que a causa exata do defeito seja maquiada.

Assim, surge a hipótese que o uso dos Métodos Formais Leves, por meio do emprego sistemático de assertivas executáveis possa ser um mecanismo eficaz e economicamente viável para assegurar a confiabilidade do software. Além de torná-lo autoverificante por viabilizar a observação, em tempo de uso produtivo, das eventuais falhas, possibilitando o controle dos potenciais danos por elas causados. Também, permite o desenvolvimento mais próximo do ideal fidedigno por construção, por possibilitar o entendimento mais minucioso dos requisitos do software antes ainda de iniciar o desenvolvimento (Woodcock, 2009; Hierons, 2009; Boyatt, 2007).

PUC-Rio - Certificação Digital Nº 1221680/CA

1.1. Objetivos gerais e específicos

O objetivo principal desta pesquisa foi avaliar a eficácia das assertivas para observação de falhas em tempo de execução. Por intermédio de uma pesquisa quantitativa as avaliações da eficácia foram feitas por meio de testes baseados em mutações de estruturas de dados. O experimento foi realizado com intuito de encontrar subsídios que respondam aos seguintes questionamentos sobre os sistemas implementados com assertivas executáveis:

1. As assertivas podem encontrar falhas dos mutantes antes dos oráculos associados aos casos de teste?
2. Caso ocorram falhas as assertivas podem observá-las em tempo de execução?

Ainda foi descrito o uso de assertivas executáveis segundo os tópicos abaixo relacionados:

1. Como usar assertivas;
2. Quando usá-las;

3. Como criar assertivas segundo seus diferentes tipos, para programas escritos nas linguagens Java e PHP, considerando os seguintes tipos de assertivas:
    a. **Assertivas pontuais** (Meyer, 1992):
        i. **Entrada ou pré-condições:** Condição que deve estar satisfeita para ativar o método.
        ii. **Saída ou pós-condição:** Condição que deve estar satisfeita após a execução do método.
    b. **Assertivas estruturais:** Condições envolvendo todos os atributos de diversos objetos pertencentes a diversas classes e que realizam uma estrutura de dados. Essas condições correspondem a invariantes da estrutura e devem ser sempre verdadeiras quando a estrutura não está sendo alterada (Staa, 2000).

4. Como possibilitar um mecanismo para habilitar e desabilitar o uso de assertivas executáveis em programas escritos nas linguagens Java e PHP.

Por fim, como objetivo secundário foi feita uma revisão da literatura envolvendo os diferentes tipos de abordagens de Métodos Formais Leves e suas viabilidades para utilização em nível industrial.

1.2. Trabalho proposto

Tendo em vista a hipótese formulada parte-se para a realização do experimento para verificar se a mesma tem aplicabilidade real ou não. O experimento foi realizado em dois momentos com objetivos distintos.

O primeiro voltado para verificação de quantas falhas injetadas artificialmente e de forma sistemática as assertivas foram capazes de detectar. Aliado a essa etapa foram propostas métricas capazes de estimar o aproveitamento das assertivas criadas.

No segundo momento do experimento a preocupação foi relativa ao tempo computacional gasto pelo uso das assertivas executáveis. Há necessidade de aferir o seu custo computacional porque elas podem ser mantidas ativas no sistema, estando esse nas fases de desenvolvimento ou de produção. Assim pôde ser demonstrado quantitativamente se as assertivas foram capazes de detectar falhas de forma economicamente viável para o uso.

PUC-Rio - Certificação Digital Nº 1221680/CA

Para a elaboração dos experimentos se fizeram necessários alguns pré-requisitos descritos no decorrer do trabalho. Sendo esses requisitos:

1. Revisão da literatura sobre Métodos Formais Leves;
2. Criação de um modelo de uso de assertivas executáveis;
3. Criação de um modelo de utilização de teste de análise mutante para verificação da eficácia das assertivas executáveis;
4. Emprego da ferramenta MuClipse (Smith, 2007) destinada à análise de testes mutantes para linguagem Java;
5. Implementação de um protótipo chamado MuPHP que possibilita análise de testes mutantes para a linguagem PHP;
6. Criação de assertivas para diversas estruturas de dados;
7. Implementação, execução da infraestrutura do experimento e análise dos dados obtidos.

1.3. Organização do trabalho

Além do capítulo de introdução, este documento apresenta mais seis capítulos que estão organizados da seguinte forma. O Capítulo 2 apresenta Fundamentação teórica e trabalhos relacionados. O Capítulo 3 descreve uma Política de instrumentação de programas proposta. O Capítulo 4 apresenta a ferramenta de testes análise mutante utilizada no experimento, e o protótipo para teste análise mutante para PHP. O Capítulo 5 apresenta os detalhes relativos da infraestrutura do experimento e demonstrações do emprego de assertivas executáveis. O Capítulo 6 relata os resultados alcançados com experimento bem como os dados obtidos. Finalmente o Capítulo 7 apresenta as considerações finais a respeito do trabalho, destacando suas contribuições e desdobramentos em trabalhos futuros.

PUC-Rio - Certificação Digital Nº 1221680/CA

2
Fundamentação teórica e trabalhos relacionados

Este capítulo apresenta a fundamentação teórica utilizada para criação da hipótese formulada. Ele contempla uma visão geral sobre os Métodos Formais Leves (Seção 2.1), aplicação dos Métodos Formais Leves na forma de assertivas executáveis (Seção 2.2), assertivas (Seção 2.3) e Diferenças e comparações na abordagem utilizada (Seção 2.4).

2.1.
Revisão da literatura sobre Métodos Formais Leves

Foi realizada uma revisão da bibliografia, a pesquisa foi voltada apenas para artigos disponíveis na web, por meio do parâmetro de consulta “Lightweight Formal Method”. Foram utilizados os seguintes motores de busca web para auxiliar a pesquisa dos artigos acadêmicos.

1. ACM
2. CiteSeerX
3. Ieeexplore
4. Sciencedirect
5. Scopus
6. Springer link

Foram encontrados 102 artigos, os quais, tinham relevância e relação com o tema em questão, a lista desses artigos se encontra no Apêndice 1 desse trabalho. A fim de selecionar um número menor de artigos, foi realizada a seguinte estratégia:

1. Leitura dos *abstracts* destes artigos;
2. Por meio da leitura dos *abstracts*, foram feitas avaliações dos artigos, segundo uma indicação preliminar de relevância. Assim, os artigos receberam as seguintes indicações numéricas. Quando apresentavam:
   a) Nenhuma relação com tema “0”
   b) Pouca relação com o tema “1”
   c) O assunto apenas sobre Métodos Formais “2”
   d) O assunto apenas sobre Métodos Formais e possuíam exemplos de aplicação e avaliação dos Métodos Formais “3”
   e) O assunto sobre Métodos Formais Leves “4”

PUC-Rio - Certificação Digital Nº 1221680/CA

f) O assunto sobre Métodos Formais Leves que possuíam exemplos de aplicação e avaliação dos Métodos Formais Leves “5”

3. Foi realizada uma breve descrição dos artigos por meio dos seus resumos.
4. Foram identificados os artigos que continham exemplos com código.
5. Foram identificadas as datas de publicação dos artigos.

Com estes dados anteriormente citados preenchidos, foi feita a ordenação dos artigos pela classificação preliminar de relevância e suas datas de publicação. Os 103 artigos encontrados inicialmente receberam as seguintes indicações descritas na tabela abaixo.

Tabela 1 – Classificação numérica dos artigos pesquisados

Número de artigos	Classificação
39	5
17	4
9	3
12	2
13	1
13	0

Como pode ser observado com a tabela acima 39 artigos receberam a classificação de número 5. Destes 39 artigos, por meio da ordenação feita, citada anteriormente, foram selecionados para a pesquisa apenas os cinco primeiros artigos, pois esses artigos apresentam exemplos práticos do uso de métodos formais leves. Seguem abaixo os títulos desses artigos escolhidos para a pesquisa. Encontra-se nos apêndice deste trabalho a lista dos artigos pesquisados com sua indicação preliminar de relevância dada.

1. *Yang, Guowei, Sarfraz Khurshid, and Miryung Kim. "**Specification-based test repair using a lightweight formal method**." FM 2012: Formal Methods. Springer Berlin Heidelberg, 2012. 455-470.*
2. *Shao, Danhua, Sarfraz Khurshid, and Dewayne E. Perry. "**An incremental approach to scope-bounded checking using a lightweight formal method**." FM 2009: Formal Methods. Springer Berlin Heidelberg, 2009. 757-772.*
3. *Larsen, P. G; Fitzgerald, J. S; Riddle, S; **Learning by Doing: Practical Course s in Lightweight Formal Methods using VDM++,** 2007.*

PUC-Rio - Certificação Digital Nº 1221680/CA

4. *Paige, Richard F., and Jonathan S. Ostroff.* **Specification-driven design with Eiffel and agents for teaching lightweight formal methods**. *Springer Berlin Heidelberg, 2004.*

5. *Feather, Martin S. "**Rapid application of lightweight formal methods for consistency analyses.**" Software Engineering, IEEE Transactions on 24.11 (1998): 949-959.*

2.2. Métodos Formais Leves

Com o objetivo de fornecer um grau de formalização nas especificações dos sistemas que não seja no nível dos tradicionais Métodos Formais, surgem os Métodos Formais Leves também chamados de Técnicas Formais. Eles possibilitam o benefício da corretude em fragmentos de códigos essenciais e módulos vitais do sistema.

Essa metodologia está se tornando uma solução viável em custo-eficiência. Como foi observado por (Staa, 2013), “as técnicas formais leves têm se mostrado eficazes e economicamente viáveis como instrumentos de verificação de especificações e de apoio à criação de software autoverificante”. E ainda segundo (Akhtar, Missen, 2014), “as especificações formais leves são flexíveis, menos rigorosas e mais práticas do que especificações formais”.

A característica fundamental deste método é a possibilidade de realizar a tradução dos requisitos do sistema em uma linguagem em nível maior de abstração. Assim, eles induzem os desenvolvedores a terem compreensão mais aprofundada dos requisitos do sistema, melhorando a clareza e precisão das especificações dos requisitos (Feather, 1998). E ainda, a possibilidade de verificar fragmentos de código, a fim de que seja garantido um estado previamente determinado em pontos do algoritmo. Com isso, esse controle proporciona a criação de um mecanismo que viabiliza a observação de falhas em tempo de execução. Os Métodos Formais Leves são capazes de viabilizar a observação de falhas em tempo de execução, além de reduzirem a incidência de defeitos remanescentes. Como pode ser observado em um projeto para Naves Espaciais, onde foi usado retirar inconsistências em banco de dados (Feather, 1998) e na reparação em testes automatizados baseando-se na especificação do sistema (Yang et al, 2012).

Segundo (Feather, 1998) Métodos Formais Leves ocupam um lugar diferente no que diz respeito às técnicas de análise. Eles têm objetivos mais modestos, e empregam as ferramentas que necessitam de menos trabalho preparatório para aplicá-los. Pois, como foi observado por (Cheon, Leavens,

PUC-Rio - Certificação Digital Nº 1221680/CA

2002) a depuração e os testes consomem uma fração significante do custo do desenvolvimento e manutenção de software.

Outra característica que merece destaque é a redução do retrabalho inútil sem aumentar em demasia o trabalho de incluir uma técnica formal leve para instrumentar um código. Pois, segundo Easterbrook (1998) os Métodos Formais Leves podem oferecer uma maneira rentável de melhoria da qualidade das especificações de software. Além do mais, o uso bem sucedido dos Métodos Formais Leves em ambiente de desenvolvimento gera resultados mais simples em tempo hábil (Feather, 1998). Existem vários exemplos bem sucedidos na literatura que demonstram os resultados obtidos com a aplicação de métodos formais na indústria, tais como: projeto de roteamento de mensagens e sistema de alerta de congestionamento de veículos (Larsen, 2006); software para Nave Espacial da NASA (Feather, 1998); ferramenta para reparação de testes (Yang, 2012); Sistema robótico multi-agente para o transporte de estoque de armazéns (Akhtar, Missen, 2014).

Outro fator de suma importância é o suporte que a especificação fornece para fase de manutenção do software. Pois, os contratos especificados conforme os requisitos do sistema também servem como uma boa documentação para projetos detalhados (Cheon, Leavens, 2002).

PUC-Rio - Certificação Digital Nº 1221680/CA

2.3. Assertivas

Assertivas também chamadas de Asserções (Pezzè, Young, 2008) segundo (Duncan, Hölzle, 1998), são expressões booleanas que devem ser satisfeitas caso o fragmento de código a que são associadas esteja operando corretamente. Elas não fazem parte da implementação de um algoritmo, mas descrevem restrições sobre os valores que são manipulados dentro desses fragmentos de códigos. Ainda Boyatt (2008) as define como métodos que têm uma base formal, mas é limitada em seu alcance por um ou mais das seguintes características: é parcial na linguagem, modelagem, análise ou na composição. Segundo Plösch (2002) as técnicas baseadas em assertivas executáveis têm efeitos positivos na qualidade de software em geral.

Existem vários trabalhos na literatura que descrevem a utilização de assertivas executáveis, por meio de especificações chamadas de contrato como exemplo a linguagem Eiffel (MEYER, 1998) que estabelecem especificações, precisas e verificáveis das interfaces para componentes de software. Por meio

de pré-condições, pós-condições e invariantes previamente elaboradas em um nível elevado de abstração, segundo os requisitos do software. Essa técnica é conhecida por Design by Contract a qual é definida como um Método Formal Leve (Paige, Ostroff, 2004). Ainda, podem ser observadas outras técnicas que utilizam Métodos Formais Leves tais como: Model Driven Development, Sistemas Self-checking, a linguagem Eiffel (MEYER, 1998) e Técnicas Alloy (Jackson, 2012).

A popularização do Design by Contract contribui para o surgimento de várias ferramentas que seguem suas especificações de interface para os componentes de software. Esta técnica também é utilizada para aplicações escritas em Java, utilizando assertivas executáveis como podem ser observadas nas ferramentas IContract (Kramer, 2001), Jcontractor (Karaorman, Hölzle and, Bruno, 1999) Jcontract (Parasoft, 2002), Jass (Bartetzko, 2001), Handshake (Duncan, Hölzle, 1998) e JML (Cheon, Leavens, 2002).

O conhecimento mais apurado dos requisitos do sistema impulsionado por meio da construção de assertivas acarreta o desenvolvimento de sistemas mais corretos. O detalhamento maior dos requisitos possibilitará um número maior de falhas observadas. Pois, serão as assertivas as observadoras de possíveis inconsistências de estados.

PUC-Rio - Certificação Digital Nº 1221680/CA

2.4. Diferenças e comparações na abordagem utilizada

A principal diferença na abordagem utilizada no trabalho proposto, comparada com trabalhos anteriores que também avaliam a eficácia de assertivas, se dá pelo fato do experimento ser feito sobre defeitos conhecidos e gerenciáveis. Os defeitos são conhecidos por serem criados por meio de mutantes gerados por operadores de mutação pré-determinados, possibilitando assim o conhecimento prévio dos defeitos injetados. Também podem ser gerenciáveis por meio da funcionalidade oferecida pela ferramenta utilizada que possibilita a identificação do estado dos mutantes (vivos ou mortos) após serem testados. Ainda a observação dos desvios sintáticos que cada mutante apresentou comparado com o programa original.

Neste experimento pode ser observado que do número total de mutantes, quantos foram mortos pelas assertivas executáveis e quantos não foram. Assim, uma estimativa da eficácia das assertivas pôde ser criada, pois o número de defeitos inseridos no código é conhecido.

Ainda neste trabalho foi levado em consideração o tempo computacional gasto pelo uso de assertivas executáveis por dois sistemas instrumentados por elas. Para isso foi comparado o tempo computacional de dois sistemas em dois momentos, o primeiro sem o uso de assertivas e segundo com o uso delas. Assim pôde ser feita a estimativa do percentual de tempo computacional adicionado em virtude do uso de assertivas executáveis.

PUC-Rio - Certificação Digital Nº 1221680/CA

3
Política de instrumentação de programas proposta

Este capítulo apresenta o modelo para verificação da eficácia das assertivas executáveis que preconiza o uso de testes de análise mutante o qual foi utilizado para a realização do experimento.

3.1.
Modelo de verificação da eficácia das assertivas executáveis por meio de testes mutantes

O experimento descrito neste trabalho foi realizado tendo por base um pequeno modelo que preconiza a utilização de assertivas executáveis com testes de analise mutante, desde a fase da implementação dessas, até a análise de sua eficácia, como pode ser observado no modelo demonstrado na Figura 1.

O modelo é constituído dos seguintes passos:

1. O programa poderá ser desenvolvido com assertivas executáveis, ou essas são acrescentadas ao programa que já tenha sido implementado;
2. Devem ser criados os casos de testes unitários para o programa que se deseja instrumentar.
3. É necessário que sejam executados os testes para o programa original juntamente com as assertivas executáveis. Neste passo necessita-se da avaliação do testador para verificar a saída gerada por meio da execução dos casos de testes, analisando o relatório desses. Caso todos os testes passem, então se deve seguir para o próximo passo do modelo, o qual é a geração dos mutantes. Se os testes não passarem, o programa original deverá ser corrigido e também as assertivas executáveis devem ser modificadas, de modo que essas observem as falhas que antes não haviam sido observadas por elas. Esse passo deve ser realizado enquanto existir algum teste que falhe.
4. No passo da geração dos mutantes devem ser escolhidos os operadores de mutação (Ma, 2005).
5. Devem ser executados os testes mutantes com as assertivas executáveis criadas. Novamente o testador deve verificar se os

PUC-Rio - Certificação Digital Nº 1221680/CA

mutantes foram mortos. Caso tenha algum vivo, deve ser analisada qual mutação o código original sofreu e assim verificar se semanticamente o código do mutante é equivalente ao programa original.

6. Segundo a avaliação do testador se isso não ocorrer devem ser atualizados os casos de testes unitários e as assertivas executáveis, para cobrirem aqueles que permanecem vivos. Caso o mutante seja equivalente ao programa original, então se deve considerar o mutante como morto.
7. Com a quantidade de mutantes mortos e a dos que permanece vivos, o testador pode verificar se esses foram mortos por assertivas executáveis. Assim pode ser feita a verificação da eficácia destas, pois por meio dos *log* gerados são verificadas se as assertivas foram capazes de matar os mutantes antes do programa retornar para o controle do teste. Desse modo, as assertivas exercitadas são consideras eficientes.
8. Finalmente o fluxo do modelo termina quando restarem vivos somente os mutantes que o testador tenha julgado como equivalentes.

PUC-Rio - Certificação Digital Nº 1221680/CA

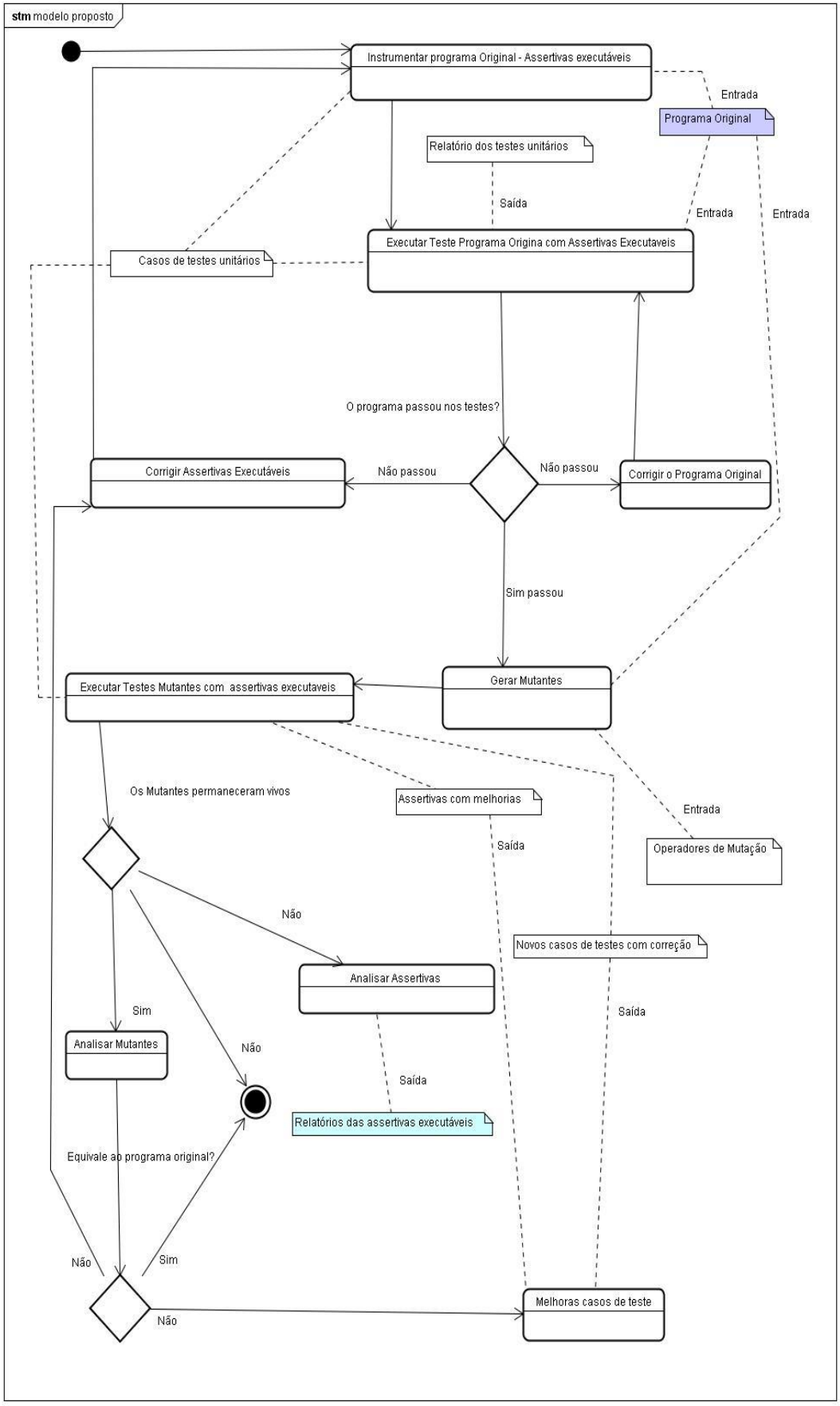


Figura 1 – Modelo de uso de assertivas com teste de análise mutante

PUC-Rio - Certificação Digital Nº 1221680/CA

3.2. Métricas utilizadas

Por meio da observação dos relatórios obtidos pode-se analisar a eficácia do uso das assertivas. Segundo (Delamaro, et al., 2007) um escore de mutação pode ser calculado com uma análise do número de mutantes que permaneceu vivo e morto, sendo que esse escore de mutação deve variar entre 0 e 1. Quanto maior for o valor do escore de mutação, mais adequado será o conjunto de casos de teste para testar o programa, o mesmo é calculado por essa forma:

$$EM(P,T) = \frac{DM(P,T)}{M(P) - EQ(P)}$$

Sendo:

- **EM (P,T):** Escore de mutação de um programa P em relação ao um caso teste T;
- **DM (P,T):** número de mutantes mortos pelo conjunto de casos de testes T;
- **M (P):** número total de mutantes gerados a partir de programa P;
- **EQ (P):** número de mutantes considerados equivalentes a P;

Para determinar a eficácia das assertivas, pode ser calculado o escore de mutação com relação às assertivas executáveis com base no que foi demonstrado no exemplo anterior. Assim, a variável DM, a que representa o número de mutantes mortos pela suíte de teste, deve ser alterado para representar o número de mutantes mortos por meio das assertivas executáveis, como segue abaixo:

$$EM(P,A) = \frac{DM(P,A)}{M(P) - EQ(P)}$$

Sendo:

- **EM (P,A)**: escore de mutação de um programa P em relação a assertivas A;
- **DM (P,A)**: número de mutantes mortos pelas assertivas executáveis;
- **M (P)**: número total de mutantes gerados a partir de programa P;
- **EQ (P)**: número de mutantes considerados equivalentes a P;

PUC-Rio - Certificação Digital Nº 1221680/CA

O escore de mutação também deve variar entre 0 e 1. Quanto maior for o valor do escore de mutação mais eficaz será o conjunto de assertivas executáveis para prevenção e observação de falhas no programa.

Com os dois escores de mutação descritos acima, pode ser feita uma estimativa da eficiência das assertivas executáveis, comparadas com a eficiência dos casos de testes em relação a um grupo de mutantes. Basta analisar qual escore de mutação tem o maior valor, como segue abaixo.

$$EM(P,T)$$

$$EM(P,A)$$

Sendo:

- **EM (P,T)**: Escore de mutação de um programa P em relação ao um caso teste T;
- **EM (P,A)**: Escore de mutação de um programa P em relação a assertivas A;

Por fim, é importante ressaltar que por meio da saída do programa, o testador poderá observar se algum mutante criado teve uma execução interrompida pelo fato de ter entrado em algum estado errôneo, gerando uma exceção que interrompeu o processamento. É de se esperar que a assertiva observe tais erros antes do programa retornar para o controle do teste. Caso isso não ocorra, o controle do teste deverá observar a falha causada pelo mutante. Portanto, nesse caso as assertivas introduzidas não foram eficazes para observar o funcionamento incorreto provocado pelo mutante.

PUC-Rio - Certificação Digital Nº 1221680/CA

3.3. Assertivas executáveis

A proposta de assertivas utilizada neste trabalho é oferecer uma instrumentação para código, que possibilite observar e detectar inconsistências no estado computacional do sistema e prevenir falhas de forma automática. Onde os desenvolvedores poderão rastrear as causas dos defeitos onde foram completa e corretamente localizados por meio da observação de *log*. Assim, as assertivas verificam a integridade de fragmento de códigos em tempo de execução. Elas serão utilizadas como observadoras dos programas em execução. As assertivas poderão ser dos seguintes tipos: pré-condição, pós-

condição e estruturais. Esses tipos são baseados na especificação de design do Design by Contract (Karaorman, Hölzle and Bruno, 1999).

Quando a expressão lógica de uma assertiva não for válida, isto indicará a ocorrência de algum estado incorreto naquele instante. Então, será disparada uma exceção gerando informações do estado do processamento que será incluída nas informações que serão armazenadas no log. Assim, fragmentos do código podem ser instrumentados, oferecendo um suporte para observação de falhas. Essa instrumentação não faz parte da implementação das funções do sistema, então a qualquer momento as assertivas poderão ser desativadas.

3.4. Modelo de criação de assertivas executáveis

Para confecção das assertivas utilizadas no experimento foram realizados os seguintes passos:

1. Estender a classe original que se deseja instrumentar a fim de que sejam herdados todos os seus métodos e atributos, os quais serão utilizados para criação das assertivas executáveis como demonstrado na Figura 2.
2. Na classe filha devem ser inseridas as assertivas executáveis, para que o conjunto de assertivas seja chamado por apenas um método, cria-se um método público que internamente invoca cada assertiva estrutural criada, como apresentado na Figura 2 por meio do diagrama de classe e ainda por meio do exemplo da estrutura do código demonstrado na Figura 3.
3. Ainda devem ser criadas as assertivas pontuais para os métodos da classe pai em que for necessário verificar seus parâmetros de entradas e suas saídas. Na Figura 3 está apresentada a forma em que as assertivas pontuais foram criadas no experimento, testando se a entrada é do tipo requerido. A saída é verificada por meio das assertivas estruturais, caso elas não observem nenhuma inconsistência espera-se que o resultado esteja correto.

PUC-Rio - Certificação Digital Nº 1221680/CA

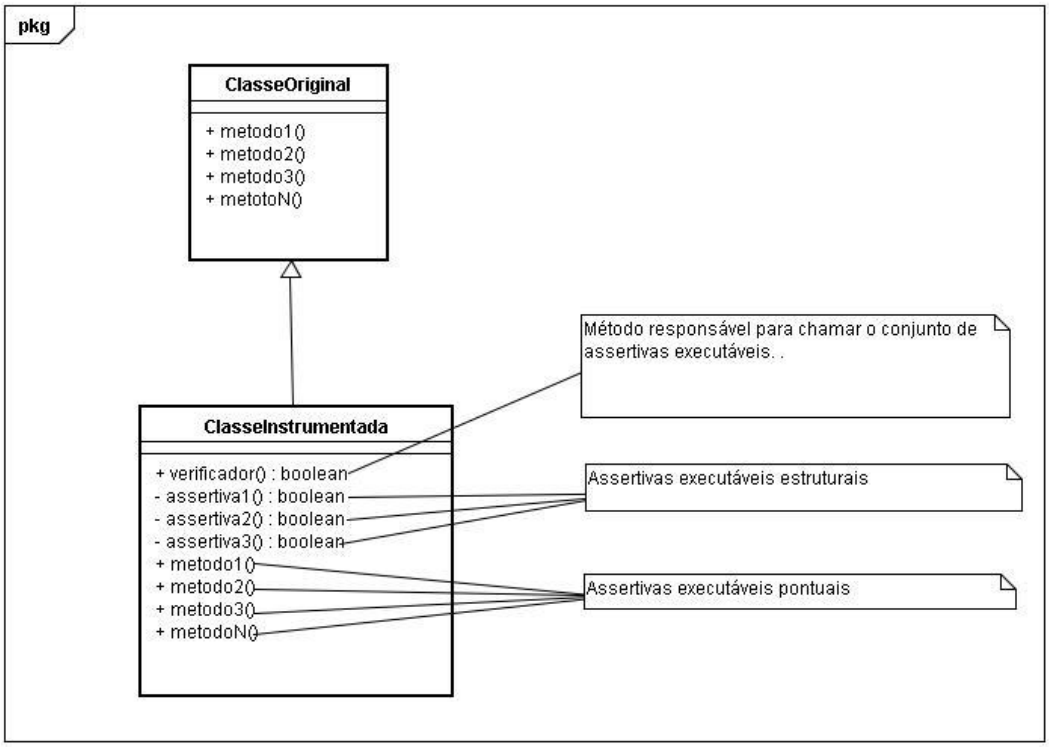


Figura 2 – Diagrama de Classe da instrumentação por meio das assertivas executáveis

PUC-Rio - Certificação Digital Nº 1221680/CA

```
1  class ClasseInstrumentada extends ClasseOriginal{
2
3      //verificador para chamar o conjunto
4      //de assertivas estruturais
5      public boolean verificador(){
6          if (!assertiva1())
7              return false;
8
9          if (!assertiva2())
10             return false;
11
12         if(!assertiva3())
13             return false;
14
15         return true;
16     }
17
18     //assertivas estruturais
19     private boolean assertiva1(){ ... }
20     private boolean assertiva2(){ ... }
21     private boolean assertiva3(){ ... }
22
23     public void metodo1(int a){
24         //assertiva pontual
25         if(a instanceof Integer){
26             super.metodo1(a);
27             verficador();
28         }else print("Entrada não válida")
29     }
30     public void metodo2(boolean b){
31         //assertiva pontual
32         if(b instanceof bollean){
33             verficador();
34             super.metodo2(b);
35         }else print("Entrada não válida")
36     }
37 }
```

Figura 3 – Exemplo da estrutura de instrumentação por meio das assertivas executáveis

Nos exemplos de assertivas estruturais descritas neste trabalho foi usado um misto de linguagens formais e português proposto por Staa (2000). Segue um exemplo de especificação formal leve para uma lista duplamente encadeada:

PUC-Rio - Certificação Digital Nº 1221680/CA

$$\forall\, pElem \in lista: ?\,(pElem \longrightarrow pAnt! = null\,) \implies$$
$$pElem \longrightarrow pAnt \longrightarrow pProx == pElem$$

Para todos pElem **pertencentes a** lista **vale se** pElem->pAnt != null **então** pElem->pAnt->pProx == pElem**.**

PUC-Rio - Certificação Digital Nº 1221680/CA

4
Demonstrações do emprego de assertivas executáveis

Este capítulo tem o objetivo de demonstrar as especificações de algumas estruturas de dados utilizadas no experimento e as instrumentações realizadas com assertivas executáveis para essas estruturas. Na seção 4.1 são descritas algumas informações necessárias para entendimento geral deste capítulo e a demais seções demonstram a criação das assertivas executáveis para as estruturas de dados.

4.1.
Introdução

Foram utilizadas estruturas de dados como amostragem de uma população de programas reais. Serão descritas as propriedades inerentes a cada estrutura de dados e as assertivas executáveis correspondentes às propriedades demonstradas. Assim, para cada estrutura de dados serão apresentados os seguintes tópicos utilizando a nomenclatura abaixo:

1. **P:** Propriedade;
2. **EFL:** Especificação Formal Leve;
3. **AEI:** Assertiva estrutural implementada;
4. **API:** Assertiva pontual implementada;

Em cada seção dentro do capítulo correspondente à estrutura de dados são listados esses tópicos. Em alguns casos a assertiva pontual será omitida na demonstração das estruturas de dados, para que sejam focadas as propriedade inerentes de cada estrutura. Os tópicos 1, 2 e 3 dentro de cada seção correspondem a mesma especificação escrita em níveis diferentes de formalização. Algumas propriedades que são repetidas em determinadas estruturas de dados não serão descritas novamente, apenas será informado que a propriedade já foi demonstrada e também se aplica a referida estrutura de dados. Ainda, em cada seção possui uma subseção que apresenta algumas anomalias na estrutura de dados que as assertivas são capazes de observar.

PUC-Rio - Certificação Digital Nº 1221680/CA

4.2. Árvore de pesquisa binária

Uma árvore é uma estrutura de dados organizada em componentes encadeados chamados de Nós, esses elementos são capazes de armazenar informações relativas aos seus nós vizinhos diretos, além de poderem armazenar dados. Dependendo da posição em que um dado nó está alocado na árvore em relação à posição aos seus vizinhos diretos, permite a esse nó receber uma qualificação particular. Essa segue a seguinte regra: o primeiro nó da árvore é chamado de raiz, caso um dado nó tenha referência a outros nós, esse nó é chamado de pai, os nós referenciados por esses nós são chamados de filhos. Por fim os nós que não tem referências de filhos são denominados folhas.

Uma árvore de pesquisa binária possui propriedades as quais serão utilizadas para criação das assertivas executáveis, pois se a árvore em uso tiver suas propriedades intactas, considera-se que ela se mantém correta e consistente. Assim, se a estrutura da árvore se mantiver correta, também as operações internas da árvore, que são extremamente dependentes da corretude estrutural desta, terão uma base apropriada para realizar suas finalidades. Desta forma, as operações poderão apresentar resultados corretos que reflitam a realidade dos dados armazenados na árvore.

4.2.1. Assertivas estruturais

4.2.1.1. Árvore sem elementos

Para uma árvore qualquer que não possua elementos a raiz deverá ser nula, sendo a raiz o elemento de entrada para acessar a árvore, nenhum elemento da árvore poderá ser acessado, caso a raiz da árvore figure como nula. Assim, certifica-se que a inicialização da árvore esteja correta antes de seu uso. Nas implementações das árvores em que são armazenados para os nós atributos como altura e nível deverão também figurar como nulo, quando árvore não possuir elementos.

P:

Se o número de elementos de uma árvore for igual a zero então a raiz deverá ser nula.

PUC-Rio - Certificação Digital Nº 1221680/CA

EFL:

$$\forall\, n \in \acute{a}rvore : ?\,(totalElements(n) == 0\,)$$
$$\Longrightarrow$$
$$(root == null)$$

AEI:

```
165    private boolean verificadorArvoreSemElementos() {
166        try {
167            if (totalDeNodos() == 0 && root != null)
168                throw new IllegalStructureException();
169        } catch (IllegalStructureException e) {
170            e.printStackTrace();
171            System.err.println("@-> A estrutura está incorreta, raiz:  "
172                    + root.element);
173        }
174
175        return true;
176    }
```

Figura 4 – Assertiva verificadora da árvore sem elementos

4.2.1.2. Árvore com apenas um elemento

A árvore com altura zero deve possuir apenas um elemento o qual figurará como raiz da árvore, e ainda nenhum outro elemento poderá pertencer à árvore. Portanto, a assertiva estrutural deverá assegurar que não existam os nós filhos da direita e esquerda da raiz da árvore.

P:

Se a raiz for diferente de nulo e a altura da raiz for igual a zero então os filhos da esquerda e da direita deverão ser nulos.

EFL:

$$?\,\big((root\,!= null\,)\;\&\&\;(root \longrightarrow height == 0\,)\big)$$
$$\Longrightarrow$$
$$((root \longrightarrow left == null)\;\&\&\;(root \longrightarrow right == null))$$

PUC-Rio - Certificação Digital Nº 1221680/CA

AEI:

```
261    private boolean verificadorArvoreUmElmento() {
262        if (root != null && root.height == 0) {
263            try {
264                if (root.left == null && root.right == null) {
265                    return true;
266                } else {
267                    throw new IllegalStructureException();
268                }
269            } catch (IllegalStructureException e) {
270                e.printStackTrace();
271                System.err.println("@-> A árvore com um elemento possui "
272                        + "filho a esquerda ou o filho a direita da raiz"
273                        + "diferentes de null");
274            }
275        }
276
277        return true;
278
279    }
```

Figura 5 – Assertiva verificadora da árvore com um elemento

4.2.1.3. Árvore com mais de um elemento

P:

Se a raiz é diferente de nulo e a altura da raiz for maior ou igual a um então o ramo da esquerda, direita ou ambos deverão ser diferentes de nulo.

EFL:

$$?\,((root\,!=null)\;\&\&\;(root\longrightarrow height\geq 1\;))$$

$$\Longrightarrow$$

$$((root\longrightarrow left\,!=null)\;||\;(\,root\longrightarrow right!=null))$$

PUC-Rio - Certificação Digital Nº 1221680/CA

AEI:

```
178  private boolean verificadorArvoreMaisDeUmElemento() {
179      if (root != null && root.height >= 1) {
180          try{
181          if ((this.root.left != null || this.root.right != null))
182                  return true;
183              else{
184                  throw new IllegalStructureException();
185              }
186          }catch (IllegalStructureException e) {
187              e.printStackTrace();
188              System.err.println("@-> A árvore com mais de um elemento possui"
189                      + "filho a esquerda ou o filho a direita da raiz"
190                      + "são nulos");
191          }
192      }
193
194      return true;
195  }
```

Figura 6 – Assertiva verificadora da árvore com mais de um elemento

4.2.1.4. Encadeamento da árvore

P:

Para todo elemento *p* pertencente à árvore se o filho esquerdo de *p* for diferente de nulo então o pai do filho esquerdo de *p* deverá ser o próprio *p.* Ainda se o filho direito de *p* for diferente de nulo então o pai do filho direito de *p* deverá ser o próprio *p*.

EFL:

$$\forall\, p \in árvore : ?\,(p \longrightarrow left != null\ ) \implies (p \longrightarrow left \longrightarrow father == p)$$

$$\forall\, p \in árvore : ?\,(p \longrightarrow right != null\ ) \implies (p \longrightarrow right \longrightarrow father == p)$$

AEI:

Para realizar a verificação do encadeamento da árvore é necessário fazer com que o nó filho guarde a referência para seu nó pai. Para isso, é chamado um método auxiliar nessa operação o qual pode ser observado na Figura 8.

PUC-Rio - Certificação Digital Nº 1221680/CA

```
178     private boolean verificadorEncadeamentoArvore(AvlNode t) {
179         if (t != null) {
180             verificadorEncadeamentoArvore(t.left);
181             if (t.left != null) {
182                 try {
183                     if (t.left.father != t) {
184                         throw new IllegalStructureException();
185                     }
186                 } catch (IllegalStructureException e) {
187                     e.printStackTrace();
188                     System.err.println("@-> A estrutura está incorreta  "
189                             + "t.left.father.element: " + t.left.father.element
190                             + "t.element: " + t.element);
191                 }
192             }
193             if (t.right != null) {
194                 try {
195                     if (t.right.father != t) {
196                         throw new IllegalStructureException();
197                     }
198                 } catch (IllegalStructureException e) {
199                     e.printStackTrace();
200                     System.err.println("@-> A estrutura está incorreta  "
201                             + "t.right.father.element: "
202                             + t.right.father.element + "t.element: "
203                             + t.element);
204                 }
205             }
206             verificadorEncadeamentoArvore(t.right);
207         }
208         return true;
209     }
210
```

Figura 7 – Assertiva verificadora do encadeamento da árvore

```
215     private void gerarEncadeamentoDuplicado(AvlNode t, AvlNode x) {
216         if (t != null) {
217             t.father = x;
218             x = t;
219             gerarEncadeamentoDuplicado(t.left, x);
220             gerarEncadeamentoDuplicado(t.right, x);
221         }
222     }
```

Figura 8 – Método para geração de encadeamento duplicado

4.2.1.5. Folhas da árvore

P:

Para todo elemento *p* pertencente à árvore se a altura de *p* for igual a zero e *p* for diferente da raiz então o filho esquerdo de *p* tem que ser diferente de nulo ou o filho direito de *p* tem que ser diferente de nulo.

PUC-Rio - Certificação Digital Nº 1221680/CA

EFL:

$$\forall\, p \in árvore: ?\left((p \longrightarrow height == 0)\ \&\&\ (p\ != root)\right)$$

$$\Longrightarrow$$

$$((p \longrightarrow left\ != null)\ ||\ (p \longrightarrow right != null))$$

AEI:

```
396    private boolean verificarFolhas(AvlNode t) {
397        if (t != null) {
398            verificarFolhas(t.left);
399
400            if (t.height == 0 && t != root) {
401                try {
402                    if (t.left != null || t.right != null)
403                        throw new IllegalStructureException();
404                } catch (IllegalStructureException e) {
405                    e.printStackTrace();
406                    System.err.println("@-> A folha"+t.element+
407                            " possui filho ou filhos");
408                }
409            }
410            verificarFolhas(t.right);
411        }
412        return true;
413    }
```

Figura 9 – Assertiva verificadora das folhas da árvore

4.2.1.6. Árvore de Pesquisa Binária

P:

Para todo elemento *p* pertencente à árvore de pesquisa binária o filho da esquerda deve ser menor que seu pai e o filho da direita deve ser maior que seu pai.

PUC-Rio - Certificação Digital Nº 1221680/CA

EFL:

$$\forall\, n \in árvore:\ (n \longrightarrow left\ ! = null)$$
$$\Longrightarrow$$
$$((n \longrightarrow elemento) > (n \longrightarrow left \longrightarrow elemento\ ))$$

$$\forall n \in árvore: (n \longrightarrow right! = null)$$
$$\Longrightarrow$$
$$((n \longrightarrow elemento) < (n \longrightarrow right \longrightarrow elemento))$$

AEI:

```
317    private boolean verificadorArvoreBinaria(AvlNode t) {
318        if (t != null) {
319            verificadorArvoreBinaria(t.left);
320            if (t.left != null && t.right != null) {
321                try {
322                    if ((((MyInteger) (t.left.element)).intValue()) > (((MyInteger) (t.element))
323                            .intValue())
324                            && (((MyInteger) (t.right.element)).intValue()) < (((MyInteger) (t.element))
325                                    .intValue())) {
326                        throw new IllegalStructureException();
327                    }
328                } catch (IllegalStructureException e) {
329                    e.printStackTrace();
330                    System.err
331                            .println("@ A árvore não é uma árvore de busca binária :"
332                                    + "filho da esquerda: "
333                                    + t.left.element
334                                    + "filho da direita: "
335                                    + t.right.element
336                                    + "elemento pai" + t.element);
337                }
338            }
339            verificadorArvoreBinaria(t.right);
340        }
341        return true;
342    }
```

Figura 10 – Assertiva verificadora da árvore binária

4.2.1.7. Número de filhos de um nó

P:

Para todo elemento *p* pertencente à árvore de pesquisa binária poderá possuir apenas um filho direto esquerdo e um filho direto direito.

PUC-Rio - Certificação Digital Nº 1221680/CA

EFL:

$$\forall\, n \in árvore : ?\,(n \longrightarrow left! = null)$$
$$\Longrightarrow$$
$$((card(n \longrightarrow left) \leq 1)\ \&\&\ (card(n \longrightarrow left) \geq 1))$$

$$\forall\, n \in árvore : ?\,(n \longrightarrow right! = null)$$
$$\Longrightarrow$$
$$((card(n \longrightarrow right) \leq 1)\ \&\&\ (card(n \longrightarrow right) \geq 1))$$

AEI:

```
189    private boolean verificadorNumFilhos(AvlNode t) {
190        if (t != null) {
191            verificadorNumFilhos(t.left);
192            if (t.left != null)
193                t.cont++;
194            if (t.right != null)
195                t.cont++;
196            try {
197                if (t.cont > 2)
198                    throw new IllegalStructureException();
199            } catch (IllegalStructureException e) {
200                e.printStackTrace();
201                System.err.println("@-> A estrutura está incorreta, "
202                        + "o elemento: " + t.element + " número de filhos:"
203                        + t.cont);
204            }
205            verificadorNumFilhos(t.right);
206        }
207        return true;
208    }
```

Figura 11 – Assertiva verificadora do número de filhos de um nó

4.2.2. Assertivas pontuais

Foram utilizadas as assertivas pontuais para possibilitar que cada método chamado tenha uma verificação de sua entrada (pré-condição), garantindo que ela seja do tipo requerido. Ainda a verificação da saída (pós-condição) é feita por meio das assertivas estruturais as quais analisam a estrutura da árvore. Caso ela se mantenha correta então a saída tende a estar correta também. Nos exemplos da Figura 12 pode ser observado que é feita uma verificação se os parâmetros de entrada dos métodos são do tipo esperado por ele. Neste exemplo deverá ser do tipo *Comparable* o qual é o tipo de parâmetro esperado pelos métodos demonstrados, essa verificação constitui a assertiva de entrada. Ainda as assertivas de saída verificam se a estrutura da árvore se mantém consistente, chamando o verificador estrutural.

PUC-Rio - Certificação Digital Nº 1221680/CA

```
76      public void insert(Comparable x) {
77          if (x instanceof Comparable) {
78              super.insert(x);
79               verificador();
80          } else
81              System.out.println("Erro entrada não permitida!");
82      }
83
84      public void remove(Comparable x) {
85          if (x instanceof Comparable) {
86              super.remove(x);
87              verificador();
88          } else
89              System.out.println("Erro entrada não permitida!");
90      }
91
92      public Comparable findMax( ){
93          verificador();
94          return super.findMax();
95      }
96
97      public Comparable find(Comparable x){
98          if (x instanceof Comparable) {
99            verificador();
100           return super.find(x);
101         }else
102             System.out.println("Erro entrada não permitida!");
103         return null;
104     }
```

Figura 12 – Assertivas pontuais para Árvore de Pesquisa Binária

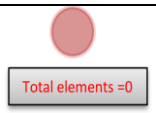
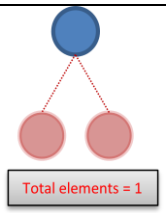
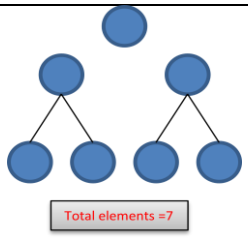
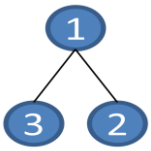
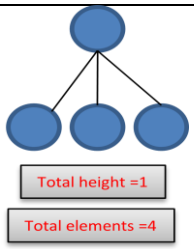
A forma em que as assertivas pontuais executáveis demonstradas na Figura 12 foram confeccionadas também se aplica as demais árvores utilizadas no experimento. Porque, mesmo que os métodos tenham implementações diferentes para as diferentes árvores demonstradas, a maneira de utilizar as assertivas pontuais foi a mesma. Portanto, elas não serão apresentadas novamente nas demais seções.

4.2.3. Possíveis anomalias para Árvore de Pesquisa Binária

As assertivas são capazes de observar anomalias que as estruturas poderão apresentar em uso. Abaixo é demonstrada para cada assertiva utilizada a anomalia que ela é capaz de observar.

PUC-Rio - Certificação Digital Nº 1221680/CA

Tabela 2 – Possíveis tipos de anomalias para Árvore de Pesquisa Binária

Anomalias	Descrição das anomalias	Assertivas exercitadas
 1- Árvore vazia	• A árvore que não teve nenhum elemento inserido, mas mesmo assim, contém um elemento.	• 4.2.1.1- **AEI** (Árvore sem elementos)
 2- Árvore com um elemento	• A árvore com um elemento inserido, porém está apresentando mais de um elemento.	• 4.2.1.2 - **AEI** (Árvore com apenas um elemento)
 3- Nós sem referência	• A árvore não possui referência para todos os elementos inseridos; • A raiz da árvore sem a referência dos nós filhos;	• 4.2.1.3 - **AEI** (Árvore com mais de um elemento) • 4.2.1.4 - **AEI** (Encadeamento da árvore)
 4- Árvore não binária	• A árvore possui elementos dispostos de maneira que fere a propriedade da árvore binária ordenada;	• 4.2.1.6- **AEI** (Árvore de pesquisa binária)
 5- Número de filhos para o nó	• A árvore possui elementos dispostos de maneira que fere a propriedade do número de filhos para cada nó;	• 4.2.1.7- **AEI** (Número de filhos de um) • 4.2.1.4- **AEI** (Encadeamento da árvore)

4.3. Árvore AVL

Em 1962, dois matemáticos russos G. M. Adelson-Velskii e E. M. Landis criaram uma estrutura de árvore binária balanceada os quais denominaram de

PUC-Rio - Certificação Digital Nº 1221680/CA

árvores AVL nome oriundo das iniciais dos seus nomes. Essa estrutura de dados foi criada para as operações de busca e de remoção possuíssem a complexidade $O(\log n)$, garantida porque árvore é balanceada automaticamente. A árvore possui um conjunto de nós ou células interligados dispostos um após o outro sendo que cada nó pode guardar a referência para no máximo dois nós sendo que um nó deve ser o filho da esquerda e outro deve ser o filho a direita. Comumente as implementações da AVL os nós também guardam informações do valor do elemento e altura (Adelson-Velskii, 1962).

A Árvore AVL apresenta todas as assertivas apresentadas para a Árvore Binária no Capítulo 4.1, pois Árvore AVL também é uma Árvore de Pesquisa Binária. Porém existem outras propriedades relativas ao balanceamento que a AVL possui, possibilitando novas assertivas que mantém a estrutura da árvore correta e consistente. Essas são apresentadas abaixo:

PUC-Rio - Certificação Digital Nº 1221680/CA

4.3.1. Assertivas estruturais

4.3.1.1. Balanceamento da árvore

P:

Para toda árvore balanceada a altura do nó à esquerda menos a altura do nó à direita deverá ser menor ou igual a uma unidade.

EFL:

$$\forall\, n \in \acute{a}rvore : ((\mathrm{n} \rightarrow \mathrm{left} \,!= \mathrm{null})\ \&\&\ (\mathrm{n} \rightarrow \mathrm{right}\ !\!= \mathrm{null}))$$

$$\Longrightarrow$$

$$((n \rightarrow left \rightarrow height) - (n \rightarrow right \rightarrow height)) \leq 1$$

AEI:

```
202    private boolean verificadorBalanceamento(AvlNode t) {
203        if (t != null) {
204            verificadorBalanceamento(t.left);
205            if (t.left != null && t.right != null) {
206                try {
207                    if ((t.left.height - t.right.height) > 1) {
208                        throw new IllegalStructureException();
209                    }
210                } catch (IllegalStructureException e) {
211                    e.printStackTrace();
212                    System.err
213                            .println("@-> A árvore não está balanceada"
214                                    +"altura da sub-árvore à esquerda: "
215                                    +t.left.height
216                                    +"altura da sub-árvore à direita: "
217                                    +t.right.height);
218                }
219            }
220            verificadorBalanceamento(t.right);
221        }
222        return true;
223    }
224
```

Figura 13 – Assertiva executável verificadora do balanceamento da árvore

PUC-Rio - Certificação Digital Nº 1221680/CA

4.3.1.2. Altura da árvore

P:

Para toda árvore de pesquisa binária a altura total deve ser menor que $1,44\log(n+2)-1,328$

EFL:

$$heightTotal() < 1.44\log(n+2) - 1.328$$

AEI:

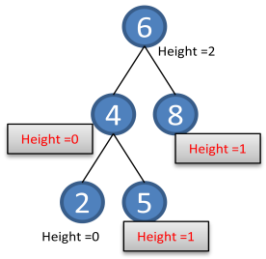
```
251    private boolean verificadorAlturaArvore(int n) {
252        try {
253        if(alturaTotal() > 1.44 * (Math.log(n + 2) / Math.log(2)) - 0.328)
254            throw new IllegalStructureException();
255        }catch (IllegalStructureException e) {
256            e.printStackTrace();
257            System.err.println("@ A altura da árvore está incorreta ");
258        }
259        return true;
260    }
261
```

Figura 14 – Assertiva executável verificadora da altura da árvore

4.3.2. Anomalias

As assertivas executáveis são capazes de observar a anomalia oriunda do balanceamento incorreto da árvore, como pode ser observado na Tabela 3. Ainda a Árvore AVL por ser uma Árvore de Pesquisa Binária é possível aplicar todas as assertivas executáveis descritas na seção 4.1, bem como as anomalias, descritas na Tabela 2, são observadas pelo conjunto de assertivas da Árvore AVL.

Tabela 3 – Possíveis tipos de anomalias para Árvore AVL

Anomalias	Descrição das anomalias	Assertivas exercitadas
 1. Altura incorreta dos elementos	• A árvore possui a altura dos elementos calculada de forma incorreta; • Elementos folhas possuem a altura diferente de zero;	• 4.3.1.1- **AEI** (Árvore balanceada) • 4.2.1.5-**AEI** (Folhas da árvore)

4.4. Árvore Vermelho e Preto

A Árvore Vermelho e Preto é uma árvore de pesquisa binária que possuiu um atributo adicional que é a cor do nó, o qual pode figurar como vermelho ou preto. Por meio da forma como os nós são coloridos na árvore a propriedade do balanceamento é mantida. Assim as assertivas executáveis demonstradas nesta seção garantem a correta disposição dos elementos na árvore segundo sua cor, a fim que árvore seja balanceada corretamente.

4.4.1. Assertivas estruturais

4.4.1.1. Cor da raiz da árvore

P:

Para todo elemento pertencente à árvore Vermelho e Preto vale se um dado elemento *n* for a raiz da árvore então a cor de *n* deverá ser preto.

PUC-Rio - Certificação Digital Nº 1221680/CA

$$\forall\, n \in árvoreVermelhoPreto : ?\, (n == root\,)$$

$$\Longrightarrow$$

$$(\, n \rightarrow color == black)$$

AEI:

```
142    public boolean verficadorCorRaiz() {
143        try {
144            if (header.color != 1) {
145                throw new IllegalStructureException();
146            }
147        } catch (IllegalStructureException e) {
148            e.printStackTrace();
149            System.err.println("@ A cor da raiz está incorreta: "
150                    + header.element + "cor: " + header.color);
151        }
152        return true;
153    }
```

Figura 15 – Assertiva executável verificadora da cor da raiz

4.4.1.2. Cor dos elementos da árvore

P:

Para todo elemento pertencente à árvore Vermelho e Preto vale se um dado elemento *n* for diferente de nulo então a cor de *n* deverá ser igual a preto ou a cor de *n* deverá ser igual a vermelho.

EFL:

$$\forall\, n \in árvoreVermelhoPreto : ?\, (n\,!\,= null)$$

$$\Longrightarrow$$

$$((n \rightarrow color == black)\;||\;(n \rightarrow color == red))$$

PUC-Rio - Certificação Digital Nº 1221680/CA

AEI:

```
125    public boolean verificadorCor(RedBlackNode t) {
126        if (t != nullNode) {
127            verificadorCor(t.left);
128            try {
129                if (t.color != 0 && t.color != 1) {
130                    throw new IllegalStructureException();
131                }
132            } catch (IllegalStructureException e) {
133                e.printStackTrace();
134                System.err.println("@ A cor do elemento está incorreta: "
135                        + t.element + "cor: " + t.color);
136            }
137            verificadorCor(t.right);
138        }
139
140        return true;
141    }
```

Figura 16 – Assertiva executável verificadora da cor dos elementos

4.4.1.3. Cor das folhas

P:

Para todo elemento pertencente à árvore Vermelho e Preto vale se um dado elemento *n* for uma folha então a cor de *n* deverá ser igual a preto.

EFL:

$$\forall\, \boldsymbol{n} \in \acute{a}\boldsymbol{rvoreVermelhoPreto} : ?\,(\boldsymbol{n} == \boldsymbol{leaf})$$

$$\Longrightarrow$$

$$(n \rightarrow color == black)$$

PUC-Rio - Certificação Digital Nº 1221680/CA

AEI:

```
237   public boolean corFolha(RedBlackNode t) {
238       if (t != nullNode) {
239           corFolha(t.left);
240           if (t.left == nullNode) {
241               try {
242                   if (t.left.color != 1)
243                       throw new IllegalStructureException();
244               } catch (IllegalStructureException e) {
245                   e.printStackTrace();
246                   System.err.println("@ A cor do elemento está incorreta: "
247                           + t.left.element + "cor: " + t.left.color);
248               }
249           }
250           if (t.right == nullNode) {
251               try {
252                   if (t.right.color != 1)
253                       throw new IllegalStructureException();
254               } catch (IllegalStructureException e) {
255                   e.printStackTrace();
256                   System.err.println("@ A cor do elemento está incorreta: "
257                           + t.left.element + "cor: " + t.left.color);
258               }
259               corFolha(t.right);
260           }
261       }
262       return true;
263   }
```

Figura 17 – Assertiva executável verificadora da cor das folhas

4.4.1.4. Cor dos filhos de um nó vermelho

P:

Para todo elemento pertencente à árvore Vermelho e Preto vale se a cor e de *n* for igual a vermelho então o filho esquerdo de *n* e o filho direito de *n* ambos deverão ser de cor preta.

EFL:

$$\forall\, n \in árvoreVermelhoPreto : ?\,(n \rightarrow color == red)$$

$$\Longrightarrow$$

$$((\mathrm{n} \rightarrow \mathrm{left} \rightarrow \mathrm{color} == \mathrm{black})\ \&\&\ (n \rightarrow right \rightarrow color == black))$$

PUC-Rio - Certificação Digital Nº 1221680/CA

AEI:

```
155    public boolean verificadorCorEstrutura(RedBlackNode t) {
156        if (t != nullNode) {
157            verificadorCorEstrutura(t.left);
158            if (t.color == 0) {
159                if (t.right != nullNode) {
160                    try {
161                        if (t.right.color != 1) {
162                            throw new IllegalStructureException();
163                        }
164                    } catch (IllegalStructureException e) {
165                        e.printStackTrace();
166                        System.err.println("@ A cor do elemento está incorreta: "
167                                + t.right.element + "cor: "
168                                + t.right.color);
169                    }
170                }
171                if (t.left != nullNode) {
172                    try {
173                        if (t.left.color != 1) {
174                            throw new IllegalStructureException();
175                        }
176                    } catch (IllegalStructureException e) {
177                        e.printStackTrace();
178                        System.err.println("@ A cor do elemento está incorreta: "
179                                + t.left.element + "cor: "
180                                + t.left.color);
181                    }
182                }
183            }
184            verificadorCorEstrutura(t.right);
185        }
186        return true;
187    }
```

Figura 18 – Assertiva executável verificadora da cor dos nós filhos

4.4.1.5. Quantidade de elementos pretos no caminho

P:

Para cada nó, todos os caminhos desde nó até as folhas descendentes contêm o mesmo número de nós pretos.

Essa regra assegura uma propriedade crítica da árvore Vermelho e Preto que o caminho mais longo da raiz a qualquer folha não seja maior do que duas vezes o caminho mais curto da raiz a qualquer outra folha naquela árvore. Essa propriedade garante o balanceamento da Árvore Vermelho e Preto.

PUC-Rio - Certificação Digital Nº 1221680/CA

EFL:

$$\forall\, r, s, t, u \in árvoreVermelhoPreto :?$$
$$\Big(\big((B(r,t) > 0\,)\&\&(B(s,t) > 0\,\big)\,\&\&(t == leaf)\&\&\,(u! = leaf)\big)\Big)$$
$$\Rightarrow$$
$$(B(r,t) == B(s,t)\;\&\&(B(r,t)! = B(r,u)\;\&\&(B(s,t)! = B(s,u)\,)$$

Sendo $B(x,y)$ o número de elementos de cor preta no caminho entre os nós x, y.

AEI:

Para a implementação dessa assertiva foi necessário criar três métodos adicionais para que seja feita uma busca em profundidade para cada nó da árvore a fim de serem contados os nós pretos de todos os caminhos de cada nó até ao nó folha. O primeiro método "*inicilazaNo"* demonstrado na Figura 19 inicializa para cada nó da árvore o atributo *isVisitado* como *false.* O segundo método "*chamaDFS*" demonstrado na Figura 21 chama a busca em profundidade para os nós pelos quais não foi realizada a contagem dos nós pretos do caminho. O terceiro método "*buscaEmProfundidade"* demonstrado na Figura 22 conta efetivamente todos os nós pretos do caminho de um dado nó até a folha.

```
124    private boolean verificaQuantidadePreto(RedBlackNode t) {
125        ArrayList<Integer> list = new ArrayList<Integer>();
126        inicializaNo(header.right);
127        chamaDSF(t, list);
128        int valor = 0;
129        valor = list.get(0);
130
131        for (int i = 0; i < list.size(); i++) {
132            try {
133                if (valor != list.get(i)) {
134                    throw new IllegalStructureException();
135                }
136            } catch (IllegalStructureException e) {
137                e.printStackTrace();
138                System.err.println("@ O caminho a partir do elemento: "
139                        + t.element
140                        + " está incorreto quanto ao número de nós pretos");
141            }
142        }
143        return true;
144    }
```

Figura 19 – Assertiva executável verificadora da quantidade de nós pretos

PUC-Rio - Certificação Digital Nº 1221680/CA

```
94     public void inicializaNo(RedBlackNode t) {
95         if (t != nullNode) {
96             inicializaNo(t.left);
97             t.isVistado = false;
98             inicializaNo(t.right);
99         }
100
101    }
102
```

Figura 20 – Método para inicialização dos nós da árvore

```
111    private ArrayList chamaDSF(RedBlackNode t, ArrayList<Integer> list) {
112        if (t != nullNode) {
113            chamaDSF(t.left, list);
114            if (t.isVistado == false) {
115                buscaEmProfundidade(header.right);
116                list.add(cont);
117                cont = 0;
118            }
119            chamaDSF(t.right, list);
120        }
121        return list;
122    }
123
```

Figura 21 – Método para controlar os nós não visitados

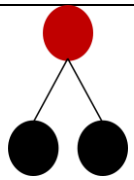
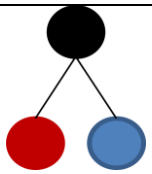
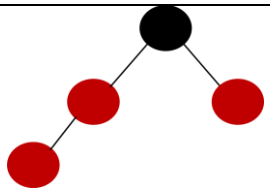
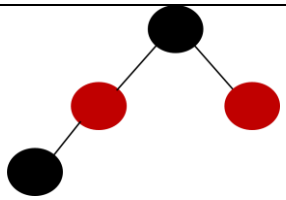
```
223    private void buscaEmProfundidade(RedBlackNode t) {
224        if (t != nullNode) {
225            if (t.right.isVistado == true) {
226                buscaEmProfundidade(t.left);
227            }
228            if (t.right.isVistado == true &&
229                    t.left.isVistado == true) {
230                t.isVistado = true;
231            }
232            if (t.color == 1) {
233                cont++;
234            }
235            if (t.right == nullNode && t.left == nullNode) {
236                t.isVistado = true;
237            }
238            if (t.right.isVistado == false) {
239                buscaEmProfundidade(t.right);
240            }
241
242        }
```

Figura 22 – Método para contar os nós pretos encontrados

PUC-Rio - Certificação Digital Nº 1221680/CA

4.4.2. Possíveis anomalias para Árvore Vermelho e Preto

Tabela 4 – Possíveis tipos de anomalias para Árvore Vermelho e Preto

Anomalias	Descrição das anomalias	Assertivas exercitadas
 1- Raiz de cor vermelha	• Caso a raiz da árvore não tenha sido inicializada com a cor preta.	• 4.4.1.1- **AEI** (Cor da raiz da árvore)
 2- Nó que não seja preto nem vermelho	• Caso algum elemento da árvore não tenha sido inicializado com a cor vermelho ou preta. Neste caso a assertiva 5.4.1.4 não poderia observar, pois o pai tem a cor preta e não vermelha.	• 4.4.1.2-**AEI** (Cor dos elementos da árvore)
 3- Filho de um nó vermelho que também seja vermelho	• Caso algum filho de um nó de cor vermelha tenha um filho que também seja de cor vermelha.	• 4.4.1.4-**AEI** (Cor dos filhos de um nó vermelho) • 4.4.1.3-**AEI** (Cor das folhas)
 4- Somatório de nós pretos no caminho	• A árvore possui elementos dispostos de maneira que fere a propriedade de que cada nó, todos os caminhos desde um nó até as folhas descendentes contêm o mesmo número de nós pretos;	• 4.4.1.5 -**AEI** (Quantidade de elementos pretos no caminho)

PUC-Rio - Certificação Digital Nº 1221680/CA

4.5. Árvore AA

O nome da Árvore de AA é oriundo das iniciais do seu inventor Arne Anderson é uma forma de árvore balanceada com uma implementação mais simples comparada a Árvore Vermelho e Preto, porém com desempenho próximo de serem similares. A facilidade de implementação é porque ela possui apenas duas formas chamadas *Skew* e *Split* para árvore manter a propriedade do balanceamento, menor que o número da Árvore Vermelho e Preto que considerada sete formas diferentes (Andersson, 1993).

A Árvore AA apresenta todas as assertivas apresentadas para a Árvore Binária no Capítulo 4.1, pois a Árvore AA também é uma Árvore de Pesquisa Binária. Porém existem outras propriedades relativas ao balanceamento que a Árvore AA possui que necessita de novas assertivas executáveis para manter a estrutura da árvore correta e consistente. Essas são apresentadas abaixo:

4.5.1. Assertivas estruturais

4.5.1.1. Nível do filho à esquerda de um nó

P:

Para todo elemento pertencente à Árvore AA vale se o filho esquerdo de *n* for diferente de nulo então, o nível do filho esquerdo de *n* menos uma unidade deverá ser igual ao nível de *n*.

EFL:

$$\forall\, n \in \acute{a}rvore\ AA : ?\,(n \longrightarrow left\ != null)$$

$$\Longrightarrow$$

$$\big((n \longrightarrow left \longrightarrow level - 1) == (n \longrightarrow level)\big)$$

PUC-Rio - Certificação Digital Nº 1221680/CA

AEI:

```
85    public boolean verificaNivelFilhoEsquerdo(AANode t) {
86
87        if (t != t.left) {
88            verificaNivelFilhoEsquerdo(t.left);
89            if (t.left != nullNode) {
90                try {
91                    if ((t.left.level + 1) != t.level) {
92                        throw new IllegalStructureException();
93                    }
94                } catch (IllegalStructureException e) {
95                    e.printStackTrace();
96                    System.err.println("@->Filho a esquerda"
97                            + t.left.element.toString() + " nivel: "
98                            + t.left.level + " Pai: " + t.element.toString()
99                            + " nivel: " + t.level);
100               }
101           }
102           verificaNivelFilhoEsquerdo(t.right);
103       }
104       return true;
105   }
```

Figura 23 – Assertiva executável verificadora do nível do filho à esquerda

4.5.1.2. Nível do filho à direita de um nó

P:

Para todo elemento pertencente à Árvore AA vale se o filho direito de *n* for diferente de nulo então, o nível do filho direito de *n* menos uma unidade deverá ser igual ao nível de *n* ou o filho direito de *n* deverá ser igual ao nível de *n*.

EFL:

$$\forall \text{ n} \in \text{árvore AA}: ?\,(\text{n} \longrightarrow \text{right} \,!= \text{null})$$

$$\Longrightarrow$$

$$\big((\text{n} \longrightarrow \text{right} \longrightarrow \text{level} - 1) == (\text{n} \longrightarrow \text{level})\big) \,||$$

$$((n \longrightarrow right \longrightarrow level) == (n \longrightarrow level))$$

PUC-Rio - Certificação Digital Nº 1221680/CA

AEI:

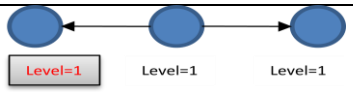
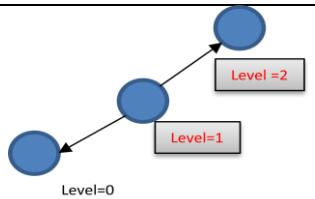
```
107    public boolean verificaNivelFilhoDireito(AANode t) {
108        if (t != t.left) {
109            verificaNivelFilhoDireito(t.left);
110            if (t.right != nullNode) {
111                try {
112                    if (((t.right.level + 1) != t.level)
113                            && ((t.right.level) != t.level)) {
114                        throw new IllegalStructureException();
115                    }
116                } catch (IllegalStructureException e) {
117                    e.printStackTrace();
118                    System.err.println("@->Filho da direita: "
119                            + t.right.element.toString() + " nivel: "
120                            + t.right.level + " Pai: " + t.element.toString()
121                            + " nivel: " + t.level);
122                }
123            }
124            verificaNivelFilhoDireito(t.right);
125        }
126        return true;
127    }
128
```

Figura 24 – Assertiva executável verificadora do nível do filho à direita

4.5.2. Possíveis anomalias para Árvore AA

As assertivas executáveis apresentadas são capazes de observar a anomalia oriunda do balanceamento incorreto da árvore, como pode ser observado na Tabela 5. Ainda para Árvore AA podem ser aplicadas todas as assertivas executáveis descritas na seção 4.1, bem como as anomalias descritas na Tabela 2 são observadas pelo conjunto de assertivas da Árvore AA.

Tabela 5 – Possíveis tipos de anomalias para Árvore AA

Anomalias	Descrição das anomalias	Assertivas exercitadas
 1- Nível do filho à esquerda	• A árvore possui uma subárvore à esquerda com um nó de mesmo nível do nó pai.	• 4.5.1.1 - **AEI** (Nível do filho à esquerda de um nó)
 2- Nível do filho à direita	• A árvore possui a subárvore à direita um nó com um nível acima do nó pai.	• 4.5.1.2 - **AEI** (Nível do filho à direita de um nó)

PUC-Rio - Certificação Digital Nº 1221680/CA

4.6. Árvore Splay

A Árvore Splay é uma árvore binária auto-ajustável que não usa regras explícitas para forçar o balanceamento como as demais árvores apresentadas. Segundo Sleator e Tarjan (1985) as árvores balanceadas não são tão eficientes quando os nodos da árvore são acessados sequencialmente. Já a Àrvore Splay ajusta a sua estrutura à frequência de acesso aos dados. Assim ela tenta minimizar o número de operações na sua estrutura colocando os nós mais frequentemente acessados mais perto da raiz, o que faz com que eles sejam acessados mais rapidamente. Aplica-se uma operação de mover para a raiz, chamada de *splaying* a cada acesso, para manter essa propriedade é utilizada a assertiva executável abaixo demonstrada (seção 4.6.1).

4.6.1. Assertivas estruturais

4.6.1.1. Elemento da raiz

P:

Para todo elemento pertencente à Árvore Splay depois dele ser encontrado deverá figurar como raiz da árvore.

EFL:

$$\forall\, n \in árvoreSplay : ?\,(find(n) == True)$$

$$\Longrightarrow$$

$$(root == n)$$

PUC-Rio - Certificação Digital Nº 1221680/CA

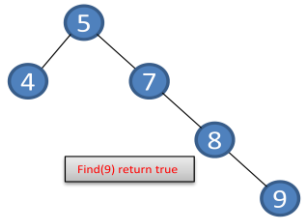
AEI:

```
60  public boolean verificaRaizArvorePorElemento(Comparable x ) {
61
62          if (x != null ) {
63              find(x);
64              try{
65              if (root.element.compareTo(x) != 0){
66                  throw new IllegalStructureException();
67              }
68              }catch (IllegalStructureException e) {
69                  e.printStackTrace();
70                  System.err
71                          .println("@ elemento: " + x + "não é a raiza da árvore. "
72                                  + "Raiz: " + root.element);
73              }
74
75          }
76      return false;
77
78  }
79
```

Figura 35 – Assertiva executável verificadora do ajuste da Árvore Splay

4.6.2. Possíveis anomalias para Árvore Splay

Tabela 6 – Possíveis tipos de anomalias para Árvore Splay

Anomalias	Descrição das anomalias	Assertivas exercitadas
 1- Elemento encontrado não figura na raiz da árvore	• Caso o elemento que tenha sido pesquisado não ser movido para a raiz da árvore;	• 4.6.1.1- **AEI** (Elemento da raiz)

4.7. Árvore B

A Árvore B é uma árvore de pesquisa binária e balanceada que ao contrário das demais, árvores de pesquisa binária demonstradas neste trabalho, pode ter nós que tenham muitos filhos com número variável. Segundo Cormen (2002) o fator de ramificação de uma Árvore B pode ser muito grande, muito maior do que a Árvore Vermelho e Preto, e com a altura consideravelmente menor do que elas. As Árvores B podem ser utilizadas para trabalhar com dispositivos de armazenamento secundário como discos magnéticos bem como

PUC-Rio - Certificação Digital Nº 1221680/CA

para implementar muitas operações sobre conjuntos dinâmicos em tempo $O(\lg n)$.

4.7.1. Assertivas estruturais

4.7.1.1. Limite superior sobre o número de filhos de um nó

P:

Para nó pertencente à Árvore B pode conter no máximo *2t - 1* chaves e o nó interno no máximo *2t* chaves

EFL:

$$\forall\, n \in árvoreB :$$

$$((n \longrightarrow \text{Keys} \longrightarrow \text{lenght} < 2 * t - 1)\ \&\&\ (n \longrightarrow \text{child} \longrightarrow \text{lenght} < 2 * t))$$

AEI:

```
126    public boolean verificaLimiteSuperior(Node node) {
127        boolean correto = true;
128        try {
129            if (node.mKeys.length > 2 * T - 1
130                    && node.mObjects.length > 2 * T - 1) {
131                throw new IllegalStructureException();
132            }
133        } catch (IllegalStructureException e) {
134            e.printStackTrace();
135            System.err.println("@ O limite superior está incorreto"
136                    + "para mKeys");
137        }
138        try {
139            if (node.mChildNodes.length > 2 * T) {
140                throw new IllegalStructureException();
141            }
142        } catch (IllegalStructureException e) {
143            e.printStackTrace();
144            System.err.println("@ O limite superior está incorreto"
145                    + "para mChildNodes");
146        }
147
148        return correto;
149    }
150
151 }
```

Figura 26 – Assertiva verificadora do limite superior

PUC-Rio - Certificação Digital Nº 1221680/CA

4.7.1.2. Nós internos e folhas

P:

Para todo elemento pertencente à árvore B se o nó é uma folha então ele não poderá ser um nó interno.

EFL:

$$\forall\, n \in árvoreB :?\, (n == leaf) \Longrightarrow (\text{node} \rightarrow \text{mIsLeafNode} == \text{true})$$

$$\forall\, n \in árvoreB :?\, (n == childNode) \Longrightarrow (\text{node} \rightarrow \text{mIsLeafNode} == \text{false})$$

AEI:

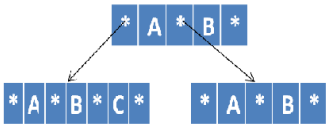
```
66    public boolean verificaNoFolhaNoInterno(Node node) {
67        boolean correto = true;
68        if (node != null) {
69            if (node.mIsLeafNode) {
70                for (int i = 0; i < node.mNumKeys; i++) {
71                    if (!node.mIsLeafNode)
72                        correto = false;
73                }
74            } else {
75                int i;
76                for (i = 0; i < node.mNumKeys; i++) {
77                    verificaNoFolhaNoInterno(node.mChildNodes[i]);
78                    if (node.mIsLeafNode) {
79                        correto = false;
80                    }
81
82                }
83                verificaNoFolhaNoInterno(node.mChildNodes[i]);
84            }
85        }
86
87        return correto;
88    }
```

Figura 27 – Assertiva executável verificadora das chaves da Árvore B

PUC-Rio - Certificação Digital Nº 1221680/CA

4.7.2. Possíveis anomalias para Árvore B

Tabela 7 – Possível tipo de anomalia para Árvore B

Anomalias	Descrição das anomalias	Assertivas exercitadas
 1- Elemento encontrado não figura na raiz da árvore	• A capacidade que o nó pode ter de filhos foi estourada no filho da esquerda;	• 4.7.1.1- **AEI** (Limite superior sobre o número de filhos de um)

4.8. Heap binária

Uma Heap Binária e uma estrutura de prioridades na forma de árvore binária balanceada. Inicialmente concebida por Vuillemin onde apresentou um algoritmo para a manipulação de filas de prioridade, em uma estrutura de dados que associa cada valor contido em um conjunto a uma chave (Vuillemin, 1978). A Heap Binária deve ser uma árvore que todos os níveis devem estar completos e o último nível deve armazenar os nós na ordem da esquerda para a direita.

Ainda ela poderá ser uma Heap Máximo onde o pai deverá ser sempre maior que os filhos e Heap Mínimo onde os filhos são maiores que o pai. Essas propriedades são descritas na forma de assertiva para a heap binária e deve se manter consistente em todo tempo de uso dessa estrutura.

4.8.1. Assertivas estruturais

4.8.1.1. Filhos maiores que os pais

P:

Para todo nó pertencente à Heap Binária os filhos de um dado nó devem ser maiores que esse nó na Heap Mínimo.

PUC-Rio - Certificação Digital Nº 1221680/CA

EFL:

$$\forall\, n \in heapBinária : ?\,(n \longrightarrow \text{children} \,!= null)$$
$$\Longrightarrow$$
$$(n \longrightarrow \text{children} \longrightarrow elements) < (n \longrightarrow elements)$$

AEI:

```
45    private boolean verificaFilhosMaioresQPai() {
46        int n = totalElementos();
47        for (int x = 1; x <= n; x++) {
48            if (array[(x * 2) + 1] != null) {
49                            try {
50                    if (array[x].compareTo(array[(x * 2) + 1]) > 0) {
51                        throw new IllegalStructureException();
52                    }
53                } catch (IllegalStructureException e) {
54                    e.printStackTrace();
55                    System.err.println("@-> A estrutura está incorreta +"
56                            + "filho:"+ array[x] + "pai: "+array[(x * 2) + 1]);
57                }
58            }
59            if (array[(x * 2) + 2] != null) {
60                try {
61                    if (array[x].compareTo(array[(x * 2) + 2]) > 0) {
62                        throw new IllegalStructureException();
63                    }
64                } catch (IllegalStructureException e) {
65                    e.printStackTrace();
66                    System.err.println("@-> A estrutura está incorreta +"
67                            + "filho:"+ array[x] + "pai: "+array[(x * 2) + 1]);
68                }
69            }
70        }
71
72        return true;
73    }
```

Figura 28 – Assertiva executável verificadora da Heap Mínimo

4.8.1.2. Heap Binária completa

P:

Para todo elemento pertencente à Heap Binária se o filho direito for diferente de nulo o filho esquerdo obrigatoriamente deverá ser diferente de nulo.

PUC-Rio - Certificação Digital Nº 1221680/CA

EFL:

$$\forall\, n \in \mathit{HeapBinária} : ?\,(n \longrightarrow \text{Right} \,!= \mathit{null})$$
$$\Longrightarrow$$
$$(n \longrightarrow \text{left} \,!= \mathit{null})$$

AEI:

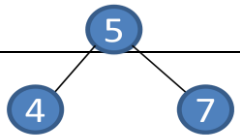
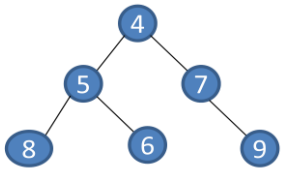
```
79    public boolean verificaHeapCompleta() {
80
81        for (int x = 1; x < totalElementos(); x++) {
82           if (array[(x * 2) + 1] == null && array[(x * 2) + 2] != null)
83                return false;
84        }
85        return true;
86    }
87
```

Figura 29 – Assertiva executável verificadora da alocação dos nós na Heap

PUC-Rio - Certificação Digital Nº 1221680/CA

4.8.2. Possíveis anomalias para Heap Binária

Tabela 8 – Possíveis tipos de anomalias para Heap Binária

Anomalias	Descrição das anomalias	Assertivas exercitadas
 1- Heap Mínimo	• O filho esquerdo da raiz está com um valor menor que a raiz;	• 4.8.1.1- **AEI** (Filhos maiores que o pai)
 2- Heap completa	• A Heap não está completa, pois o elemento de valor sete possui o filho da direita sem ter o da esquerda;	• 4.8.1.2- **AEI** (Heap Binária completa)

4.9. Heap Leftist

A Leftist Heap é uma heap binária em que, diferente das heaps mais comuns, a árvore não necessariamente é completa. Ela tem a propriedade de ser mais profunda do lado esquerdo, ou seja, ela tem subárvores esquerdas maiores do que às direitas. Para essa heap também se aplica a assertiva executável descrita na seção 5.8.1.1 Filhos maiores que os pais e não se aplica da seção 5.8.1.2, pois a leftist heap não é completa. Porém para manter sua característica que qualquer subárvore esquerda seja maior que a subárvore direita é apresentada ainda a assertiva executável do Caminho nulo.

4.9.1. Assertivas estruturais

4.9.1.1. Caminho nulo da Heap Leftist

P:

Caminho nulo da subárvore esquerda tem que ser maior ou igual ao caminho nulo da subárvore direita

Sendo o caminho nulo de uma árvore definido como o tamanho do menor caminho entre a raiz da árvore e um nó externo.

EFL:

$$\forall\, n \in leftist$$
$$\forall\, s \in subárvore\ \ a\ esquerda\ de\ n$$
$$\forall\, t \in subárvore\ \ a\ direita\ de\ n : ?\ (n == root)$$
$$(caminhoNulo(s) < caminhoNulo(t))$$

PUC-Rio - Certificação Digital Nº 1221680/CA

AEI:

```
71 public boolean verificaCaminhoNulo() {
72     try {
73         if (caminhoEsquerdo() < caminhoDireito()) {
74             throw new IllegalStructureException();
75         }
76     } catch (IllegalStructureException e) {
77         e.printStackTrace();
78         System.err.println("@-> O caminho nulo da árvore "
79                 + "a esquerda é menor que o caminho da árvore"
80                 + " a direita");
81     }
82     return true;
83 }
```

```
104 private int caminhoEsquerdo() {
105     LeftHeapNode t = root;
106     int menor = t.npl;
107     while (t != null) {
108
109         if (t.npl < menor)
110             menor = t.npl;
111
112         t = t.left;
113     }
114     return menor;
115 }
```

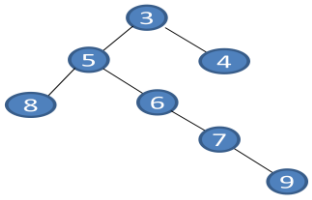
```
117 private int caminhoDireito() {
118     LeftHeapNode t = root;
119     int menor = t.npl;
120     while (t != null) {
121         if (t.npl < menor)
122             menor = t.npl;
123
124         t = t.right;
125     }
126     return menor;
127 }
```

Figura 30 – Assertiva executável verificadora do caminho nulo e métodos auxiliares

PUC-Rio - Certificação Digital Nº 1221680/CA

4.9.2. Possíveis anomalias para Heap Leftist

Tabela 9 – Possíveis tipos de anomalias para Heap Leftist

Anomalias	Descrição das anomalias	Assertivas exercitadas
 1- Subárvores da Heap Leftist	• A subárvore direita é maior do que subárvore esquerda para o nó de valor cinco	• 4.9.1.1- **AEI** (Caminho nulo da Heap Leftist)

4.10. Heap Fibonacci

A Heap Fibonacci é uma coleção de árvores que mantém a propriedade descrita para Heap Binária onde os filhos devem ser maiores que os pais. Ainda essa heap mantém seus nós organizados em um encadeamento duplo, onde o pai tem referência para seus filhos diretos e filho tem referência para seu pai. Além da assertiva para garantir que os filhos sejam maiores que os pais, a assertiva executável para garantir a propriedade do encadeamento da Heap Fibonacci é apresentada nesta seção.

4.10.1. Assertivas estruturais:

4.10.1.1. Encadeamento da Heap Fibonacci

P:

Para todo elemento *p* pertencente à Heap Fibonacci se o próximo elemento de *p* for diferente de nulo então o elemento anterior do próximo elemento de *p* deverá ser o próprio *p*.

EFL:

$$\forall\, p \in Heap\ Fibonacce : ?\,(p \longrightarrow prev ! = null\ )$$

$$\Longrightarrow$$

$$(p \longrightarrow next \longrightarrow prev == p)$$

AEI:

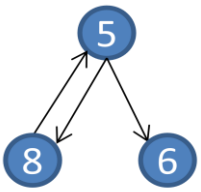
```
53  public boolean verificaListaEncadeada() {
54
55      Entry temp = mMin;
56      while (temp != null) {
57          temp.mIsMarked = true;
58          if (temp.mPrev != null) {
59              try {
60                  if (temp != temp.mNext.mPrev) {
61                      throw new IllegalStructureException();
62                  }
63              } catch (IllegalStructureException e) {
64                  e.printStackTrace();
65                  System.err.println("@-> A estrutura está incorreta  "
66                          + "temp.mNext.mPrev.mElement: " + temp.mNext.mPrev
67                          + "temp.mElement: " + temp.mElem);
68              }
69          }
70          if (temp.mNext.mIsMarked == true) {
71              temp = null;
72          } else {
73              temp = temp.mNext;
74          }
75      }
76      return false;
77  }
```

Figura 31 – Assertiva executável verificadora do encadeamento da Heap Fibonacci

PUC-Rio - Certificação Digital Nº 1221680/CA

4.10.2. Possíveis anomalias para Heap Fibonacci

Tabela 10 – Possíveis tipos de anomalias para Heap Fibonacci

Anomalias	Descrição das anomalias	Assertivas exercitadas
 1- Encadeamento da Heap Fibonacci	• O elemento de valor seis não tem a referência do seu nó pai de valor cinco	• 4.9.1.1- **AEI** (Caminho nulo da Heap Leftist)

4.11. Assertivas executáveis para Árvore AVL implementada em PHP

As assertivas criadas para árvore AVL na linguagem JAVA foram traduzidas para linguagem PHP. Uso de uma segunda linguagem se dá para demonstrar que instrumentar programas com assertivas pode ser utilizado para demais linguagens não apenas em programas escritos em JAVA e ainda se as assertivas são eficazes naquela linguagem.

Foi utilizada para essa demonstração apenas a árvore AVL as demais assertivas executáveis criadas para as estruturas de dados anteriormente apresentadas em Java não foram traduzidas para PHP. Serão descritas para essa seção apenas o exemplo de implementação na linguagem PHP, pois a descrição de cada assertiva criada foi demonstrada na seção 4.2.1.

4.11.1. Árvore sem elemento

```
53      public function verificadorArvoreSemElementos() {
54          if ($this->root != null && $this->size == 0) {
55              throw new Exception('A árvore com sem elementos está incorreta');
56          }
57          return true;
58      }
59
```

Figura 32 – Assertiva executável verificadora da árvore sem elementos

PUC-Rio - Certificação Digital Nº 1221680/CA

4.11.2. Árvore com apenas um elemento

```
40  private function verificadorArvoreUmElmento() {
41      if ($this->root != null && $this->size == 1) {
42         if ($this->root->getLeft() == null && $this->root->getRight() == null) {
43                 return true;
44            } else {
45              throw new Exception('A árvore com um elemento possui
46                 filho a esquerda ou filho a direita da raiz diferentes de null');
47              }
48          }
49          return true;
50  }
```

Figura 33 – Assertiva executável verificadora da árvore com um elemento

4.11.3. Árvore com mais de um elemento

```
55  public function verificadorArvoreMaisDeUmELemento() {
56      if ($this->root != null && $this->size >= 0) {
57          if ($this->root->getLeft() != null || $this->root->getRight() != null) {
58                 return true;
59            } else {
60              throw new Exception('A árvore com mais de um elemento posssui
61              filho a esquerda ou o filho a direita da raiz  diferentes de null');
62              }
63         }
64          return true;
65  }
```

Figura 34 – Assertiva executável verificadora da árvore com mais de um elemento

4.11.4. Encadeamento da árvore

```
104 public function verificadorEncadeamentoArvore() {
105     $this->gerarEncadeamentoDuplicado($this->root, $this->root);
106       return $this->verificadorEncadeamento($this->root);
107   }
108
109 public function gerarEncadeamentoDuplicado(BinaryTree $node = NULL,
110           BinaryTree $node2 = NULL){
111       if ($node != null) {
112           $node->setFather($node2);
113           $this->gerarEncadeamentoDuplicado($node->getLeft(), $node);
114           $this->gerarEncadeamentoDuplicado($node->getRight(), $node);
115          }
116   }
```

Figura 35 – Métodos auxiliares para criação das assertivas executáveis

PUC-Rio - Certificação Digital Nº 1221680/CA

```php
public function verificadorEncadeamento(BinaryTree $node = NULL){
      if ($node != null) {
          $this->verificadorEncadeamento($node->getLeft());
             if($node->getLeft() != null){
              if($node->getLeft()->getFather()!= $node){
                throw new Exception('O encadeamento da árvore está incorreto.');
                   }
              }
            if($node->getRight() != null){
               if($node->getRight()->getFather()!= $node){
                 throw new Exception('O encadeamento da árvore está incorreto:');
              }
               }
          $this->verificadorEncadeamento($node->getRight());
         }
  }
```

Figura 36 – Assertiva executável verificadora do encadeamento da árvore

4.11.5. Folhas da árvore

```php
public function verificadorFolhas($node){
      if ($node != null) {
        $this->verificadorFolhas($node->getLeft());
           if($this->size == 0 && $node != $this->root ){
                if($node->getLeft()!= null && $node->getRight()!= null){
                 throw new Exception('Estutura incorreta ');
                  }
              }

             $this->verificadorFolhas($node->getRight());
         }
         return true;
     }
 }
```

Figura 37 – Assertiva executável verificadora das folhas da árvore binária

4.11.6. Árvore de Pesquisa Binária

```php
    public function verificadorArvoreBinaria(BinaryTree $node = NULL) {
        if ($node != null) {
            $this->verificadorBalanceamento($node->getLeft());

            if ($node->getLeft() != null && $node->getRight() != null) {

                if ($node->getLeft()->getValue() > $node->getValue()
                        && $node->getRight()->getValue() < $node->getValue()) {
                    throw new Exception('A árvore não está balanceada');
                }
            }

            $this->verificadorBalanceamento($node->getRight());
        }
        return true;
    }
```

Figura 38 – Assertiva executável verificadora da propriedade da árvore binária

PUC-Rio - Certificação Digital Nº 1221680/CA

5
Teste de análise mutante

Este capítulo apresenta a fundamentação teórica necessária para o entendimento das definições e terminologias com relação aos testes de análise mutante (seção 5.1). Ainda foram descritas duas ferramentas: a primeira é a principal, pois foi utilizada na maior parte do experimento (seção 5.2) já a segunda é um protótipo implementado exclusivamente para este trabalho, que possibilita a criação de mutantes na linguagem PHP (seção 5.3) por fim é apresentada uma seção reservada para descrição dos testes unitário utilizados no experimento (seção 5.4).

5.1.
Teste de análise mutante

Teste de mutação ou análise de mutantes é uma técnica de avaliação e melhoria da qualidade dos testes, com base na determinação da sua capacidade de detectar falhas provocadas por defeitos injetados artificialmente no código sendo testado, e que são representativas de falhas reais. Com a injeção de defeitos no programa original são criadas várias versões alteradas desse programa que constituem os mutantes. Os defeitos são injetados pela aplicação de operadores de mutação, representando defeitos típicos. Segundo Delamaro (2007) os operadores de mutação surgiram de estudos que determinavam os erros mais comuns cometidos por programadores e erros mais comuns em linguagens de programação específicas.

Quando um operador de mutação é aplicado ao programa original, e este possui a estrutura sintática que um dado operador de mutação consegue modificá-lo, então essas modificações são exercidas e um conjunto de mutantes é gerado. Assim, normalmente nas ferramentas de teste de análise mutante, o mutante criado tem o nome do operador de mutação que originou sua mudança sintática. Ainda a cada ocorrência encontrada dessa estrutura sintática no programa original é criado um novo mutante. Sendo que cada mutante tem alguma pequena modificação em relação ao programa original, então cada mutante criado é diferente dos demais mutantes para o um mesmo programa.

PUC-Rio - Certificação Digital Nº 1221680/CA

Segundo a sintaxe e semântica do programa escrito, os mutantes são gerados por meio de pequenas variações do programa original e não no universo de todas as variações possíveis, essa prática é formulada como base na hipótese do programador competente que estabelece: um programa criado por um programador competente está correto ou está próximo do correto e na hipótese do efeito do acoplamento que preconiza: defeitos complexos estão ligados a defeitos simples e, por isso, a detecção de um defeito simples pode levar a descoberta de defeitos complexos (Demillo, Sayward,1978).

Para cada mutante, é aplicado o conjunto de testes original. Nos casos de testes se algum falhar, o correspondente mutante diz-se neutralizado ou morto. Neste caso, para o mesmo caso de teste o programa original e o mutante criado geram saídas diferentes. Caso os testes passem o mutante é considerado como vivo. Os mutantes que permanecem vivos poderão ser equivalentes ao programa original, ou então é um indicador de que o teste é insuficiente. Para se chegar a essa conclusão é necessária uma análise feita pelo testador, e definir se o mutante manteve vivo por ser realmente equivalente ao programa original ou se os testes não foram capazes de perceberem o defeito injetado, com isso os testes devem ser melhorados. Se os testes detectarem as falhas artificiais assume-se que detectarão falhas reais (Delamaro, et al., 2007).

Para entendimento dos termos utilizados no experimento será utilizada a seguinte terminologia:

- ***Mutante:*** o programa original modificado.
- ***Operador de mutação:*** o agente que definirá qual será o tipo de transformação que o programa original terá para ser considerado um mutante.
- ***Mutante morto ou neutralizado:*** quando a suíte de teste consegue distinguir o mutante do seu respectivo programa original, ou seja. Neste caso um ou mais testes não passaram.
- ***Mutante vivo:*** quando a suíte de teste não consegue distinguir o mutante do seu respectivo programa original, ou seja. Neste caso todos os testes passaram.
- ***Mutante equivalente:*** quando por análise de um desenvolvedor experiente o mutante é considerado equivalente ao seu respectivo programa original.

5.2. MuClipse: ferramenta para teste análise mutante para Java

Por causa dos mutantes gerados serem em grande número, torna-se indispensável o uso de uma ferramenta que tenha as seguintes funções: (i)

PUC-Rio - Certificação Digital Nº 1221680/CA

criação de mutantes, segundo os operadores de mutação escolhidos, (ii) execução de cada mutante segundo os casos de testes fornecidos e (iii) indicação do estado do mutante após a execução dos testes (DELAMARO, et al., 2007). Além do mais, é importante que seja demonstrada a diferença entre o mutante e o seu respectivo programa original. Isso se faz necessário para facilitar a análise do testador na hipótese da equivalência entre o programa original e o mutante. Existem algumas ferramentas para testar programas escritos em Java que apresentam essas características tais como: Jabuti (Vincenzi, et al., 2003), Jester (Moore, 2001) e MuClipse (Smith, Williams 2007).

O MuClipse comparado com as demais ferramentas pesquisadas foi utilizado para a realização do experimento por causa das seguintes qualidades:

1. Variedade de operadores de mutações;
2. Facilidade na gerência dos mutantes criados;
3. Poucas restrições para instalação;
4. Configuração rápida e descomplicada;
5. Desempenho em tempo considerável;
6. Documentação disponível;
7. Integração com JUnit;
8. Por ser um plug-in para Eclipse e assim aproveitar os benéficos desse ambiente de desenvolvimento sem ter custo maior para aprendizagem do ambiente.

As principais desvantagens encontradas com a utilização da ferramenta MuClipse foi porque ela (i) não permite a criação de mutantes para mais de uma classe ao mesmo tempo, e (ii) não permite que o programa original tenha o uso do *Generics* do Java. No experimento realizado alguns algoritmos das estruturas de dados eram escritos em mais de uma classe e ainda utilizavam o recurso do Generics do Java. Mas essas pendências foram sanadas para a utilização da ferramenta, pois as qualidades e principalmente pela facilidade de utilização dela compensaram as modificações feitas nas estruturas de dados.

O MuCplise é um plugin do Eclipse que fornece uma ponte entre a API Mujava existente e o Eclipse Workbench, assim MuJava é uma nova implementação das especificações do Mujava para um plugin do Eclipse (Smith, Williams 2007). Ele utiliza dois grupos de operadores de mutação para gerar os mutantes, esses são identificados como: o primeiro operador relativo à classe e o segundo operador relativo ao método. Sendo que no total são quarenta e três operadores divididos em vinte oito de classe e quinze de método.

PUC-Rio - Certificação Digital Nº 1221680/CA

Os operadores de mutação para classe foram projetados por meio da observação de falhas mais comuns em Orientação a Objetos. Estes são separados em quatro grupos (Ma, Offutt, 2005):

- Encapsulamento;
- Herança;
- Polimorfismo;
- Características específicas da Orientação a Objetos em Java.

Ainda os operadores de mutação reativos ao método são baseados nos tipos primitivos do Java, separados em seis grupos de operadores (Ma, Offutt, 2005):

- Aritmético;
- Relacional;
- Condicional;
- Deslocamento;
- Lógico;
- Atribuição.

Na Tabela 11 e na Tabela 13 são demonstrados os operadores de mutação por meio de seus respectivos tipos, nomes e uma descrição de sua ação no programa original. Ainda podem ser visualizados na Tabela 12 e na Tabela 14 fragmentos de códigos extraídos das estruturas de dados utilizadas no experimento, onde são demonstradas as modificações que cada operador de mutação realizou.

Dos quarenta e três operadores de mutação disponíveis na ferramenta será demonstrado apenas aqueles em que a ferramenta MuClipse conseguiu utilizar para criar os mutantes no experimento. É necessária a demonstração destes operadores para conhecimento dos tipos de falhas inseridas nas estruturas de dados utilizadas no experimento.

PUC-Rio - Certificação Digital Nº 1221680/CA

Tabela 11 – Operadores de mutação relativos à classe

Tipo	Sigla	Nome	Ação
Características específicas do Java	EAM	Arithmetic Operator Replacement	É modificado o nome do método “get” por outro método com assinatura compatível
	EMM	Modifier method change	É modificado o nome do método “set” por outro método com assinatura compatível
	JDC	Java-supported default constructor creation	É excluído o construtor criado
	JID	Member variable initialization deletion	É excluída a inicialização da variável de classe
	JSI	Static modifier insertion	É inserido o tipo “static” na variável de classe
	JSD	Static modifier deletion	É excluído o tipo “static” da variável de classe
	JTD	This keyword deletion	É excluída a palavra reservada "this" da instrução
	JTI	This keyword insertion	É inserida a palavra reservada "this" na instrução
Polimorfismo	OAN	Argument number change	É modificado o número de argumentos que são chamados.
	OMR	Overloading method contents change	Dentro de um método é adicionado a chamada a outro método de mesmo nome, porém com número de parâmetros diferentes.
	PRV	Reference assignment with other comparable variable	É substituído o tipo do valor que a variável recebe por um tipo descendente ao tipo da variável
Herança	IOD	Overriding method deletion	É excluída toda declaração da classe filha, sendo executada a classe pai

PUC-Rio - Certificação Digital Nº 1221680/CA

Tabela 12 – Operadores de mutação relativos à classe em uso

Sigla	Programa Original	Mutante
EAM	atual = atual.**getProxima();**	atual = atual.**getAnterior();**
EMM	nova.**setAnterior**(this.ultima);	nova.**setProxima**(this.ultima);
IOD	**public String toString(){...}**	// public java.lang.String toString(){ ... }
JDC	**public SplayTree(){ root = nullNode;}**	**// public JDC_1() { ... }**
JID	private static BinaryNode **newNode = null**;	private static BinaryNode **newNode**;
JSI	protected BinaryNode root;	protected **static** BinaryNode root;
JSD	private **static** BinaryNode newNode = null;	private BinaryNode newNode = null;
JTD	**this.**primeira = primeira;	primeira = primeira;
JTI	this.primeira = primeira;	this.primeira = **this**.primeira;
OAN	delete**(mRootNode**, key);	delete(key);
OMR	private void printTree(AANode t){ **if (t != t.left) { printTree(t.left); printTree(t.right); }** }	private void printTree(AANode t){ **printTree();** }
PRV	public void setPrimeira(Celula primeira) { this.primeira = **primeira**; }	public void setPrimeira(Celula primeira) { this.primeira = **ultima**; }

Tabela 13 – Operadores de mutação relativos ao método

PUC-Rio - Certificação Digital Nº 1221680/CA

Sigla	Nome	Ação
AODU	Arithmetic Operator Deletion, Unary	É excluído operador aritmético unário
AODS	Arithmetic Operator Deletion, Shortcut	É excluído operador aritmético de atalho
AOIS	Arithmetic Operator Insertion, Shortcut	É excluído operador aritmético de atalho
AOIU	(Arithmetic Operator Insertion, Unary	É Inserido operador aritmético unário
AORB	Arithmetic Operator Replacement, Binary	É substituído o operador aritmético binário
AORS	Arithmetic Operator Replacement, Shortcut	É substituído o operador aritmético de atalho
COD	Conditional Operator Deletion	É excluído operador condicional
COI	Conditional Operator Insertion:	É inserido operador condicional.
COR	Conditional Operator Replacement	É substituído operador condicional
LOI	Logical Operator Insertion	É inserido operador lógico.
ROR	Relational Operator Replacement:	É substituído operador relacional.
SOR	Shift Operator Replacement	É substituído operador de shift.
ASRS	Assignment Operator Replacement	É substituído operador de atribuição

Tabela 14 – Operadores de mutação relativos ao método em uso

Sigla	Programa Original	Mutante
AODU	return **-**1000;	return 1000;
AODS	if (t.right.level > **--**t.level)	if (t.right.level > t.level)
AOIS	if (t.left.level == t.level)	if (**++**t.left.level == t.level)
AOIU	array[hole]=array[hole/2]	array[hole]=array[**-**hole/2]
AORB	array[hole]=array[hole/2]	array[hole]=array[hole**+**2]
AORS	**i++**;	**i--**;
COD	if (**!**(x.compareTo(t.element) < 0))	if (x.compareTo(t.element) < 0)
COI	if (t == null)	if (**!**(t == null))
COR	if (t.left != null **&&** t.right != null)	if (t.left != null **||** t.right != null)
LOI	if (t.left.level == t.level)	if (**~**t.left.level == t.level)
ROR	if (t **==** nullNode)	if (t **!=** nullNode)
SOR	return (1 **<<** theTrees.length) - 1;	return (1 **>>** theTrees.length) - 1;
ASRS	currentSize **-=** deletedQueue.currentSize + 1;	currentSize **+=** deletedQueue.currentSize + 1;

A Figura 39 demonstra o processo de escolha dos operadores de mutação na ferramenta MuEclipse. Ainda na Figura 40 pode ser observado um mutante

PUC-Rio - Certificação Digital Nº 1221680/CA

criado (código fonte da direita) a partir do programa original (código fonte da esquerda).

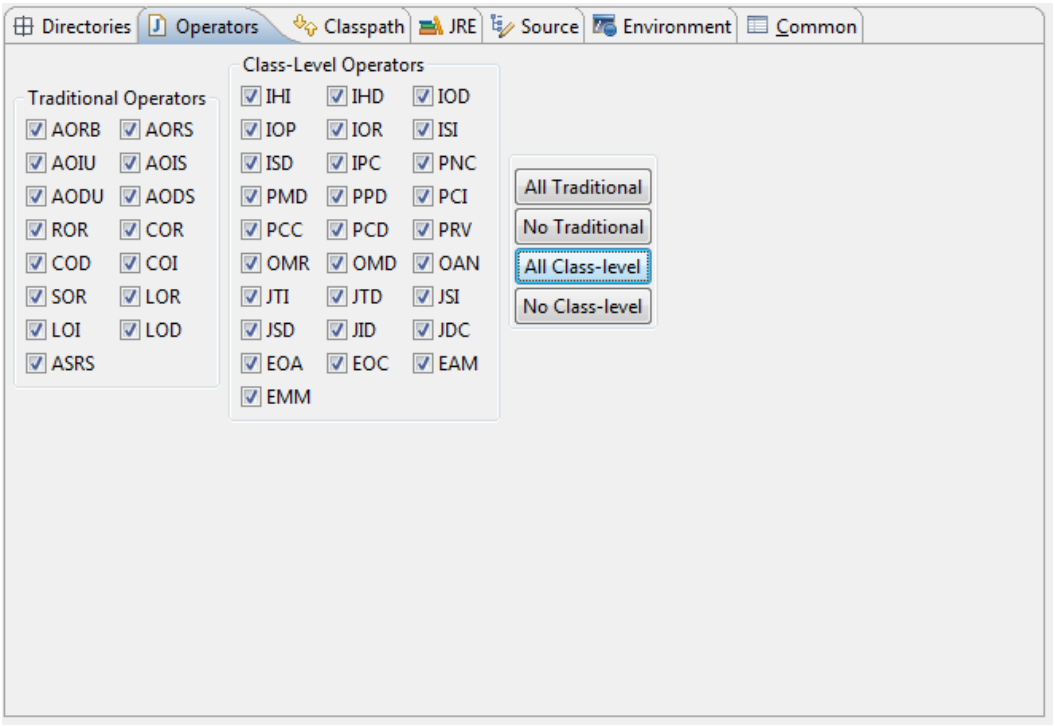


Figura 39 – Interface gráfica para selecionar operadores de mutação MuCplise

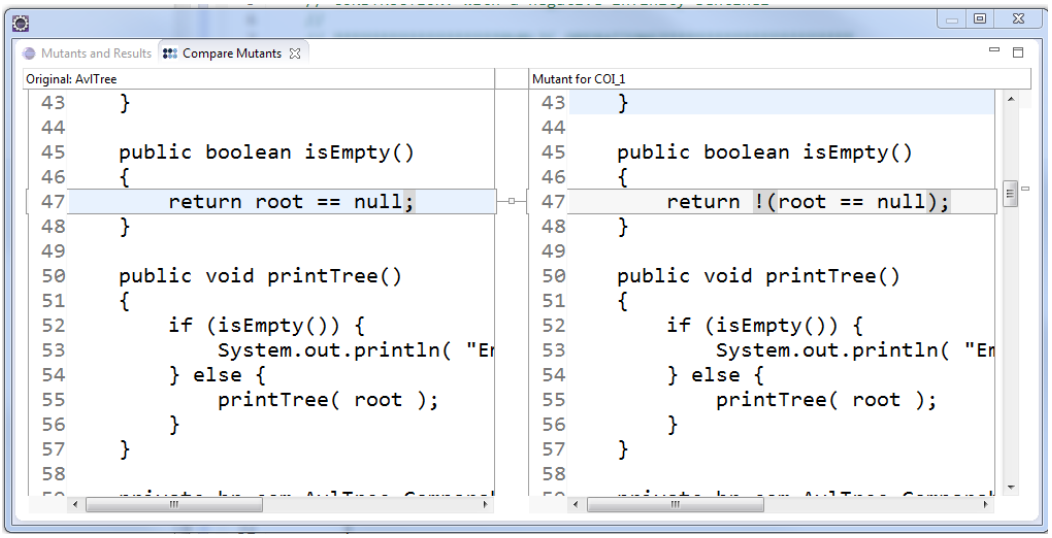


Figura 40 – Demonstração de um mutante criado

5.3. MuPhp: Protótipo de ferramenta para testes de análise mutante em PHP

Foi desenvolvido neste trabalho um protótipo de ferramenta para testes de análise mutante que possibilita a criação de mutantes para a linguagem PHP. A

PUC-Rio - Certificação Digital Nº 1221680/CA

razão motivadora para a criação desse protótipo se deu para que a eficácia das assertivas executáveis pudesse ser analisada em outra linguagem diferente de Java e ainda, por não ter sido encontrada na literatura uma ferramenta disponível de teste análise mutante para PHP.

O protótipo construído foi baseado nas especificações e operadores de mutação descrito para o MuJava, assim alguns operadores de mutação foi implementado a fim de ser criado mutantes a partir de um programa original escrito em PHP.

O protótipo em questão não traz nenhuma inovação para literatura com relação aos testes de análise mutante, pois ele apenas foi uma implementação de partes da especificação do MuJava. Assim não serão demonstrados detalhes da sua implementação. Porém na Figura 41 pode ser observada a interface do MuPHP referente à visualização de um mutante criado.

PUC-Rio - Certificação Digital Nº 1221680/CA

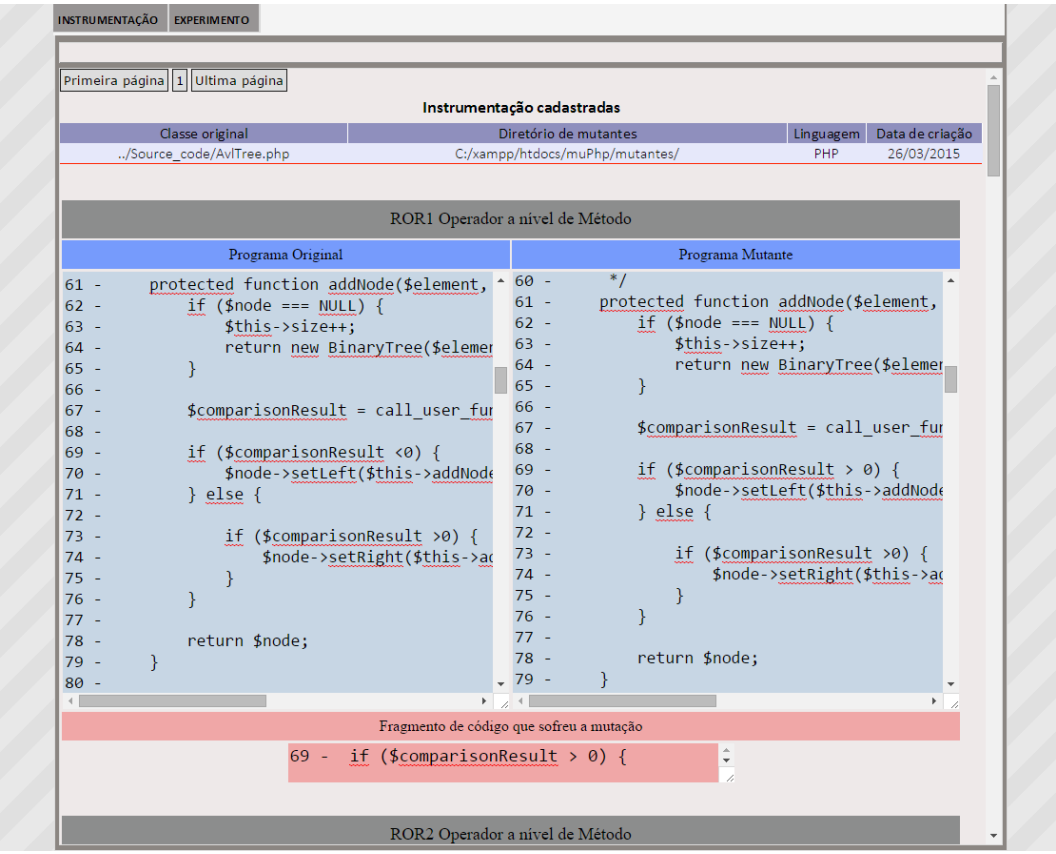


Figura 41 – Demonstração da criação de mutantes via a ferramenta MuPHP

5.4. Teste Unitário

Para a utilização das ferramentas de testes de análise mutante se faz necessário que sejam criados testes unitários no formato JUnit no caso do Java e PHPUnit para o PHP.

O teste unitário é uma modalidade de testes que se concentra na verificação de partes do código que podem ser exercitadas individualmente e que contenham regras de domínio. Por meio do fornecimento de dados suficientes para se testar apenas a lógica da unidade em questão e sabendo qual deverá ser o resultado para a unidade testada o resultado corresponde deve ser avaliado se é o resultado esperado. Essa verificação é feita pelo uso dos métodos *Assert* integrantes à biblioteca JUnit.

A Figura 42 e a Figura 43 demonstram os casos de testes criados para Árvore AA. Não serão demonstrados os casos de teste criados para todas as demais estruturas de dados utilizadas no experimento.

PUC-Rio - Certificação Digital Nº 1221680/CA

```
83      @Test
84      public void testeMinMax() {
85          int NUMS = 10000;
86          int GAP = 37;
87          boolean correto = true;
88
89          for (int i = GAP; i != 0; i = (i + GAP) % NUMS) {
90              t.insert(new MyInteger(i));
91          }
92          for (int i = 1; i < NUMS; i++) {
93              if (((MyInteger) (t.findMin())).intValue() != 1
94                      && ((MyInteger) (t.findMax())).intValue() != NUMS - 1)
95                  correto = false;
96          }
97          assertTrue(correto);
98      }
35          int GAP = 37;
36          boolean correto = true;
37
38          for (int i = GAP; i != 0; i = (i + GAP) % NUMS) {
39              t.insert(new MyInteger(i));
40          }
41
42          for (int i = 1; i < NUMS; i++) {
43           if (((MyInteger) (t.find(new MyInteger(i)))).intValue() != i)
44                  correto = false;
45          }
46          assertTrue(correto);
47      }
49      @Test
50      public void testeInserirNumerosAleatorios() {
51          int NUMS = 4000;
52          int GAP = 37;
53          boolean correto = true;
54          for (int i = GAP; i != 0; i = (i + GAP) % NUMS) {
55              t.insert(new MyInteger(i));
56          }
57          for (int i = 1; i < NUMS; i++) {
58              if (((MyInteger) (t.find(new MyInteger(i)))).intValue() != i)
59                  correto = false;
60          }
61          assertTrue(correto);
62      }
65      @Test
66      public void testeInserirRemover() {
67          int NUMS = 999;
68          boolean correto= true;
69
70          for (int i = 0; i <=NUMS; i++) {
71              t.insert(new MyInteger(i));
72          }
73
74          for(int i=NUMS; i>=0; i--){
75              if(((MyInteger) (t.findMax())).intValue()!=i){
76                  correto = false;
77              }
78              t.remove(new MyInteger(i));
79          }
80          assertTrue(correto);
81      }
83      @Test
84      public void testeMinMax() {
85          int NUMS = 10000;
86          int GAP = 37;
87          boolean correto = true;
88
89          for (int i = GAP; i != 0; i = (i + GAP) % NUMS) {
90              t.insert(new MyInteger(i));
91          }
92          for (int i = 1; i < NUMS; i++) {
93              if (((MyInteger) (t.findMin())).intValue() != 1
94                      && ((MyInteger) (t.findMax())).intValue() != NUMS - 1)
95                  correto = false;
96          }
97          assertTrue(correto);
98      }
```

Figura 42 – Caso de testes unitários para Árvore AA, parte 1

PUC-Rio - Certificação Digital Nº 1221680/CA

```
100     @Test
101     public void testeRemoverMin(){
102         int NUMS = 1000;
103         int x=1000;
104         int cont=0;
105         boolean correto= true;
106
107         for (int i = 0; i <=NUMS; i++) {
108             t.insert(new MyInteger(i));
109         }
110
111         while(cont<=500){
112             t.remove(new MyInteger(cont));
113             if ((((MyInteger) (t.findMin())).intValue() == cont)&&
114                     (((MyInteger) (t.findMax())).intValue() != x)){
115                 correto = false;
116             }
117             cont++;
118         }
119         assertTrue(correto);
120     }
121     @Test
122     public void testeRemoverMax(){
123         int NUMS = 1000;
124         int x=1000;
125         int y=0;
126         int cont=0;
127         boolean correto= true;
128
129         for (int i = 0; i <=NUMS; i++) {
130             t.insert(new MyInteger(i));
131         }
132
133         while(cont<=500){
134             t.remove(new MyInteger(x));
135             if ((((MyInteger) (t.findMax())).intValue() == x)&&
136                     (((MyInteger) (t.findMin())).intValue() != y)){
137                 correto = false;
138             }
139             x--;
140             cont++;
141         }
142         assertTrue(correto);
143     }
146     @Test
147     public void testeInserirProcurar() {
148         int NUMS = 40000;
149         int GAP = 37;
150         boolean correto = true;
151
152         for (int i = GAP; i != 0; i = (i + GAP) % NUMS) {
153             t.insert(new MyInteger(i));
154         }
155         for (int i = 1; i < NUMS; i++) {
156             if (((MyInteger) (t.find(new MyInteger(i)))).intValue() != i)
157                 correto = false;
158         }
159         assertTrue(correto);
160     }
161
162     @Test
163     public void testeLimparArvore() {
164         int NUMS = 4799;
165         int GAP = 37;
166
167         for (int i = GAP; i != 0; i = (i + GAP) % NUMS) {
168             t.insert(new MyInteger(i));
169         }
170         t.makeEmpty();
171         assertTrue(t.isEmpty());
172     }
173
174     @Test
175     public void testeArvoreSemElementos() {
176         assertTrue(t.isEmpty());
177     }
178
179     @After
180     public void tearDown() throws Exception {
181     }
```

Figura 43 – Caso de testes unitários para Árvore AA, parte 2

PUC-Rio - Certificação Digital Nº 1221680/CA

6
Experimentos e resultados

Este capítulo tem como objetivo demonstrar de forma quantitativa os resultados obtidos com o experimento realizado. Na seção 6.1 foram demonstrados os resultados da primeira parte do experimento onde foi avaliada a eficácia das assertivas executáveis. Os resultados relativos à segunda parte do experimento estão descritos na seção 6.2 a qual teve como objetivo aferir o tempo gasto nos sistemas instrumentados com assertivas executáveis.

6.1.
Resultados obtidos com as estruturas de dados

PUC-Rio - Certificação Digital Nº 1221680/CA

As implementações das estruturas de dados utilizadas no experimento foram extraídas do livro *Data Structures and Algorithm Analysis in Java* (Weiss, 2012). Assim para essas implementações foram inseridas assertivas executáveis como descritas no Capítulo 3 e no Capítulo 4, uma suíte de testes unitários e a geração de mutantes segundo o Capítulo 5.

Os resultados obtidos com os testes análise mutantes para verificar a eficácia das assertivas executáveis estão apresentados na Tabela 15. Nessa tabela são descritos os tipos de mutantes criados por meio de operadores de mutação de métodos ou classe. Estão inseridos os resultados para as estruturas em Java e para a Árvore AVL escrita em PHP.

As colunas da tabela contêm as seguintes informações: A primeira coluna descreve o nome da estrutura de dados utilizada. Na segunda coluna são apresentados os mutantes de métodos e na terceira os mutantes de classe. A próxima coluna descreve o número total de mutantes criados que consiste no somatório dos tipos de mutantes de método e classe. Também consta na Tabela 16 o detalhamento dos mutantes criados segundo os operadores de mutação que puderam exercer desvios sintáticos no programa original.

Ainda na quinta coluna é informado o número de mutantes equivalentes. O critério para análise dos mutantes equivalentes utilizado neste trabalho baseou-se na comparação da saída que o mutante produz em relação à saída que o programa original apresenta para uma mesma instância de entrada. Quando a

saída for a mesma em ambos os casos, considerou-se que o mutante é equivalente ao programa original.

Na sexta coluna, subdividida em três sub-colunas, foram descritos o número de mutantes não equivalentes mortos, vivos e o escore de mutação respectivamente para programas que não foram instrumentados com as assertivas executáveis. Da mesma forma a coluna sete da tabela descreve os resultados obtidos com programas que foram utilizados com assertivas executáveis.

Por fim a última coluna da tabela descreve a diferença entre os escores de mutação para programas instrumentados e para os não instrumentados segundo as métricas propostas neste trabalho na seção 3.2. A média obtida demonstra que as assertivas para o experimento criado puderam ser mais eficientes que os testes criados em relação ao escore de mutação numa escala de 0,3. Traduzindo em dados percentuais resultaria em 30% a mais de eficácia das assertivas para neutralizar os mutantes em relação ao oráculo dos testes.

Ainda as assertivas tiveram o escore de mutação igual a 1,0. Isso demonstra que puderam detectar todas as modificações que os mutantes apresentaram em relação ao programa original.

PUC-Rio - Certificação Digital Nº 1221680/CA

Tabela 15 – Resultados obtidos com teste mutantes e assertivas executáveis

ESTRUTURAS DE DADOS	MÉTODO	CLASSE	TOTAL	EQUIVALENTES	SEM INSTRUMENTAÇÃO			COM INSTRUMENTAÇÃO			DIFERENÇA DOS ESCORES DE MUTAÇÃO
					MORTO	VIVO	ESCORE DE MUTAÇÃO	MORTO	VIVO	ESCORE DE MUTAÇÃO	
AA Tree	133	56	189	2	147	42	0,8	189	0	1,0	0,2
AVL Tree	139	16	155	6	88	67	0,6	155	0	1,0	0,4
Binary Heap	191	2	193	1	43	150	0,2	193	0	1,0	0,8
Binary Search Tree	50	5	55	1	39	16	0,7	55	0	1,0	0,3
BinomialQueue	225	7	232	0	184	48	0,8	232	0	1,0	0,2
Black Red Tree	88	88	176	5	119	57	0,7	176	0	1,0	0,4
BTree	1582	30	1612	16	939	673	0,6	1612	0	1,0	0,4
Deterministic Skip List	32	40	72	0	57	15	0,8	72	0	1,0	0,2
Fibonacci Heap	167	39	206	0	206	0	1,0	206	0	1,0	0,0
Leftist Heap	32	6	38	0	23	15	0,6	38	0	1,0	0,4
Linked List	173	87	260	12	104	156	0,4	260	0	1,0	0,6
Pair Heap	203	87	290	1	247	43	0,9	290	0	1,0	0,1
SpalyTree	54	142	196	3	158	38	0,8	196	0	1,0	0,2
Treap	72	23	95	1	67	28	0,7	95	0	1,0	0,3
AVL Tree - PHP	19	16	35	2	30	5	0,4	35	0	1,0	0,6
	Total: 3160	Total: 644	Total: 3804	Média: 48	Total: 2451	Total: 1353	Média: 0,7	Total: 3823	Total: 0	Média: 1,0	Média: 0,3

PUC-Rio - Certificação Digital Nº 1221680/CA

Tabela 16 – Número de mutantes criados para as estruturas de dados usadas

	Linked List	Deterministic Skip List	Binomial Queue	AA Tree	AVL Tree	Binary Search Tree	B Tree	Black Red Tree	Spaly Tree	Binary Heap	Fibonacci Heap	Leftist Heap	Pair Heap	Treap
EAM	7	-	-	-	-	-	-	-	-	-	-	-	-	-
EMM	7	-	-	-	-	-	-	-	-	-	-	-	-	-
EOA	-	-	-	-	-	-	-	-	-	-	-	-	-	-
IOD	1	-	-	-	-	-	-	-	-	-	-	-	-	-
IOP	-	-	-	-	-	-	-	-	-	-	-	-	-	-
IOR	-	-	-	-	-	-	-	-	-	-	-	-	-	-
IHD	-	-	-	-	-	-	-	-	-	-	-	-	-	-
IHI	-	-	-	-	-	-	-	-	-	-	-	-	-	-
IPC	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ISI	-	-	-	-	-	1	-	1	-	-	-	-	-	
JDC	-	-	1	1	1	1	1	-	1	-	-	1	1	-
JID	-	2	-	1	2	-	4	-	2	-	4	-	1	1
JSI	3	4	2	1	2	-	6	-	1	1	6	1	2	1
JSD	-	-	1	2	-	-	3	4	2	1	-	-	-	-
JTD	3	-	-	-	-	-	-	-	-	-	-	-	-	-
JTI	-	-	-	-	-	-	-	-	-	-	-	-	-	-
OMD	-	-	-	-	-	-	-	-	-	-	-	-	-	-
OAN	-	-	-	-	-	-	6	-	-	-	-	-	-	-
OMR	1	-	-	1	2	1	3	1	1	-	-	-	-	1
PRV	62	34	3	50	9	2	7	82	135	-	27	4	33	20
PPD	-	-	-	-	-	-	-	-	-	-	-	-	-	-
PMD	-	-	-	-	-	-	-	-	-	-	-	-	-	-
PCI	-	-	-	-	-	-	-	-	-	-	-	-	-	-
PCD	-	-	-	-	-	-	-	-	-	-	-	-	-	-
PCC	-	-	-	-	-	-	-	-	-	-	-	-	-	-
AODU	-	-	-	-	-	-	1	-	-	-	-	-	-	-
AODS	6	-	-	2	1	-	14	-	-	3	5	-	-	-
AOIS	54		70	39	27	-	724	30	-	67	55	12	76	26
AOIU	4	-	9	2	1	-	104	6	-	9	11	1	11	1
AORB	20	-	16	8	27	-	308	-	-	28	4	4	32	-
AORS	8	-	6	4	1	-	26	-	-	6	3	-	2	-
COD	4	-	-	-	-	-	16	-	-	-	-	-	-	-
COI	15	15	17	22	21	20	77	11	18	10	30	7	19	15
COR	2	2	2	4	-	2	21	2	2	4	10	-	-	-
LOI	29	-	45	13	11	-	294	13	-	30	18	3	28	7
LOD	-	-	-	-	-	-	-	-	-	-	-	-	-	-
LOR	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ROR	31	15	36	39	42	28	167	26	34	30	33	5	27	23
SOR	-	-	4	-	-	-	-	-	-	-	-	-	-	-
ASRS	-	-	20	-	-	-	4	-	-	4	-	-	8	-
total:	260	72	232	189	155	55	1582	176	196	193	206	38	240	95

PUC-Rio - Certificação Digital Nº 1221680/CA

Os operadores de mutação utilizados para criação dos mutantes para a Árvore AVL em PHP estão demonstrados na Tabela 17.

Tabela 17 – Número de mutantes criados a Árvore AVL em PHP

Operador de mutação	ROR	COI	JID	OMR	PRV	AODU	AOIS	JDC	AORB
Número de mutantes	11	2	4	3	6	1	4	2	1

6.2.Medidas e comparações do tempo computacional no uso das assertivas executáveis

A eficácia das assertivas executáveis quanto ao tempo computacional foram analisadas em dois sistemas utilizando algumas das estruturas de dados descritas neste trabalho. O primeiro foi o Problema de Programação Hiperbólica (PPH) e o segundo Problema da Árvore Geradora Mínima (PAGM).

Foi observado o tempo computacional no uso desses dois sistemas, em duas situações: (i) o programa no seu estado original; e (ii) instrumentado com assertivas executáveis. Assim foi possível fazer uma estimativa do acréscimo de tempo ocasionado pela execução das assertivas executáveis considerando os problemas utilizados no experimento. Como essas verificações, em última análise, são redundâncias, espera-se que o tempo computacional sofra um acréscimo.

Os dois sistemas foram utilizados com uma grande quantidade de dados lidos de instâncias que fazem parte do *benchmark* específico de cada problema. Para esses sistemas não foram utilizados mutantes, pois o intuito da utilização desses sistemas foi observar a média do acréscimo de tempo computacional em sistema que utilizam assertivas em tempo de uso.

PUC-Rio - Certificação Digital Nº 1221680/CA

6.2.1.Problema de Programação Hiperbólica (PPH)

Dado um conjunto de pares ordenados {(a1, b1)...(an, bn)} e um par obrigatório (a0, b0) onde ai, bi ∈ Z+, R* o valor da razão máxima obtida, S* um subconjunto de N tal que R(S*)= R* e dada a fórmula:

$$R(S) = \frac{a_0 + \sum_{t \in S} a_t}{b_0 + \sum_{t \in S} b_t}$$

Lema: Seja R* o valor da razão máxima obtida para o (PPH) e S* um subconjunto de N tal que R(S*) = R*. Então, um par t pertence a qualquer S* se e somente se (at/bt) > R*.

O sistema do PPH de maneira geral funciona desta forma: Os pares ordenados são obtidos pela leitura de um arquivo criado por um gerador de instâncias específico, que gera vários pares ordenados de números inteiros. A razão é iniciada com a razão do par ordenado (a0, b0). Após todos os pares serem adicionados a uma lista, o algoritmo de ordenação coloca as razões que são criadas pela operação (ai/bi) em ordem crescente. Para cada elemento do conjunto N (Ai,Bi), é testado se a razão do elemento corrente é maior que a razão atual. Em caso positivo, é adicionado o par ao conjunto S. Assim atualiza-se o somatório e obtém-se a nova razão. Por fim, é verificado se o Lema é válido. Se existir algum par (ai/bi) que seja menor do que a razão atual, este par será removido do conjunto S e uma nova razão deverá ser calculada.

Para realização deste experimento foram escolhidos três algoritmos de ordenação de Merge Sort, Quick Sort e Selection Sort:

6.2.2.Assertivas:

6.2.2.1. Vetor ordenado:

P:

Para todo elemento pertencente ao *array* vale se o tamanho do *array* for maior que um então o elemento na posição *array[i]* deverá ser menor que o elemento na posição *array[i+1]*.

PUC-Rio - Certificação Digital Nº 1221680/CA

EFL:

$$\forall\, n \in array : ?\,(array \longrightarrow size > 1)$$
$$\Longrightarrow$$
$$(array[i] < array[i+1])$$

AEI:

Ainda o método contém uma assertiva de entrada que garante que o tamanho do vetor seja maior que um (implementado na linha 39 da Figura 44), pois quando o método é invocado o arquivo contendo os pares ordenados já terá sido lido. O arquivo que contém as instâncias utilizadas não possui nenhum elemento que seja igual a zero, assim a razão final será sempre maior que zero, ou seja, sempre será maior que a razão inicial. Logo, a assertiva de saída para esse método garante que a razão final seja maior que zero (implementado na linha 58 da Figura 44). Por meio dessas assertivas o processamento principal do sistema é monitorado para que não apresente um estado inconsistente.

PUC-Rio - Certificação Digital Nº 1221680/CA

```
38    public void specificProcess(List<OrderedPair> listNOfOrderedPairs) {
39        if (listNOfOrderedPairs.size() != 0) {
40            super.specificProcess(listNOfOrderedPairs);
41        } else
42            throw new IllegalArgumentException();
43        for (int i = 0; i < (listNOfOrderedPairs.size() - 1); i++) {
44            try {
45                if (listNOfOrderedPairs.get(i + 1) != null
46                        && listNOfOrderedPairs.get(i) != null) {
47                    if (listNOfOrderedPairs.get(i + 1).getRatio() < listNOfOrderedPairs
48                            .get(i).getRatio()) {
49                        throw new IllegalResultException();
50                    }
51                }
52            } catch (IllegalResultException e) {
53                e.printStackTrace();
54                System.err.println("@-> Os dados não estão ordenados");
55            }
56        }
57        try {
58            if (finalRatio < 0) {
59                throw new IllegalExitException();
60            }
61        } catch (IllegalExitException e) {
62            e.printStackTrace();
63            System.err
64                    .println("@-> O valor da razão não está correto, finalratio:"
65                            + finalRatio);
66        }
67    }
```

Figura 44 – Assertiva executável verificadora da ordenação do vetor

6.2.3.Ambiente do experimento

O experimento para o PPH foi realizado para três algoritmos de ordenação Merge Sort, Quick Sort e Selection Sort, foram utilizadas diversas instâncias de tamanhos variados: 10, 100, 1.000, 10.000, 100.000, e 1.000.000. Para cada tamanho de instância foram gerados dez arquivos diferentes por meio do

gerador de instâncias. Essas foram executadas para os três tipos de algoritmo de ordenação, com tempo de execução de no mínimo dez segundos. Foi necessário deixar o sistema rodar as instâncias por esse tempo, pois o relógio do processador não possui precisão para poder medir com rigor a diferença de tempos. Pois se não fosse feito isso as instâncias 10, 100 e 1000 por serem muito pequenas não apresentariam resultados confiáveis.

Justifica-se o uso de instâncias diferentes de tamanhos iguais a fim de que a análise do tempo computacional seja realizada com entradas distintas, possibilitando uma melhor estimativa do custo do uso de assertivas em sistema em uso.

Os testes foram realizados em uma máquina de configurações: processador Intel Core i3 de 2,53GHz, 3GB de memória RAM. O sistema operacional utilizado foi Windows 8 e a IDE de desenvolvimento foi o Eclipse.

6.2.4. Resultados obtidos para o PPH:

6.2.4.1. Resultados obtidos para o PPH com Merge Sort.

Como pode ser observado na Tabela 18 por meio dos resultados obtidos, demonstram que o percentual de aumento do custo computacional total em utilizar assertivas para o problema do PPH com o algoritmo de ordenação Merge Sort foi de 5,8% em relação ao mesmo programa sem a utilização de assertivas executáveis.

Tabela 18 – Resultados obtidos para o Merge Sort

Tamanho da Entrada	Sem instrumentação	Com instrumentação	Diferença	Aumento
	Média de tempo execução	Média de tempo execução		
10	0,004	0,004	0	4,3%
100	0,056	0,060	0	8,9%
1.000	0,765	0,785	0,02	2,6%
10.000	9,822	10,336	0,51	5,2%
100.000	136,244	141,683	5,44	3,9%
1.000.000	2034,780	2235,160	200,38	9,8%
Total: 6	**Total:** 146,89	**Total:** 152,87	**Média:** 1,20	**Média:** 5,8%

PUC-Rio - Certificação Digital Nº 1221680/CA

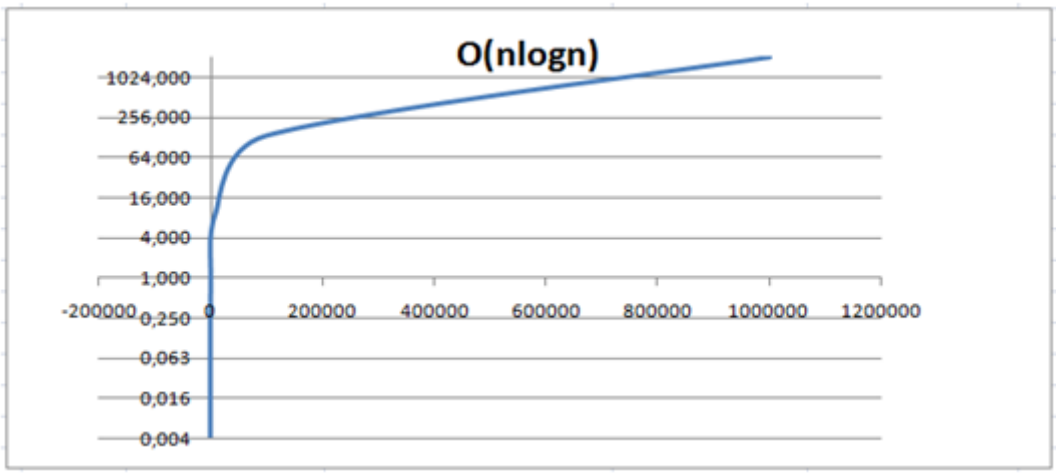


Figura 45 – Gráfico do resultado do Merge Sort sem instrumentação

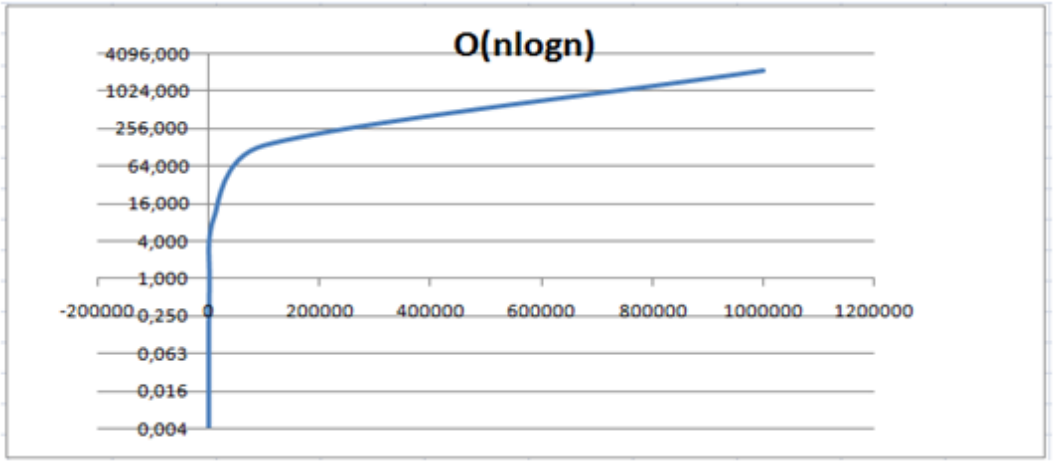


Figura 46 – Gráfico do resultado do Merge Sort com instrumentação

6.2.4.2. Resultados obtidos para o PPH com Quick Sort.

Para o PPH com o algoritmo Quick Sort o percentual de aumento do custo computacional no uso de assertivas executáveis foi de 4,8%, como pode ser observado na Tabela 19.

PUC-Rio - Certificação Digital Nº 1221680/CA

Tabela 19 – Resultados obtidos para o Quick Sort

Quick Sort				
Tamanho da Entrada	**Sem instrumentação**	**Com instrumentação**	**Diferença**	**Aumento**
	Média de tempo execução	**Média de tempo execução**		
10	0,003	0,003	0,00	7,7%
100	0,034	0,034	0,00	0,1%
1.000	0,393	0,406	0,01	3,5%
10.000	4,700	5,158	0,46	9,7%
100.000	60,1280014	65,860469	5,73	9,5%
1.000.000	1076,526709	1164,233289	87,71	8,1%
Total: 6	**Total:** 64,25	**Total:** 72,32	**Média:** 1,61	**Média:** 4,8%

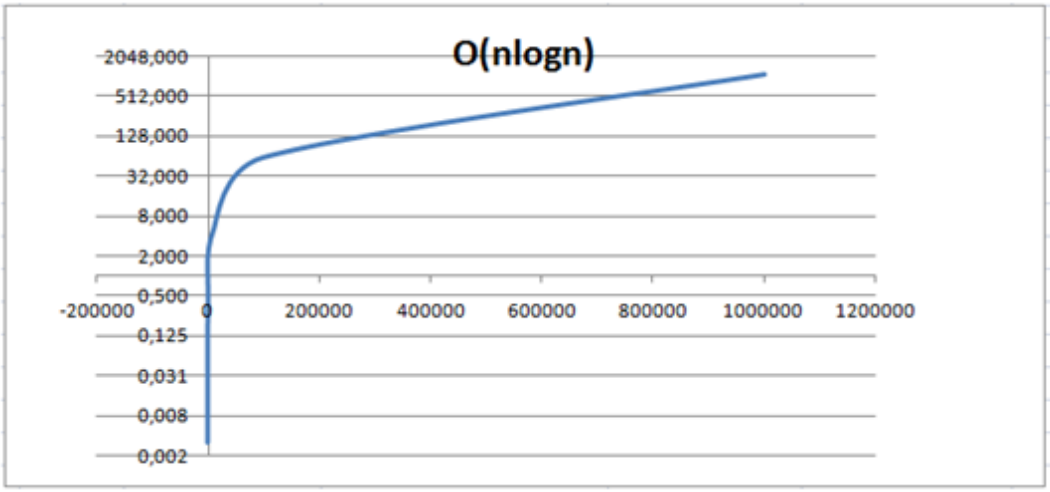


Figura 47 – Gráfico dos resultados do Quick Sort sem instrumentação

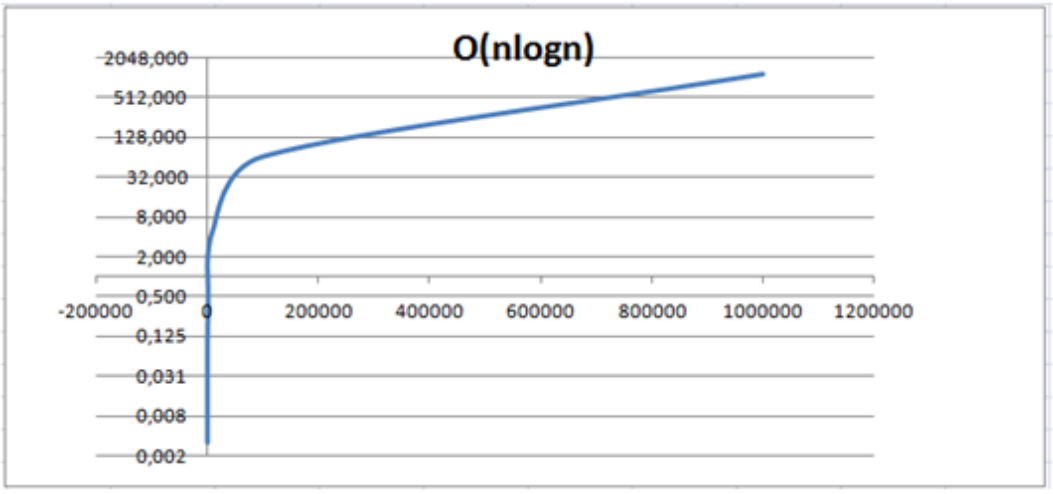


Figura 48 – Gráfico dos resultados do Quick Sort com instrumentação

PUC-Rio - Certificação Digital Nº 1221680/CA

6.2.4.3. Resultados obtidos para PPH com Selection Sort.

Para o algoritmo Selection Sort que tem uma complexidade no pior caso na ordem de N^2 o crescimento do custo no uso de assertivas executáveis ainda se manteve na média dos demais algoritmos de ordenação o qual foi do percentual de 4,9%, como pode ser observado na Tabela 20.

Tabela 20 – Resultados obtidos para o Selection Sort

Selection Sort				
Tamanho da Entrada	**Sem instrumentação**	**Com instrumentação**	**Diferença**	**Aumento**
	Média de tempo execução	**Média de tempo execução**		
10	0,003	0,003	0,000	13,36
100	0,092	0,097	0,01	5,62
1.000	8,197	8,271	0,07	0,89
10.000	829,174	833,202	4,03	0,49
100.000	836,13	840,565	4,43	0,53
1.000.000	18.104.824,00	18.968.840,00	864.016,00	4,77
Total: 6	**Total:** 18106497,60	**Total:** 18970522,14	**Média:** 1,71	**Média:** 4,9%

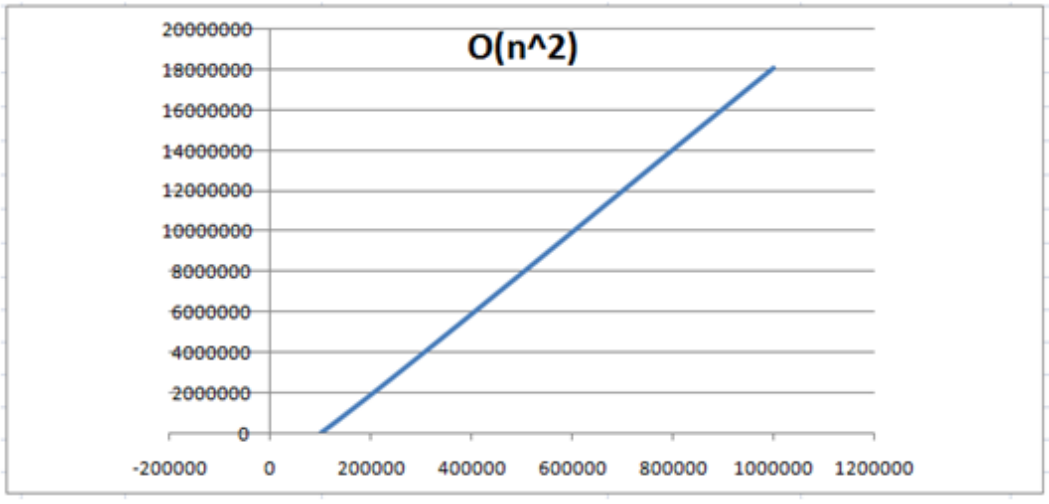


Figura 49 – Gráfico dos resultados do Selection Sort sem instrumentação

PUC-Rio - Certificação Digital Nº 1221680/CA

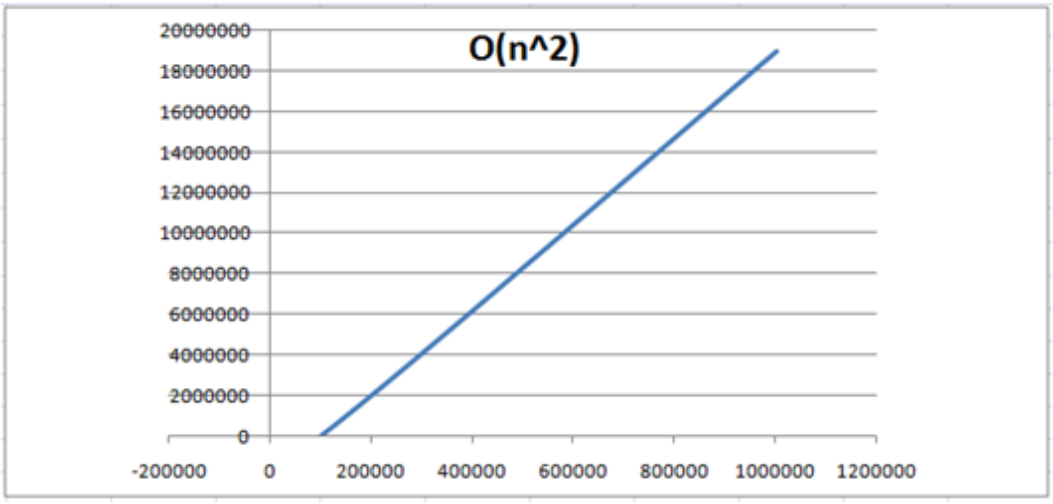


Figura 50 – Gráfico dos resultados do Selecion Sort com instrumentação

6.2.4.4. Média de tempo de execução obtida para PPH.

Por fim pode ser observada na Tabela 21 a média do custo computacional no uso de assertivas executáveis para o PPH utilizando os três algoritmos de ordenação descritos anteriormente. Segundo os dados obtidos a assertivas executáveis teve um percentual de aumento 5,7%. Como pode ser observado nos gráficos demonstrados anteriormente a complexidade de nenhum algoritmo foi alterada.

Tabela 21 – Média dos resultados obtidos para o PPH

Algoritmo	**Percentual de aumento no uso de assertivas executáveis**
Merge Sort	5,8%
Quick Sort	4,8%
Selection Sort	4,9%
	Média: 5,7%

6.2.5. Resultados obtidos para o PAGM:

No experimento do Problema da Árvore Geradora Mínima (AGM) o conjunto de assertivas foi invocado três vezes no código para cada interação. Sendo primeiramente ao inicializar árvore AVL, seguinte quando é colocado o

PUC-Rio - Certificação Digital Nº 1221680/CA

primeiro conjunto de arestas na árvore e por fim quando é adicionado um novo conjunto de arestas que sejam melhores que as tenham sido adicionadas.

Segue a tabela com a indicação dos tempos obtidos com o experimento, no programa sem a instrumentação e com a instrumentação por meio de assertivas executáveis. Ainda é disponibilizada a diferença do tempo computacional entre esses dois casos e o percentual de aumento. Com a finalidade de ser encontrar o tempo computacional exato utilizado na execução dos programas, cada instância foi executada dez vezes e uma média do tempo de execução dessas instâncias foi utilizada para a comparação nos dois casos. Ainda foi deixado o programa rodar por dez segundos e foi retirada uma média do tempo em cada iteração dentro desses dez segundos.

PUC-Rio - Certificação Digital Nº 1221680/CA

6.2.5.1. Resultados obtidos para AGM utilizando a árvore AVL

Tabela 22 – Resultados obtidos para o PAGM com AVL

Algoritmo: Prim com AVL Tree				
Instâncias	**Sem instrumentação**	**Com instrumentação**	**Diferença**	**Aumento**
	Média de tempo execução	**Média de tempo execução**		
alue2087.stp	6,76	6,85	0,09	1,29%
alue2105.stp	6,77	6,77	0,00	0,04%
alue3146.stp	31,15	31,20	0,04	0,14%
alue5067.stp	25,64	25,69	0,04	0,18%
alue5345.stp	49,54	49,76	0,22	0,45%
alue5623.stp	42,14	42,46	0,33	0,77%
alue5901.stp	136,11	136,85	0,74	0,54%
alue6179.stp	22,80	22,81	0,01	0,03%
alue6457.stp	30,68	30,97	0,29	0,95%
alue6735.stp	35,32	35,48	0,16	0,46%
alue6951.stp	17,67	17,76	0,09	0,51%
alue7066.stp	62,03	62,49	0,46	0,73%
alue7229.stp	4,71	4,98	0,28	5,93%
alut0787.stp	10,05	10,07	0,02	0,23%
alut0805.stp	6,47	6,61	0,13	2,04%
alut1181.stp	33,98	34,21	0,23	0,68%
alut2010.stp	89,00	91,95	2,95	3,32%
alut2288.stp	131,08	132,44	1,36	1,04%
alut2566.stp	55,88	56,09	0,21	0,38%
alut2764.stp	2,22	2,23	0,02	0,75%
dmxa0296.stp	0,96	0,98	0,02	2,41%
dmxa0368.stp	18,36	18,51	0,15	0,84%
dmxa0454.stp	16,33	16,47	0,14	0,86%
dmxa0628.stp	0,69	0,70	0,01	1,62%
dmxa0734.stp	3,78	3,83	0,05	1,29%
dmxa0848.stp	2,57	2,60	0,03	1,18%
dmxa0903.stp	3,35	3,39	0,05	1,34%
dmxa1010.stp	31,47	32,06	0,59	1,89%
dmxa1109.stp	1,57	1,60	0,03	1,76%
dmxa1200.stp	4,87	4,89	0,02	0,39%
dmxa1304.stp	1,39	1,40	0,01	0,66%
dmxa1516.stp	4,29	4,29	0,00	0,07%
dmxa1721.stp	7,51	7,56	0,05	0,65%
dmxa1801.stp	19,64	19,64	0,00	0,00%
Total: 33	**Total: 916,78**	**Total: 925,62**	**Média: 0,26**	**Média: 1,04%**
Tempo total de execução:		1.842,41		

PUC-Rio - Certificação Digital Nº 1221680/CA

6.2.5.2. Resultados obtidos para AGM utilizando a Leftist Heap

Tabela 23 – Resultados obtidos para o PAGM com Leftis Heap

Algoritmo: Prim com leftist Heap				
Instâncias	Sem instrumentação	Com instrumentação	Diferença	Aumento
	Média de tempo execução	Média de tempo execução		
alue2087.stp	7,16	7,17	0,01	0,16%
alue2105.stp	6,50	6,56	0,05	0,79%
alue3146.stp	56,86	58,40	1,54	2,71%
alue5067.stp	53,56	54,24	0,68	1,27%
alue5345.stp	110,57	110,83	0,27	0,24%
alue5623.stp	83,32	83,58	0,26	0,31%
alue5901.stp	572,07	572,36	0,29	0,05%
alue6179.stp	47,92	47,92	0,00	0,01%
alue6457.stp	48,59	64,65	16,06	33,04%
alue6735.stp	63,87	73,65	9,78	15,31%
alue6951.stp	34,34	35,45	1,11	3,24%
alue7066.stp	176,07	176,21	0,14	0,08%
alue7229.stp	3,97	3,98	0,00	0,13%
alut0787.stp	6,87	6,89	0,01	0,18%
alut0805.stp	4,51	4,53	0,02	0,46%
alut1181.stp	44,08	44,37	0,29	0,66%
alut2010.stp	165,51	167,63	2,12	1,28%
alut2288.stp	370,12	371,08	0,96	0,26%
alut2566.stp	115,95	116,27	0,31	0,27%
alut2764.stp	0,73	0,74	0,01	1,19%
dmxa0296.stp	0,34	0,34	0,00	0,73%
dmxa0368.stp	19,31	19,37	0,06	0,29%
dmxa0454.stp	16,06	16,07	0,01	0,08%
dmxa0628.stp	0,17	0,17	0,00	0,71%
dmxa0734.stp	2,02	2,03	0,00	0,10%
dmxa0848.stp	1,27	1,28	0,00	0,16%
dmxa0903.stp	2,01	2,01	0,00	0,00%
dmxa1010.stp	74,34	74,61	0,27	0,36%
dmxa1109.stp	0,62	0,63	0,00	0,74%
dmxa1200.stp	2,92	2,95	0,03	0,92%
dmxa1304.stp	0,46	0,46	0,00	0,29%
dmxa1516.stp	2,55	2,56	2,36	0,22%
dmxa1721.stp	4,90	4,91	0,01	0,20%
dmxa1801.stp	25,90	25,94	0,04	0,16%
Total: 33	**Total:** 2125,49	**Total: 2159,84**	**Média: 1,08**	**Média: 1,96%**
Tempo total de execução:		4.285,33		

PUC-Rio - Certificação Digital Nº 1221680/CA

6.2.5.3. Resultados obtidos para AGM utilizando a Fibonacci

Tabela 24 – Resultados obtidos para o PAGM com Fibonnaci Heap

Algoritmo: Prim com Fibonacci Heap				
Instâncias	**Sem instrumentação**	**Com instrumentação**	**Diferença**	**Aumento**
	Média de tempo execução	**Média de tempo execução**		
alue2087.stp	4,42	4,43	0,01	0,30%
alue2105.stp	4,33	4,37	0,05	1,11%
alue3146.stp	14,82	14,84	0,02	0,11%
alue5067.stp	13,93	14,04	0,11	0,81%
alue5345.stp	21,14	21,17	0,03	0,16%
alue5623.stp	18,33	18,52	0,19	1,05%
alue5901.stp	50,73	51,66	0,93	1,84%
alue6179.stp	13,06	13,14	0,07	0,57%
alue6457.stp	15,86	15,94	0,08	0,51%
alue6735.stp	16,71	16,80	0,09	0,56%
alue6951.stp	10,66	10,68	0,02	0,20%
alue7229.stp	3,25	3,29	0,04	1,34%
alut0787.stp	4,48	4,52	0,05	1,00%
alut0805.stp	3,54	3,57	0,02	0,65%
alut1181.stp	12,61	12,62	0,01	0,10%
alut2010.stp	26,65	26,71	0,07	0,25%
alut2288.stp	41,19	41,44	0,25	0,60%
alut2566.stp	21,94	22,24	0,30	1,36%
alut2764.stp	1,37	1,38	0,01	0,77%
dmxa0296.stp	0,82	0,87	0,05	6,22%
dmxa0368.stp	8,09	8,14	0,05	0,59%
dmxa0454.stp	7,10	7,20	0,10	1,41%
dmxa0628.stp	0,56	0,57	0,01	1,33%
dmxa0734.stp	2,42	2,43	0,01	0,39%
dmxa0848.stp	1,81	1,83	0,02	1,00%
dmxa0903.stp	2,32	2,35	0,03	1,26%
dmxa1010.stp	16,54	16,63	0,09	0,53%
dmxa1109.stp	1,17	1,18	0,01	0,74%
dmxa1200.stp	2,90	2,91	0,01	0,51%
dmxa1304.stp	1,03	1,03	0,00	0,45%
dmxa1516.stp	2,67	2,69	0,01	0,52%
dmxa1721.stp	3,74	3,79	0,04	1,20%
dmxa1801.stp	9,08	9,12	0,04	0,44%
Total: 33	**Total:** 359,27	**Total:** 362,12	**Média: 0,09**	**Média: 0,91%**
Tempo total de execução:		721,39		

PUC-Rio - Certificação Digital Nº 1221680/CA

Tabela 25 – Média dos resultados obtidos para o PAGM

Algoritmo	**Percentual de aumento no uso de assertivas executáveis**
Prim com AVL Tree	1,0%
Prim com lefttist Heap	1,9%
Prim com Fibonacci Heap	4,9%
	Média: 2,6%

6.2.5.4. Resultados obtidos para AGM utilizando a árvore AVL.

Nesta parte do experimento o conjunto de assertivas foi utilizado para verificar a integridade da árvore AVL cada vez que se tentava adicionar a uma nova aresta a Árvore Geradora Mínima. Assim o custo computacional foi bem maior comparado aos resultados obtidos anteriormente, como pode ser observado na Tabela 26 o percentual de aumento foi de 83,72%.

PUC-Rio - Certificação Digital Nº 1221680/CA

Tabela 26 – Resultados obtidos para o PAGM com AVL com instrumentação extra

Algoritmo: Prim com AVLTree				
Instâncias	**Sem instrumentação**	**Com instrumentação**	**Diferença**	**Aumento**
	Média de tempo execução	**Média de tempo execução**		
alue2087.stp	6,76	10,46	3,70	54,72%
alue2105.stp	6,77	9,77	3,00	44,29%
alue3146.stp	31,15	62,61	31,46	100,97%
alue5067.stp	25,64	44,87	19,22	74,97%
alue5345.stp	49,54	109,55	60,02	121,16%
alue5623.stp	42,14	92,58	50,44	119,72%
alue5901.stp	136,11	338,55	202,44	148,73%
alue6179.stp	22,80	36,76	13,95	61,20%
alue6457.stp	30,68	57,04	26,36	85,93%
alue6735.stp	35,32	67,17	31,85	90,17%
alue6951.stp	17,67	27,86	10,19	57,64%
alue7066.stp	62,03	130,82	68,78	110,88%
alue7229.stp	4,71	6,81	2,11	44,74%
alut0787.stp	10,05	19,82	9,78	97,33%
alut0805.stp	6,47	10,77	4,30	66,35%
alut1181.stp	33,98	76,85	42,88	126,20%
alut2010.stp	89,00	235,90	146,90	165,06%
alut2288.stp	131,08	325,11	194,03	148,03%
alut2566.stp	55,88	129,33	73,46	131,46%
alut2764.stp	2,22	3,77	1,55	69,86%
dmxa0296.stp	0,96	1,24	0,28	29,69%
dmxa0368.stp	18,36	37,39	19,03	103,68%
dmxa0454.stp	16,33	32,21	15,88	97,24%
dmxa0628.stp	0,69	0,96	0,28	40,43%
dmxa0734.stp	3,78	6,06	2,28	60,23%
dmxa0848.stp	2,57	3,83	1,25	48,76%
dmxa0903.stp	3,35	5,05	1,70	50,88%
dmxa1010.stp	31,47	59,06	27,59	87,69%
dmxa1109.stp	1,57	2,23	0,66	42,22%
dmxa1200.stp	4,87	8,17	3,30	67,66%
dmxa1304.stp	1,39	2,03	0,64	45,69%
dmxa1516.stp	4,29	7,03	2,74	63,86%
dmxa1721.stp	7,51	14,49	6,98	93,02%
dmxa1801.stp	19,64	38,64	18,99	96,67%
Total de instâncias:	**Total:**	**Total:**	**Média da diferença**	**Média do percentual de aumento**
33	916,78	2014,77	32,29	83,72%
Tempo total de execução:		2.931,55		

PUC-Rio - Certificação Digital Nº 1221680/CA

7
Considerações finais

O conhecimento sobre o domínio de um problema tende a ser maior quando a solução em software para o mesmo é implementado muitas vezes, consequentemente levando as ultimas versões do sistema a terem um ganho de qualidade. Isso remete ao método de tentativas e erros, pois por meio das observações das falhas geradas, quando detectadas, um novo conhecimento vai sendo criado. Logo as causas geradoras dessas falhas podem ser eliminadas para as novas implementações.

Porém na tentativa do software ser correto por construção isso não se aplica. Pois se espera que o software tenha uma qualidade elevada logo na sua primeira versão. Assim se faz necessário usar técnicas para que o software possa aproximar-se de ser correto por construção.

Um dos benefícios que se adquire utilizando os métodos formais é o conhecimento apurado sobre os requisitos do sistema. Isso ocorre porque os desenvolvedores são forçados a encontrar, entender e exercitar os detalhes das regras de negócio do domínio do sistema a ser implementado. Assim esse esforço adicional para melhor entender os requisitos, reflete em sistemas mais corretos e menos passíveis de conter defeitos.

Os Métodos Formais Leves na forma de assertivas executáveis também trazem o benefício do entendimento dos detalhes dos requisitos. Porém com custos menores comparados com os métodos formais. Para a confecção das assertivas executáveis esse entendimento apurado se faz necessário, obrigando aos desenvolvedores a se preocuparem com partes críticas do código fonte, segundo o domínio do sistema, procurar fragmentos de códigos onde são passíveis de apresentar estados computacionais errôneos causando ao sistema inconsistências gerando defeitos.

No experimento realizado o custo do entendimento das especificações dos programas, os quais foram estruturas de dados, foi nulo por causa das suas propriedades serem bem definidas e provadas na literatura. Foram utilizadas implementações distintas para a árvore AVL na linguagem Java em uma implementação para linguagem PHP e pôde ser observado que o custo em traduzir as propriedades da AVL para assertivas foi muito pequeno em todas as

PUC-Rio - Certificação Digital Nº 1221680/CA

implementações propostas. Mesmo com as mudanças de implementações as assertivas, uma vez criadas para a primeira versão, puderam ser traduzidas facilmente para as demais, sendo que as versões de implementação foram distintas. Em ambos os casos as assertivas executáveis puderam detectar as anomalias do programa original, geradas por meio dos testes de análise mutante.

O segundo momento do experimento foi realizado para observação de quanto o custo computacional foi acrescido com o emprego das assertivas executáveis utilizadas em tempo de uso. O percentual de aumento em relação ao programa original foi de 5,7% para o sistema do Problema da Programação Hiperbólica e 2,6% para o sistema Problema da Árvore Geradora Mínima. Portanto as assertivas executáveis podem ser consideradas viáveis para permanecerem acionadas nos sistemas em uso.

Ainda foi demonstrado no modelo de instrumentação com assertivas executáveis que elas podem ser ligadas ou desligadas nos sistemas em uso. Também pode haver uma equalização entre agilidade do programa que se deseja, levando-se em conta a velocidade do programa original, com o grau de verificação que permanecem acionadas nos fragmentos de código instrumentados. Portanto, para uma dada operação do sistema quanto mais as verificações forem invocadas no código, maior será o tempo computacional gasto. Como pôde ser observado no experimento para PAGM com a Árvore AVL que teve um percentual de aumento 1,04% quando o conjunto de assertivas foi invocado em três partes do código, mas não verificada para cada tentativa de incluir uma nova aresta. Porém o percentual de aumento se elevou para 83,72% quando o conjunto de assertiva foi chamado a cada tentativa de incluir uma nova aresta.

Portanto por meio dos experimentos: (i) análise mutante nas estruturas de dados e (ii) medidas do tempo computacional com o PPH e PAGM as assertivas foram eficazes e economicamente viáveis como forma de instrumentação e observação de falhas.

Ainda os questionamentos propostos nos objetivos específicos desse trabalho puderam ser respondidos com base no experimento realizado. As assertivas puderam encontrar falhas dos mutantes antes dos oráculos associados aos casos de teste. Como foram demonstrados todos os mutantes criados ficaram neutralizados pela intervenção das assertivas executáveis criadas. Elas foram 100% eficazes na detecção das anomalias que os mutantes apresentaram em relação o programa original. Também as falhas inseridas

PUC-Rio - Certificação Digital Nº 1221680/CA

sistematicamente foram detectadas pelas assertivas executáveis em tempo de execução.

Por fim, a hipótese formulada na introdução do trabalho a qual preconiza que o uso dos Métodos Formais Leves, por meio do emprego sistemático de assertivas executáveis pôde ser um mecanismo eficaz e economicamente viável para assegurar a confiabilidade do software. Além de torná-lo autoverificante por viabilizar a observação de defeitos e possibilitar uma forma de reduzir falhas em tempo de uso produtivo. Por meio dos dados quantitativos obtidos essa hipótese pôde ser tomada como verdade.

7.1. Contribuições

A maior contribuição do trabalho é demonstrar de forma quantitativa por meio do experimento realizado que o uso de assertivas executáveis é economicamente viável para a utilização em produção, reforçando o conhecimento adquirido com a revisão realizada que já tinha chegado a essa conclusão.

Ainda como contribuições podem ser destacadas nos seguintes itens:

1. **Modelo de instrumentação de assertivas:** foi demonstrado na prática um mecanismo para instrumentar programas com assertivas executáveis;
2. **Modelo para a utilização de teste análise mutante**: foi demonstrado um modelo de utilização de assertivas e teste mutante por meio de forma prática como utilizar essas técnicas;
3. **Métrica para avaliação de software:** foram propostas métricas para avaliar a eficácia das assertivas executáveis;
4. **MuPHP:** protótipo de ferramenta para testes análise mutantes para a linguagem PHP;
5. **Revisão da bibliografia:** foi pesquisada em vários artigos a aplicação prática dos Métodos Formais Leves.

7.2. Limitações e trabalhos futuros

A redação de assertivas reduz o número de defeitos, mas não os elimina. Além disso, o conjunto de assertivas muitas vezes é incompleto, e pode até ser incorreto, permitindo a ocorrência de falsos negativos**.**

PUC-Rio - Certificação Digital Nº 1221680/CA

Ainda se faz necessário o esforço adicional para o entendimento de detalhes dos requisitos, pois se as assertivas não contemplarem os pontos críticos do sistema, falhas não serão observadas por elas. Assim no experimento realizado neste trabalho o custo em tomar conhecimento das partes críticas foi nulo, porque as propriedades das estruturas de dados dos são bem definidas e provadas na literatura relacionada. Portanto não foi levado em consideração esse fator que é primordial que a criação do conhecimento dos requisitos do sistema. Em outro experimento que se tenha que levantar os requisitos para confecção das assertivas, elas podem ficar incompletas e não terem a eficácia de 100% que foi observado no experimento deste trabalho.

Além disso, as confecções das assertivas e da suíte de teste foram feitas por apenas um programador. Assim para se ter uma confiabilidade maior no resultado do experimento seria importante que essas atividades fossem feitas por programadores distintos.

Na segunda parte do experimento foi aferido o tempo computacional. Faz-se necessário um estudo mais detalhado para identificar as possíveis causas do custo adicional no tempo computacional, a fim de que se possa encontrar um melhor balanceamento entre velocidade do programa original com uso do conjunto de assertivas executáveis.

Ainda existe a questão da escolha de estruturas de dados utilizadas no experimento como amostragem de uma população de sistemas. Seria necessário se fazer uma medição de quão boa é essa amostragem.

Não foi objetivo comparar a eficácia das assertivas com a suíte de testes, mas em dado momento do trabalho foi feita essa comparação. Porém essa comparação para que seja mais justa, seria importante pegar assertivas e testes criados por vários programadores e em diferentes tipos de programas.

O protótipo desenvolvido foi implementado segundo as especificações do MuJava seguindo seus operadores de mutação. Mas as falhas estão em muito relacionadas à linguagem em que o programa está escrito. Assim seria necessário criar operadores de mutação para a linguagem PHP.

PUC-Rio - Certificação Digital Nº 1221680/CA

8
Referências Bibliográficas

AKHTAR, N. and MISSEN, M. M. S. (2014). “**Practical application of a light-weight formal implementation for specifying a multi-agent robotic system**” International Journal of Computer Science Issues, Vol. 11, Issue 1, No 2, January 2014.

ANDERSSON, A; “**Balanced Search Trees Made Simple"**, *Workshop on Algorithms and Data Structures*, pages 60-71. Springer Verlag; 1993.

ARAÚJO, O.L.R; “**Test-Driven Maintenance: uma abordagem para manutenção de sistemas legados”.** Dissertação de Mestrado – PUC-Rio – Departamento de Informática; 2011.

ARAÚJO,T.P; CERQUEIRA,R; STAA, A.V; “**Annotating Logs With Meta-information to Support Failure Diagnosis”.** Monografias em Ciência da Computação – PUC-Rio – Departamento de Informática, 2014.

BARTETZKO, DETLEF, et al. "**Jass—Java with assertions**." Electronic Notes in Theoretical Computer Science 55.2: 103-117, 2001.

BOYATT, R. C., and J. E. SINCLAIR. "A **lightweight formal methods” perspective on investigating aspects of interactive systems**", Pre-Proc. 2nd Int. Workshop Formal Methods Interact. Syst. 2007.

BERRY, D.M.; Academic Legitimacy of the Software Engineering Discipline; **Technical Report 92-TR-34**, Software Engineering Institute, Carnegie Mellon Academic, University; 1992.

BROWN, A, and D. A. PATTERSON. "**To err is human."** Proceedings of the First Workshop on evaluating and architecting system dependability (EASY’01). 2001.

PUC-Rio - Certificação Digital Nº 1221680/CA

BLACKBURN, M.R., BUSSER, R and A. NAUMAN. "**Removing Requirement defects and Automating test**." STAREAST, May (2001).

CALINESCU, R, and KIKUCHI, S. "**Formal methods@ runtime**". Springer Berlin Heidelberg, 2011.

CHEON,Y; LEAVENS, G,T; "**A Runtime Assertion Checker for the Java Modeling Language JML"**, 2002.

CORMEN T, H; LEISERSON C, E; RIVEST, R,L; Stein, C; "**Algoritmos Teoria e Prática",** Elsevier; 2002.

DELAMARO, M. E., MALDONADO, J. C., and JINO, M. (2007). "**Introdução ao teste de software**" Rio de Janeiro, RJ, BR: Editora Campus, 2007. 408 p.

DEMILLO, R.; LIPTON, R.; SAYWARD, F. "**Hints on Test Data Selection: Help for the Practicing Programmer"**. Computer, 11(4):34–41, Apr. 1978.

Duncan, A., and Hölzle, U. (1998). "**Adding contracts to Java with Handshake**". Technical Report TRCS98-32, Department of Computer Science, University of California, Santa Barbara, CA.

EASTERBROOK, S., et al. "**Experiences using lightweight formal methods for requirements modeling**." Software Engineering, IEEE Transactions on 24.1; 1998.

*FEATHER, M. S. "**Rapid application of lightweight formal methods for consistency analyses.**" Software Engineering, IEEE Transactions on 24.11 (1998): 949-959.*

HIERONS, R. M., et al. "**Using formal specifications to support testing**." ACM Computing Surveys (CSUR) 41.2; 2009

JACKSON, D; **"Alloy Tutorial"**; acessado em 13/01/2014 http://alloy.mit.edu/alloy/, 2012.

PUC-Rio - Certificação Digital Nº 1221680/CA

KARAORMAN,M; HÖLZLE,U; BRUNO,J. "**jContractor: A reflective Java library to support design by contract**." Meta-Level Architectures and Reflection. Springer Berlin Heidelberg, 1999.

Plösch, Reinhold. "**Evaluation of assertion support for the java programming language**." Journal of Object Technology 1.3 (2002): 5-17.

KNEUPER, R.; "**Limits of formal methods**." Formal Aspects of Computing 9.4 (1997): 379-394.

KRAMER,R; "**iContract-the Java TM design by contract TM tool**." Technology of Object-Oriented Languages, 1998. TOOLS 26. Proceedings. IEEE, 1998.

Larsen, P.G., Fitzgerald, J.S., Riddle, S. (2006). "**Learning by Doing: Practical Courses in Lightweight Formal Methods using VDM++**". Technical Report CS-TR:992, School of Computing Science, Newcastle University.

MA, Y.-S., Offutt, A. J. and Kwon, Y.-R. MuJava: "**An Automated Class Mutation System. 2, June 2005, Software Testing, Verification & Reliability"**, Vol. 15, pp. 97 - 133.

MA, Y.-S, Y.R KWON, and J. OFFUTT. "**Inter-class mutation operators for Java**." Software Reliability Engineering, 2002. ISSRE 2003. Proceedings. 13th International Symposium on. IEEE, 2002.

Ma, Y.S, and Jeff OFFUTT. "**Description of class mutation mutation operators for java**", 2005.

MAGALHÃES, J., STAA, A.v, and de LUCENA, C. J. P. (2009). "**Evaluating the recovery oriented approach through the systematic development of real complex applications".** Software: Practice and Experience", 39(3), 315-330.

MALDONADO J. C; A. M VINCENZI; R; DELAMARO M. E.; "**JaBUTi – Java Bytecode Understanding and Testing"**, 2003.

PUC-Rio - Certificação Digital Nº 1221680/CA

MEYER, B. "**Eiffel: A language and environment for software engineering.**" Journal of Systems and Software 8.3 (1988): 199-246.

MOORE, I. "**Jester-a JUnit test tester**." Proc. of 2nd XP (2001): 84-87.

Paige, Richard F., and Jonathan S. Ostroff. "**Specification-driven design with Eiffel and agents for teaching lightweight formal methods"**. Springer Berlin Heidelberg, 2004

PARASOFT; "**Using Design by Contract to Automate Java Software and Component Testing"**, acessado em 13/02/2015 http://char.tuiasi.ro/doace/www.parasoft.com/products/jtest/papers/tech_dbc.htm 2002.

PEZZÈ, M; YOUNG M. "**Teste e análise de software: processos, princípios e técnicas**". Bookman, 2008.

SHAO, D, S KHURSHID, and D E. PERRY. "**An incremental approach to scope-bounded checking using a lightweight formal method"**. FM 2009: Formal Methods. Springer Berlin Heidelberg, 2009. 757-772.

SIMÕES, S.C.F; "**Automatização De Testes De Mutação Em Java"**, 2014.

SLEATOR, D, D.; TARJAN, R, E; "**Self-Adjusting Binary Search Trees**" *Journal of the ACM*; 1985.

SMITH, B. H., and WILLIAMS, L. (2007, September). "**An empirical evaluation of the MuJava mutation operators**". In Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007. TAICPART-MUTATION 2007 (pp. 193-202). IEEE.

STAA, A.v. (2000) "**Programação Modular, Desenvolvendo programas complexos de forma organizada e segura**". Rio de Janeiro. Editora Campus/Elsevier.

PUC-Rio - Certificação Digital Nº 1221680/CA

STAA, A.v; “**Qualidade se Atinge com Profissionais de Qualidade”**, VIII Simpósio Brasileiro de Qualidade de Software, 2009. Acessado em 25 de maio de 2014 no endereço: http://www.lbd.dcc.ufmg.br/bdbcomp/servlet/Trabalho?id=8818.

STAA, A.V; “**Controle da Qualidade Baseada em Técnicas Formais”**, nota de aula PUC-Rio, 2013. Acessado em 25 de maio de 2014 no endereço: http://www.inf.puc-rio.br/~inf2134/docs/INF2134_Modulo08_TecnicasFormais.pdf.

STAA, A.V; “**Engenharia de Software Fidedigno Qualidade de Software”**, nota de aula PUC-Rio, 2014. Acessado em 20 de março de 2015 no endereço: http://www.inf.puc-rio.br/~inf2134/docs/INF2134_Modulo03_IntroducaoControleQualidade.pdf

STAA, A.V; “**Engenharia de Software Fidedigno”**, Monografias em Ciência da Computação- PUC-Rio - Departamento de Informática; 2006.

THOMAS, D.; “**The Deplorable State of Class Libraries”**; Journal of Object Technology; Zürich, CH: ETH Zürich; 2002; pags 21-27

VUILLEMIN, Jean. "**A data structure for manipulating priority queues**." Communications of the ACM 21.4 (1978): 309-315.

WEBER, T.S. "**Um roteiro para exploração dos conceitos básicos de tolerância a falhas**." Relatório técnico, Instituto de Informática UFRGS (2002).

WEISS, M,A. (2012) “**Data Structures and Algorithm Analysis in Java**”. 3rd ed. ISBN-13: 978-0-13-257627-7.

WIKIPÉDIA; **Métodos formais**, 2013, acessado em 03 de agosto de 2014 http://pt.wikipedia.org/wiki/M%C3%A9todos_formais.

WOODCOCK, Jim, et al. "**Formal methods: Practice and experience**." ACM Computing Surveys (CSUR) 41.4 (2009): 19.

PUC-Rio - Certificação Digital Nº 1221680/CA

YANG, G; KHURSHID,S; KIM, M;. "**Specification-based test repair using a lightweight formal method.**" FM 2012: Formal Methods. Springer Berlin Heidelberg, 2012. 455-470.

YI, Q., YANG, Z., LIU, J., ZHAO, C., and WANG, C. (2015). "**A synergistic analysis method for explaining failed regression tests**". In *International Conference on Software Engineering*.

PUC-Rio - Certificação Digital Nº 1221680/CA

Apêndice 1 **Classificação dada aos artigos pesquisados**

Título	Ano Pub.	Exemplos práticos	Relação com tema
Specification-Based Test Repair Using a Lightweight Formal Method	2012	ok	5
An Incremental Approach to Scope-Bounded Checking Using a Lightweight Formal Method	2009	ok	5
Learning by Doing: Practical Course s in Lightweight Formal Methods using VDM++	2006	ok	5
Specification-Driven Design with Eiffel and Agents for Teaching Lightweight Formal Methods	2004	ok	5
Practical application of "lightweight" Z in DEVS framework	2003	_	5
Automatic analysis of consistency between requirements and designs	2001	ok	5
Rapid Application of Lightweight Formal Methods for Consistency Analyses	1998	ok	5
Tekes Ubicom program LIME project LightweIght formal Methods for distributed component-based Embedded systems (2010)	2010	ok	5
Formal Methods: Practice and Experience	2009	ok	5
Transformation techniques can make students excited about formal methods	2008	ok	5
Rapid System Prototyping Creating and Validating Embedded Assertion Statecharts	2007	ok	5
Triumphs and Challenges for the Industrial Application of Model- Oriented Formal Methods	2007	ok	5
Lightweight Formal Methods for Scenario-Based Software Engineering	2005	ok	5
Ontology Based Requirements Analysis: Lightweight Semantic Processing Approach	2005	ok	5
UML2ALLOY: A TOOL FOR LIGHTWEIGHT MODELLING OF DISCRETE EVENT	2005	ok	5

PUC-Rio - Certificação Digital Nº 1221680/CA

SYSTEMS			
A Lightweight LTL Runtime Verification Tool for Java	2004	ok	5
Formal methods for smart cards: an experience Report	2004	ok	5
Lightweight coarse-grained coordination: a scalable system-level approach	2004	ok	5
Lightweightcoarse-grainedcoor dination: ascalablesystem-levelapproach	2004	ok	5
General Test Result Checking with Log File Analysis	2003	ok	5
Language Support for Lightweight Transactions	2003	ok	5
Lurch: a Lightweight Alternative to Model Checking	2003	ok	5
Managerial Issues for the Consideration and Use of Formal Methods	2003	ok	5
RBAC Schema Verifications Using Lightweight Formal Model and Constraint Analysis	2003	ok	5
Lightweight validation of natural language requirements	2002	ok	5
AProposal for a Lightweight Rigorous UML-Based Development Method for Reliable Systems	2001	ok	5
Exploring the Design of an Intentional Naming Scheme with an Automatic Constraint Analyzer	2000	ok	5
Support for Teaching Formal Methods	2000	ok	5
Questions and Answers about ten Formal Methods	1999	ok	5
A lightweight Approach to Formal Methods	1998	ok	5
Experience of using a lightweight formal method for Requirements Modeling	1998	ok	5
Formal Methods for Verification and Validation of Partial Specifications: A Case Study	1998	ok	5
Lightweight formal method for Computer	1998	ok	5

PUC-Rio - Certificação Digital Nº 1221680/CA

Algebra Systems			
On the Need for practical Formal Methods	1998	_	5
Pragmatic Formal Design: A Case Study in Integrating Formal Methods into the HCI Development Cycle	1998	ok	5
Formal Modeling and Validation Apllied to a Commercial Coherent Bus: A case Study	1997	ok	5
The use of Industrial- Strength Formal Methods	1997	_	5
Lightweight causal and atomic group multicast	1991	ok	5
Balancing Insight and Effort: The Industrial Uptake of Formal Methods	2007	_	5
Trustable Formal Specification for Software Certification	2010	_	4
Lightweight Self-Protecting JavaScript	2009	ok	4
Automated Reasoning in Kleene Algebra	2007	ok	4
EXPLODE: a Lightweight, General System for Finding Serious Storage System Errors	2006	ok	4
Experience of using a lightweight formal specification method for a commercial embedded system product line	2005	ok	4
Ten Commandments Revisited	2005	_	4
A Formal Monitoring-based Framework for Software Development and Analysis	2004	ok	4
How the Design of JML Accommodates Both Runtime Assertion Checking and Formal Verification	2004	ok	4
Java-MaC: A Run-Time Assurance Approach for Java Programs	2004	ok	4
Model Comparison: A Key Challenge for Transformation Testing and	2004	ok	4
Lightweight Reasoning about Program Correctness	2002	_	4
Intrusion Tolerant Software Architectures	2001	ok	4
Lightweight Analysis of Object Interactions	2001	ok	4

PUC-Rio - Certificação Digital Nº 1221680/CA

Modular semantics for a UML statechart diagrams kernel and its extension to multicharts and branching	2001	ok	4
Specification of the Javacard API in JML	2000	ok	4
NASA Langley's research and technology-transfer program in formal methods	1998	_	4
A ligtweight Approach to Formal Methods	1997	ok	4
Software engineering and formal methods	2008	_	3
Formal Methods for Specifying, Validating, and Verifying Requirements	2007	ok	3
A Lightweight Formal Framework for Service-Oriented Applications Design	2005	_	3
Application of Lightweight Formal Methods to Software Security	2005	_	3
Remote Integration and Coordination of Verification Tools in JETI	2005	ok	3
Practical Application of Formal Methods in Modeling and Simulation	2003		3
JML: notations and tools supporting detailed design in Java	2000	_	3
Formal Development of an Embedded Verifier for Java Card Byte	1999	ok	3
Lightweight formal method (1996) - A Specifier's Introduction to Formal Methods	1990	_	3
Formal Methods @ Runtime	2010	_	2
MoDeST - A Modelling and Description Language for Stochastic Timed Systems	2006	_	2
Lightweight Object Specification with Typestates	2005	_	2
Dear Sir, Yours faithfully: an Everyday Story of Formality	2004	_	2
Formal Techniques for Java-like Programs(FTfJP)	2004	_	2
Modeling and Validating Distributed Embedded Real-Time Systems with VDM++	2004	_	2
A Lightweight Formal Analysis of a Multicast	2003	ok	2

PUC-Rio - Certificação Digital Nº 1221680/CA

Key Management Scheme			
Deriving Operational Software Specifications from System Goals	2002	-	2
Experiences Using Lightweight Formal Methods for Requirements Modeling	2002	_	2
Semantic Web for Extending and Linking Formalisms	2002	ok	2
Validating Voice Communication Requirements Using Lightweight Formal Methods (2000)	2000	ok	2
A pragmatic approach to formalizing object-oriented modeling and development	1997	ok	2
Transforming OntoUML into Alloy: towards conceptual model validation using a lightweight formal method	2009	ok	1
Applying a Formal Requirements Method to three NASA Systems: Lessons Learned	2007	_	1
A "Lightweight Formal Methods" Perspective on Investigating Aspects of Interactive Systems (conjunto de artigos)	2007	ok	1
Formal Methods Light	2006	_	1
Automating commutativity analysis at the design level	2004	ok	1
Formal Specification and Static Checking of Gemplus' Electronic Purse Using ESC/Java	2004	_	1
Goal-Oriented Requirements Engineering: From System Objectives to Uml Models to precise Software Specifications	2003	_	1
The application of Dependence Analysis to Software Architecture Descriptions	2003	ok	1
A General Framework for Formalizing UML with Formal Languages	2001	ok	1
Analyzing Uml Active Classes and Associated State machines - A Lightweight Formal Approach	2000	_	1

PUC-Rio - Certificação Digital Nº 1221680/CA

Requirements validation of a voice communication system used in air traffic control. An industrial application of light-weight formal methods	2000	_	1
Formal methods for extensions to CAS	1999	ok	1
Real-Time Reactive System Development -A Formal Approach Based on UML and PVS	1998	ok	1
The LIME Interface Speciation Language and Runtime Monitoring Tool	2009	ok	0
Security Requirements for the Rest of Us: A Survey	2008	_	0
Motivating language learners: a classroom-oriented investigation of teachers' motivational practices and students' motivation	2007	_	0
Validating UML and OCL Models in USE by Automatic Snapshot Generation	2005	_	0
Eliciting security requirements with misuse cases	2004		0
Towards an FCA based tool for visualizing	2003		0
Chase: a Static Checker for JML's Assignable Clause	2002	_	0
equality in computer in Computer Algebra and Beyond Agile Specifications	2002	_	0
Automata-based verification of temporal properties on running programs	2001	_	0
Automating First-Order Relational Logic	2000	_	0
Broad Spectrum Studies of log File Analysis	2000	_	0
Desugaring JML Method Specifications	2000	_	0
Viewpoints: a framework for integrating multiple perspectives system development	1992	ok	0

PUC-Rio - Certificação Digital Nº 1221680/CA