Uma Estratégia para Geração de Oráculos para Teste de Software a partir de Especificação Formal

Edson Eustáchio Azevedo

Universidade Federal do Rio Grande do Sul

Instituto de Informática

Av. Bento Gonçalves, 9500, Agronomia, Porto Alegre – RS, Brasil

eustakio@inf.ufrgs.br

and

Ana Maria de Alencar Price

Universidade Federal do Rio Grande do Sul

Instituto de Informática

Av. Bento Gonçalves, 9500, Agronomia, Porto Alegre – RS, Brasil

anaprice@inf.ufrgs.br

Resumo

Este artigo apresenta uma estratégia para geração de oráculos para teste de software a partir de especificação formal que visa aumentar a abrangência da aplicabilidade dos geradores de oráculo. A estratégia permite gerar oráculos para implementações não derivadas diretamente da estrutura da especificação e que podem ser aplicados a casos de testes derivados de qualquer técnica de seleção de casos de teste. A estratégia apresentada é a base para a implementação da ferramenta OZJ, a qual automatiza parte do processo de geração do oráculo para classes Java a partir de especificações Object-Z.

Palavras-Chave: Oráculo, teste de software, engenharia de software, métodos formais

Abstract

This paper presents a strategy for oracle generation for software testing from formal specification that aims increase the aplicability of the oracle generators. By this estrategy is possible generate oracles for implementations that are not generated directy from the formal specification strucutre and that can be applied to any test case generation technique. The estrategy is the base to OZJ tool which automates part of the oracle generate process for class Java from Object-Z specifications.

Key words: Oracle, software testing, software engineering, formal methods.

1 Introdução

A aferição da qualidade é considerada uma tarefa essencial no ciclo de desenvolvimento de *software*. Desde o surgimento da engenharia de *software*, estudos vêm sendo realizados para desenvolver metodologias e técnicas eficientes para medir a qualidade dos sistemas em relação a seus requisitos iniciais.

Com o crescimento da complexidade do *software*, devido a grande quantidade de testes necessárias à realização prática de provas de correção através de técnicas dinâmicas, a atividade de teste mostrou-se inviável e, com isso, os estudos na área de teste passaram a procurar por técnicas e metodologias capazes de afirmar, não a correção, mas a confiabilidade do *software* em relação a seus requisitos iniciais. O desafio principal estabelecido para as pesquisas nesta área tornou-se, então, determinar estratégias que reduzissem a quantidade de casos de testes necessárias para garantir confiabilidade ao *software*.

Ao longo dos anos, muitas pesquisas sobre geração de casos de teste vêm sendo desenvolvidas e diversas metodologias e estratégias de testes vêm sendo propostas para a seleção de casos de teste. Muito embora tais metodologias e estratégias se mostrem eficientes na descoberta de erros, comumente, são necessárias quantidades relativamente grandes de casos de teste para realizar a validação do *software*. Dois casos ilustram essa afirmação: 1) Stocks através de uma combinação de estratégias de seleção de casos de teste funcionais [1,2] derivou 36 casos de teste diferentes para verificar um problema simples de classificação de triângulos pelo comprimento dos seus lados e 2) White e Coehn [3] estabeleceram que para cada decisão bi-dimensional contida no código a ser testado, para

detectar eficientemente erros de domínio são necessárias três entradas – dois pontos ON sobre a borda da representação geométrica do domínio e um ponto OFF fora da borda. Se essa estratégia for aplicada a instruções de decisão com mais de duas variáveis – bordas N-Dimensionais – são necessárias N pontos ON para cada estrutura de decisão. Logo, para aplicações minimamente complexas essas estratégias geram grandes quantidades casos de teste.

Nesse contexto, mesmo que as técnicas de seleção de casos de teste produzam valores eficientes na descoberta de erros e mesmo que ferramentas automatizem essas técnicas, a tarefa de comparação dos resultados obtidos com os resultados esperados pode se tornar demasiadamente cara e passível de erros para ser realizada manualmente. Este inconveniente justifica o estudo e o desenvolvimento de mecanismos automáticos para a comparação dos resultados do teste com os valores esperados. Tais mecanismos são conhecidos como *oráculos automáticos*.

Dessa forma, pode-se destacar duas vantagens no uso de oráculos automáticos [4]:

1) Reduzem o custo de aplicação das técnicas de teste;
2) Aumentam a confiabilidade da atividade de teste.

Um estudo sobre literatura sobre geradores de oráculos mostra que duas características marcantes são encontradas:

1) Algumas abordagens geram oráculos aplicáveis a implementações derivadas diretamente da estrutura da especificação;
2) Algumas abordagens geram oráculos para verificar o resultado de casos de teste gerados por técnicas de seleção de casos de teste específicas.

A conseqüência negativa dessas características é a limitação do uso de oráculos 1) apenas àquelas implementações desenvolvidas através de um processo de desenvolvimento baseado em especificações formais e 2) apenas aos testes derivados especificamente pelas técnicas de seleção de casos de teste que dão origem ao oráculo.

Este artigo apresenta uma estratégia para geração de oráculos para implementações de classe Java a partir de especificações escritas em Object-Z [5]. Esta estratégia leva em consideração a dependência tanto ao processo de desenvolvimento e como às técnicas de seleção de casos de teste e propõe uma abordagem mais abrangente no que tange aplicabilidade dos oráculos. Além disso, apresenta-se a ferramenta OZJ (da frase, Oráculo a partir de Object-Z para classes Java) [6,7], a qual automatiza a aplicação desta estratégia através de um processo de geração de oráculo semi-automático.

2. Trabalhos Relacionados

Uma das primeiras estratégias de geração de oráculos a partir de especificações formais foi proposta por Gannon, McMullin e Hamlet para a ferramenta DAISTS (*Data Abstraction, Implementation and Testing System*) [8]. Esta abordagem consiste em inserir instruções que verificam os axiomas definidos em especificações algébricas no código da implementação. Os axiomas da especificação são traduzidos de forma a realizar chamadas aos procedimentos da especificação. Os testes são realizados pela instanciação das variáveis livres do axioma. Quando o código resultante da tradução dos axiomas é executado, o teste é considerado bem sucedido se os dois lados do axioma retornam o mesmo resultado. Para tipos complexos, é necessário definir seus próprios axiomas para a operação de igualdade.

Hayes [9] desenvolveu outra estratégia para verificar se a especificação de tipos abstratos de dados foi corretamente implementada para linguagens de especificação baseadas em modelos. A idéia central é relativamente direta e consiste em aplicar as restrições descritas na especificação à implementação para, posteriormente, avaliar o resultado durante a execução do teste. Hayes demonstra como gerar oráculos para implementações em linguagem PASCAL a partir de especificações escritas em linguagem Z [13]. A estratégia considera que a implementação foi escrita utilizando uma estrutura de dados definida segundo os tipos da especificação. Ou seja, os tipos da implementação devem necessariamente satisfazer as características dos tipos da especificação. O oráculo é gerado manualmente através da implementação das expressões lógicas extraídas da especificação, classificadas como invariantes, pré-condições e relação entrada-saída. Para realizar o teste, cada expressão dá origem a um procedimento de verificação que é incorporado à implementação original.

Richardson [4] apresentou uma estratégia onde o oráculo é gerado em conjunto com critérios de seleção de casos de teste. Esta abordagem utiliza a especificação tanto para gerar casos de teste como para validar os resultados obtidos de uma forma genérica e não dependente da linguagem de especificação. O primeiro passo da construção do oráculo define três tipos de espaços de variáveis, um para a especificação, um para a implementação e um para o oráculo. A partir dos espaços de variáveis constrói-se mapeamentos entre eles. Comumente, considera-se que o espaço de variável da especificação e do oráculo é o mesmo. O mapeamento entre a especificação e a implementação consiste em um conjunto de pontos onde tanto a especificação como a implementação deve representar o mesmo estado. Esse conjunto de pontos compõe o *mapeamento de controle*. Além disso, define-se associações entre as variáveis que representam o estado interno da especificação e da implementação. Essas associações são chamadas *mapeamento de*

dados. A verificação ocorre pela comparação entre o estado interno da implementação e da especificação nos pontos de controle. O estado da implementação é capturado através de funções de abstração. Posteriormente, esta abordagem foi incorporada à ferramenta TAOS (*Testing with Analysis and Oracle Suport*) [10]. Com esta abordagem Richardson definiu uma estrutura básica para o mapeamento, seguida nas abordagens de O'Malley [11] e McDonald [12].

O'Malley propôs um modelo genérico para a construção de oráculos baseados em especificação, chamado EASOF (*Execution-time Analysis for Specification-based Oracle Failures*). A abordagem não descreve uma estratégia e sim um modelo genérico capaz de gerenciar múltiplas linguagens de especificação, considerando inclusive, estilos diferentes. Cada linguagem de especificação adotada pode ser usada para descrever características específicas do sistema, e para cada caso, apresenta-se formas adequadas de compor o mapeamento entre a especificação e a implementação. Outros fatores abordados são a instrumentação do código e a abrangência da verificação do oráculo.

McDonald [12] aborda a geração de oráculos para implementações orientadas a objetos em C++ a partir de especificações Object-Z [13]. A abordagem consiste em aplicar invariantes, pré e pós-condições ao código da implementação de forma semelhante à proposta em [9], tirando proveito da semelhança estrutural entre a linguagem de especificação e a linguagem de implementação. O oráculo proposto por McDonald foi projetado de forma a ser integrado à ferramenta *ClassBench Testing* [14], a qual gera *drivers* e casos de teste específicos para as classes C++ em teste. A estratégia de verificação segue o modelo proposto por Richardson [15], o qual avalia o comportamento da especificação através de um espaço de variáveis abstrato. Esta estratégia também utiliza funções de abstração para obter o estado abstrato a partir do estado concreto da implementação. A derivação de um protótipo da implementação a partir da especificação é uma característica marcante nesta abordagem, facilitando a construção das funções de abstração e automatizando a definição dos mapeamentos.

Stocks definiu uma abordagem onde é possível obter oráculos de acordo com critérios de seleção de casos de teste [1], chamada TTF (*Test Templates Framework*). A TTF é uma estratégia estruturada e formal para o teste baseado em especificação e tem como objetivo derivar casos de teste a partir de especificações escritas em linguagem Z. O TTF não é uma estratégia para geração de casos de teste e sim uma estrutura onde as estratégias de geração de casos de teste, tais como o particionamento de domínio [16], são especificadas. A partir da especificação, os TT's (*Test Templates*) são obtidos. Os TT's são expressões lógicas que descrevem conjuntos de valores de entrada, chamados VIS's (*valid input states*), os quais são usados para derivar os casos de teste. Cada VIS dá origem a um OS (*output state*) através da relação entre os estados de entrada e os correspondentes estados de saída. Dessa forma, as expressões lógicas que definem os OS's são usadas como oráculos para verificar o resultado da aplicação dos casos de teste. Vale comentar, que a abordagem de Stocks é puramente teórica e restringe-se à definição das expressões lógicas do oráculo, não entrando no mérito da construção de um sistema real.

Hörcher [17] e Mikk [18] apresentam uma abordagem para a geração automática de oráculos a partir da linguagem Z, onde a verificação do comportamento também é realizada em um espaço de variáveis abstrato. Nesta abordagem, as expressões da especificação são traduzidas para funções em linguagem C, as quais são usadas para analisar o resultado do teste. Um tópico interessante apresentado nesta abordagem é uma forma de contornar a avaliação de especificações não executáveis.

A abordagem proposta por Peters and Parnas [19] apresenta algumas variações em relação às demais. Uma diferença está no uso de uma linguagem de especificação que descreve o comportamento através de expressões tabulares, denominadas *LD-Relations* [20]. *LD-Relations* corresponde a um documento da linguagem de especificação que descreve a funcionalidade do sistema sob a forma de documentação de programas. A descrição do comportamento, propriamente dita, é realizada sobre estruturas de dados usadas tanto na especificação como na implementação, dispensando portanto, a necessidade de mapeamentos. A avaliação do resultado ocorre através de funções que representam as relações descritas na especificação.

Antoy e Hamlet [21] descrevem um oráculo aplicável a ADT's (do inglês, *Abstract Data Type)* a partir de especificações algébricas. Esta abordagem está relacionada com a abordagem descrita em [8] e, assim como a abordagem de McDonald [12], também está voltada para aplicações orientadas a objetos, embora não aborde questões relacionadas às hierarquias de classe. A estratégia proposta consiste em instrumentar a classe sob teste com funções de abstração fornecidas pelo usuário, as quais mapeiam o estado concreto no estado abstrato da classe. A verificação consiste em simular os axiomas da especificação, a fim de garantir que a implementação satisfaça a especificação. O uso de funções de abstração indica o uso de espaço de variáveis abstratas e, assim como nas demais abordagens, essas funções devem ser definidas explicitamente pelo usuário.

Cita-se ainda, Bieman e Yiu [22] que propõem um oráculo sobre uma linguagem de especificação que define pré e pós-condições aplicáveis a linguagens de programação puramente funcionais, PROSPER (*Prototypes and Specifications with Relative Types*), e Fletcher e Sajeev [23] que apresentam um oráculo para implementações em *Eiffel* [24] a partir de especificações em Object-Z, chamado OZTEST. Uma característica marcante desta última abordagem é que a avaliação do resultado do teste não é feita durante a sua execução. Os valores produzidos pelo teste são capturados e posteriormente submetidos a um oráculo independente do programa sob teste.

3 Estratégia para Geração de Oráculos

Conforme apresentado na Seção 2, a literatura sobre oráculos para teste de *software* apresenta várias abordagens. A partir dessas abordagens é possível identificar uma estrutura geral relativamente simples para o funcionamento dos oráculos. Esta estrutura se divide em três etapas: a captura do estado da implementação, a identificação do estado correspondente na especificação e, posteriormente, a comparação entre esses estados. Embora a estrutura geral seja simples, as tarefas que as compõem não são triviais. Os maiores problemas estão relacionados à necessidade de estabelecer uma correspondência entre a representação do estado da implementação e o estado esperado na especificação, ou seja, compor um mapeamento entre eles.

A estratégia mais utilizada para gerar oráculos segundo esta estrutura vem sendo a instrumentação da implementação com assertivas geradas a partir da especificação do programa. Cita-se Peters e Parnas [19] que propõem uma estratégia de instrumentação a partir de uma linguagem para documentação de programas para implementações em linguagem C; Antoy e Hamlet que partem de especificações algébricas para gerar assertivas e aplicá-las a implementações de ADT's – do inglês *abstract data type* – na linguagem C++; e McDonald e Strooper [12] que extraem assertivas de especificações formais escritas em Object-Z e as aplicam em implementações de classes em C++.

Seguindo esta linha, a abordagem para geração de oráculos adotada pelo OZJ consiste na aplicação de assertivas derivadas de predicados escritos em Object-Z a classes implementadas em Java. A instrumentação gera uma nova classe – a qual deste ponto em diante será chamada simplesmente *oráculo* – como uma cópia da classe original acrescida de instruções capazes de detectar, quando submetida ao teste, comportamentos incorretos em relação a especificação da classe.

Embora considere o mesmo principio da maioria das abordagens citadas, a abordagem adotada no OZJ objetiva ser uma estratégia mais abrangente no que diz respeito à aplicabilidade do gerador de oráculo. O termo "mais abrangente" se refere à possibilidade de utilização da estratégia em um maior número de ambientes de desenvolvimento. Essa característica não é plenamente atendida pelas abordagens apresentadas, por exemplo em [12, 18, 20] devido elas estarem intimamente ligadas ao processo de desenvolvimento de *software* baseado em especificação formal. Além disso, abordagens como as descritas em [1, 4, 16] associam a geração do oráculo ao processo de seleção de casos de teste, limitando a utilização do oráculo ao uso de técnicas de teste específicas.

Nesse contexto, a estratégia adotada para a ferramenta OZJ apresenta duas características básicas:

- independência do processo de desenvolvimento;
- independência da técnica de seleção de casos de teste.

A independência do processo de desenvolvimento envolve uma reestruturação da forma como a especificação e a implementação se relacionam. Ou seja, é necessário estabelecer restrições flexíveis ao mapeamento, a fim de permitir que a implementação sob teste não precise ter necessariamente uma estrutura idêntica à estrutura da especificação.

A independência da técnica de seleção de casos de teste está relacionada com o fato da instrumentação não alterar nem características funcionais e nem características estruturais da classe sob teste. Isso equivale a garantir que se um conjunto de casos de teste for eficaz na descoberta de erros quando aplicado à classe original, ele também deve ser eficaz quando aplicado ao oráculo.

As características funcionais são mantidas inalteradas quando se garante que os domínios de entrada do oráculo e da classe originais sejam o mesmo. Dessa forma, mudanças na interface dos métodos no oráculo em relação à classe original devem ser evitadas. Isso permite que técnicas funcionais de teste como o particionamento de equivalências [25, 26] ou a geração de casos de teste para erros de domínio [27] sejam aplicadas ao oráculo da mesma forma que seriam aplicadas à classe original.

De forma semelhante, as características estruturais não se alteram quando se garante que o grafo de fluxo de controle do oráculo seja idêntico ao grafo da classe original. Para isso é necessário que a instrumentação não introduza no oráculo instruções de decisão ou de repetição que manipulem variáveis da implementação. Assim, técnicas de geração de casos de teste estruturais [28] quando aplicadas à classe original e ao oráculo produzem o mesmo conjunto de casos de teste.

Além das características funcionais e estruturais, é importante que a instrumentação não altere o valor das variáveis de implementação, sob pena de modificar o comportamento originalmente implementado e introduzir falhas no oráculo, implicando na detecção de erros que não estão presentes na implementação original.

3.1 Etapas do processo de geração de oráculos

O processo de geração de oráculos pode ser dividido em uma série de passos segundo sugere a Figura 1.

A primeira etapa consiste no mapeamento, onde a estrutura da especificação é associada a estruturas correspondentes na implementação. A seguir, a partir das informações do mapeamento, as expressões são traduzidas da sintaxe do Object-Z para uma sintaxe do Java. Na prática, a tradução consiste em reescrever as expressões da especificação segundo uma notação de chamadas às funções que implementam os operadores relacionais e lógicos, e chamadas às operações da teoria dos conjuntos.

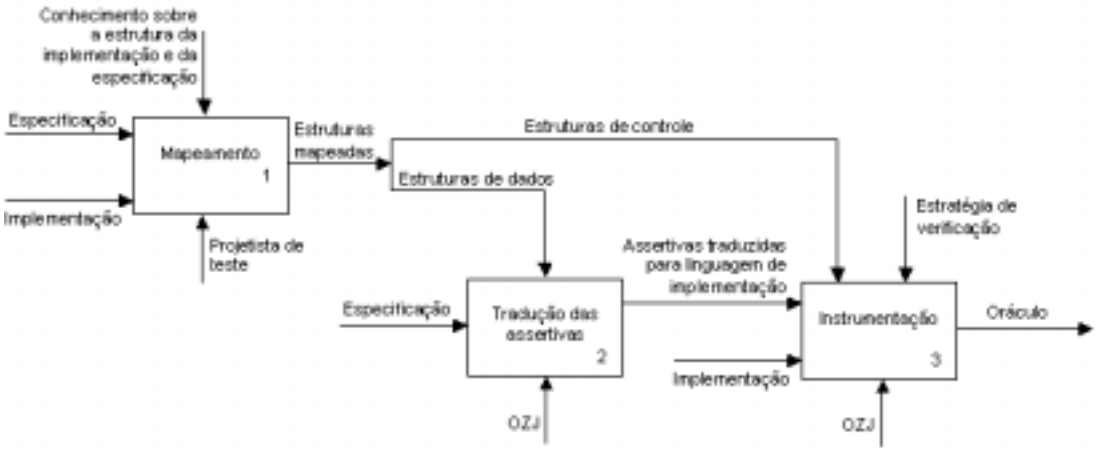

FIGURA 1 – Etapas do processo de geração de oráculos

A última etapa, realizada pelo gerador de instrumentação, consiste em inserir convenientemente instruções capazes realizar a verificação do comportamento do software. A instrumentação é regida pela estratégia de verificação e pelo mapeamento, descritos nas seções 3.3 e 3.4, respectivamente.

3.2 Estratégia de verificação do oráculo

O conceito de pré-condições e pós-condições vem sendo usado há muito tempo na verificação de programas. Hoare [29] propôs uma das abordagens mais antigas, onde pré-condições e pós-condições são definidas como expressões lógicas aplicadas às variáveis da implementação. Nesta abordagem, a verificação é feita através de provas cujo objetivo é determinar uma expressão especial, denominada pré-condição mais fraca, a partir de uma dada pós-condição. Para obter a prova, essas expressões são "posicionadas" antes e depois de cada computação no código do programa de forma a descrever seus estados nesses instantes. Através desse processo, dado um programa e sua pós-condição, determinar a expressão válida no início do programa corresponde a obter uma especificação completa. Nesse contexto, a abordagem de Hoare sugere que a posição em que as expressões são inseridas no código corresponde ao ponto onde, necessariamente, o estado alcançado pela execução da implementação deve satisfazê-las, caso contrário, considera-se que o programa está incorreto.

A verificação baseada na avaliação de pré e pós-condições vem sendo largamente utilizada entre as abordagens para construção de oráculos [20, 18, 12, 16, 4]. As diferenças em relação à abordagem de Hoare é que as abordagens citadas não objetivam determinar a correção do programa através de provas e, tanto as pré como as pós-condições são derivadas de linguagens de especificação formal. O objetivo dessas estratégias é determinar a correção da execução para um caso de teste segundo o seguinte critério: "*se durante a execução do programa a avaliação das assertivas é valida, considera-se que o programa apresentou o comportamento esperado para o caso de teste aplicado como entrada*".

Nesse contexto, traçando um paralelo com a estratégia de Hoare, para afirmar que a execução de um programa é bem sucedida para um caso de teste seria necessário aplicar pré e pós-condições a cada instrução do programa, o que é, de certa forma, pouco viável na prática. Para evitar esse inconveniente, define-se um conjunto de instruções que represente uma computação significativa, chamado unidade de execução.

Considerando o programa inteiro como uma unidade de execução, pode-se aplicar uma pré-condição no início e uma pós-condição no final. Embora esta idéia esteja correta, para programas não triviais seria difícil perceber qual falha acarretou o erro detectado. Além disso, levando em conta o axioma da antidecomposição de Weyuker [30], o qual diz que um programa adequadamente testado não implica que seus componentes estão adequadamente testados; não é possível garantir que os procedimentos internos funcionam corretamente.

Por outro lado, se procedimentos forem adotados como unidade de execução, a percepção da falha é facilitada mas, novamente, segundo Weyuker, pelo axioma da anticomposição, o qual diz que o fato de todos os componentes de um programa terem sido adequadamente testados não implica que o programa como um todo está adequadamente testado; não é possível garantir que a execução do programa produz o resultado esperado. Logo, a solução para

inferir confiabilidade ao programa é testar cada procedimento separadamente, ou seja, realizar um teste de unidade e, posteriormente, testar o programa como um todo em um teste de integração.

Aplicando esta abordagem ao teste orientado a objetos, considera-se os métodos de uma classe como unidade de execução. Portanto, a estratégia de verificação para o oráculo proposto neste trabalho consiste em aplicar pré-condições no início de cada método de classe e pós-condições ao final do mesmo. Dessa forma, testes de classe podem ser aplicados pela execução individual dos métodos, para em seguida, verificar-se o efeito da interação entre eles. Testes de *clusters* também podem ser realizados gerando-se oráculos para todas as classes do cluster.

A estratégia apresentada é válida tanto se um método é testado isoladamente, durante testes de unidade, ou em seqüência de chamadas, durante testes de *clusters*. Para ilustrar como as assertivas são verificadas, a Figura 2 apresenta em uma seqüência de chamadas a métodos, dada uma classe com um construtor, um método *A* que altera o estado da classe, um método *B* que altera o estado da classe e retorna um valor e, um método *C* que não altera o estado da classe.

Considerando-se o autômato finito da Figura 2 como uma representação dos estados internos de uma classe em uma seqüência de chamadas de métodos, onde assertivas são associadas a cada estado de acordo com a estratégia apresentada. Dessa forma, sempre que a classe atinge um estado, a pós-condição do método chamado (POS_{CONST}, POS_A, POS_B e POS_C) e a invariante (INV) de classe são avaliadas; antes de uma mudança de estado uma pré-condição (PRE_A, PRE_B, PRE_C) e a invariante é avaliada e, se houver valores de retorno, a assertiva que descreve o retorno (RET_B) é sempre avaliada no estado resultante da execução. Dessa forma, em qualquer seqüência de execução garante-se que se qualquer um dos estados alcançados violar a especificação antes ou depois da execução de um método, um erro é detectado.

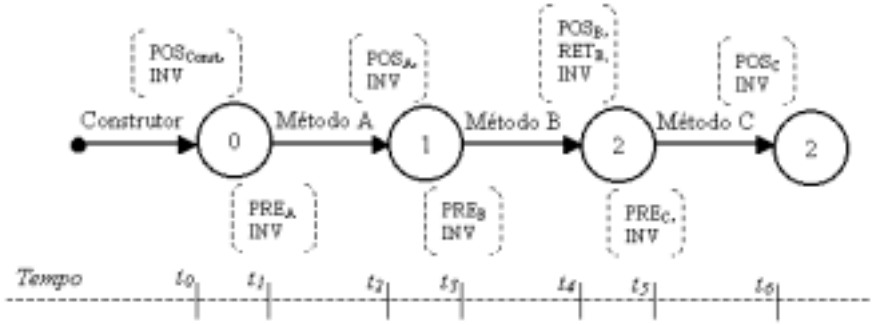

FIGURA 2 – Avaliação das assertivas em uma seqüência de chamadas de métodos

Além disso, deve-se notar que mesmo que o estado da classe não seja alterado através da execução de métodos, como por exemplo, através de quebra de encapsulamento, esta estratégia ainda é capaz de detectar violações.

3.3 Mapeamento

A composição do mapeamento entre a especificação e a implementação é a etapa central da geração de um oráculo, uma vez que as tarefas subseqüentes, como a captura do estado esperado na especificação e a comparação entre os estados, são definidas a partir de informações fornecidas por ele. Entretanto, segundo O'Malley [11], esta também é a tarefa mais difícil de ser realizada e automatizada de forma abrangente. A razão dessa dificuldade está fundamentada em dois fatores principais: a diferença de nível de abstração e a falta de homogeneidade entre especificações e implementações.

Dentro do processo de desenvolvimento de *software* baseado em especificações formais, a tarefa de reduzir a diferença de níveis de abstração entre a especificação e a implementação é chamada *refinamento* [31]. Através de refinamentos, uma especificação pode ser gradativamente reescrita sob uma forma mais concreta, ou seja, uma forma que utilize estruturas mais próximas às estruturas com as quais a implementação será construída. Entretanto, o processo de refinamento como um todo é uma atividade de alto custo para especificações complexas [32].

Muitas abordagens para geração de oráculo consideram o processo de desenvolvimento baseado em especificações formais, mas minimizam a atividade de refinamento à medida que realizam uma prototipação da implementação. Estas abordagens tomam como base a estrutura da especificação para gerar automaticamente um protótipo de implementação. Como exemplo, Antoy e Hamlet [21] derivam a assinatura dos métodos de implementações de classe em C++ segundo a estrutura sugerida por especificações algébricas e, de modo semelhante, McDonald e Stropper [12] geram protótipos de classes em C++ a partir da estrutura de especificações em Object-Z.

A principal vantagem decorrente do uso da prototipação é a possibilidade de estabelecer uma clara relação estrutural entre a implementação e a especificação. Dessa forma, o mapeamento pode ser automatizado em alguns aspectos

[15]. A abordagem de McDonald e Strooper tira proveito da prototipação em dois aspectos para compor o mapeamento:

1) por relacionar automaticamente pelos nomes, as variáveis e os métodos da implementação C++ às variáveis e esquemas em Object-Z;

2) por estabelecer uma relação direta entre os tipos primitivos da linguagem C++ e tipos da lógica de predicados e da teoria dos conjuntos, utilizados na linguagem Object-Z.

O uso de prototipação tem relação direta com o conceito de homogeneidade. O'Malley [11] sugere que a homogeneidade "*reflete a* tendência *de decompor a especificação e a implementação de uma forma similar*". No contexto apresentado, tomando a abordagem de McDonald e Strooper [12], pode-se considerar que um protótipo derivado de uma especificação possui *homogeneidade total*, uma vez que toda operação e toda a variável da especificação possui um correspondente na implementação exatamente sob a mesma estrutura.

Entretanto, é possível aplicar a estratégia de verificação, descrita na Seção 3.2, a implementações que não são totalmente homogêneas, mas que possuem uma representação equivalente para todas as variáveis e métodos, mesmo que elas não sejam definidas exatamente sob a mesma estrutura. Neste trabalho, define-se este conceito como *homogeneidade parcial*.

Para exemplificar, considera-se a Figura 3 que apresenta a especificação de um conjunto de nomes que pode conter no máximo mil caracteres. Nota-se que a implementação da representa um conjunto de seqüências de letras – variável list da especificação - através de uma variável list do tipo StringBuffer. Embora a variável *length* da especificação não esteja explicitamente representada por uma variável na implementação, seu valor correspondente pode ser obtido pela chamada ao método *list.length()* da classe *StringBuffer* da linguagem Java.

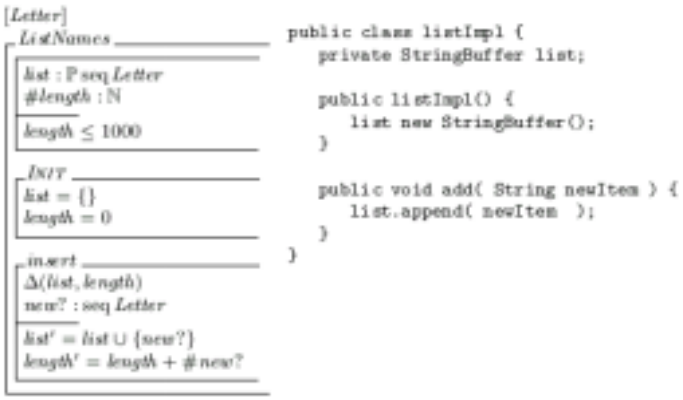

FIGURA 3 – Especificação e uma implementação de uma lista de nomes

Nesse contexto, quando se assume homogeneidade total restringe-se a aplicação dos oráculos às implementações construídas através de prototipação, não se permitindo, portanto, a utilização desses oráculos em implementações desenvolvidas segundo outros processos de desenvolvimento.

A abordagem seguida neste trabalho considera a homogeneidade parcial, portanto, aumenta-se a aplicabilidade do oráculo. Com isso, assume-se que a forma de transpor a diferença de abstração entre especificação e a implementação é a utilização do conhecimento do projetista de teste a respeito do significado semântico de cada item.

Além disso, neste trabalho como em outras abordagens citadas, o conceito do mapeamento é diretamente derivado da abordagem de Hoare [29] que introduz o conceito de função de abstração. Na prática, devido a tipagem das linguagens de programação, a função de abstração é representada por uma função capaz de converter os tipos e, por conseguinte, os valores concretos em abstratos. Como se considera homogeneidade parcial, não há restrições aos tipos que podem ser usados na linguagem de programação em função dos tipos usados na especificação. Da mesma forma não há restrições relativas à assinatura do método implementado em função da assinatura sugerida na especificação. A questão a ser interpretada pelo projetista é determinar "*que estrutura na implementação representa qual estrutura na especificação*".

Ainda sobre a composição dos mapeamentos a utilização de Object-Z como linguagem de especificação e de Java como linguagem de implementação facilita a identificação de estruturas correspondentes devido a estas duas linguagens seguirem o paradigma orientado a objetos. Para o caso específico apresentado neste trabalho, o mapeamento pode ser classificado em dois tipos. O mapeamento de controle, que determina pontos onde a especificação e a implementação devem representar o mesmo estado e o mapeamento de dados que indica estruturas de dados correspondentes [15].

De acordo com a estratégia de verificação, os itens do mapeamento de controle são determinados diretamente como o início e o fim dos métodos da implementação, cabendo ao projetista definir a qual operação da especificação cada método da implementação está associado. O mapeamento de dados, entretanto, pode não ser tão direto, tal como no exemplo da Figura 3, sendo dependente da semântica atribuída a ele pelo projetista.

Além da composição do mapeamento, cabe ao projetista a tarefa de implementação das funções de abstração. Como a abordagem adotada representa os mapeamentos como uma forma de conversão entre tipos concretos e abstratos, e não simplesmente como uma relação pré-definida baseada em nomes e em tipos, obtém-se a flexibilidade necessária para mapear especificações a implementações parcialmente homogêneas. Essa abordagem também minimiza o custo da implementação, uma vez que permite a reutilização de mapeamentos previamente implementados.

3.4 Tradução das Assertivas

A tarefa de tradução de assertivas consiste em transformar a notação da lógica de predicados e da teoria dos conjuntos, presente em Object-Z, para a forma de chamadas a métodos, sob a sintaxe Java. A representação abstrata dos tipos e operações definidas no Object-Z está implementada no pacote ZMTK.

Um exemplo simples é visto a seguir. Considerando-se a expressão

$$\#length \leq 1000$$

o processo de tradução produz como assertiva equivalente em Java

```
oracle.lessOrEqual( length.cardinality(), new ZMTKInteger(1000) );
```

onde *lessOrEqual*() é um método do oráculo e *cardinality*() é um método da classe *Set* do pacote ZMTK.

A avaliação das assertivas em Java é feita pela substituição dos valores capturados pelas funções de abstração. Os resultados são computados pela avaliação de tabelas verdade, quando a expressão de origem é uma fórmula do cálculo proposicional. Quando a expressão de origem contém definições implícitas de conjuntos, operadores universais e operadores existenciais os resultados são obtidos através algoritmos que simulam iterações sugeridas por esse tipo de construção.

3.5 Instrumentação

Tendo em mente os requisitos de praticidade, independência do processo de desenvolvimento e independência da estratégia de teste definidos neste artigo, a utilização da instrumentação de código-fonte apresenta maiores vantagens. Para se obter a independência requerida, é essencial que a captura das informações provida pela instrumentação não dependa de outros recursos além dos recursos estritamente necessários para execução da classe original.

As demais abordagens de instrumentação apresentam inconvenientes no que se refere à independência do oráculo. Uma vez que Java é usado como linguagem de programação, a instrumentação de código-objeto exige a utilização de uma JVM modificada, o que não é comum para a maioria dos ambientes de desenvolvimento. De forma semelhante, a instrumentação baseada em hardware é restritiva por exigir o uso de equipamentos específicos. O uso de instrumentação de ambiente através de instruções oriundas do sistema operacional está limitado aos recursos disponibilizados por eles. A instrumentação de ambiente através de reflexão computacional pode ser uma opção viável, uma vez que Java possui recursos nativos de reflexão. Entretanto, esta para o uso desta abordagem, é necessária a utilização de um ambiente de reflexão.

4 Funcionamento do Oráculo

Na prática, o oráculo gerado através da ferramenta OZJ é o resultado da instrumentação de uma cópia da classe sob teste com assertivas geradas a partir da especificação e do mapeamento. Portanto, durante a atividade de teste, os casos de teste devem ser efetivamente aplicados ao oráculo, e não à classe original. O resultado da realização dos testes sobre o oráculo é a geração de um histórico de execução, que contém o resultado da avaliação das assertivas, apontando violações na especificação, caso existam.

Como exemplo, tomando o oráculo gerado para a classe *StackInteger* apresentado pela Figura 4 e, considerando um teste onde os casos de teste 10 e 20 são submetidos ao método *push()*, a execução deste teste gera a seqüência de chamada a métodos ilustrada pela Figura 5.

Nesta seqüência, cada método provoca alterações no estado concreto da classe e, conseqüentemente, no estado abstrato correspondente. A seguir, a evolução da representação concreta e abstrata do estado após a execução de cada instrução é apresentada. Nota-se que no mapeamento definido entre a especificação e a implementação, a

variável *list* está mapeada para a variável *items* na implementação, o esquema INIT está mapeado para o construtor da classe *StackInteger*, e os esquemas *push* e *pop* estão mapeados, respectivamente, para os métodos *push()* e *pop()* na implementação. Os trechos de código contidos na primeira coluna das tabelas 1, 2, 3 e 4 correspondem a trechos do oráculo gerado automaticamente pela ferramenta OZJ. Maiores detalhes sobre a estrutura do código do oráculos podem se encontrados em [7]

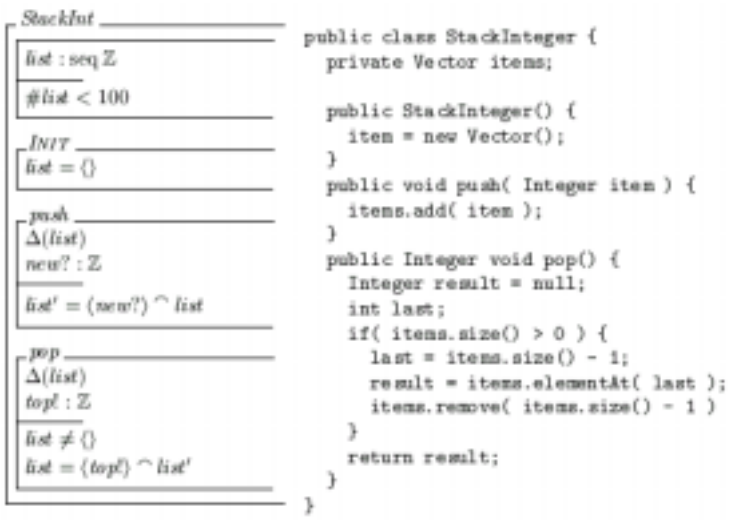

FIGURA 4 – Especificação e uma implementação de uma pilha de inteiro

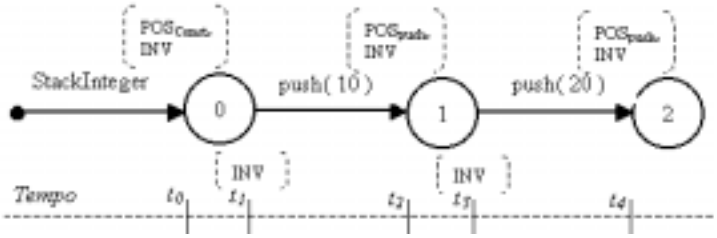

FIGURA 5 – Seqüência de chamadas a métodos da classe *StackInteger*

A Tabela 1 apresenta as alterações no estado concreto e abstrato pela execução do construtor da classe *StackInteger*.

TABELA 1 – Estado concreto e abstrato do oráculo pela execução do método *StackInteger()*

StackInteger()	Estado concreto	Estado abstrato
items = new Vector()	items ← ( )	
abstractFunction_list(concrete_items())		list ← <>
posCondition('init')		list = <> → OK
invariant('init')		#list < 100 → OK

O espaço de variáveis concreto da classe *StackInteger* é composto apenas pela variável *items*. A única instrução do construtor da classe *StackInteger* instancia *items* como um *Vector* vazio. Através da função de abstração – *abstractFunction_list* – entre *items* e *list*, a variável *list* é instanciada como uma seqüência vazia, criando o espaço de variáveis abstrato do oráculo. A seguir, a pós-condição e a invariante de classe são avaliadas sobre o espaço de variáveis abstrato.

A Tabela 2 ilustra a execução do método push(), tendo o valor "10" como entrada.

TABELA 2 – Estado concreto e abstrato do oráculo pela execução durante *push(10)*

push(10)	Estado concreto	Estado abstrato
	items ← ( )	list ← <>
	item ← 10	

abstractFunction_new_in(item)		new? ← 10
items.add(item)	items ← (10)	
abstractFunction_list_pos(concrete_items())		list' ← <10>
posCondition(‘push’)		list' = <new?> ⌢ list → OK
		list ← <10>
invariant(‘push’)		#list < 100 → OK

A primeira chamada ao método *push* introduz a variável *item* como parâmetro formal. Esta variável está mapeada para *new?* na especificação. Através da função de abstração entre *item* e *new? – abstractFunction_new_in()* – o valor 10 é atribuído à variável *new?* no espaço de variáveis abstrato. Em seguida, o valor de *item* é inserido em *items*, configurando uma mudança de estado. Para representar a mudança de estado no espaço de variáveis abstrato, a função de abstração preenche a variável *list’* com o conteúdo alterado de *items*. A seguir, a pós-condição é avaliada como verdadeira, acarretando na atualização do valor da variável *list*. Por fim, a invariante também é avaliada como verdadeira.

A Tabela 3 apresenta a execução do método push(), tendo o valor “20” como estrada.

TABELA 3 – Estado concreto e abstrato do oráculo durante a execução de *push(20)*

push(20)	Estado concreto	Estado abstrato
	items ← (10)	list ← < 10 >
	item ← 20	
abstractFunction_new_in(concrete_item())		new? ← 20
items.add(item)	items ← (10, 20)	
AbstractFunction_list_pos(concrete_items())		list' ← <10, 20>
posCondition(‘push’)		list' = <new?> ⌢ list → ERRO
invariant(‘push’)		#list < 100 → INDEFINIDO

A segunda chamada ao método *push* funciona da mesma forma, acrescentando o valor 20 tanto à variável *items* como a sua correspondente abstrata. Entretanto, a pós-condição é avaliada como falsa. Isso ocorre devido à característica de ordenamento inerente ao tipo *Sequence* da linguagem Z. Observa-se que *list’* é preenchida pelo estado alterado de *items*, que contém os valores 10 e 20, nesta ordem. De acordo com a expressão que define *list’*, ou seja o próximo estado, cada novo item deve ser inserido no início da seqüência. Portanto, o valor de *list’* deveria ser <20,10> e não <10,20>. Em decorrência do erro detectado na pós-condição, a variável *list* não foi atualizada e a invariante não pôde ser avaliada.

A descoberta de um erro é informada ao projetista através do histórico de execução. O principal objetivo do histórico de execução é mostrar a seqüência de chamadas a métodos, os erros encontrados e os valores que provocaram o erro.

O histórico de execução gerado para o exemplo apresentado é listado, na Figura 6.

Specification: StackInt	push applied to void push(Integer item)
Implementation: StackInteger	POS_CONDITION
Sequence trace	* list' = < new? > ⌢ list → false
INIT applied to StackInteger()	- list' → < 10, 20 >
POS_CONDITION → true	- list → < 10 >
INVARIANT → true	- new? → 20
push applied to void push(Integer item)	INVARIANT → undefined
POS_CONDITION → true	
INVARIANT → true	

FIGURA 6 – Exemplo de histórico de execução

O histórico de execução apresentado pela Figura 6 contém o registro da execução de casa assertiva avaliada pelo oráculo, seguindo a seqüência de chamadas de métodos executados no teste. Quando uma violação a especificação é encontrada a assertiva é apresentada em detalhes, dando ao projetista de teste informações sobre o conteúdo das variáveis envolvidas na expressão que falhou. Essa informação pode ser usada para localizar a falha que provocou o erro detectado.

5 Conclusão

Este artigo apresentou uma estratégia para geração de oráculo para teste de software a partir de especificações de classe escritas em Object-Z para classes Java. A estratégia aborda duas características que diminuem a abrangência de utilização dos oráculos gerados a partir de especificação formal: a dependência ao processo de desenvolvimento e a dependência à técnica de geração de casos de teste.

A partir dessas duas características, através do conceito de homogeneidade parcial, definiu-se uma forma de mapeamento mais flexível de forma a permitir a geração de oráculos mesmo a implementações não derivadas diretamente da estrutura da especificação. O mapeamento proposta, é baseado em tipos, ou seja, e dessa forma, para defini-lo é necessário ao projetista identificar uma relação entre as estruturas da especificação e da implementação e definir uma função de abstração entre os tipos das estruturas mapeadas.

A estratégia apresentada constitui o núcleo da ferramenta OZJ que automatiza parte do processo de geração do oráculo. A ferramenta está implementada em Java, e automatiza a etapa de tradução de assertivas, instrumentação e geração do oráculo, além de definir uma estrutura de apóio à composição do mapeamento. A partir da ferramenta, define-se uma linguagem para composição de mapeamento – chamada *OZJ-mapping* – a qual permite ao projetista representar as relações entre as estruturas mapeadas e definir suas funções de abstração.

Agradecimentos

Este trabalho foi financiado pelo CNPQ.

Referências

[1] STOCKS, P. A. **Applying Formal Methods to Software Testing**. 1993. PhD Thesis - The University of Queesland, Brisbane.

[2] STOCKS, P. A.; CARRINGTON, D. A tale of two paradigms: Formal methods and Software Testing: 8th ENGLISH Z USER MEETING, 1994. **proceedings**. Cambridge:Spinger-Verlag, 1994. p.51-68.

[3] WHITE, J. L.; COHEN, A. E. A Domain Strategy for Computer Program Testing. **IEEE Transaction on Software Engineering**, v.6, n.3, p.249-257, May 1980.

[4] RICHARDSON, D. J.; O'Malley, O.; TITTLE, C. Approaches to Specification-based Testing. **Software Engineering Notes**, n.14, v.8, p.86-96, December 1989.

[5] ROSE, G.; DUKE, R.; SMITH, G. **Object-Z: A Specification Language Advocated for Description of Standards**. Software Verification Research Centre, The University of Queensland, 1994. (Technical Report 94-45). Disponível em: <ftp://svrc.it.uq.edu.au/pub/techreports/tr94-45.ps>. Acesso em Jul. 2000.

[6] AZEVEDO, E. E., PRICE, A. M. A. Oracle Generation for Software Testing from Formal Specifications: A Practical Approach. In **Proceeding** of 30th Jornadas Argentinas de Informatica e Investigatión Cooperativa, Argentine Symposium on Software Engineering , Buenos Aires, Argentina. Setembro 2001.

[7] AZEVEDO, E. E. **OZJ – Uma Ferramenta para Geração de Oráculos para Teste de Software a partir de Especificações Formais**. 2002. Dissertação de Mestrado – Universidade Federal do Rio Grande do Sul, Porto Alegre.

[8] GANNON, J.; McMULLIN, P.; HAMLET, R. Data-Abstraction Implementation, Specification and Testing. **ACM Transaction on Programming Languages and Systems**. New York, v.3, n.3, p.211-223, July 1981.

[9] HAYES, I. J. Specification Direced Module Testing. **IEEE Transaction on Software Engineering**, n.12, v.1, p.124-133, January 1986.

[10] RICHARDSON, D. J. TAOS: Testing with Analysis and Oracle Support: INTERNATIONAL SYMPOSIUM ON SOFTWARE TESTING AND ANALYSIS, 1994. **proceedings**. Seatle:[S.n.], 1994. p. 138-153.

[11] O'MALLEY, Owen. **A Model of Specification-based Test Oracles**. 1996. PhD Thesis - University of California, Irvine.

[12] McDONALD, J.; STROOPER, P.; MURRAY, L.. Translating Object-Z Specifications to Object-Oriented Test Oracles: ASIAN-PACIFIC SOFTWARE ENGINEERING CONFERENTE, 1997. **proceedings**. Hong Kong:[S.n], 1997. p. 414-432.

[13] SPYLEY, Mike. **The Z Notation: A Reference Manual**. 2nd Ed. Hemel Hempstead, Prentice-Hall, 1992.

[14] HOFFMAN, D. M.; STROOPER, P. A. A Methodology and Framework for Automated Class Testing. Software Practice and Experience, **IEEE Transactions on Software Engineering**, n.27, v.5, p.573-597, May 1997.

[15] RICHARDSON, D. J.; O'Malley, O.; LEIF, S. Specification-based Test Oracle for Reactive Systems: 14TH INTERNATIONAL CONFERENCE ON SOFTWARE TESTING, 1992. ICSE: **proceedings**. Melborne:[S.n], 1992. p.105-118.

[16] OSTRAND, T. J.; BALCER, M. J. The Category-Partition Method for Specifying and Generating Functional Testes. **Communications of the ACM**, n.31, v.6, p.676-686, June 1988.

[17] HÖRCHER, Hans-Martin. Improving Software Testing Using Z Specifications: 9th ANNUAL Z USER MEETING, 1995. **proceeding**. Limeck: [S.n], 1995. p.152-166.

[18] MIKK, Erich. Compilation of Z Specifications into C for Automatic Test Result Evaluation: Z USER MEETING, 1995: Workshop in Computing: **proceedings**. Berlin: Springer-Verlag, 1995. p. 167-180 (Lecture Notes in Computer Science v.967)

[19] PETERS, D. K., PARNAS, D. L. Using Test Oracles Generated from Program Documentation. **IEEE Transactions on Software Engineering**, n.24, v.3, p.161-173, March 1998.

[20] PARNAS, D. L, MADEY, J. Functional Documentation for Computer Systems. **Science of Computer Programming**. v.25, n.1, p.41-61, May 1995.

[21] ANTOY, S.; HAMLET, D. Automatically Checking an Implementation against its Formal Specification. **IEEE Transactions on Software Enginerring**, New York, v.26, n.1, p.55-69, January 2000.

[22] BIEMAN, J. M.; YIN H. Design for Software Testability Using Automated Oracles: INTERNATIONAL TEST CONFERENCE, 1992. **Proceeding.** Baltimore:[S.n.]. ITC, September 1992.

[23] FLETCHER, R.; SAJEEV, A. S. M. A Framework for Testing Object Oriented Software Using Formal Specifications. ADA-EUROPE INTERNATIONAL CONFERENCE ON RELIABLE SOFTWARE TECHNOLOGIES, **Reliable Software Technologies**: proceeding. Mountreux: Springer-Verlag, 1996. p.159-170. (Lecture Notes in Computer Science, v.1008)

[24] MEYER, B. **Eiffel The Language**, Hertfordshire, Prentice-Hall, 1992.

[25] WEYUKER, E. J.; JENG, B. Analyzing Partition Testing Strategies. **IEEE Transactions of Software Engineering**, v.2, v.17, p.703-711, July 1991

[26] OSTRAND, T. J.; BALCER, M. J. The Category-Partition Method for Specifying and Generating Functional Testes. **Communications of the ACM**, n.31, v.6, p.676-686, June 1988.

[27] WHITE, J. L.; COHEN, A. E. A Domain Strategy for Computer Program Testing. **IEEE Transaction on Software Engineering**, v.6, n.3, p.249-257, May 1980.

[28] RAPPS, S.; WEYUKER, E. J. Selecting Software Test using Data Flow Information. **IEEE Transcation on Software Engineering**, n.11, v.4, p. 367-375, April 1985.

[29] HOARE, C. A. R. Proof of Correctness of Data Representations. **Acta Informatica**, v.1, n.4, p. 271-281, February 1972.

[30] WEYUKER, E. J. Axiomatizing Software Test Data Engineering, **IEEE Transactions on Software Engineering**, v.16, n.2, pp.121-128, February 1986.

[31] BERNOR, G.; GAUDEL, M.-C.; MARRE, B. Software Testing based on Formal Specifications: a Theory and a Tool. **Software Engineering Journal**, v.6, n.6, p.387-405, November 1991.

[32] RUSHBY, John. **Formal Methods and Certification of Critical Systems**. Computer Science Laboratory , SRI International, 1993. (Technical Report CSL-93-7). Disponível em <www.csl.sri.com/reports/postscript/sree-thesis.ps.gz>. Acesso em: Ago. 2000.