Uma Análise da Eficácia de Assertivas Executáveis como Indicadoras de Falhas em Software

Fischer J. Ferreira[1], Arndt von Staa[1], Eduardo Figueiredo[2]

[1]Departamento de Informática – Pontifícia Universidade Católica do Rio de Janeiro (PUC- Rio) Rio de Janeiro – RJ – Brasil.

[2]Universidade Federal de Minas Gerais (UFMG) – Belo Horizonte – MG – Brasil.

{fferreira,arndt}@inf.puc-rio.br, figueiredo@dcc.ufmg.br

Abstract. *During software debugging, a significant amount of effort is needed so that programmers can identify the root cause of a failure. In an attempt to make the software capable of detecting bugs on runtime, we analyze a mechanism for indicating failures on software systems with systematic use of executable assertions. A quasi-experiment was conducted by inserting bugs in some data structures through mutation testing. Results showed that the executable assertions were effective in killing all mutants created during the experiment.*

Resumo. *Durante a depuração de software, uma quantidade significativa de esforço é necessária para que os programadores possam identificar a raiz de uma falha. Na tentativa de tornar o software capaz de detectar defeitos em tempo de uso, analisa-se um mecanismo para indicar falhas em sistemas de software com emprego sistemático de assertivas executáveis. Um quase-experimento foi realizado inserindo defeitos em algumas estruturas de dados por meio de teste baseado em mutantes. Os resultados mostraram que as assertivas executáveis para um grupo particular de programas foram eficazes em matar todos os mutantes criados no experimento.*

1. Introdução

Dia a dia aumenta a nossa dependência com relação a sistemas informatizados. Porém sistemas de software, mesmo quando confeccionado seguindo regras rígidas de qualidade, não estão livres das ocorrências de falhas durante a sua vida útil [Brown and Patterson 2001]. São utilizados cada vez mais bibliotecas e serviços remotos que muitas vezes possuem qualidade duvidosa. Além disso, independentemente do cuidado dos projetistas, especificações erradas ou incompletas são também frequentes, pois advém de possível falta de conhecimento por parte dos projetistas [Armour 2000]. 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]. Então, faz-se necessário que exista um mecanismo para facilitar o processo de observação de falhas e de identificação dos fragmentos de códigos relacionados com elas [Pullum 2001].

Ao exercitar um defeito é possível que seja gerado um erro, ou seja, um desvio entre o estado do sistema e o estado que havia sido especificado. Erros podem passar um tempo sem que sejam observados. No momento em que são observados passam a ser falhas. Para poder diagnosticar corretamente e com pouco esforço a causa exata (o

defeito) é importante que seja pequena a latência entre o momento da geração do erro e da sua observação [Magalhães et.al. 2009]. Quanto menor a latência, menor será o volume de código analisado, menor será a interferência de dados e fragmentos de código não relacionados ao defeito. Em adição, é necessário saber o porquê da falha, para que mantenedores ou desenvolvedores sejam capazes de remover os correspondentes defeitos de forma eficaz [Khoshnood 2015].

A violação de uma assertiva executável (AE) indica a presença de uma falha [Khoshnood 2015]. Além de reportar a ocorrência da falha, pode-se também reportar a parte relevante do estado do sistema no momento da sua observação. Surgem então as seguintes questões de pesquisa: (i) as AEs seriam um mecanismo eficaz para observar falhas? (ii) dado um número conhecido de falhas, as AEs podem indicá-las em totalidade?

Neste trabalho não discutimos quais políticas de inserção de AEs devem ser utilizadas. Também não foi objetivo avaliar o custo de sua redação. Em Magalhães et.al. [2009], foi mostrado que AEs requerem em torno de 10% de linhas de código a mais. Como elas tornam necessária a formalização, mesmo que parcial, de aspectos implementacionais do programa, o custo do desenvolvimento diminuiu em virtude da redução do retrabalho inútil. Também não foi objetivo verificar a aceitação pelos desenvolvedores do uso sistemático de AEs ao desenvolver programas. Em Araújo et.al. [2012], foi demonstrado que até mesmo desenvolvedores pouco experientes puderam dar manutenção em um programa com apoio de um conjunto de AEs.

Neste trabalho analisamos, por meio de um experimento, o uso de AEs quanto a sua eficácia para observar falhas. Para tal medimos quantos mutantes são mortos por intermédio das AEs. Um mutante é uma versão propositalmente adulterada do programa sob teste, injetando um ou poucos defeitos nele. Um mutante é dito morto, se um ou mais casos de teste da suíte de teste acuse uma falha. A eficácia da suíte de teste pode ser estimada observando-se quantos mutantes são mortos pela suíte de teste. Espera-se, porém, que um programa devidamente instrumentado com assertivas executáveis seria capaz de acusar a ocorrência de uma falha provocada por um mutante, por meio da observação de alguma das assertivas. A eficácia das assertivas é agora estimada pelo número de mutantes que são mortos por falha detectada por alguma assertiva. Os resultados alcançados no experimento realizado demonstraram que AEs criadas puderam matar todos os mutantes para o grupo de estruturas de dados utilizado.

3. Fundamentação teórica

Esta seção apresenta a fundamentação teórica para o entendimento deste trabalho. Ela contempla uma visão geral sobre assertivas executáveis (Subseção 3.1) e conceitos sobre teste baseado em mutantes (Subseção 3.2).

3.1. Assertivas executáveis (AEs)

As falhas que os programas apresentam geralmente são demonstradas como saídas inconsistentes, inesperadas ou resultados indesejáveis. Tais saídas fornecem aos desenvolvedores poucas informações para rastrearem a causa raiz do problema [Clarke and Rosenblum 2006]. Sem um mecanismo que possa informar o fragmento de código associado às falhas, o processo de inspeção no código requer muito esforço, sujeito a muitos erros humanos e tende a ser impreciso [Yi et.al. 2015].

Segundo Duncan e Hölzle [1998], AEs são expressões booleanas que devem ser satisfeitas caso o fragmento de código a que são associadas esteja operando corretamente. Elas são uma das mais úteis técnicas automatizadas disponíveis para detecção de falhas e fornecimentos de informações sobre pontos onde o defeito está localizado no código [Clarke and Rosenblum 2006]. As AEs são essencialmente especificações executáveis e verificáveis, viabilizam assim atuar como oráculos dinâmicos [Staa 2000].

Redigir AEs no desenvolvimento inicial do código contribui para escrever código mais correto, e estimula os desenvolvedores a pensarem no problema a ser resolvido, em vez de começar a codificar, antes que a solução seja suficientemente bem entendida [Araújo et.al. 2012]. Elas obrigam a formalização de especificações. Essa formalização, mesmo que incompleta [Akhtar and Missen 2014], reduz significativamente o volume de defeitos acidentalmente inseridos pelo programador. Observa-se também que o custo do desenvolvimento diminui em virtude da redução significativa do volume de retrabalho inútil [Hall 1990, Bowen and Hinchey 1994].

Segundo Araújo et.al. [2012] o conjunto de AEs representa entre 8 e 12% de acréscimo de linhas de código ao código original e não fazem parte da implementação de um algoritmo ou classe, portanto não participam no processamento dos algoritmos. As AEs podem ser facilmente ativadas e desativadas em sistema em produção ou desenvolvimento.

Uma das vantagens das AEs é que podem ficar ativadas em sistema em produção, podendo observar inconsistências para entradas reais em todo o ciclo de vida do software. Porém, AEs incompletas podem levar a falsos negativos e assertivas incorretas podem levar a falsos positivos [Staa 2015]. Assim, o entendimento do domínio do sistema é fundamental para qualidade das AEs criadas. Os falsos positivos são facilmente sanados. Os falsos negativos podem ser, em grande parte, sanados ainda durante os testes, já desde os testes de unidade.

3.2. Teste baseado em mutantes

Teste baseado em mutantes é uma técnica utilizada para aferir a eficácia da suíte de teste, inicialmente proposto por Demillo et.al. [1978]. Por meio de injeções de defeitos no programa original são criadas várias versões alteradas desse programa cada qual corresponde a um mutante. Os defeitos são injetados pela aplicação sistemática de operadores de mutação. Esses operadores correspondem aos defeitos típicos. Segundo Delamaro et.al. [2007], os operadores de mutação surgiram de estudos que determinavam os erros mais comuns cometidos por programadores considerando linguagens de programação específicas.

Quando um operador de mutação é aplicado ao programa original, e o programa original possui a estrutura sintática que o operador de mutação consegue modificá-lo, então essas modificações são exercitadas e um conjunto de mutantes é gerado. A cada ocorrência encontrada no programa original dessa estrutura sintática é criado um novo mutante. Cada mutante criado é sintaticamente diferente dos demais mutantes para o um mesmo programa.

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 em duas hipóteses. (i) Hipótese

do programador competente que estabelece: um programa criado por um programador competente está correto ou está próximo do correto e (ii) 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 et.al. 1978].

Para cada mutante, é aplicado o conjunto original de testes. Nos casos de testes, se algum falhar, o mutante ao qual foi testado diz-se morto. Diante disso, a suíte de teste foi capaz de observar o desvio sintático correspondente ao mutante. Se os testes passarem o mutante é considerado como vivo, pois a suíte de teste não foi capaz de observar o comportamento incorreto provocado pelo mutante. Contudo, alguns mutantes que permanecem vivos poderão ser equivalentes ao programa original. Para determinar se um mutante é equivalente se faz necessária uma análise feita pelo testador, e definir se o mutante se manteve vivo por ser realmente equivalente ao programa original ou se os testes não foram capazes de perceberem o defeito injetado. Caso os testes detectarem as falhas artificiais, assume-se que detectarão falhas reais [Delamaro et.al. 2007]. Evidentemente isso requer a geração substantiva e sistemática de mutantes.

Como exemplo de criação de um mutante, na Figura 1 é demonstrado um fragmento de código de um programa original e o seu respectivo código modificado. O operador de mutação ROR, que substitui os operadores relacionais, foi escolhido para aplicar a mutação no código original. No caso específico do exemplo o operador relacional igual (==) foi substituído por diferente (!=).

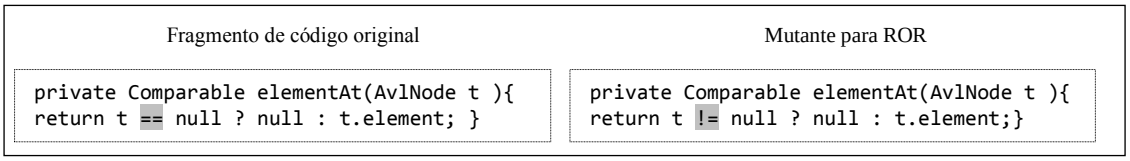

Figura 1. Exemplo de mutante criado por meio do operador de mutação ROR

Para entendimento dos termos utilizados no experimento será utilizada a seguinte terminologia: (i) *Mutante:* o programa original modificado; (ii) *Operador de mutação:* o agente que definirá qual será o tipo de transformação a que o programa original será submetido; (iii) *Mutante morto:* quando a suíte de teste consegue distinguir o mutante do seu respectivo programa original, ou seja, neste caso um ou mais testes não passaram. (iv) *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. (v) *Mutante equivalente:* quando por análise de um desenvolvedor experiente o mutante é considerado equivalente ao seu respectivo programa original.

4. Configuração do experimento

Esta seção apresenta detalhes da configuração do experimento. Ela contempla o modelo e demonstração do uso de assertivas executáveis (Subseção 4.1), métrica utilizada (Subseção 4.2) e justificativas do uso de mutantes para inserção de defeitos (Subseção 4.3).

4.1 Modelo e demonstração do uso de assertivas executáveis

No experimento realizado foram utilizadas apenas assertivas executáveis estruturais. Estas envolvem condições e diversos objetos pertencentes às classes que realizam a estrutura de dados. Essas condições devem ser sempre verdadeiras quando a estrutura

não está sendo alterada [Staa 2000]. Para confecção das AEs utilizadas no experimento foram realizados os seguintes passos: (i) 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 AEs. (ii) Na classe filha devem ser inseridas as AEs, para elas sejam chamadas por apenas um método, cria-se um método público que internamente invoca as AEs criadas. (iii) Cada AE será codificada em um método privado que retornará *false,* caso o fragmento de código em que ela estiver associada tiver alguma inconsistência ou desvio do estado computacional pretendido. A AE deverá informar qual a inconsistência observada, bem como os dados manipulados e em que ponto do código foi observado. Se nenhum defeito for observado o conjunto de AEs deve retornar *true.*

A Figura 2 demonstra um exemplo de utilização do modelo no qual uma implementação da árvore AVL é instrumentada. Nesse exemplo apenas uma AE é chamada pelo verificador. O exemplo de como a AE é implementada pode ser observado na Figura 4.

```
public class AvlTreeInstrumentada extends AvlTree {
public boolean verificador() {
if (!verificadorArvoreBinaria(root)) {return false;}
return true;
}
```

Figura 2. Exemplo do modelo de criação de AE

Para descrever as assertivas foi usado um misto de linguagens formais e português proposto por Staa [2000]. Um exemplo pode ser observado na Figura 3, no qual uma assertiva é criada para verificar se uma dada árvore mantém a propriedade de árvore binária de pesquisa. Assim, essa assertiva define que cada elemento da árvore em questão que tenha um filho à esquerda, seja menor que seu pai direto. Além disso, existindo o filho à direita, seja maior que seu pai direto. Essa forma de descrever assertiva também poderá ser usada para documentá-las.

$$\forall\, n \in \acute{a}rvore\colon\ (n \longrightarrow left\ != null) \implies ((n \longrightarrow elemento) > (n \longrightarrow left \longrightarrow elemento\ ))$$

$$\forall n \in \acute{a}rvore\colon (n \longrightarrow right != null) \implies ((n \longrightarrow elemento) < (n \longrightarrow right \longrightarrow elemento))$$

Figura 3. Assertiva para uma árvore binária de pesquisa

A Figura 4 demonstra a implementação na linguagem Java da assertiva descrita na Figura 3. Caso a propriedade de árvore binária de pesquisa seja violada (linhas seis e sete do código exemplo), uma exceção será gerada com informações relativas ao exato elemento em que foi observada a inconsistência (linhas de oito a doze do código exemplo).

```
1.  private boolean verificadorArvoreBinaria(AvlNode t) {
2.  if (t != null) {
3.  verificadorArvoreBinaria(t.left);
4.  if (t.left != null && t.right != null) {
5.  try {
6.  if ((((MyInteger) (t.left.element)).intValue())>(((MyInteger) (t.element)).intValue())
7.  && (((MyInteger) (t.right.element)).intValue())<(((MyInteger) (t.element)).intValue())){
8.  throw new IllegalStructureException();}
9.  }catch (IllegalStructureException e) { e.printStackTrace();
10. System.err.println("@ A árvore não é uma árvore de busca binária :"
11. + "filho da esquerda: " + t.left.element
12. + "filho da direita: "  + t.right.element+ "elemento pai" + t.element);} }
13. verificadorArvoreBinaria(t.right);} return true;}
```

Figura 4. AE implementada

4.2. Métrica utilizada

Segundo Delamaro et.al. [2007] um escore de mutação pode ser calculado com uma análise do número de mutantes que permaneceram vivos e mortos, sendo que esse escore de mutação deve variar entre zero e um. Quanto maior for o valor do escore de mutação, mais adequado será o conjunto de casos de teste para testar o programa. O escore de mutação originalmente criado foi adaptado para aferir a eficácia das AEs. Assim, a Figura 5 demonstra o cálculo do escore de mutação para AEs:

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

Figura 5. Escore de mutação para o conjunto de AEs

Sendo: (i) *EM(P,A):* escore de mutação de um programa P em relação ao conjunto de AEs; (ii) *DM(P,A):* número de mutantes mortos pelas AEs; (iii) *M(P):* número total de mutantes gerados a partir de programa P; (iv) *EQ(P):* número de mutantes considerados equivalentes a P;

4.3. Justificativa do uso de mutantes para inserção de defeitos

O uso de AEs não mede a eficácia da suíte de testes, uma vez que não é possível estimar-se o número de defeitos existentes. Uma das formas de criar uma estimativa é por meio do uso de mutantes. Por meio do emprego de operadores de mutação a um programa original, com apoio de uma ferramenta para testes de análise mutante, torna-se possível conhecer e gerenciar as falhas que correspondam aos mutantes. Quanto maior o número e a variedade de mutantes, maior a confiabilidade assegurada pela suíte de teste para ser capaz de identificar um número grande desses mutantes, mediante aos oráculos contidos na suíte de teste.

Quando as AEs são utilizadas e essas acusam as falhas observadas ao executar a suíte de testes, os oráculos de teste possivelmente não chegam a acusar falhas, pois as AEs já as detectaram. Então se as AEs funcionam a contento é de se esperar que elas observem o defeito antes do programa retornar para o controle do teste. Assim, deixa-se de medir a qualidade da suíte de teste, forma pela qual o teste de análise mutante originalmente foi concebido, e passa-se a medir a eficácia do conjunto de AEs em identificar as falhas provocadas pelos mutantes. Com isso, as AEs agora podem matar os mutantes, quando elas conseguem observar o erro gerado ao executar o mutante. Como, por definição da abordagem de teste baseado em mutantes, cada mutante contém um ou poucos defeitos interdependentes, a falha observada por uma AE corresponde a ter detectado o erro gerado pela alteração, portanto o mutante terá sido morto.

O uso sistemático de AE torna possível assegurar-se que, existindo algum defeito, as falhas por ele provocadas serão observadas por alguma AE. Evidentemente, isso implica a necessidade de uma política de instrumentação (inserção de assertivas) suficientemente abrangente. Neste trabalho não procuramos discutir que políticas de inserção de assertivas devem ser idealmente utilizadas. Baseamos o estudo no uso de assertivas executáveis estruturais que são capazes de verificar se as propriedades formais, invariantes estruturais, das estruturas de dados são satisfeitas.

5. Resultados obtidos com o experimento

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 estruturas de dados foram inseridas AEs como descritas na Seção 4.1. As AEs utilizadas no experimento foram criadas pelo autor principal desse trabalho e revisada pelos demais autores.

Com a finalidade de criar os mutantes para cada estrutura de dados foram utilizados todos os operadores de mutação que a ferramenta MuClipse [Smith and Williams 2007] fornece. Foi configurado na ferramenta MuClipse o tempo de 4500 ms para execução dos mutantes.

Os resultados obtidos com os testes de análise mutante para verificar a eficácia das AEs estão apresentados na Tabela 1. Nessa tabela são descritos os tipos de mutantes criados por meio dos operadores de mutação de métodos ou classe.

Tabela 1: Resultados obtidos com teste mutantes e assertivas executáveis

Estrutura de dados	Método	Classe	Total	Equivalentes	Morto	Vivo	EM
AA Tree	133	56	189	2	189	0	1,0
AVL Tree	139	16	155	6	155	0	1,0
Binary Heap	191	2	193	1	193	0	1,0
Binary Search Tree	50	5	55	1	55	0	1,0
Binomial Queue	225	7	232	0	232	0	1,0
Black Red Tree	88	88	176	5	176	0	1,0
BTree	1582	30	1612	16	1612	0	1,0
Deterministic Skip List	32	40	72	0	72	0	1,0
Fibonacci Heap	167	39	206	0	206	0	1,0
Leftist Heap	32	6	38	0	38	0	1,0
Linked List	173	87	260	12	260	0	1,0
Pair Heap	203	87	290	1	290	0	1,0
Spaly Tree	54	142	196	3	196	0	1,0
Treap	72	23	95	1	95	0	1,0

As colunas da Tabela 1 contêm as seguintes informações: A primeira coluna descreve o nome da estrutura de dados utilizada. Na segunda e terceira colunas são apresentados os mutantes oriundos de operadores de mutação sobre código de método e de classe respectivamente. A próxima coluna descreve o número total de mutantes criados que consiste no somatório dos tipos de mutantes de método e classe. Ainda na quinta coluna é informado o número de mutantes equivalentes. O critério para análise dos mutantes equivalentes utilizado neste trabalho (i) 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 e (ii) por meio da análise dos autores comparando o código original e seu respectivo mutante equivalente. Quando a saída for a mesma em ambos os casos, e quando o programa original e o mutante equivalente forem sintaticamente equivalentes, considerou-se que o mutante é equivalente ao programa original. Para isso os mutantes vivos e o programa original foram submetidos a uma entrada de cinquenta mil elementos aleatórios.

Nas próximas colunas, foram descritos o número de mutantes não equivalentes mortos, vivos e o escore de mutação respectivamente para as estruturas de dados instrumentados com AEs. Por fim a última coluna da tabela descreve os escores de mutação (EM) para programas instrumentados segundo a métrica proposta neste

trabalho na Seção 3.2. As assertivas tiveram o escore de mutação igual a 1,0. Isso demonstra que elas foram 100% eficaz para detectar as modificações que os mutantes apresentaram em relação ao programa original.

No link: http://labsoft.dcc.ufmg.br/doku.php?id=about:mutants_list podem ser visualizados arquivos utilizados no experimento, separados da seguinte forma: (i) código original da estruturas de dados e implementações das AEs, (ii) código fonte dos mutantes criados e (iii) suíte de teste.

6. Riscos à Validade

A redação de AE reduz o número de defeitos, mas não os elimina. Além disso, o conjunto de AE muitas vezes é incompleto, e pode até ser incorreto, permitindo a ocorrência de falsos negativos**.** Assim 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 de grande monta não serão observadas por elas. No experimento realizado neste trabalho o custo em tomar conhecimento das partes críticas foi nulo, porque as propriedades das estruturas de dados 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 AEs, elas podem ficar incompletas e não terem a eficácia de 100% que foi observado no experimento deste trabalho.

Ainda as AEs confeccionadas no experimento foram criadas pelo primeiro autor e validada pelos demais autores, no entanto o resultado seria mais fidedigno se as AEs fossem criadas e validadas por um grupo maior de participantes, a fim de que, detalhes da criação e custos associados fossem analisados. Além do mais, os sistemas alvo do experimento foram apenas estruturas de dados. Faz-se necessário que experimento seja replicado para outros tipos de sistemas, como exemplo sistemas de informação.

No experimento realizado, o custo do entendimento das especificações das estruturas de dados foi nulo, por suas propriedades serem definidas e provadas na literatura. Como a qualidade das AEs está diretamente relacionada ao entendimento dos requisitos do sistema de software. Não se pode afirmar se os resultados alcançados no presente trabalho poderiam ser replicados para outros experimentos, que fosse necessário o levantamento dos requisitos para elaboração das assertivas.

7. Trabalhos relacionados

Existem vários exemplos bem sucedidos na literatura que demonstram os resultados obtidos com a aplicação de AEs na indústria, tais como: projeto de roteamento de mensagens e sistema de alerta de congestionamento de veículos [Larsen et.al. 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 and Missen 2014].

Uma análise do uso de AEs em um sistema em uso foi realizada por Magalhães et.al. (2009) no qual demonstrou que vinte e duas falhas puderam ser observadas por assertivas durante testes comparadas com cinco falhas observadas por outros meios durante testes [Magalhães et.al. 2009].

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 deste trabalho ser feito sobre defeitos conhecidos e gerenciáveis. Os defeitos são conhecidos por meio dos mutantes. Sendo esses gerados por operadores de mutação pré-determinados, possibilitando assim o conhecimento prévio do total dos defeitos injetados. Também podem ser gerenciáveis por meio da funcionalidade oferecida pela ferramenta utilizada a qual foi MuClipse [Smith and Williams 2007], que possibilita a identificação do estado dos mutantes (vivos ou mortos) após serem testados. Ainda, a observação dos erros que cada mutante apresenta comparado com o programa original.

8. Conclusões e trabalhos futuros

Por meio da implementação em forma de AEs das propriedades das estruturas de dados utilizadas, as AEs puderam encontrar falhas dos mutantes antes dos oráculos associados aos casos de teste. Como demonstrado, todos os mutantes criados foram mortos pela intervenção das AEs criadas. Elas foram 100% eficazes na detecção das anomalias que os mutantes apresentaram em relação o programa original. Portanto, as falhas inseridas sistematicamente foram detectadas pelas assertivas executáveis em tempo de execução.

Por fim, as questões de pesquisa levantadas para este quase-experimento podem ser respondidas. (i) As AEs seriam um mecanismo eficaz para observar falhas? Sim, como observado para as estruturas de dados utilizados as AEs foram capazes de observar as falhas. (ii) Dado um número conhecido de falhas as AEs podem indicá-las em totalidade? Sim, todos os mutantes criados foram mortos pelas as AEs.

A partir desses resultados pôde-se replicar o experimento para um número maior de programas a serem instrumentados e mais indivíduos confeccionando as AEs. Assim, pode ser analisados dados relativos à confecção de AEs tais como: (i) custo de sua redação; (ii) capacidade e aceitação de desenvolvedores para criarem AEs e corrigir falhas apontadas por elas e (iii) tempo utilizado para confecção.

Referências

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.

Araújo, T., Wanderley, C. and von Staa, A. (2012). “An introspection mechanism to debug distributed systems”. In Software Engineering (SBES), 2012 26th Brazilian Symposium on (pp. 21-30). IEEE.

Armour, P. G. (2000). “The five orders of ignorance”. Communications of the ACM, 43(10), 17-20.

Bowen, J. P., and Hinchey, M. G. (1994, January). “Seven more myths of formal methods: Dispelling industrial prejudices”. In FME'94: Industrial Benefit of Formal Methods (pp. 105-117). Springer Berlin Heidelberg.

Brown, A., and Patterson, D. A. (2001). “To err is human”. In Proceedings of the First Workshop on evaluating and architecting system dependability (EASY’01).

Clarke, L. A., and Rosenblum, D. S. (2006). “A historical perspective on runtime assertion checking in software development.” ACM SIGSOFT Software Engineering Notes, 31(3), 25-37.

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 (1978). “Hints on Test Data Selection: Help for the Practicing Programmer”. Computer, 11(4):34–41.

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.

Feather, M. S. (1998). “Rapid application of lightweight formal methods for consistency analyses”. Software Engineering, IEEE Transactions on, 24(11), 949-959.

Hall, A. (1990). “Seven myths of formal methods”. Software, IEEE, 7(5), 11-19.

Khoshnood, S. (2015). “Constraint Solving for Diagnosing Concurrency Bugs“ (Doctoral dissertation, Virginia Tech).

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

Pullum, L. L. (2001). “Software fault tolerance techniques and implementation”. Boston London. Artech House Computing Library.

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.

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.

Staa, A.v. (2015) “Teste Automatizado 2” nota de aula PUC-Rio, 2015. Acessado em 15-06-2015 no endereço: http://www.inf.puc-rio.br/~inf1413/docs/INF1413_Aula23_TesteAutomatizado-2.pdf

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

Yi, Q., Yang, Z., Liu, J., Zhao, C., & Wang, C. (2015). “A synergistic analysis method for explaining failed regression tests”. In *International Conference on Software Engineering*.