UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

OZJ – Uma Ferramenta para Geração de Oráculos para Teste de Software a partir de Especificação Formal

por

EDSON EUSTÁCHIO OLIVEIRA DE AZEVEDO

Dissertação submetida à avaliação, como requisito parcial para obtenção do grau de Mestre em Ciência da Computação

Profa. Dra. Ana Maria de Alencar Price
Orientadora

Porto Alegre, maio de 2002

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Azevedo, Edson Eustáchio Oliveira de

OZJ – Uma Ferramenta para Geração de Oráculos para Teste de Software a partir de Especificação Formal / por Edson Eustáchio Oliveira de Azevedo – Porto Alegre: PPGC da UFRGS, 2002.

Dissertação (Mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-graduação em Computação, Porto Alegre, BR-RS, 2002. Orientadora: Price, Ana Maria de Alencar

1. Oráculo para Teste de Software, 2. Teste de Software, 3. Especificação Formal, 4. Engenharia de Software. I. Price, Ana Maria de Alencar. II. Título.

Dedico esta dissertação a minha mãe Maria Eliete,
a minha irmã Heloisa Helena, ao meu irmão Luiz
Eustáchio e a minha namorada e incentivadora
Rejane Frozza

"A canoa vai de proa e de proa eu chego lá."

Do poema Indauê Tupã, de Paulo André Barata e Ruy Barata

Agradecimentos

Inicialmente, quero fazer um agradecimento especial a minha mãe e aos meus irmãos, pelo apoio, em todos os sentidos e em todos os momentos, durante os dois anos de estudo necessários para a realização desta dissertação.

Este trabalho contou com a contribuição de uma infinidade de pessoas que, direta ou indiretamente, me ajudaram durante o mestrado. Assim como qualquer autor de um trabalho desta natureza, é muito difícil citar nominalmente todos aqueles que merecem minha gratidão, por isso, peço desculpas àqueles não citados e peço, também, que se sintam incluídos neste agradecimento.

Ademais, gostaria de agradecer a Profa. Dra. Ana Price, antes de tudo, por ter me concedido a oportunidade de cursar o mestrado no Instituto de Informática da UFRGS, e também, pela dedicação demonstrada durante a realização deste trabalho e pelo tratamento sempre cordial e amigável dispensado a mim em todas as muitas conversas realizadas para esclarecer dúvidas e obter opiniões

Agradeço também a minha querida namorada Rejane Frozza por me transmitir segurança, a sua experiência nos momentos difíceis e por me servir de exemplo de vida, tanto pessoal como profissional.

Não posso deixar de agradecer o grande amigo Abraham Rabelo, pelo companheirismo, pelas inúmeras e produtivas conversas tanto sobre meu trabalho como sobre outros assuntos. Agradeço, também, por cada nova dica repassada e por todos os comentários que, muito freqüentemente, apontavam erros e novas direções a seguir. Não posso deixar de citar, também, o grande amigo Sandro Sawicki por ter sempre uma palavra amiga, pelos momentos agradáveis e descontraídos nos concorridos "*carreteiros*" e pelos convites para assistir aos jogos no Beira-Rio.

Por fim, quero agradecer ao professor Dr. Gustavo Campos, meu orientador de trabalho de conclusão de curso na Universidade Federal do Pará, pela inspiração e incentivo para ingressar no mestrado; agradeço a todos os professores da pós-graduação da UFRGS, e em especial ao Prof Dr. Daltro Nunes pela atenção dada às conversas sobre este trabalho; agradeço aos funcionários do Instituto de Informática e, em especial, a Eliane e ao Astrogildo, que sempre conseguem fazer com que os alunos se sintam em casa; e agradeço a todos os funcionários da biblioteca do Instituto de Informática, pelo excelente trabalho realizado e pela extrema simpatia e eficiência com que atendem e ajudam a todos os usuários da biblioteca.

Agradeço ao CNPq pelo indispensável apoio financeiro fornecido.

Muito obrigado a todos.

Sumário

Lista de Figuras

Lista de Tabelas

Resumo

A literatura sobre Teste de Software apresenta diversas estratégias e metodologias que definem critérios eficazes e automatizáveis para selecionar casos de teste capazes de detectar erros em *softwares*.

Embora eficientes na descoberta de erros, as técnicas de seleção de casos de teste exigem que uma quantidade relativamente grande de testes seja realizada para satisfazer os seus critérios. Essa característica acarreta, em parte, um alto custo na atividade de teste, uma vez que, ao fim de cada teste deve-se verificar se o comportamento do software está ou não de acordo com os seus requisitos.

Oráculo para teste de software é um mecanismo capaz de determinar se o resultado de um teste está ou não de acordo com os valores esperados. Freqüentemente, assume-se que o próprio projetista de teste é o responsável por esta tarefa. A automatização da atividade dos oráculos deu origem a oráculos automáticos, os quais são capazes de determinar o bom ou mau funcionamento do software a partir de uma fonte de informação confiável.

Ao longo dos anos, a especificação formal vêm sendo largamente utilizada como fonte de informação para oráculos automáticos. Diversas estratégias vêm propondo geradores de oráculos baseados em especificações formais. Dentre as características marcantes dessas estratégias, cita-se aquelas que são aplicáveis a implementações derivadas a partir da estrutura da especificação e aquelas que geram oráculos a partir de técnicas específicas de seleção de casos. Essas características, entretanto, limitam a aplicação abrangente dos oráculos por restringi-los tanto a implementações derivadas diretamente de especificações como ao uso de técnicas específicas de seleção de casos de teste.

Este trabalho apresenta um estudo sobre os geradores de oráculos para teste de software, identifica aspectos fundamentais que regem seu processo de construção e propõe uma estratégia que permite a geração de oráculos semi-automaticamente, mesmo para implementações não derivadas diretamente da estrutura da especificação. A estratégia proposta é, também, aplicável aos casos de teste derivados de qualquer técnica de seleção de casos de teste.

Palavras-Chaves: Oráculos para Teste de Software, Teste de Software, Especificação Formal, Engenharia de Software

TITLE: "OZJ – A TOOL FOR GENERATING ORACLES FOR SOFTWARE TESTING FROM FORMAL SPECIFICATION"

Abstract

The Software Testing literature presents general strategies that define criteria for selecting test cases able to detect errors. These strategies are efficiently employed to uncover errors and they can be processed automatically.

Although efficienties to uncover errors, the strategies for selecting test cases require an amount relatively large of cases to satisfy most of criteria. This feature contributes to the high cost of the testing activity because at the end of each individual test is necessary to verify if the software behavior meets to the software specification.

Oracle for software testing is a mechanism able to determine if the result of one test is or not is according to the expected values. Frequently, the testing engineer is responsible for this task. The automatization of the oracle activity had produced automatic oracles, which are able to determine if software behavior is good or bad from a confident information source.

In the last decade, the application formal specifications have been widely used by automatic oracles as information source. Several strategies have been proposed for oracle generators based on formal specifications. Among the main features of these estrategies, we emphasize both these ones where the implementation is directly derived from the specification structure and these ones that generate oracles from specific test case selection techniques.

This work shows a study about oracle generator for software testing, it identifies the fundamenetal aspects that guide the construction process of semi-automatic oracles and proposes an strategy to generate this kind of oracles. This strategy is adequate even if the implementation is not derived directely from specification strutucture and it can be employed for testing cases derived from any test case selection technique.

Keywords: Oracles for Software Testing, Software Testing, Formal Specification, Software Engineering

1 Introdução

A aferição da qualidade dos programas 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 do *software* 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 [STO 93, STO 94] 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 [WHI 80] estabeleceram que para cada decisão para 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 de imediato, pelo menos, duas vantagens no uso de oráculos automáticos:

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

Nesse contexto, as linguagens de especificação formal são consideradas uma boa fonte de informações funcional [WEY 80], uma vez que, por estarem fundamentadas em entidades matemáticas, permitem uma descrição abstrata e livre de ambigüidades do comportamento esperado dos sistemas [RUS 93, CLA 96]. Além disso, outra conseqüência positiva do uso de especificações formais é que uma descrição sistemática

do comportamento do *software* facilita a automatização dos processos de seleção de casos de teste [STO 93, AMM 98, OFF 99] e da geração de oráculos [ANT 2000, HOF 91].

1.1 Motivação e Objetivos

A principal motivação deste trabalho está relacionada à abrangência de utilização de geradores de oráculos. Um estudo da 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.

Nesse cenário, o objetivo principal deste trabalho é apresentar um estudo sobre o processo de geração de oráculos, propondo uma estratégia mais abrangente e que contorne as duas conseqüências negativas citadas. Dessa forma, a estratégia deve ser aplicável a implementações construídas sob qualquer processo de desenvolvimento, mesmo aqueles que não consideram especificações formais. Além disso, a estratégia deve permitir o uso de casos de teste selecionados por qualquer técnica de seleção

A partir da estratégia proposta, outro objetivo deste trabalho é implementar um protótipo de uma ferramenta, chamada *OZJ* (Gerador de Oráculos a partir de Object-Z para Java) capaz de gerar oráculos semi-automaticamente a partir de especificações formais, escritas na linguagem Object-Z para implementações de classe em Java.

A estratégia apresentada neste trabalho é projetada para atuar na geração de oráculos que podem ser aplicados em testes de aplicações orientadas a objetos durante os testes de classe e/ou testes de *clusters*.

1.2 Terminologia

Nesta seção define-se alguns termos usados ao longo desta dissertação.

Primeiramente define-se os termos *defeito*, *erro* e *falha,* os quais são bastante usados na literatura sobre teste de *software*, mas comumente têm sido empregados com significados diferentes em outros trabalhos. Neste trabalho adota-se a definição apresentada em [MAR 98]:

- *Defeito* é a evidência de um processamento incorreto.
- *Erro* é a parte do estado de um sistema que pode levar a um defeito;
- *Falha* é a causa direta de um erro;

Outro par de termos largamente utilizado na literatura sobre teste de *software* e que costuma causar confusões é: *verificação* e *validação*. Nesta dissertação, segue-se a definição da IEEE.

- *Verificação* é o processo de avaliar um sistema ou componente para determinar se o produto de uma determinada fase do processo de desenvolvimento satisfaz as condições impostas pela fase inicial. Pode ser resumido pela pergunta: "*o sistema correto está sendo construído?*".
- *Validação* é o processo de avaliar o sistema ou componente durante ou ao fim do processo de desenvolvimento para determinar se ele satisfaz os requisitos necessários. Pode ser resumido pela pergunta: "*o sistema correto foi construído?*".

Define-se, ainda, os termos *caso de teste* , *teste*, e *driver*. Embora estes termos sejam de conhecimento geral, entende-se ser conveniente apresentar o sentido com o qual eles são utilizados nesta dissertação.

- *Caso de teste* é formado pelos valores de entrada de um teste e sua(s) respectiva(s) saída(s).
- *Teste* é a atividade de aplicar um determinado caso de teste em *uma* execução do programa para verificar ou validar o comportamento do sistema.
- *Driver* é um mecanismo usado para efetivamente aplicar os casos de teste aos componentes de um sistemas ou a sistemas completos.

Devido ao escopo deste trabalho, o termo *assertiva* é muito utilizado e, a seguir, apresenta-se a sua definição segundo o glossário da IEEE.

- *Assertiva* é uma expressão lógica que especifica o estado interno ou um conjunto de condições que as variáveis devem satisfazer durante a execução de um programa.

1.3 Organização

Esta dissertação está organizada da forma descrita a seguir.

O Capítulo 2 apresenta uma introdução ao Teste de *Software*. Nesse capítulo, comenta-se sobre os objetivos do teste, apresenta-se informalmente a teoria do teste e, discorre-se, brevemente, sobre o teste orientado a objetos e sobre a aplicação de especificações formais na atividade de teste.

O Capítulo 3 trata de especificações formais. O objetivo deste capítulo e apresentar uma pequena introdução às especificações formais mostrando uma visão geral de seus pontos fortes e uma de suas formas de classificação mais usadas. Neste capítulo, também, é apresentada uma introdução informal à sintaxe e à semântica da linguagem Object-Z.

O Capitulo 4 consiste de um estudo sobre a literatura relacionada a oráculos. Neste capitulo, comenta-se sobre diversas abordagens diferentes para a construção de oráculos e identifica-se os fatores que influenciam a aplicabilidade dos geradores de oráculos e quais os aspectos fundamentais abordados pelas pesquisas anteriores.

O Capítulo 5 apresenta uma estratégia mais abrangente para a geração de oráculos para teste de *software*. A estratégia é apresentada como discurssões sobre os aspectos fundamentais para a geração de oráculos definidos no Capítulo 4. Os tópicos abordados neste capítulo são a estrutura geral da estratégia, a estrutura física do oráculo, a estratégia de verificação adotada, o mapeamento, a forma de tradução das assertivas do

oráculo, o tipo de instrumentação adotado e, por fim, um exemplo do funcionamento do oráculo proposto.

O Capítulo 6 apresenta a ferramenta OZJ segundo suas características e suas funcionalidades, define o perfil do projetista de oráculos e apresenta alguns critérios que devem ser seguidos para sua utilização. A seguir, este capítulo apresenta o projeto resumido da implementação da ferramenta.

O Capitulo 7 é composto por dois estudos de casos. O objetivo deste capítulo é ilustrar situações reais de uso dos oráculos gerados pela ferramenta OZJ e realizar uma validação preliminar das funcionalidades implementadas.

O Capítulo 8 descreve as conclusões sob forma das contribuições identificadas neste trabalho e dos aspectos que dão origem a trabalhos futuros. Além disso, algumas limitações são identificadas na estratégia proposta e na ferramenta implementada.

2 Teste de Software

Este capítulo introduz o tema Teste de *Software*, apresentando-o através de uma visão geral da teoria, das metodologias clássicas da área, das estratégias de teste voltadas para o paradigma orientado a objetos e da aplicação de métodos formais ao teste. A Seção 2.1 apresenta uma breve introdução ao teste de *software*, no qual são comentadas as etapas e abordagens clássicas do teste; a Seção 2.2 apresenta uma introdução aos estudos que formalizam a atividade de teste; a Seção 2.3 apresenta uma visão geral das técnicas de teste desenvolvidas para o paradigma orientado a objetos e na Seção 2.4 comenta-se sobre as abordagens que utilizam métodos formais como apoio ao teste de *software*.

2.1 Visão Geral do Teste de Software

Intuitivamente, quando se pensa em teste, acredita-se que a idéia imediata seja colocar um produto à prova, simulando sua utilização ou funcionamento em um ambiente real. Seguindo esse raciocínio, considerando-se que o produto em questão é um veículo, por exemplo, procura-se submetê-lo ao tráfego em diversos tipos de estradas e em condições extremas de funcionamento. Com base em um padrão de qualidade previamente determinado, os resultados obtidos nessa simulação são comparados, caracterizando os pontos fortes e os pontos fracos do objeto em teste.

Traçando-se um paralelo desse conceito intuitivo com o *software*, o teste poderia ser resumido como a atividade de executar programas e observar se seu comportamento é adequado em relação ao inicialmente projetado. Provavelmente, os primeiros programadores devem ter testado seus programas com essa idéia em mente. Entretanto, logo ficou claro que quando se trata de *software*, alguns problemas tornam essa simulação simples e aleatória pouco viável.

Com o crescimento da complexidade dos sistemas, a inviabilidade de simular o funcionamento de um *software* em todas as situações possíveis se tornou um problema de difícil solução. Esse fator se tornou mais grave quando se considera questões mais específicas relacionadas ao ambiente operacional e à usabilidade. Devido a essa constatação, observa-se que parte da literatura sobre o assunto limita o escopo de atuação dos métodos e estratégias de teste ao que diz respeito à funcionalidade do programa, ou seja, se o programa atende aos requisitos funcionais para o qual foi projetado.

Levando em consideração essa restrição de escopo, uma definição genérica dos objetivos do teste de *software* é "*medir a qualidade do software através de um exercício sistemático em um ambiente de execução controlado*" [PRA 83].

Embora o objetivo final das atividades de teste seja determinar uma medida de qualidade, o teste de *software* não se resume à verificação do código do programa. Novamente, comparando a produção de *software* com a linha de produção de um outro artefato, encarando o *software* como um produto industrial e seu ciclo de vida como uma linha de produção, a metodologia utilizada para medir a qualidade deve tirar proveito das informações geradas no decorrer da produção. Entretanto, deve-se observar que o teste de *software* não é o responsável pelo controle de qualidade, e sim pela medida da qualidade [MOS 91]. O controle de qualidade está inserido no processo de

desenvolvimento de *software*, e corresponde a realizar a análise de requisitos, a análise do sistema, o projeto e a implementação de acordo com critérios e estratégias sistemáticas. O teste de *software*, então, é aplicado no decorrer dessas fases com a intenção de determinar uma medida da qualidade com que cada passo foi realizado.

A atividade de teste pode ser visualizada em quatro passos básicos: planejamento, seleção dos casos de teste, execução do teste e avaliação do resultado. Esses passos podem ser aplicados aos artefatos produzidos ao longo do processo de desenvolvimento do *software*, sendo que cada fase do processo conta com técnicas específicas.

Quando o objeto do teste é o programa, propriamente dito, pode-se dividir o teste em três categorias: teste de unidade, teste de integração e teste de sistema. O teste de unidade é aplicado a componentes do programa que realizam funções específicas. Quando o *software* é desenvolvido em módulos, cada módulo é testado individualmente durante o teste de unidade. O teste de integração visa verificar a interação entre os diversos módulos. Nesse caso, testa-se os módulos em conjunto. O teste de sistema consiste em aplicar testes à versão do sistema produzida após a integração dos módulos.

As técnicas de seleção de casos de teste de *software* são classificadas em duas abordagens básicas, de acordo com a origem da informação usada para tal: abordagem funcional e abordagem estrutural.

Pela abordagem funcional[1], os casos de teste são derivados a partir do domínio de entrada do programa. Através dessa abordagem procura-se dividir o domínio de entrada do sistema em subconjuntos que contenham casos de teste que provoquem o mesmo comportamento quando aplicados ao programa. Dessa forma, para todo caso de teste selecionado em um subconjunto e aplicado ao programa deve-se, necessariamente, obter um resultado bem sucedido ou um resultado mal sucedido [WEY 91].

A abordagem estrutural[2] obtém dados para derivar casos de teste a partir da estrutura do programa. Nesse caso, o grafo de fluxo de controle do programa é analisado de forma a determinar os possíveis caminhos percorridos durante a execução do programa. Pela análise do grafo de fluxo de controle do programa, critérios de cobertura pré-estabelecidos determinam quais caminhos devem ser percorridos. Dessa forma, os casos de testes são derivados para provocar a execução dos caminhos necessários para satisfazer o critério de cobertura usado no teste [RAP 85].

Embora as abordagens apresentem enfoques diferentes, estudos vêm mostrando o caráter complementar do teste funcional e do teste estrutural [GOO 75, WEY 80, PRA 83, HAM 91], uma vez que o teste estrutural exercita o código implementado baseado no que o programa realmente faz e o teste funcional exercita a implementação baseado no que o programa deveria fazer.

2.2 Teoria do Teste

Estudos sobre os passos que compõem a atividade de teste vêm sendo desenvolvidos ao longo dos anos e formalizaram alguns conceitos empíricos estabelecidos por pesquisadores pioneiros do Teste de *Software*. Esta seção apresenta uma breve descrição informal da teoria do teste.

---

[1] Também chamada de teste de caixa-preta ou teste baseado em especificação.
[2] Também chamada de teste caixa-branca, teste caixa-aberta ou teste baseado em código.

Uma das primeiras tentativas de formalização do Teste de *Software* foi apresentada por Goodenough e Gerhart [GOO 75] e, posteriormente, incrementada por Weyuker e Ostrand [WEY 80]. Nessa abordagem, o programa e a especificação são vistos como funções que representam o resultado da execução do programa *P* e da avaliação da especificação *E* para uma determinada entrada, respectivamente,

$$P{:}D \rightarrow R \; e \; E{:}D \rightarrow R$$

onde *D* é o domínio formado pelo conjunto de todos os valores de entrada e *R* o contra-domínio formado pelos valores de saída. O programa estará de acordo com a especificação se o predicado

$$\forall\, d \in D \bullet P(d) = E(d)$$

for satisfeito. Tal predicado corresponde ao teste exaustivo, o qual é freqüentemente inviável na prática. Logo, uma forma mais viável de verificação se fez necessária.

O primeiro passo para definir uma estratégia de teste mais viável seria determinar um conjunto finito de dados de entrada *T*, capaz de satisfazer o predicado

$$T \subset D \;,\; \forall\, t \in T \bullet \text{success}(T) \Rightarrow \text{correct}(P)$$

onde a função *success* indica que a execução do programa foi bem sucedida no conjunto *T*, e a função *correct* indica que o programa *P* está correto em relação a sua especificação.

O conjunto *T* é chamado *conjunto ideal* e o predicado indica que o programa estará correto se o teste de todos os elementos do conjunto ideal for bem sucedido implicar que o teste de todos os elementos do domínio de entrada também seja bem sucedido. Considerando o conjunto ideal, o teste se torna mais viável por considerar um conjunto de entrada finito. Entretanto, não é trivial descobrir tal conjunto.

A fim de estabelecer uma forma de determinar o conjunto ideal, assume-se que existe um critério *C* de seleção de casos de teste. Esse critério deve satisfazer duas propriedades: ser confiável e válido.

O critério é *confiável* para um programa *P* se dado dois subconjuntos $T1, T2 \subseteq T$, tem-se

$$C(T1) \wedge C(T2) \Rightarrow (\; success(T1) \Leftrightarrow success(T2) \;)$$

ou seja, *C* é confiável se, e somente se, todos ou nenhum dos elementos selecionados por *C* são executados de forma bem sucedida pelo programa *P*. Além disso, um critério é dito *válido* se para um dado programa *P* e um conjunto *T*

$$\exists\, t \in T \bullet \neg\text{correct}(P) \Rightarrow C(T) \wedge \neg success(T)$$

ou seja, se um programa está incorreto, então *C* seleciona pelo menos um caso de teste que provoca uma execução mal sucedida. A conseqüência dessas definições consiste no teorema de Goodenough e Gerhart.

Teorema de Goodenough e Gerhart*: se um critério C é confiável e válido, então C(T) seleciona um conjunto de teste ideal T.*

Por outro lado, considerando-se que uma especificação pode ter inúmeras implementações, Weyuker e Ostrand [WEY 80] mostraram que conceitos de validade e confiabilidade, para um determinado conjunto ideal, são aplicáveis a apenas uma, dentre as diversas implementações possíveis para uma especificação, e que para realizar a

tarefa de determinar que um conjunto é confiável e válido seria necessário assumir que o programa *P* está correto.

Se forem consideradas todas as possíveis implementações representadas pela função $P:D \rightarrow R$, o que corresponde considerar *uniformemente* os conceitos de confiabilidade e validade, o resultado é

$$T = D$$

ou seja, o conjunto ideal corresponde a todos os valores do domínio de entrada, que significa o teste exaustivo. Essa constatação é sintetizada no Teorema de Weyuker e Ostrand.

Teorema de Weyuker e Ostrand*: Se C é uniformemente confiável e válido, então C(T) implica que T =D.*

Uma conseqüência importante derivada dos estudos de Weyuker e Ostrand [WEY 80] é que se nada é conhecido sobre os erros procurados, então o critério para determinar o conjunto ideal de teste seleciona todos os elementos do domínio de entrada. Sobre essa premissa, um critério de seleção mais fraco em relação ao original e capaz de selecionar subconjuntos de casos de testes que detectem a existência de erros específicos foi proposto. Ou seja, dado um programa *P* e um erro *R* que afeta a execução de uma entrada $t \in T$, quando *R* está presente em *P*, então $\neg success(T)$.

O enfoque dado pelos estudos citados é basicamente funcional, entretanto, analisando através de enfoque estrutural, pode-se traçar um paralelo entre o problema de partição de domínio e a partição de caminhos [RAP 85], uma vez que a partição de caminhos também divide o domínio de entrada em classes de entrada que representam o mesmo comportamento [PRA 83]. Por outro lado, dado o mesmo domínio de entrada, não é garantido e nem necessário que a partição de domínio e a partição de caminhos coincidam e, inclusive, segundo Weyuker, essas diferenças parecem um bom local para procurar erros.

Pela teoria apresentada, nota-se que o oráculo está representado internamente no predicado *success*, o qual deve decidir se, para cada entrada $t \in T$, o resultado da execução do programa foi ou não bem sucedido. A definição de um oráculo tal como representado pelo predicado *success* não é uma tarefa trivial e, em certos casos, pode ser impossível definir um oráculo automatizado para alguns tipos de casos de teste [BER 91].

2.3 Teste Orientado a Objetos

Quando se considera *software* orientado a objeto, as técnicas de teste tradicionais enfrentam dois grandes problemas: 1) técnicas de teste procedimentais não são adequadas ao *software* orientado a objetos, uma vez que há diferenças intrínsecas entre a abordagem procedimental e a orientação a objetos; 2) as características introduzidas pelo paradigma orientado a objetos, tais como o encapsulamento, a herança e o polimorfismo acarretam em novos problemas para a atividade de teste.

Segundo McGregor [MCG 96], alguns dos principais problemas que a orientação a objetos acarreta ao teste de *software* são:

- O encapsulamento das informações torna a avaliação do comportamento mais complicada;

- Os diversos pontos de entrada para um objeto dificultam o gerenciamento do teste;
- O polimorfismo e a ligação dinâmica aumentam as combinações de caminhos de execução possíveis.

No *software* orientado a objetos, os métodos não podem ser considerados como a unidade básica para o teste de unidade. Os métodos devem ser considerados em conjunto para determinar o comportamento da classe. Dessa forma, no paradigma orientado a objetos, a menor unidade a ser testada é a instância de uma classe, ou seja, o objeto.

As técnicas de teste clássicas são usadas em orientação a objetos com algumas alterações [FIE 89] e, ao longo dos anos, estudos vêem propondo técnicas de teste específicas ao paradigma orientado a objetos [KUN 95, BIN 94, HAR 92].

Considerando-se as características inerentes ao paradigma orientado a objetos, novas categorias de teste correspondentes ao teste de unidade, teste de integração e teste de sistemas foram definidas [SMI 92]:

- Teste de Classe

  Correspondente ao teste de unidade da programação estruturada, consiste em validar um componente isolado da classe e, posteriormente, verificar o efeito da interação entre cada um de seus componentes.

- Teste de *Cluster*

  Cluster consiste em um grupo de classes que interagem entre si e juntas implementam alguma funcionalidade. O objetivo do teste de *cluster* é verificar o efeito da interação entre as classes. Essa etapa deve acontecer após o teste individual de cada classe que compõe o *cluster*.

- Teste de Sistema

  O conceito de teste de sistema para *software* orientado a objetos é idêntico ao correspondente teste de sistema aplicado aos programas estruturados e consiste em aplicar casos de teste ao sistema para analisar sua funcionalidade como um todo.

Uma vez que as instâncias da classe são a unidade de teste, as estratégias de teste orientadas a objetos devem considerar o comportamento da classe como um todo. Nesses casos, o comportamento é representado pelos estados da classe, ou seja, os valores das variáveis internas. O teste baseado em estados se concentra no comportamento dos estados do objeto e não nas estruturas de controle ou de dados.

Um exemplo de abordagem de teste baseado em estado aplicada à orientação a objetos é o *framework* FREE (*Flattened Regular Expressions*) [BIN 94]. FREE destina-se à geração de casos de teste, o quais podem ser aplicados tanto a classes como a *clusters.* A abordagem consiste em derivar uma máquina de estados finitos onde os estados são definidos a partir da especificação da classe e as transições entre os estados são representadas por chamadas aos métodos da classe. A partir da máquina de estados, casos de teste são desenvolvidos para provocar a execução de seqüências de teste.

Outra metodologia desenvolvida de acordo com as características do paradigma orientado a objetos é o *Teste Incremental* [HAR 92]. O Teste Incremental consiste de uma técnica pra validação de classes que explora a estrutura hierárquica de conjuntos de

classes relacionadas pelo conceito da herança. Nesta abordagem a classe de maior nível na hierarquia é testada através de técnicas funcionais e estruturais e o resultado do teste é guardado. A seguir, apenas as características novas, redefinidas e os itens herdados mas afetados pelas novas características da subclasse são testadas. Essa estratégia é aplicada também sobre as interações entre as características da subclasse.

2.4 Teste Baseado em Especificação Formal

O termo teste baseado em especificação é geralmente usado na literatura para referenciar as técnicas de teste baseadas unicamente na especificação do programa, sem uso de informações estruturais. Nota-se que essas técnicas se referem tanto ao uso de especificações informais como formais. Esta seção apresenta algumas considerações à cerca do uso de especificação formal como uma ferramenta de apoio à atividade de teste.

O emprego de abordagens formais vem sendo continuamente pesquisado e, ao longo da última década, experimentou um crescimento considerável devido, em grande parte, ao desenvolvimento de linguagens de especificação formais mais poderosas e expressivas [LAM00].

A razão para o emprego de métodos formais ao teste de *software* está na possibilidade de trazer para as técnicas de teste de *software* tradicionais alguns dos pontos fortes que o uso de linguagens de especificação formal trouxeram para o processo de desenvolvimento de *software*.

Primeiramente, a propriedade de descrever o comportamento de um sistema de forma não-ambígua é vantajosa à medida que disponibiliza ao projetista de teste uma fonte de informação funcional confiável. Além disso, o poder de abstração das linguagens de especificação, em muitos casos, elimina detalhes procedimentais usados nas linguagens de programação, o que evita que a especificação seja "contaminada" por erros semelhantes aos erros da implementação [RUS 93].

As linguagens de especificação têm sintaxe e semântica formalmente definida, o que acarreta em uma estrutura sistemática na descrição do sistema, facilitando o desenvolvimento de ferramentas automáticas para o teste. Devido à semântica formal, as linguagens de especificação ainda têm disponível uma base de prova, a qual permite que sejam feitas provas de correção das propriedades descritas pela especificação.

Estudos a cerca de teste baseado em especificação apresentam duas aplicações básicas dos métodos formais ao teste de *software*: geração de casos de teste e definição de oráculos.

Quanto que tange à geração de casos de teste nota-se que nas abordagens citadas, ocorre a reutilização de métodos já segmentados, como particionamento de equivalências, relação de causa-efeito, análise do valor-limite, dentre outras, em conjunto com linguagens de especificação formal [STO 93, MEU 97, OFF 99, DON 97, SIN 97, CHA 96]. De uma forma geral, a especificação formal é utilizada como uma ferramenta de apoio, com o objetivo de sistematizar procedimentos que nos métodos originais eram deixados a cargo da habilidade do projetista, confirmando a característica de facilitar a automatização do teste.

A afirmação acima é vista em prática pelas abordagens citadas. Pode-se notar que um procedimento uniforme é adotado por essas abordagens. Primeiramente, os requisitos do *software* são formalmente especificados, quer seja por linguagens de

especificação formal como Z – visto em [STO93, SIN97, DON97] - quer seja por linguagens específicas para requisitos – utilizadas em [CHA99, OFF99]. Em seguida, se aplica algum tipo de partição no domínio de entrada descrito pela especificação - tais como DNF (*Disjuntive Normal Form*) em [SIN 97] ou o analisador de especificações ADL em [CHA 99] – gerando-se um conjunto de predicados que representam partições do domínio de entrada do sistema sob teste. A partir desses predicados, os casos de teste são derivados pela determinação de valores que os tornam válidos.

Outra forma de utilização de métodos formais na geração de casos de teste é o uso dos verificadores de modelos. Estudos mostram que os verificadores de modelos podem ser aplicados não apenas para mostrar que uma determinada propriedade não está presente no modelo, mas também para determinar quais valores provocam uma violação [CAL 96, HEI 96, AMM 98]. Dessa forma, uma estratégia comumente usada nesse tipo de abordagem consiste em construir modelos que representem situações de erro. Tirando-se proveito da característica dos verificadores para gerar contra-exemplos quando uma violação é encontrada, determina-se os valores que provocam o erro especificado para, posteriormente, gerar casos de teste.

Os métodos formais vêm sendo largamente utilizados em geradores de oráculos e, sendo esse o tópico principal desta dissertação, os oráculos e a abordagem proposta neste trabalho são discutidos em mais detalhes a partir do Capítulo 4.

2.5 Resumo do Capítulo

Neste capítulo foi apresentada uma breve introdução à teoria geral do Teste, ao teste orientado a objetos e ao uso de métodos formais como apoio ao teste. O objetivo deste capítulo é apresentar os aspectos básicos de um dos tópicos principais dessa dissertação: o Teste de *Software*. O próximo capítulo introduz o outro tópico: a especificação formal.

3 Especificação Formal

Este capítulo apresenta uma breve introdução as linguagens de especificação formal e a sua utilização no processo de verificação de *software*. O capítulo está organizado da seguinte forma. A Seção 3.1, comenta sobre alguns aspectos relativos aos requisitos básicos das linguagens de especificação; a Seção 3.2, apresenta uma classificação das linguagens de especificação e a Seção 3.3, apresenta uma introdução às características principais da linguagem Object-Z.

3.1 Visão Geral da Especificação Formal

Os métodos formais viabilizam a criação de descrições não ambíguas e abstratas do comportamento desejado dos sistemas. Portanto, técnicas formais levam a um único significado da leitura da especificação, enquanto que outras formas de descrição não formais ou semiformais, geralmente, permitem interpretações diferentes para a mesma especificação, dando margem à utilização da intuição humana [RUS 93]. A vantagem mais imediata que este tipo de especificação traz para o processo de construção de *software* é a comunicação clara e exata entre os indivíduos envolvidos e entre as diferentes fases do processo de desenvolvimento de *software*, reduzindo a ocorrência de erros ocasionados por interpretações divergentes da especificação.

A utilização da especificação formal facilita descrições abstratas devido a esse tipo de linguagem estar fundamentado em estruturas matemáticas igualmente abstratas. Pensando no processo de construção de *software*, descrições abstratas permitem que a mesma especificação possa ser utilizada como fonte de informação a diversas implementações, sem considerar linguagens de programação, ambiente operacional ou o hardware utilizado.

As descrições formais são feitas através de linguagens de especificação formal. Por especificação entende-se ser "*o processo de descrever sistemas e suas propriedades desejadas*" [CLA 96]. Assim, as linguagens de especificação descrevem as propriedades de um sistema fundamentadas em conceitos matemáticos enriquecidos com estruturas provenientes das linguagens de programação [RUS 93]. Entretanto, para uma linguagem de especificação ser considerada formal, ela deve apresentar pelo menos três características [BER 91]: sintaxe, semântica e um sistema de inferência.

- A sintaxe define exatamente o que é possível representar através da linguagem de especificação. Tal como nas linguagens de programação, a sintaxe bem definida dá ao conteúdo especificado uma estrutura sistemática e possibilita um tratamento computacional automatizado.
- Através do modelo matemático fundamenta-se o comportamento descrito na especificação. Freqüentemente, as propriedades descritas na especificação são escritas sob fórmulas de alguma lógica.
- O sistema de inferência serve para definir as deduções podem ser realizadas através da linguagem de especificação formal. Tais deduções podem ser usadas para automatizar parcialmente provas de teoremas, testes funcionais e prototipação.

Existem outros aspectos relacionados às linguagens de especificação formal além do processo de desenvolvimento e das provas de correção [GAU 95]. Além da estruturação

oriunda dos métodos formais, a formalização de um sistema requer que o especificador vá a fundo em casos particulares relacionados ao sistema especificado, especialmente em questões ligada ao tratamento de exceções. Boa parte desses casos são ocultados pela falta de estrutura das linguagens de especificação não formais.

Uma das características mais importantes dos métodos formais é a possibilidade de permitir a realização de refinamentos sucessivos na especificação. Isso corresponde a tornar a especificação original mais e mais concreta em pequenos passos. Cada passo consiste em provar que a especificação mais concreta *implementa* a especificação mais abstrata.

O passo final consiste em derivar o programa sob a sintaxe de uma linguagem de programação. Esse é o maior problema do processo de desenvolvimento de *software* baseado em especificações formais, uma vez que as linguagens de especificação e as linguagens de programação não são escritas sob a mesma sintaxe, o que torna a provas de correção demasiadamente caras para aplicações em larga escala [RUS 93].

O alto custo da provas de correção e da aplicação do processo de desenvolvimento baseado em especificação através dos refinamentos sucessivos justifica o estudo das técnicas de teste de *software* como uma alternativa viável para a verificação do *software* em larga escala. Embora, como visto no Capítulo 2, para casos minimamente complexos as provas de correção através de técnicas dinâmicas de teste sejam inviáveis, o teste de *software* é uma importante ferramenta para a aferição da qualidade de *software*.

3.2 Classificação das Linguagens de Especificação Formal

As linguagens de especificação formal são freqüentemente classificadas em dois grandes grupos: linguagens baseadas em modelos e linguagens baseadas em propriedades, embora outros estilos possam ser encontrados na literatura, tais como as especificações operacionais. No estilo operacional, a partir de ações iniciais, seqüências de ações que o sistema pode realizar são descritas. Exemplos desse estilo são as Redes de Petri [REI 85].

3.2.1 Linguagens de Especificação Baseadas em Modelos

A origem das linguagens de especificação baseadas em modelos está fundamentada nos estudos de Hoare [HOA 69, HOA 72], os quais propõem a aplicação de dois predicados lógicos sobre o estado da implementação, sendo um representando a pré-condição e outro representando a pós-condição de uma computação. Além disso, Hoare introduziu o conceito de uma função capaz de mapear as estruturas de dados da implementação para um espaço matemático. Essa função é chamada de função de abstração e esse conceito é introduzido no contexto deste trabalho no Capítulo 5.

Nas linguagens de especificação baseadas em modelos, as idéias de Hoare são combinadas com a estrutura das linguagens de programação. Dessa forma, o especificador constrói um modelo definido sobre um estado abstrato e descreve o comportamento de operações utilizando estruturas de dados e operações primitivas que a linguagem de especificação oferece.

São exemplos de linguagens de especificação baseadas em modelos VDM [JON 90], Z [SPY 92] e B [ABR 96]. Nesses casos, um programa está correto em relação à especificação se as funções que ele implementa têm o mesmo comportamento que o especificado no modelo [BER 91].

3.2.2 Linguagens de Especificação Baseados em Propriedades

Nas linguagens de especificação baseadas em propriedades, o especificador declara uma lista de funções. Assume-se que existem infinitos modelos que representem cada função. Sobre as funções, são especificadas as propriedades requeridas, sob a forma de fórmulas - comumente chamadas axiomas - as quais não precisam ser provadas. Entre os modelos definidos, apenas alguns satisfazem as propriedades descritas pelos axiomas, sendo que aqueles que não satisfazem a especificação são descartados. Nesse estilo de linguagem de especificação, um programa é considerado correto se ele implementa todas as funções requeridas e define o modelo que satisfaz todas as propriedades da especificação. Exemplos de linguagens baseadas em propriedades são as linguagens de especificação algébricas como OBJ [GOG 93].

3.3 A Linguagem Object-Z

A linguagem de especificação formal Object-Z [SMI 99] é uma extensão orientada a objetos da Linguagem Z [SPY 92]. A linguagem Z, por sua vez, sendo uma linguagem de especificação baseada em modelos, descreve o comportamento de sistemas pela representação do estado, sob o qual, um conjunto de operações atua. Em Z, estados e operações são representados através de esquemas. Através dos esquemas de estado, declara-se as variáveis e define-se os relacionamentos entre seus valores. Cada combinação dos valores das variáveis representa um estado do sistema modelado. Sobre os estados, os esquemas de operação definem o relacionamento entre o estado inicial – antes da operação – e o estado final – depois da operação.

A linguagem Object-Z impõe uma estrutura orientada a objetos às especificações Z, agrupado um conjunto de operações a uma única representação de estado. Esse agrupamento constitui uma classe e, tal como nas linguagens de programação orientadas a objetos, uma classe é um modelo do comportamento de um objeto. Dessa forma, se um objeto é dito uma instância da classe, seu estado é uma instância do esquema de estado da classe e as operações do objeto são restritas às operações definidas para a classe. As variáveis de estado e as operações são normalmente referenciadas como *características* da classe.

Especificações Object-Z, através do conceito de classe, são adequadas para descrever partes isoladas de um sistema. Classes complexas podem ser especificadas através de herança e instanciação. Por outro lado, a especificação de um sistema completo pode exigir a definição de várias classes. A interação entre as diversas classes de um sistema é representada em Object-Z através do mecanismo de agregação.

A seguir, apresenta-se, resumidamente e informalmente, a estrutura sintática da linguagem Object-Z. Maiores detalhes sobre a sintaxe e semântica da linguagem Object-Z podem ser encontradas em [SMI 99, ROS 94].

3.3.1 Classes

A estrutura sintática de um esquema de classe em Object-Z é um retângulo aberto na extremidade direita e rotulado pelo nome da classe na borda superior esquerda, tal como ilustra a Figura 3.1. Dentro do esquema define-se os componentes da classe. São eles: lista de visibilidade, lista de superclasses, definição de tipos e constantes, o esquema de estado, o esquema de inicialização, os esquemas de operação e histórico. Opcionalmente

o nome da classe pode ser acompanhado de uma lista de parâmetros para a definição de classes parametrizadas.

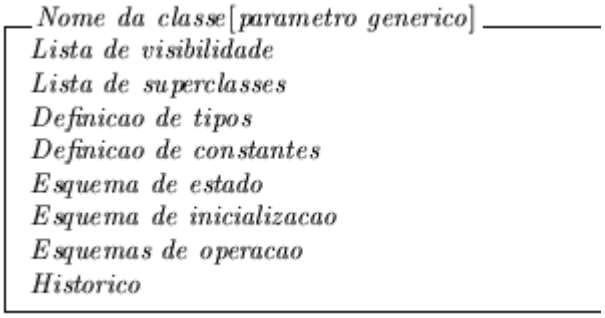


FIGURA 3.1 – Estrutura geral de uma especificação Object-Z

- Lista de visibilidade

  Quando definida, a lista de visibilidade determina quais características são visíveis fora da classe. Quando essa lista é omitida, todas as características são visíveis. Nota-se que a lista de visibilidade não é herdada e deve ser definida em todas as classes, sempre que a alguma característica deva ser ocultada.

- Lista de superclasses

  Define as superclasses das quais uma classe herda características.

- Definição de tipos

  Define tipos primitivos de usuário ou novos tipos pela composição de tipos primitivos da linguagem Z.

- Definição de constantes

  Define variáveis de estados que não podem ser alteradas assim como predicados que impõem restrições a essas variáveis.

- Esquema de estado

  O esquema de estado é semelhante ao esquema de estado da Linguagem Z, porém, sem nome. As variáveis declaradas nesse esquema são as variáveis que representam o estado do sistema e os predicados representam condições invariantes. Os predicados definidos para as constantes também são considerados invariantes.

- Esquema de inicialização

  Esse esquema é rotulado pela palavra-chave *INIT* e define o estado inicial da classe.

- Esquemas de operação

  Os esquemas de operação definem a relação entre o estado inicial e o estado final de uma operação da classe e/ou definem os valores de saída fornecidos por essa operação. Os esquemas que alteram o estado devem conter uma lista delta (Δ), a qual indica todas as variáveis que têm seus valores alterados. O uso do delta em Object-Z é diferente do uso em Z, uma vez que em Z o delta é aplicado a todas as variáveis do esquema de estado.

- Histórico

  Contem predicados que definem propriedades sobre comportamentos futuros e passados dos objetos na forma de lógica temporal.

3.3.2 Herança

O conceito de herança está presente em Object-Z de forma semelhante às linguagens de programação orientadas a objetos, ou seja, um mecanismo para construção incremental de especificações.

O mecanismo de herança funciona para tipos e constantes como uma sobreposição das definições da classe definida, sobre as definições da superclasse. O mesmo processo se aplica aos esquemas. Qualquer característica que tenha o mesmo nome na classe definida e na superclasse é sobreposta pela definição mais nova. Quando se deseja que a sobreposição não ocorra, pode-se renomear as características da superclasse.

3.3.3 Exemplo de Especificação Object-Z

Para ilustrar as características das especificações Object-Z apresenta-se um exemplo de descrição do comportamento de vários tipos de pilhas.

Primeiramente, considera-se a especificação do comportamento de uma pilha sobre um tipo genérico, tal como ilustra a Figura 3.2.

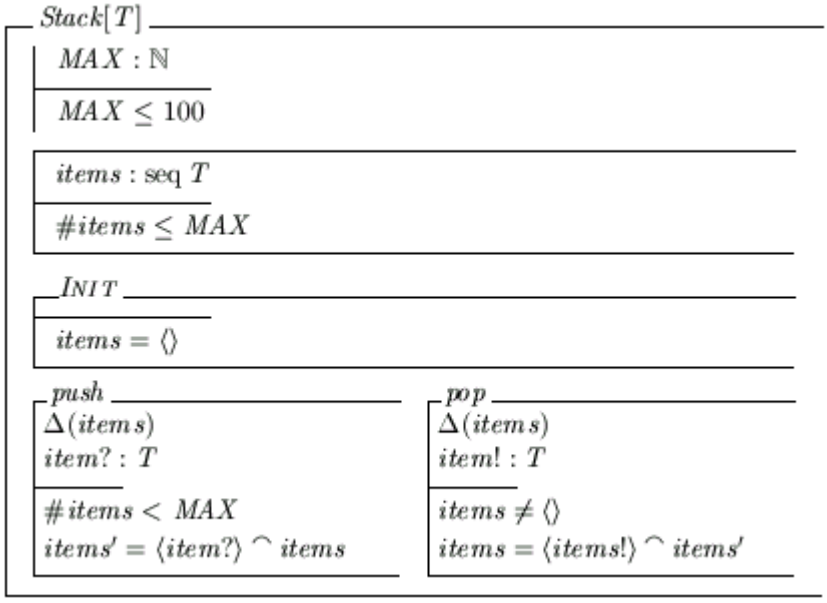


FIGURA 3.2 – Especificação de uma pilha para um tipo genérico

A especificação da Figura 3.2 descreve o comportamento de uma pilha de um tipo genérico *T*. Através do esquema de declaração de constantes, define-se que a variável *MAX* não pode ter um valor maior que 100. Nota-se que objetos diferentes podem ter valores diferentes para a constante *MAX*.

O esquema de estado declara a variável *items* como uma seqüência de elementos *T* e define uma invariante limitando o número de elementos a um valor menor ou igual ao valor da constante *MAX*. O esquema de inicialização define que a pilha sempre iniciará vazia.

A especificação *Stack* declara duas operações: *push* e *pop*. A operação *push* descreve a inserção de um novo elemento - variável de entrada *item?* - no topo da pilha. A pré-condição de *push* determina que a quantidade de elementos da lista deve ser menor que o estipulado em *max* e, caso seja satisfeita a pré-condição, *items'* deve ser igual à variável *items* concatenada com o novo elemento *item?*.

A operação *pop* descreve o comportamento da retirada de um elemento do topo da pilha. A pré-condição de *pop* determina que a operação não pode ocorrer se a lista está vazia, e se satisfeita a pré-condição, o novo estado, representado através de *items'*, é descrito como a lista *items* sem o elemento que está no topo, o qual é representado pela variável de saída *item!*.

Nota-se que Object-Z seque a notação da linguagem Z para variáveis. Dessa forma, representa-se variáveis de estado e constantes por identificadores alfanuméricos, variáveis que denotam o próximo estado, variáveis de entrada e variáveis de saída são representados por identificadores seguidos de "'", "?" e "!", respectivamente.

A partir da classe genérica *Stack*, pode-se definir pilhas para qualquer tipo. A Figura 3.3 ilustra a definição de uma pilha de naturais.

$$Stack[\mathbb{N}][nats/items, nat?/item?, nat!/item!]$$

FIGURA 3.3 – Definição de uma classe a partir de uma classe genérica

A classe definida na Figura 3.3 tem o mesmo comportamento da classe *Stack*, porém, é capaz de manipular apenas números naturais. Note que na definição da nova classe, os nomes das variáveis de estado e das variáveis de entrada foram renomeados. Assim, referências às variáveis de estado da classe definida na Figura 3.3, devem ser feitas à variável *nats* e não à *items*.

O mecanismo de herança presente em Object-Z é ilustrado pela Figura 3.4. Define-se uma pilha que permite, além das operações de inserção e retirada, recuperar qualquer elemento através do seu índice.

Em outro exemplo visto na Figura 3.4, a classe *IndexedStack* é uma subclasse de *Stack*, portanto, herda suas características e acrescenta a variável de estado *index* e as operações *setIndex* e *indexedPeek* à especificação. A variável *index* deve sempre ter um valor contido no domínio da seqüência *items*, conforme determina a invariante da classe *IndexedStack*, e seu valor determina a posição cujo valor pode ser recuperado na pilha.

Tomando como exemplo a operação *push*, o comportamento descrito na superclasse continua valendo e, na classe *IndexedStack*, acrescenta-se o controle da variável *index*. Na operação *push*, *index* deve assumir o valor 1 se a pilha não contiver elementos.

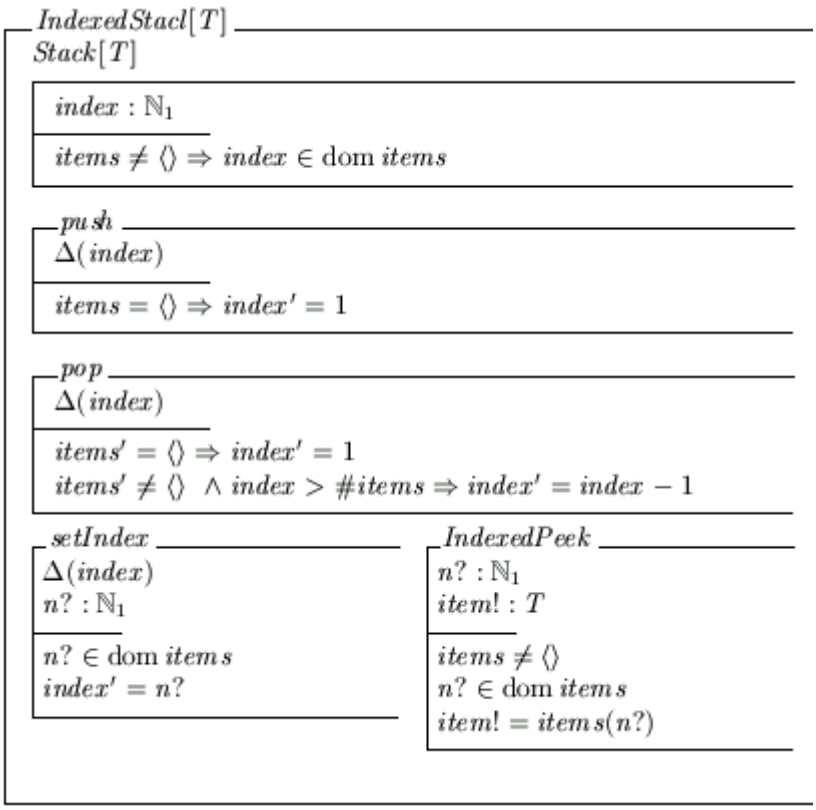


FIGURA 3.4 – Especificação IndexedStack

Para ilustrar o mecanismo de herança em Object-Z, apresenta-se na Figura 3.5 a especificação expandida do esquema push da classe *IndexedStack*.

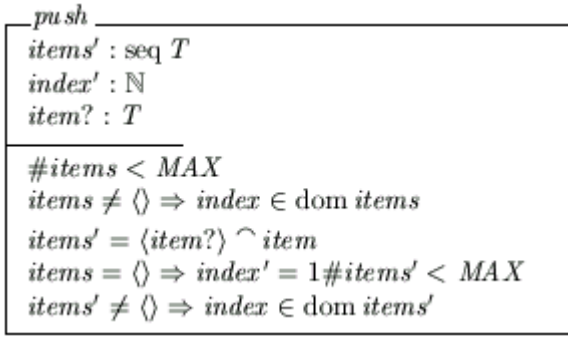


FIGURA 3.5 – Especificação completa do esquema *push* da classe *IndexedStack*

O esquema apresentado na Figura 3.5 ilustra o resultado da redefinição da operação *push* na classe IndexedStack. Primeiramente, a lista *Delta* foi substituída pelo seu significado literal, ou seja, a declaração das variáveis que representam o estado posterior a operação. São declaradas variáveis correspondentes, tanto à variável *items*, herdadas da super-classe, como à variável *index*, definida em *IndexedStack*. A seguir, os predicados invariantes são definidos sobre o estado inicial e sobre o estado posterior, e por fim o predicado que define o comportamento da superclasse é declarado após o predicado que define o comportamento de *push* em *IndexedStack*.

3.4 Resumo do Capítulo

Neste capítulo foram introduzidos alguns aspectos básicos sobre especificação formal e as estruturas básicas da Linguagem Object-Z. Este capítulo e o Capítulo 2 apresentaram, resumidamente, dois tópicos importantes deste trabalho, respectivamente, a Especificação Formal e o Teste de *Software*. O próximo capitulo aborda o assunto central desta dissertação: os oráculos para teste de *software*.

4 Oráculos para Teste de Software

Este capítulo apresenta uma visão geral da literatura especializada em oráculos para teste de *software* e está organizado da seguinte forma: a Seção 4.1 apresenta uma classificação para os oráculos; a Seção 4.2 descreve abordagens para a geração de oráculos, destacando uma estrutura geral a todas elas, e a Seção 4.3 relaciona alguns aspectos que influenciam as estratégias de definição dos oráculos.

4.1 Classificação dos Oráculos

Oráculo para teste de *software* é um dispositivo automático ou não, capaz de determinar se a execução de um teste é bem sucedida [STO 93]. Logo, toda a atividade de teste necessariamente requer um oráculo para avaliar o seu resultado.

Hoffman [HOF 91] descreve várias classes de oráculos segundo a forma usada para realizar a verificação. São elas:

- Comparação manual de cada saída após a execução do teste.
- Geração manual de cada saída, a qual é comparada automaticamente com o resultado do teste.
- Geração automática de pares $<x,y>$, onde y é a saída correta da entrada x.
- Geração automática da saída correta y para qualquer entrada x.
- Programas que determinam a correção de qualquer par de entrada e saída $<x,y>$.

Comumente, o próprio projetista de teste assume o papel do oráculo, caracterizando um *oráculo manual*. Essa abordagem, entretanto, é passível de erros, demorada e enfadonha, particularmente, quando se consideram grandes quantidades de casos de teste. Outro tipo de oráculo manual requer que as saídas esperadas sejam descritas pelo projetista, o que não é facilmente aplicado em sistemas que geram muitas saídas.

O uso de dispositivos automáticos como oráculo é importante para tornar a atividade de teste mais eficiente e confiável. Nesse caso, tem-se os *oráculos automáticos*. Os dois primeiros tipos de oráculos automáticos relacionam um valor de saída para cada entrada. Na prática, estes oráculos realizam uma computação alternativa ao programa sob teste, o que, em muitos casos, pode não ser possível. Na última classe estão os oráculos que verificam a correção de entrada e saída. Estes oráculos dispensam a computação alternativa dos valores de saída e utilizam restrições ao estado interno e externo do programa para inferir a correção da execução do teste.

4.2 Abordagens para geração de Oráculos

A comunidade científica tem apresentado várias abordagens para geração de oráculos. Devido à necessidade de uma fonte de informação estruturada, o uso de especificação formal é uma alternativa largamente utilizada. São apresentadas nesta seção as abordagens [GAN 81, HAY 86, RIC 92, OMA 96, MCD 97, STO 93, HOR 95, PET 98, ANT 2000, BIE 92, FLE 96].

4.2.1 Descrição das Abordagens

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*) [GAN 81]. 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 [HAY 86] 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 [SPY 92]. 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 [RIC 92] 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*) [RIC 94]. Com esta abordagem Richardson definiu uma estrutura básica para o mapeamento, seguida nas abordagens de O'Malley [OMA 96] e McDonald [MCD 97].

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 [MCD 97] aborda a geração de oráculos para implementações orientadas a objetos em C++ a partir de especificações Object-Z [ROS 94]. A abordagem consiste em aplicar invariantes, pré e pós-condições ao código da implementação de forma semelhante à proposta em [HAY 86], 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* [HOF 97], 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 [RIC 92], 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 [STO 93], 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 [OST 88], 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 [HOR 95] e Mikk [MIK 95] 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 [MIK 95].

A abordagem proposta por Peters and Parnas [PET 98] 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* [PAR 95]. *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 [ANT 2000] 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 [GAN 81] e, assim como a abordagem de McDonald [MCD 97], 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 [BIE 92] 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 [FLE 96] que apresentam um oráculo para implementações em *Eiffel* [MEY 92] 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.

4.2.2 Estrutura Geral dos Oráculos

Pela análise das diversas abordagens apresentadas na seção 4.2.1, é possível determinar uma estrutura genérica que rege o funcionamento dos oráculos para teste de *software* a partir de especificações formais. A Figura 4.1 ilustra os artefatos e as etapas que compõem esta estrutura através de um diagrama SADT. Destaca-se dois artefatos essenciais: a especificação[3] e a implementação. Quatro etapas estão presentes em todas as abordagens: mapeamento, captura dos resultados do teste, comparação dos resultados e a resposta do oráculo.

Através da notação SADT descreve-se as etapas como as ações do diagrama, as quais são representadas pelos retângulos. As entradas e as saídas são representadas pelas flechas à esquerda e à direita de cada ação, respectivamente. A fechas abaixo das ações indicam o mecanismo pelo qual a ação se realiza e a fechas superiores representam informações de controle para realização da ação.

A estrutura geral é válida para todas as abordagens citadas na seção 4.2.1, com exceção da abordagem de Stocks, uma vez que esta é restrita à definição lógica do oráculo e não considera fatores relacionados com a linguagem de implementação e a sua real execução. As demais abordagens seguem a estrutura da Figura 4.1, variando na forma como cada etapa é tratada.

Pela Figura 4.1 nota-se que o mapeamento, quando existe, é sempre realizado pelo projetista do teste e a informação gerada por nesta etapa serve como dado de controle a todas as demais etapas. Além disso, nota-se que o mecanismo básico usado pelos oráculos para interagir com a implementação é a instrumentação de código.

No que diz respeito ao estilo da linguagem de especificação observa-se que a maioria utiliza linguagens baseadas em modelo, sendo a linguagem Z e sua extensão

[3] Este trabalho restringe-se às abordagens que utilizam especificações formais como fonte de informação, embora existam abordagens baseadas em especificações não formais.

orientada a objetos - Object-Z, usada por quase todos que adotam este estilo. A exceção é a abordagem de Peters e Parnas. Nota-se ainda que os oráculos citados são preferencialmente voltados às linguagens de programação imperativas. Biemar foge a regra e utiliza uma linguagem funcional. Fletcher, McDonald e Antoy se detêm ao paradigma orientado a objetos.

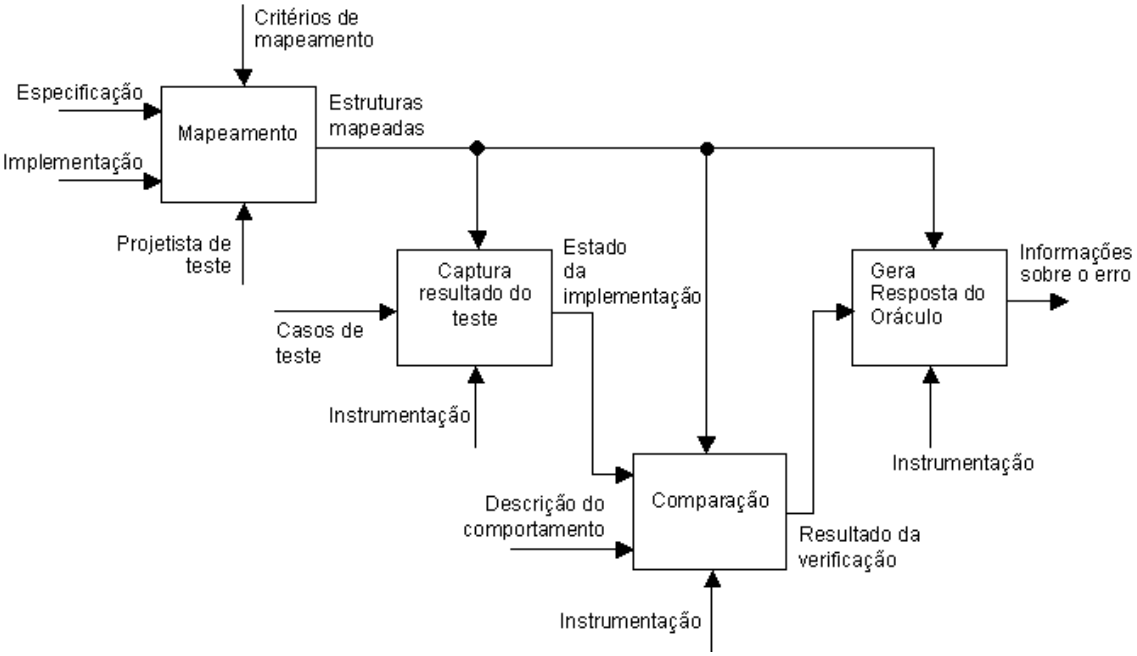


FIGURA 4.1 – Estrutura geral do funcionamento dos oráculos

O mapeamento representa a forma como a especificação e a implementação se relacionam. Pode-se identificar duas formas de relacionamento: o mapeamento implícito e o mapeamento explícito.

Quando não é necessário relacionar as estruturas da especificação às estruturas da implementação, tem-se casos de mapeamento implícito. As abordagens de Gannon, Hayes e Peters são exemplos típicos, onde a especificação e a implementação usam as mesmas estruturas de dados. Outro exemplo de mapeamento implícito é visto na abordagem de McDonald, na qual, embora a especificação não utilize estruturas da implementação, a relação entre elas é definida diretamente pelos nomes das variáveis e das operações. Nesse caso, a implementação é obtida através de prototipação da estrutura da especificação.

Quando a relação entre a especificação e a implementação não é obtida diretamente, faz-se necessária à definição explícita do mapeamento. Três abordagens exemplificam este caso. Richardson e Antoy definem o mapeamento através de funções de abstração e Fletcher define pares de correspondência entre as estruturas de dados e as estruturas de controle.

A captura dos resultados dos testes é realizada através de instrumentação por todos os autores. Uma pequena variação nesse tópico ocorre na forma como a instrumentação é feita, onde se nota duas alternativas: instrumentação de código-fonte e a instrumentação de código-objeto. Pode-se considerar que Gannon [GAN 81] adotou instrumentação de código-objeto, uma vez que na sua abordagem cabe ao compilador criar as chamadas às instruções que capturam as informações e realizam a verificação. As demais abordagens adotam a instrumentação de código-fonte, o qual é alterado antes da compilação.

A etapa de comparação de resultados pode ser analisada sob dois aspectos: a forma de tradução das expressões que definem o oráculo e como essas expressões são avaliadas. Hayes, Bieman e McDonald realizam traduções manuais. Gannon, Hörcher, Fletcher, Peters e Antoy empregam processos automatizados na tradução. Nesse tópico, Hörcher, em conjunto com Mikk [MIK 95] apresenta uma estratégia para contornar o problema das assertivas não executáveis.

A forma de avaliação das expressões pode ser classificada segundo propôs Richardson [RIC 92] ao definir espaços de variáveis diferentes para a especificação, para a implementação e para o oráculo. Dessa forma, Gannon, Hayes, Bieman e Peters realizam a avaliação das assertivas em um espaço de variáveis concretas, usando diretamente as variáveis da implementação. Richardson, Hörcher, McDonald e Antoy usam um espaço de variáveis abstratas, criando variáveis que representem o estado da especificação. Nota-se que todas as abordagens que adotam essa alternativa usam funções de abstração para gerar os valores abstratos das variáveis. A abordagem de Fletcher, por usa vez, usa o conceito de espaço de variáveis do oráculo, uma vez que os resultados não são avaliados durante a execução do teste.

Outro aspecto apresentado na estrutura da Figura 4.1 é a resposta do oráculo. Este aspecto é importante, uma vez que pode auxiliar a busca pela solução da falha que provocou o erro detectado pelo oráculo. Entretanto, a literatura que descreve as abordagens citadas não apresenta detalhes mais aprofundados sobre esse assunto.

A Tabela 4.1 resume o enfoque dado a cada abordagem em relação às etapas do funcionamento dos oráculos descritas nesta seção.

TABELA 4.1 – Etapas genéricas do funcionamento dos oráculos

	Artefatos		Mapeamento	Captura do resultado do teste	Comparação dos resultados		Resposta do oráculo
	Estilo da linguagem de especificação	Estilo da linguagem de implementação	Forma de relacionamento	Instrumentação	Forma de tradução das assertivas	Forma de avaliação das assertivas	Informação apresentada
Gannon, McMullin e Hamlet	Algébrico	Imperativo	Implícito	Código-Objeto	Automática	Variáveis concretas	Sucesso ou falha
Hayes	Modelo	Imperativo	Implícito	Código-fonte	Manual	Variáveis concretas	Sucesso ou falha
Stocks	Modelo						Sucesso ou falha
Bieman	Propriedades	Funcional	Implícito	Código-fonte	Manual	Variáveis concretas	Sucesso ou falha
Richardson	Propriedades, Modelos	Imperativo	Explícito	Código-fonte		Variáveis abstratas	Sucesso ou falha
Hörcher	Modelo	Imperativo	Explícito	Código-fonte	Automática	Variáveis abstratas	Sucesso ou falha
Fletcher e Sajeev	Modelo	Orientado a Objetos	Explícito	Código-fonte	Automática	Variáveis específicas do oráculo	Sucesso ou falha
McDonald	Modelo	Orientado a Objetos	Implícito	Código-fonte	Manual	Variáveis abstratas	Sucesso ou falha
Peters e Parnas	Modelo	Imperativo	Implícito	Código-fonte	Automática	Variáveis concretas	Sucesso ou falha
Antoy e Hamlet	Algébrico	Orientado a Objetos	Explícito	Código-fonte	Automática	Variáveis abstratas	Sucesso ou falha

4.3 Considerações sobre as Abordagens para a Construção de Oráculos

As abordagens apresentadas na seção 4.2 podem ainda ser analisadas segundo aspectos ligados ao processo de desenvolvimento de *software*, à estratégia de teste, à abrangência da verificação e ao escopo de execução.

O processo de desenvolvimento tem influência marcante no modo como se realiza o mapeamento. Essa característica pode ser notada nas abordagens de Gannon, Hayes, McDonald, Antoy e Peters, onde todas apresentam mapeamento implícito. Este fato sugere que quando o *software* é derivado a partir da estrutura da especificação, a definição do relacionamento entre seus pontos correspondentes se torna direta. A abordagem de McDonald tira proveito dessa característica, relacionando as variáveis e os métodos de classes implementadas em C++ diretamente às variáveis e esquemas de especificações Object-Z. Nos demais casos, o mapeamento não ocorre na prática, uma vez que a especificação e a implementação usam as mesmas estruturas de dados.

A estratégia de teste pode influenciar na construção do oráculo quando este é definido a partir do critério de seleção de casos de teste. Esta idéia foi introduzida na abordagem para geração de oráculos proposta por Richardson [RIC 92] e é seguida por Stocks [STO 93] e Hörcher [HOR 95]. Nesses casos, define-se oráculos para cada conjunto de caso de teste derivado pela estratégia de seleção de casos de teste.

A abrangência da verificação determina que tipos de erros os oráculos descritos são capazes de detectar. Uma característica comum a todas as abordagens citadas é que a verificação ocorre pela observação do estado da implementação, o qual corresponde aos valores das suas variáveis e, em alguns casos, ao estado do ambiente de execução, ao tempo e aos eventos ocorridos.

O escopo de execução das abordagens citadas está quase totalmente restrito aos sistemas seqüenciais, uma vez que nenhum autor faz referência a fatores de sincronização e concorrência. Algumas variações são vistas na abordagem de Richardson [RIC 92] que aborda a verificação de sistemas reativos e na abordagem de Bieman [BIE 92], que leva em consideração sistemas tempo-real.

Relembrando os aspectos ligados à abrangência de utilização dos oráculos levantados como motivação deste trabalho, observa-se que apenas os oráculos gerados pelas abordagens de Bieman e de Fletcher e Sarjeev são independentes do processo de desenvolvimento e independentes da estratégia de seleção de casos de teste. Entretanto, alguns fatores devem ser acrescentados.

Primeiramente a abordagem de Bieman é restrita a uma linguagem de programação funcional, o que por sua vez, acarreta em uma forte restrição ao uso desse tipo de oráculo em boa parte dos ambientes acadêmicos e comerciais. O oráculo de Fletcher e Sarjeev é aplicado a uma linguagem orientada a objetos – Eiffel – e suas assertivas são derivadas de Object-Z. Entretanto, deve-se considerar que a avaliação dos resultados não é realizada durante a execução da implementação sob teste, o que acarreta na necessidade de captura do estado completo da implementação. Essa característica leva a dificuldades no teste de valores complexos.

A seguir, a Tabela 4.2 resume os tópicos apresentados nesta seção.

TABELA 4.2 – Aspectos que influenciam a geração de oráculos

	Relação com o processo de desenvolvimento	Vínculo com a geração de casos de teste	Abrangência da verificação	Escopo de utilização
Gannon, McMullin e Hamlet	Vinculado à especificação	Não	Estado interno	Sistemas seqüenciais
Hayes	Vinculado à especificação	Não	Estado interno	Sistemas seqüenciais
Stocks		Sim	Estado interno	Sistemas seqüenciais
Bieman	Livre	Não	Estado interno	Sistemas seqüenciais e tempo-real
Richardson	Livre	Sim	Estado interno, valores de ambiente	Sistemas seqüenciais e reativos
Hörcher	Livre	Sim	Estado interno	Sistemas seqüenciais
Fletcher e Sajeev	Livre	Não	Estado interno	Sistemas seqüenciais
McDonald	Vinculado à especificação	Não	Estado interno	Sistemas seqüenciais
Peters e Parnas	Vinculado à especificação	Não	Estado interno	Sistemas seqüenciais
Antoy e Hamlet	Vinculado à especificação	Não	Estado interno	Sistemas seqüenciais

4.4 Resumo do Capítulo

Este capítulo introduziu o tema oráculo para teste de *software* através de um estudo sobre diversas abordagens de oráculos baseados em especificação formal. Este estudo apresentou os aspectos que mais influenciam o processo de geração de oráculos. A partir desses aspectos, o próximo capítulo apresenta uma estratégia que considera os fatores relacionados à aplicabilidade, apontados pela motivação deste trabalho.

5 Estratégia para a Geração de Oráculos a partir de Especificações Formais

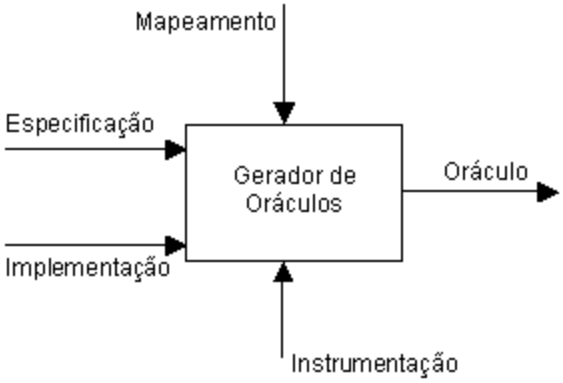
Este capítulo trata da estratégia para a geração de oráculos adotada pela ferramenta OZJ. Uma breve explanação a respeito de trabalhos relacionados é apresentada em cada seção, a fim de comparar e justificar os aspectos incorporados neste trabalho.

Este capítulo está organizado da seguinte forma: a Seção 5.1 apresenta a estrutura geral da estratégia de geração de oráculos proposta neste trabalho, estabelece os requisitos de projeto principais e define alguns pontos a cerca da linguagem de especificação, tipo do oráculo e representação dos tipos abstratos; a Seção 5.2 mostra como o oráculo é representado e aplicado às classes e à hierarquia de classes Java; a Seção 5.3 apresenta a estratégia de verificação adotada, apontando como os erros são detectados e delimitando o escopo de atuação da verificação; a Seção 5.4 trata do mapeamento entre as estruturas da especificação e da implementação, apontando suas dificuldades e as soluções adotadas; a Seção 5.5 apresenta como as expressões lógicas são traduzidas para instruções equivalentes em Java e como se contorna o problema dos predicados não executáveis; a Seção 5.6 apresenta a abordagem adotada para instrumentação de acordo com os requisitos da estratégia proposta neste trabalho; e por fim, a Seção 5.7 apresenta uma simulação do funcionamento do oráculo.

5.1 Estrutura Geral do Oráculo

A literatura sobre oráculos para teste de *software* apresenta várias abordagens [ANT 2000, PET 98, MCD 97, CHA 96, HOR 95, RIC 92, RIC 89]. De acordo com o apresentado na Seção 4.2.2, a partir dessas abordagens é possível identificar uma estrutura geral relativamente simples para o funcionamento dos oráculos. Esta estrutura, ilustrada pela Figura 4.1, consiste da captura do estado da implementação, da identificação do estado correspondente na especificação e, posteriormente, da 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 comportamento correto do sistema. Cita-se Peters e Parnas [PET 98] 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 na linguagem C++; e McDonald e Strooper [MCD 97] que extraem assertivas de especificações formais escritas em Object-Z e as aplicam em implementações de classes em C++. A Figura 5.1 apresenta uma visão geral da geração de oráculos através de um diagrama SADT.



URA 5.1 – Estrutura geral da geração de oráculos

O diagrama da Figura 5.1 ilustra a geração de oráculos como uma ação que tem a instrumentação como mecanismo principal e que recebe como entrada a especificação e uma implementação e produz como saída um oráculo. A Figura 5.1 ilustra também que o mapeamento é a principal fonte de controle do processo de geração de oráculos.

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 prática no que diz respeito à aplicabilidade do gerador de oráculo. O termo "mais prática" 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 [MCD 97, PET 98, ANT 2000] por elas estarem intimamente ligadas ao processo de desenvolvimento de *software* baseado em especificação formal.

Além disso, abordagens como as descritas em [STO 93, RIC 92, HOR 95] 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 [WEY 91, OST 88] ou a geração de casos de teste para erros de domínio [WHI 80] sejam aplicadas ao oráculo da mesma forma que seriam aplicadas à classe original.

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 [RAP 85] 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.

A seguir, são apresentados alguns aspectos a cerca da estrutura dos oráculos gerados pela ferramenta OZJ.

5.1.1 Linguagem de Especificação

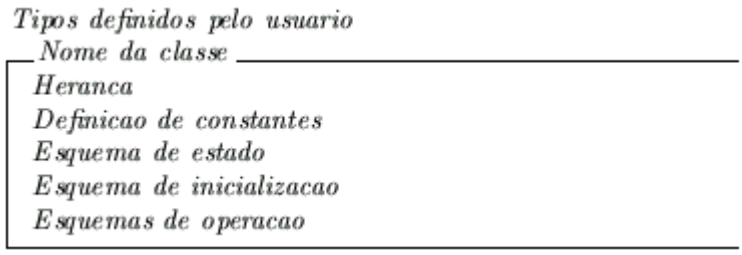
OZJ é uma ferramenta para geração de oráculos para classes implementadas em Java utilizando especificações formais Object-Z [ROS 94] como fonte de informação.

A linguagem Object-Z é uma extensão orientada a Objetos da linguagem Z [SPY 92] e se propõe a descrever o comportamento de classes sob a forma de modelos. Esses modelos são construídos pela representação dos estados válidos do sistema utilizando a sintaxe da lógica de predicados de primeira ordem e da teoria dos conjuntos.

Como OZJ deve ser utilizado no teste de classes, o fato de Object-Z fazer parte do paradigma orientado a objetos facilita a identificação das diretrizes que regem o mapeamento entre as especificações e implementações Java. Essas diretrizes são apresentadas na Seção 6.3.

Com o objetivo de simplificar a construção do protótipo do OZJ, considera-se neste trabalho apenas um sub conjunto da linguagem Object-Z, chamada Object-OZJ. Deve-se notar que, posteriormente, pode-se estender a abordagem apresentada para um escopo maior da linguagem Object-Z.

As simplificações mais importantes são relacionadas à definição e à utilização dos esquemas na especificação. Uma especificação Object-OZJ permite a definição de tipos de usuário e de uma única classe. Os tipos de usuário devem ser obrigatoriamente definidos unicamente através dos tipos primitivos do Object-Z. A estrutura de uma especificação em Object-OZJ é apresentada a seguir:



A declaração das variáveis permite apenas o uso dos tipos primitivos do Object-Z, portanto, tipos definidos por enumeração ou por composição estão fora da linguagem Object-OZJ.

Object-OZJ não prevê a definição de classes com parâmetros genéricos. Além disso, não é permitida a definição de esquemas na forma horizontal, exceto quando o esquema é definido, pela negação, pela conjunção ou pela disjunção de outros esquemas. A alteração de nomes de variáveis (*renaming*) ou de operações só é permitida na definição de hierarquias de classe.

As operações excluídas, relativas à lógica de predicados e à teoria dos conjuntos, são listadas a seguir.

- Esquemas

  Ocultação de variável, composição, *piping*, implicação, bi-implicação, conjunção distribuída, disjunção distribuída, composição distribuída.

- Operadores para formação de termos

  Definições definitivas, definições locais (*let*), termo condicional (*if them*), definições através da notação *lambda*.

- Números

  Função sucessora de números inteiros.

- Conjuntos

  União generalizada, intersecção generalizada, diferença simétrica, mínimo e máximo inteiro.

- Relações binárias e funções

  Composição relacional, composição relacional inversa, fechamento transitivo, fechamento transitivo reflexivo, potenciação.

- Seqüências

  Concatenação distribuída, disjunção e partições.

Maiores informações sobre as operações citadas podem ser encontradas em [SPY 92, DIL 94].

Em Object-OZJ, o conceito de agregação foi desconsiderado. Desta forma, qualquer variável declarada na especificação deve ser obrigatoriamente de um tipo primitivo do Object-Z. Os demais conceitos e operações não citados seguem a mesma semântica do Object-Z, tais como polimorfismos e herança.

A gramática da linguagem Object-OZJ é apresentada no Anexo 1, e descreve a sintaxe das especificações sob a forma de documentos Latex [WEN 96].

Deste ponto em diante, a toda referência à linguagem Object-Z deve-se considerar o subconjunto definido para a linguagem Object-OZJ.

5.1.2 Tipo de Oráculo e Estratégia de Avaliação de Assertivas

Os oráculos podem ser classificados segundo a forma utilizada para avaliar o seu comportamento [MCD 97], a qual é uma variação da classificação proposta por Hoffman [HOF 91], apresentada na Seção 4.1. De acordo com esta classificação:

- oráculos ativos são aqueles que implementam o comportamento do sistema sob teste produzindo um par (*entrada*, *saída*), o qual será comparado com os valores capturados durante o teste;
- oráculos passivos são aqueles que verificam o comportamento do sistema sem reproduzi-lo. Nesse caso, o resultado é obtido pela avaliação de assertivas que determina se o sistema atingiu o estado esperado.

Uma análise entre essas duas opções mostra que oráculos passivos são mais viáveis. Considerando oráculos ativos, o uso de processos manuais para desenvolver uma implementação alternativa, além de inviável, é uma atividade pouco confiável [RUS 93]. Considerado um processo automático, obter oráculos ativos é uma atividade de alto custo, uma vez que, geralmente, as especificações não fornecem detalhes algorítmicos [HOR 95], tornando difícil a tarefa de determinar o par (*entrada*, *saída*). Outro problema está no fato de que as especificações podem ser não determinísticas, acarretando que uma entrada pode ter mais de uma saída como resposta [MCD 97].

O uso de oráculos passivos leva a duas alternativas de avaliação do resultado. Uma consiste em derivar assertivas avaliadas diretamente sobre as variáveis da implementação, tal como em [PET 98]. Entretanto, a abordagem mais usual consiste em transferir os valores das variáveis da implementação para um espaço que contém as variáveis da especificação, onde as assertivas são avaliadas. Esta abordagem é adotada em [MCD 97, HOR 95, RIC 92]. Nesse caso há necessidade de se definir funções de abstração, capazes de converter os valores das variáveis da implementação para valores equivalentes no espaço de variáveis da especificação.

A principal vantagem de avaliar os resultados do teste usando diretamente as variáveis da implementação está no fato de evitar a definição da função de abstração. Nesses casos, entretanto, o poder de abstração da linguagem de especificação se torna dependente dos tipos disponíveis na linguagem de implementação [RIC 92].

A alternativa mais freqüentemente usada na avaliação dos resultados do teste, consiste em criar uma representação abstrata dos tipos da linguagem de especificação, capturar os valores das variáveis da implementação através de funções de abstração e compará-lo com o comportamento esperado em um espaço de variáveis abstrato.

Em relação a este trabalho, um ponto positivo na utilização da representação abstrata de variáveis é que a influência da aplicação da instrumentação na implementação é minimizada [OMA 96]. Dessa forma, é possível avaliar as assertivas realizando apenas acessos às variáveis da implementação garantindo que as características estruturais e funcionais da classe original permaneçam inalteradas no oráculo.

5.1.3 Representação Abstrata de Tipos

A fim de permitir que a avaliação das assertivas seja realizada fora do espaço de variáveis da implementação, a ferramenta OZJ conta com um pacote de classes que implementa a funcionalidade dos tipos do Object-Z chamada ZMTK – do inglês *Z Language Mathematical Tool Kit*.

As classes do ZMTK são estruturadas de forma a permitir que cada tipo contido nas variáveis das expressões na especificação possa ser representado por uma classe correspondente. Da mesma forma, cada operação possível sobre os tipos também é representada por um método correspondente.

Como exemplo, considerado o tipo *Set* e suas operações, a classe correspondente no ZMTK é apresentada na Tabela 5.1:

TABELA 5.1 – Representação das operações sobre o tipo *Set* no ZMTK

Set	ZMTK.Set
$\in$	ZMTKBoolean membership(ZMTKObject element)
$=$	ZMTKBoolean equality(Set another)
$\subseteq$	ZMTKBoolean subset(Set another)
$\subset$	ZMTKBoolean properSubset(Set another)
$\cup$	Set union(Set another)
$\cap$	Set intersection(Set another)
$\varnothing$	ZMTKBoolean emptySet()

De forma semelhante ao tipo *Set*, os tipos *BinaryRelation*, *Function*, *Number*, *Sequence* e *Bag* são representados por classes correspondentes no ZMTK. Uma documentação reduzida do ZMTK é apresentada no Anexo 2. Detalhes sobre a especificação dos tipos de operações do Object-Z podem ser encontrados em [SPY 92, DIL 94].

5.2 Estrutura Física do Oráculo

Devido ao conceito de encapsulamento do paradigma orientado a objeto, uma classe pode conter atributos ou operações inacessíveis às suas especializações. A abordagem adotada por McDonald [MCD 97] gera o oráculo como uma especialização da classe sob teste. Logo, para contornar o problema de acesso gerado pelo encapsulamento é necessário alterar o modificador de visibilidade dos atributos e métodos privados ou adicionar novos métodos para recuperar seus valores.

Instrumentando diretamente uma cópia da classe original ganha-se acesso às suas estruturas de dados e operações privadas sem a necessidade de alterações nos modificadores de visibilidade. Nesta linha, a abordagem adotada por Antoy e Hamlet [ANT 2000] gera oráculos para ADT's aplicando a instrumentação diretamente na classe sob teste. Entretanto, embora não tenha o problema da visibilidade na classe sob teste, esta abordagem não considera a geração de oráculos para uma hierarquia de classe. Isso traz de volta o problema da visibilidade, uma vez que, mesmo tendo acesso

aos atributos definidos na classe sob teste, comumente, é necessário ter acesso a atributos privados na super-classe para realizar a verificação do comportamento.

A estratégia adotada neste trabalho contorna o problema de manipulação de atributos inacessíveis utilizando o recurso de herança do Object-Z. Para tirar proveito do conceito de herança, o resultado do processo de geração de oráculo do OZJ cria, além do oráculo, uma classe auxiliar chamada *OracleManager*.

Para cada hierarquia, uma classe *OracleManager* é criada de acordo com a especificação, a implementação original e um mapeamento entre eles. Na prática, uma instância dessa classe é inserida no oráculo com a função de gerenciar todo o processo de verificação. Seus atributos são a representação abstrata das variáveis da implementação, e seus métodos implementam as operações da lógica de predicados, os operadores relacionais, as funções de abstração e a execução das assertivas. A captura dos valores das variáveis da implementação é realizada através de funções inseridas no corpo do oráculo.

Para exemplificar a instrumentação, a Figura 5.2 apresenta a especificação de uma pilha de números inteiros e uma implementação correspondente.

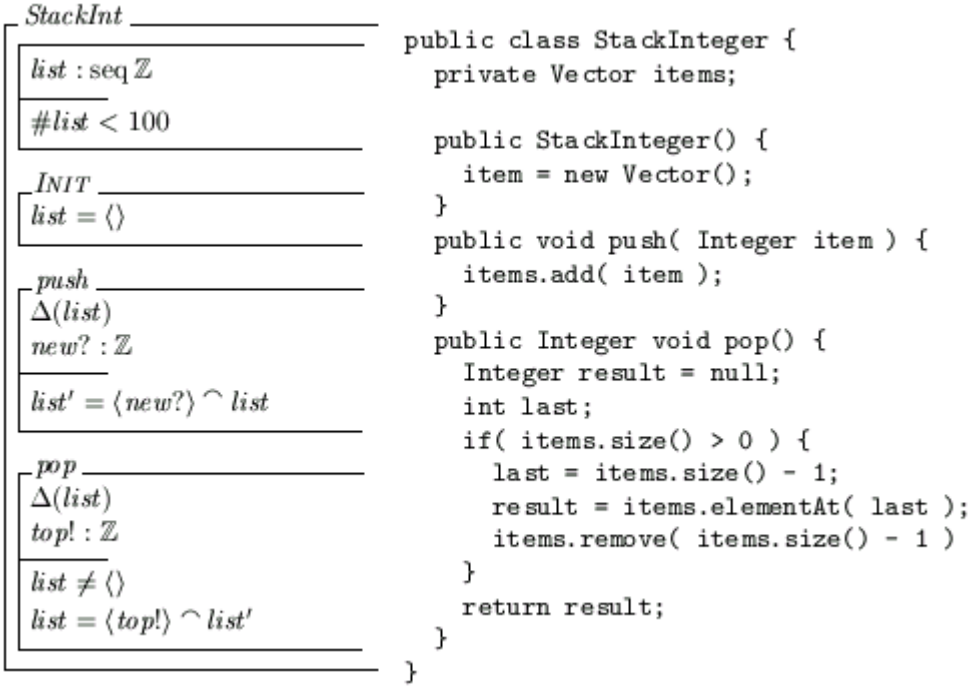


FIGURA 5.2 – Especificação e uma implementação de uma pilha de inteiro

Considerando um mapeamento entre as variáveis *list* e *items*, entre *INIT* e o construtor de *StackInteger*, e entre as operações *push* e *pop*, o oráculo gerado para a *StackInteger* é listado na Figura 5.3.

Deve-se notar que os métodos *abstract_* capturam os valores das variáveis de implementação, preenchendo as variáveis abstratas internas ao oráculo. Por exemplo, a variável *items* tem um atributo *Sequence list* como correspondente interno à classe *OracleManager*. Este atributo é preenchido com os valores de *items* pela chamada de *oracle.abstract_list(concrete_items())*, onde *abstract_list()* é a função de abstração para a variável *list* e *concrete_items()* é uma função que simplesmente retorna o valor da variável *items*. Os métodos *preCondition()*, *posCondition()* e *invariant()* contêm chamadas à implementação das assertivas que verificam o comportamento

implementado. Por exemplo, *oracle.preCondition('pop')*, realiza a avaliação da assertiva

$$list \neq \langle\rangle$$

a qual, corresponde à pré-condição do esquema *pop* aplicado ao método *pop()* da implementação.

O exemplo da figura 5.2 mostra uma situação onde não há problemas de acesso aos atributos privados. Entretanto, a Figura 5.4 mostra uma situação onde esse problema acontece devido à hierarquia de classes. Considerando uma pilha que pode conter apenas valores naturais, *StackNat*, é especificada como uma subclasse de *StackInt* acrescida de uma restrição aos seus elementos. Portanto, para gerar um oráculo para *StackNatural* a partir da especificação *StackNat* é necessário ter acesso ao valor da variável privada *items* em *StackInteger*.

```
public class StackInteger {
  private Vector items;
  // ===== Begin Instrumentation =====
  // Objeto oráculo
  protected OracleManager oracle;
  // ===== End Instrumentation =====

  public StackInteger() {
    items = new Vector();
    // ===== Begin Instrumentation =====
    // Cria gerenciador do oráculo
    oracle = new OracleManager();
    // Captura valores items -> list
    oracle.abstract_list( concrete_items() );
    // Avalia pós_condição
    oracle.posCondition( 'INIT' );
    oracle.invariant( 'INIT' );
    oracle.verifySchema( 'INIT');
    // ===== End Instrumentation =====
  }
  public void push( Integer item ) {
    // ===== Begin Instrumentation =====
    // Captura valor de item -> new?
    oracle.abstract_new_in( item );
    oracle.invariant( 'push' );
    // ===== End Instrumentation =====
    items.add( item );
    // captura valores items -> list'
    // ===== Begin Instrumentation =====
    oracle.abstract_list_pos( concrete_items() );
    // Avalia pos-condição e invariante
    oracle.posCondition( 'push' );
    oracle.invariant( 'push' );
    oracle.verifySchema( 'push' );
   // ===== End Instrumentation =====
```

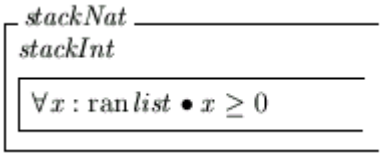
```
  public Integer void pop() {
    // ===== Begin Instrumentation =====
    // Avalia pre-condição e invariante
    oracle.inavariant( 'pop' );
    oracle.preCondition( 'pop' );
    // ===== End Instrumentation =====
    Integer result = null;
    if( items.size() > 0 ) {
      result = (Integer)items.elementAt( items.size() - 1 );
      items.remove( items.size() - 1 )
    }
    // ===== Begin Instrumentation =====
    // captura valores items -> list'
    oracle.abstract_list_pos( concrete_items() );
    // captura valores  result -> top!
    oracle.abstract_top_out( concrete_result() );
    // Avalia pos-condição e valor de saída
    oracle.posCondition( 'pop' );
    oracle.output( 'pop' );
    oracle.invariant( 'pop' );
    oracle.verifySchema( 'pop' );
    // ===== End Instrumentation =====
    return result;
  }
}
```

FIGURA 5.3 – Oráculo gerado para a classe *StackInteger*

Para contornar o problema, oráculos são gerados para todas as classes dessa hierarquia. Nesse caso, o oráculo para *StackInteger* é idêntico ao listado na Figura 5.3, com exceção do método *push*, que não recebe instrumentação por ser redefinido na subclasse. O oráculo para *StackNatural* consiste na instrumentação dos métodos novos ou redefinidos, ou seja, o método *push*, tal como mostra a Figura 5.5. A instrumentação é a mesma anteriormente criada no oráculo para o método *push* de *StackNatural* acrescida da chamada ao método *oracle.invariant('pushNat')*, que consiste na avaliação da expressão:

$$\forall x : \operatorname{ran} list \bullet x \geq 0$$

O problema da visibilidade foi contornado na medida que a instância da classe *OracleManager* e as funções que capturam os valores das variáveis da implementação são declaradas como itens protegidos na classe de maior grau na hierarquia, dando acesso aos seus valores a partir da subclasse.




```
public class StackNatural
       extends StackInteger {
    public void push( Integer item ) {
    if( ( (Integer)item ).intValue < 0 )
        throws new Exception();
    else
        super.push( item );
   }
}
```

FIGURA 5.4 – Especificação e uma implementação de uma pilha de naturais

Tanto a classe *OracleManager* como os oráculos são gerados automaticamente pela ferramenta OZJ, cabendo ao usuário as tarefas de compor o mapeamento e a implementação do corpo das funções de abstração.

```
public class StackNatural extends StackInteger {
    public void push( Integer item ) {
        if( ( (Integer)item ).intValue < 0 )
            throws new Exception();
        else
            super.push( item );

        // captura valores items' -> list
        oracle.abstract_list_post( concrete_items() );
        // Avalia pos-condição e invariante
        oracle.posCondition( 'push' );
        oracle.invariant( 'push' );
        // Invariante
        oracle.invariant( 'pushNat' );
   }
}
```

FIGURA 5.5 – Oráculo para gerado para classe *StackNatural*

O processo de geração de oráculos descrito nesta seção, pode ser dividido em uma série de passos segundo sugere a Figura 5.6.

A primeira etapa consiste no mapeamento, onde a estrutura da especificação é associada a estruturas correspondentes na implementação. A implementação das funções de abstração está associada ao mapeamento. No Capítulo 6 mostra-se como a estratégia da ferramenta ajuda o projetista a reutilizar o código de funções de abstração, minimizando o custo desta etapa.

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.

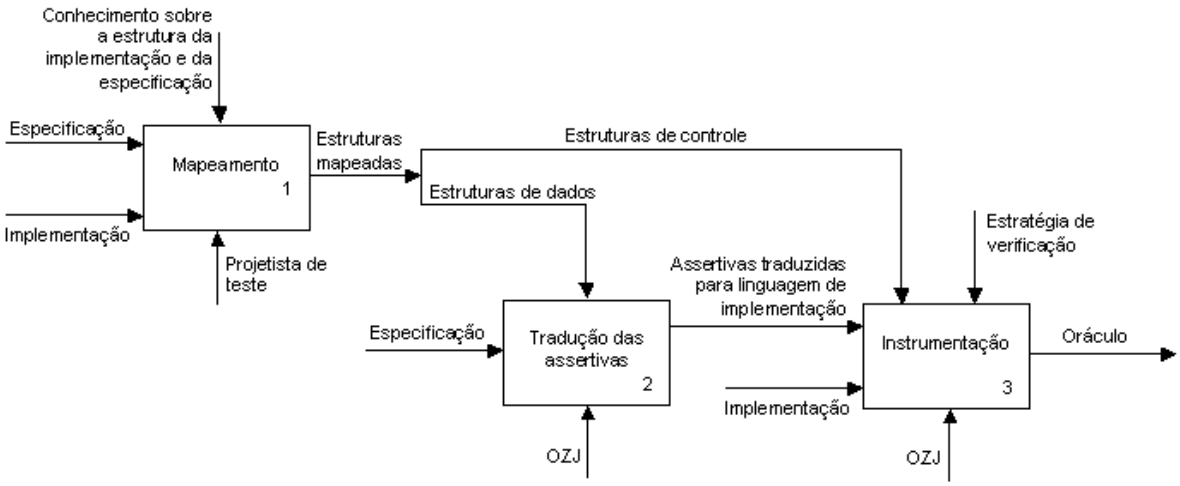


FIGURA 5.6 – Etapas do processo de geração de oráculos

A última etapa, realizada pelo gerador de instrumentação, consiste em criar a classe *OracleManager* e, a seguir, inserir convenientemente as chamadas às funções de abstração e às assertivas no código do oráculo. A instrumentação é regida por uma estratégia de verificação e pelo mapeamento, descritos nas seções 5.3 e 5.4, respectivamente.

A seguir, discute-se a abordagem e as decisões de projeto adotadas neste trabalho para cada uma dessas etapas.

5.3 Estratégia de Verificação

O conceito de pré-condições e pós-condições vem sendo usado há muito tempo na verificação de programas. Hoare propôs uma das abordagens mais antigas, onde pré-condições e pós-condições são definidas como expressões lógicas aplicadas às variáveis da implementação [HOA 69]. 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.

Nesse estilo de verificação, como as variáveis do programa são usadas diretamente nas expressões lógicas e há uma grande diferença de nível de abstração entre as expressões e as instruções dos programas, o poder de abstração da lógica de predicados fica limitado. Além disso, o uso de variáveis e tipos de dados diretamente ligados ao código da implementação pode levar a ocultação de erros, uma vez que estes estão diretamente relacionados com a estrutura de dados e ao tipo de dado usados na especificação [OMA 96].

Devido a esse problema, posteriormente, Hoare estendeu sua técnica de verificação ao propor o conceito de função de abstração [HOA 72]. Considerando que há um espaço de variáveis concretas e um espaço de variáveis abstratas, uma função de abstração é

capaz de mapear variáveis do espaço concreto para o espaço abstrato. Logo, se uma função de abstração é aplicada às variáveis do programa, ou seja, ao estado concreto, o resultado obtido é o estado abstrato.

Pela definição deste conceito, a verificação passa a lidar com o espaço abstrato de variáveis, evitando a desvantagem de manipular diretamente as variáveis do programa. Outra vantagem desta abordagem é que eventuais alterações nas estruturas de dados da implementação não alteram o predicado de verificação se as variáveis da implementação estiverem corretamente mapeadas às variáveis abstratas.

5.3.1 Unidade de Verificação

A verificação baseada na avaliação de pré e pós-condições vem sendo largamente utilizada entre as abordagens para construção de oráculos [ANT 2000, PET 98, MCD 97, HOR 95, RIC 92, RIC 89]. 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, determina-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 [WEY 86], 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[4]. 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-

[4] O conceito de unidade de execução não corresponde ao conceito de unidade de teste orientado a objeto comentado na Seção 2.3.

se o efeito da interação entre eles. Testes de *clusters* também podem ser realizados gerando-se oráculos para todas as classes do cluster.

5.3.2 Escopo de Utilização

Embora haja um ganho em praticidade em não considerar como unidade de execução cada instrução do código, há uma perda na abrangência da verificação. Isso ocorre porque possíveis erros internos a uma unidade de execução não são detectados. Por exemplo, quando existe uma condição invariante para uma classe, ou seja, uma condição que não pode ser violada em momento algum durante a execução. Nesse caso, é possível que a execução de um método satisfaça a pré-condição, mas posteriormente, leve a classe a um estado que viole a invariante e, em seguida, atinja um estado que satisfaça a pós-condição. Nesse caso, considerando os métodos de classe como unidade de execução, a violação interna da invariante não será detectada.

As características desta estratégia excluem sua aplicação a classes que realizem computação em tempo-real, uma vez que não há qualquer referência a fatores ligados ao tempo de execução de métodos [OMA 96, RIC 92]. Sistemas reativos também estão fora do escopo deste trabalho pelo fato da verificação estar restrita ao estado interno da classe, o qual não faz referência a fatores de ambientes. Igualmente, esta estratégia não leva em conta fatores ligados a aplicações concorrentes. Logo, o escopo da estratégia de verificação deste trabalho está restrito as classes estritamente seqüenciais.

5.3.3 Tipos de Assertivas e Abrangência da Verificação

Na estratégia para geração de oráculos proposta neste trabalho, assertivas são geradas a partir de expressões retiradas de especificações Object-Z. A partir da sintaxe do Object-Z é possível identificar, além de pré e pós-condições, outros dois tipos de expressões: as invariantes de classe e os valores de retorno.

Apresenta-se, a seguir, um critério para classificação de assertivas, baseado no tipo das variáveis que as compõem. Os exemplos refere-se à especificação *StackInt*, apresentada na Seção 5.2.

- Pós-condição

  São as assertivas que possuem pelo menos uma variável que denota o estado posterior da operação.

  A expressão

  $$list' = \langle new? \rangle \frown list$$

  é um exemplo e indica que no estado resultante da operação, *new*? deve ser inserido no início da seqüência *list* gerando *list'*.

- Valores de Retorno

  Se a assertiva contém uma variável de saída e não contém qualquer variável que denote um estado posterior, então, a assertiva é um valor de retorno.

  Por exemplo, considerando a expressão

  $$top! = first\ list$$

  onde uma variável de saída *top!* é definida como o primeiro elemento da seqüência *list*.

- Pré-condição

  As pré-condições são as assertivas que contêm apenas variáveis de entrada e/ou variáveis de estado.

  A expressão

  $$list \neq \langle\rangle$$

  é um exemplo, indicando que a operação não pode ocorrer em um estado onde a seqüência *list* é vazia.

- Invariantes

  Toda assertiva gerada a partir de uma expressão definida dentro de um esquema de definição de estado ou dentro de um esquema de definição de constantes é considerada uma invariante.

  A expressão

  $$\#list < 100$$

  é a invariante da especificação *StackInt*, ou seja, nenhuma operação pode iniciar ou resultar em um estado onde a seqüência *list* contenha cem ou mais elementos.

Uma vez identificados os tipos de assertivas, é possível detectar se após a execução de qualquer método, a classe atinge um estado inválido. Cada tipo de assertiva realiza uma verificação específica, conforme mostra a Tabela 5.2.

TABELA 5.2 – Função das assertivas usadas na verificação

Tipo de Assertiva	Verificação
Pré-condição	Verifica se o estado é válido no início da execução da operação.
Pós-condição	Verifica se o estado é válido após a execução da operação.
Invariante	Verifica se o estado corrente respeita a invariante.
Valor de Retorno	Verifica se o valor retornado é o esperado.

A estratégia apresentada é válida tanto se o método é testado isoladamente, durante testes de unidade, ou em seqüência, durante testes de integração. Para ilustrar como as assertivas são verificadas, a Figura 5.7 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 5.7 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.

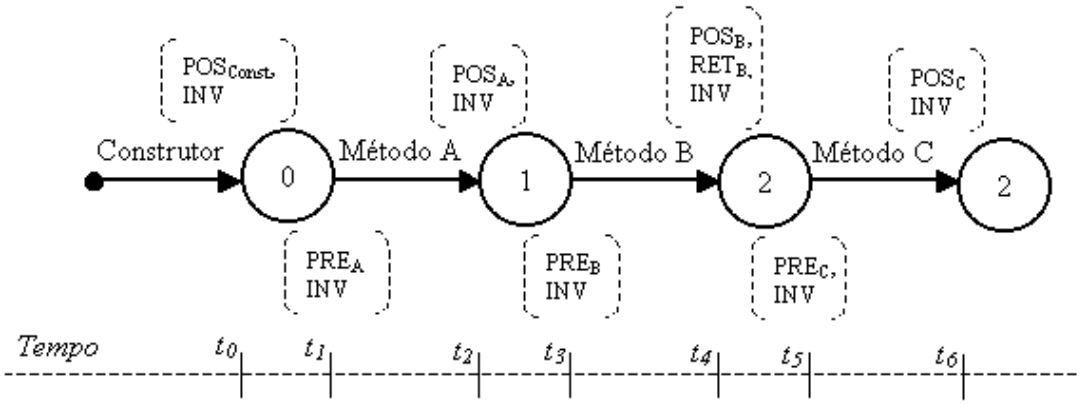


FIGURA 5.7 – Avaliação das assertivas em uma seqüência de chamadas de métodos

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.

5.4 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 [OMA 96], 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* [BER 91]. 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 [RUS 93].

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 [ANT 2000] 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 [MCD 97] 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 [RIC 89]. Como exemplo, 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 [OMA96] 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 [MCD 97], 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, apresentada na Seção 5.3.2, 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. Define-se este conceito como *homogeneidade parcial*.

Para exemplificar, considera-se a Figura 5.8 que apresenta a especificação de um conjunto de nomes que pode conter no máximo mil caracteres. 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.

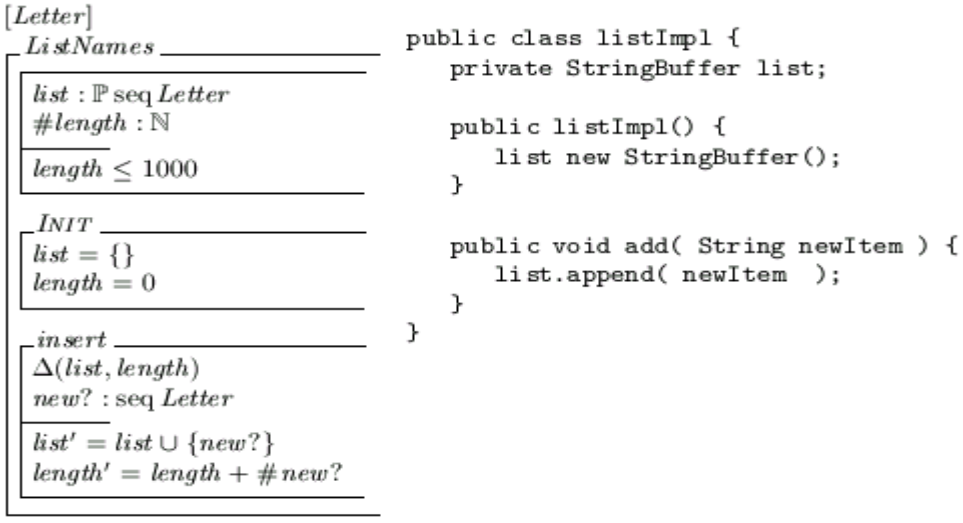


FIGURA 5.8 – Especificação e uma implementação de uma lista de nomes

Nesse contexto, quando se assume homogeneidade total restringe-se a aplicação dos oráculos às implementações construídas através de prototipação, não 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 [HOA 72] que introduz o conceito de função de abstração, comentada na Seção 5.2 . 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, nem 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, conforme comentado na Seção 5.1.1, 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 [RIC 89].

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 5.8, sendo dependente da semântica atribuída a ele pelo projetista. A seção 6.3 apresenta alguns critérios que auxiliam a composição do mapeamento considerando especificações e implementações parcialmente homogêneas.

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. A Seção 6.2, apresenta como a ferramenta OZJ tira proveito dessa característica para reutilizar as funções de abstração.

5.5 Tradução de Assertivas

A etapa seguinte ao mapeamento consiste em obter uma forma de avaliação das expressões definidas pela especificação em tempo de execução. Considerando que as assertivas são introduzidas na implementação através de instrumentação do código, é necessário traduzi-las para a sintaxe da linguagem de programação de forma que elas possam ser executadas.

5.5.1 Execução das Assertivas

A necessidade de executar as assertivas traduzidas a partir do Object-Z leva a um problema devido à notação do Object-Z ser uma versão modificada da lógica de predicados de primeira ordem [SPY 92], a qual é indecidível. Uma abordagem que leva em conta esse problema foi proposta por Mik[MIK 95] e consiste em definir

predicados executáveis como aqueles que resultam em computações que terminam. A partir desta definição, a identificação de predicados executáveis restringe-se a determinar se eles manipulam ou não estruturas infinitas. Desta forma, aplicando-se esta definição à notação do Object-Z, a ocorrência de predicados não executáveis limita-se às construções que sugerem iterações, e por conseguinte, podem resultar em iterações infinitas. Segundo Mikk, estas construções são: quantificadores (existencial e universal) e definições implícitas de conjuntos. Logo, o processo de tradução classifica como expressão não executável qualquer uma das expressões citadas que manipule:

- os conjuntos infinitos pré-definidos, tais como os naturais, os inteiros e os reais;
- definição implícita de conjuntos aplicada a conjuntos infinitos.

Estes casos não fazem referência a conjuntos definidos pelo usuário, uma vez que a abordagem proposta por Mikk só pode ser aplicada se existir uma definição explícita da sua cardinalidade [SPY 92].

Neste trabalho, como toda estrutura descrita na especificação deve ter uma correspondente na implementação, assume-se que os conjuntos definidos pelo usuário são sempre finitos.

5.5.2 Processo de Tradução

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, conforme apresentado no Anexo 2.

Um exemplo simples pode ser extraído da Figura 5.8. 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 da classe *OracleManager* e *cardinality*() é um método da classe *Set*.

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.

O processo de tradução é simples quando aplicado aos operadores lógicos e relacionais e às operações derivadas da Teoria dos Conjuntos. Nesse caso, a tradução se divide em duas etapas. Primeiro, transforma-se a notação original das expressões para uma notação infixada, aninhando-se os operadores segundo uma ordem de precedência.

A ordem de precedência é baseada na quantidade de operandos e na posição dos operandos em relação ao identificador da operação. Para os operadores lógicos, leva-se em consideração ainda a precedência entre as operações e as regras de associatividade.

A tabela 5.3 apresenta a ordem de precedência adotada no processo de tradução, onde tem maior precedência o operador associado a um número de maior ordem

TABELA 5.3 – Ordem de precedência usada na tradução para notação infixada

Ordem	Operador	Ordem	Operador
1	Bi-implicação (_⇔_)	6	Operadores relacionais Exemplo: _=_, _>_, _<_, _≥_, _≤_
2	Implicação (_⇒_)	7	Operadores binários infixos Exemplo: _∈_, _⊂_
3	Disjunção (_∨_)	8	Operadores unários prefixos Exemplo: *dom* _, *head* _
4	Conjunção (_∧_)	9	Operadores binários posfixos Exemplo: imagem de função (_(_))
5	Negação (¬_)	10	Operadores unários posfixos Exemplo: inversa de funções (_~)

De acordo com a tabela de precedência, reescreve-se as expressões na forma de chamada de funções, aninhando mais internamente as funções que correspondem aos operadores de maior precedência.

Como exemplo, considerando-se a expressão:

$$x \Rightarrow y \in B \Leftrightarrow C \subset D$$

obtém-se

$$\Leftrightarrow (\Rightarrow (x, \in (y, B)), \subset (C, D))$$

O operador de bi-implicação (⇔) é o mais externo por ser o de menor precedência. Pela mesma razão, o operador da relação de pertinência (∈) é traduzido como um argumento do operador de implicação (⇒) por ter maior precedência.

A partir da notação intermediária, o símbolo dos operadores é substituído pela chamada aos métodos da biblioteca de tipos definida no pacote ZMTK. Tomando como exemplo a expressão acima, o resultado da tradução é:

```
oracle.biImplication( oracle.implication( x, B.membership( y ) ),
                      C.subset( D ) );
```

onde, *oracle* é uma instância da classe *OracleManager*, *x* é uma instância da classe *ZMTKBoolean*, *y* é uma instância de qualquer subclasse de *ZMTKObject*, e *B*, *C* e *D* são instâncias da classe *Set*.

Ao contrário dos demais operadores, quando a expressão envolve quantificadores ou definição explícita de conjuntos, é necessária a geração de um procedimento que simule as iterações sugeridas na expressão. Um fator que dificulta a tradução dos quantificadores e das definições explícitas de conjunto é a variação permitida pela notação, o que, em geral, leva a implementações diferentes. Além disso, os quantificadores e a definição explícita de conjuntos funcionam sobre outros operadores, o que torna o significado de cada expressão diferente.

Nesta abordagem, com a intenção de simplificar a utilização desses operadores, define-se restrições a sua utilização.

Tomando o quantificador existencial como exemplo, uma restrição consiste em não permitir a declaração de mais de uma variável interna à expressão. Ou seja, expressões do tipo

$$\exists x, y : \mathbb{Z} \bullet x < y$$

não são aceitas pelo tradutor. Isso simplifica a tradução por fixar a quantidade de estruturas de repetição necessárias na implementação correspondente. Obedecendo esta restrição, toda expressão que utilizar o operador existencial será implementada com uma única estrutura de repetição.

Uma outra restrição limita a declaração da variável interna como uma estrutura finita, seguindo a abordagem de Mikk [MIK 95]. Na prática, a variável interna deve ser, obrigatoriamente, um conjunto finito previamente declarado ou obtido através das operações sobre conjuntos definidas no ZMTK.

A Figura 5.9, apresenta um exemplo de tradução de uma expressão que utiliza o quantificador existencial para uma função Java, mostrando a relação entre a notação e o código gerado.

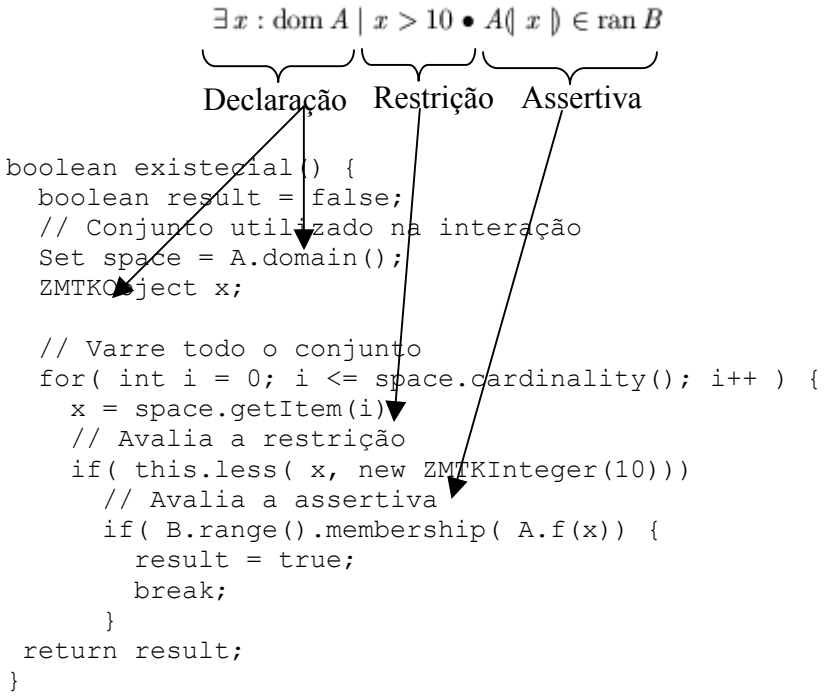


FIGURA 5.9 – Resultado da tradução de uma expressão com quantificador existencial

Conforme indica a Figura 5.9, a partir da declaração da variável, define-se o conjunto – *space* – que contém a informação e a instrução que varre todos os elementos do conjunto. A expressão referente à restrição é traduzida como uma decisão necessária, mas não suficiente para tornar toda a expressão verdadeira. Se a expressão referente à restrição for verdadeira, a expressão principal é avaliada e, ser for verdadeira para pelo menos uma iteração, toda a expressão é verdadeira. Deve-se observar que as variáveis *A* e *B* são objetos da classe *BinaryRelation*.

A tradução do quantificador universal (∀) segue a mesma linha, diferenciando-se apenas pelo fato de que se pelo menos uma iteração avaliar a assertiva principal como falsa, toda a expressão é considerada falsa. Para a definição implícita de conjunto, a tradução utiliza uma estratégia semelhante na construção das iterações e na avaliação

das expressões, diferenciando-se por acrescentar, ao conjunto que esta sendo definido todos os itens que tornam a assertiva principal verdadeira.

Esta estratégia de tradução gera a implementação de uma função para cada expressão que utilize quantificadores e definição implícita de conjuntos. Essas funções são acrescentadas à classe *OracleManager* e usadas pelos métodos de avaliação de assertivas.

5.6 Instrumentação

A instrumentação visa capturar as informações necessárias à verificação do comportamento do programa durante sua execução. Dependendo da finalidade do sistema sob teste, essas informações podem ser os valores das variáveis internas, eventos ocorridos, dados sobre o ambiente de execução, tempo e dados sobre o caminho percorrido. Por exemplo, sistemas que realizam computações em tempo-real necessariamente precisam de um *clock* para analisar restrições que envolvem tempo; verificações em sistemas reativos, exigem conhecimentos a cerca do ambiente onde ele está sendo executado.

A literatura apresenta, basicamente, quatro tipos de instrumentação:

- *Instrumentação de código-fonte*

  Consiste na inserção de instruções no código-fonte antes da compilação.

- *Instrumentação de código-objeto*

  Este tipo de instrumentação utiliza um compilador modificado capaz de inserir instruções no código-objeto sem alteração do código-fonte.

- *Instrumentação de ambiente*

  Nesse caso, o ambiente de execução deve prover operações de monitoração de programas em tempo de execução. Os sistemas operacionais são exemplos, mas ambientes desenvolvidos pelo usuário, como aqueles que realizam reflexão computacional também são usados.

- *Instrumentação baseada em hardware*

  Este tipo de instrumentação utiliza equipamentos especificamente desenvolvidos para este fim e visa, principalmente, diminuir o impacto causado no desempenho do sistema sob teste. Um exemplo deste tipo de instrumentação é encontrado em [GOR 91].

O escopo deste trabalho está restrito à verificação do comportamento de classes estritamente seqüenciais, não reativas e sem requisitos de tempo-real, onde o comportamento é representado pelo estado das variáveis internas. Nesse caso, as informações necessárias para a verificação são os valores das variáveis e a identificação dos eventos de início e fim de execução de um método de classe.

Tendo em mente os requisitos de praticidade, independência do processo de desenvolvimento e independência da estratégia de teste definidos no início deste capítulo, a utilização da instrumentação de código-fonte é mais viável. 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[5] 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, para o uso desta abordagem, é necessária a utilização de um ambiente de reflexão.

Por outro lado, a construção de oráculos através da instrumentação de código-fonte apresenta dois pontos negativos principais: efeitos colaterais sobre o significado da implementação original e a diminuição de desempenho.

Os efeitos colaterais na implementação original são provocados, basicamente, por alterações das variáveis internas e alterações no ambiente, tempo e fluxo original de execução. Para o escopo definido neste trabalho, alterações no tempo de execução e no ambiente externo do sistema não são consideradas.

Os efeitos colaterais provocados pela alteração das variáveis internas são minimizados neste trabalho pelo uso do espaço abstrato de variáveis, onde são criadas cópias abstratas preenchidas com valores equivalentes aos valores das variáveis de implementação. Através desta estratégia, qualquer alteração necessária à verificação é feita em variáveis distintas e de tipos diferentes das variáveis da implementação. A utilização de tipos diferentes é importante, uma vez que o uso de variáveis distintas, mas do mesmo tipo usadas na implementação pode provocar efeitos colaterais. Por exemplo, em classes que declaram atributos de classe[6], a modificação do estado interno de uma instância acarreta a alteração do estado interno de todas as demais instâncias.

A estratégia de instrumentação apresentada também evita alterações no fluxo de controle por não inserir no oráculo instruções de decisão ou de repetição. Dessa forma, analisando os métodos sem considerar sub-rotinas, o que é o caso, pois conforme a Seção 5.2, assume-se que os métodos são a unidade de execução da verificação, o mesmo fluxo de controle da implementação original é obtido no oráculo.

Entretanto, o uso do espaço de variáveis abstrato pode levar a complicações no que diz respeito ao desempenho do oráculo. A criação de variáveis distintas e a execução das assertivas podem provocar uma sobrecarga de execução, dependendo da quantidade de dados manipulados pelo sistema.

Relembrando que as assertivas são executadas segundo implementações das operações da lógica de predicados e da teoria dos conjuntos, basicamente, deve-se evitar a manipulação de conjuntos que contenham um número muito elevado de elementos. Essa situação pode levar a uma sensível degradação no desempenho pois, comumente, é necessário percorrer todo o conjunto para obter o resultado das operações.

A sobrecarga de execução deve ser considerada no projeto dos casos de teste de forma a não degradar a execução até o ponto onde não seja mais viável prosseguir com o teste. Essa medida de tolerância para a sobrecarga de execução depende,

[5] JVM (do inglês, *Java Virtual Machine*) é o interpretador de programas Java pré compilados.
[6] Em Java, este tipo de atributo é declarado através do modificador *static*.

principalmente, dos recursos de hardware disponíveis, tais como velocidade do processador, quantidade de memória ou espaço em disco. Uma questão primordial a ser levada em conta na seleção dos casos de teste é que os oráculos gerados pela ferramenta OZJ visam, essencialmente, o teste da funcionalidade da classe, não sendo adequados para os demais tipos de teste, como, por exemplo, o teste de carga ou o teste de *stress*. Embora este cuidado não evite que a tolerância de sobrecarga seja atingida, certamente ajuda a minimizar os seus efeitos.

5.7 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 5.2 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 5.10.

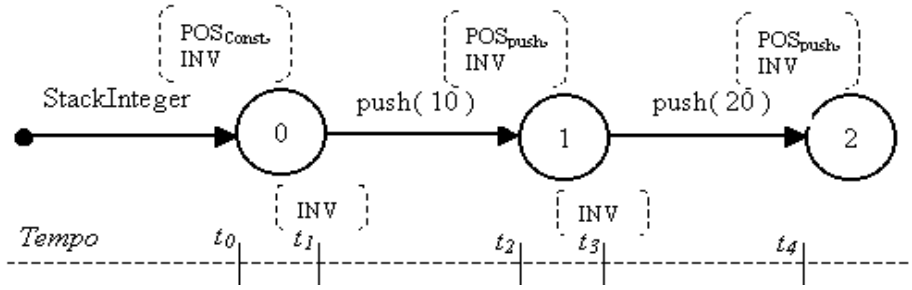


FIGURA 5.10 – Seqüência de chamadas a métodos da classe *StackInteger*

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 conforme sugerido na Seção 5.2, 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.

A Tabela 5.4 apresenta as alterações no estado concreto e abstrato pela execução do construtor da classe *StackInteger*.

TABELA 5.4 – Estado concreto e abstrato de *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 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 5.5 ilustra a execução do método push(), tendo o caso de teste "10" como entrada.

TABELA 5.5 – Estado concreto e abstrato de *StackInteger* 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?*, o valor 10 é atribuído à variável *new?* no espaço de variáveis abstrato. Em seguida, *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 5.6 apresenta a execução do método push(), tendo "20" como entrada.

TABELA 5.6 – Estado concreto e abstrato de StackInteger 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'*, 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 5.11.

Specification: StackInt

Implementation: StackInteger

Sequence trace

INIT applied to StackInteger()

POS_CONDITION → true

INVARIANT → true

push applied to void push(Integer item)

POS_CONDITION → true

INVARIANT → true

push applied to void push(Integer item)

POS_CONDITION

* list' = < new? > ⌢ list → false

- list' → < 10, 20 >

- list → < 10 >

- new? → 20

INVARIANT → undefined

FIGURA 5.11 – Exemplo de histórico de execução

A partir do erro detectado, a classe original deve ser corrigida e um novo oráculo deve ser gerado para a execução de um novo teste. A geração do novo oráculo, entretanto, só implica na alteração do mapeamento se ocorrerem mudanças na estrutura de dados, se novos métodos forem inseridos ou se métodos forem retirados da classe. Alterações nos tipos das variáveis podem acarretar em mudanças na função de abstração associada aos mapeamentos.

Para o exemplo citado, a correção para o erro seria apenas a troca da instrução

```
items.add( item );
```

para

```
items.insertElementAt( item, 0 );
```

Nesse caso, nenhuma alteração no mapeamento é necessária.

5.8 Resumo do Capítulo

Este capítulo abordou o tema central desta dissertação e apresentou uma estratégia para geração de oráculos para classe Java a partir de especificações Object-Z que considera os requisitos de independência do processo de desenvolvimento e independência da técnica de seleção de casos de teste. A estratégia foi delineada através de discurssões sobre os aspectos fundamentais dos geradores de oráculos apresentados no Capitulo 4. Baseado no conteúdo apresentado neste capítulo, o capítulo 6 descreve a as características, funcionalidades e a forma de utilização da ferramenta OZJ.

6 A Ferramenta OZJ

Este capítulo apresenta uma visão geral da ferramenta OZJ e está organizado da seguinte forma: a Seção 6.1 apresenta as características básicas requeridas pela estratégia de geração de oráculo adotada neste trabalho; a Seção 6.2 descreve as funcionalidades implementadas a partir das características; a Seção 6.3 apresenta alguns critérios que devem ser seguidos na composição do mapeamento; Seção 6.4 demonstra a forma de utilização da ferramenta e traça um perfil básico de quem pode utilizá-la e, por fim, a Seção 6.5 apresenta, em linhas gerais, o projeto de implementação da ferramenta OZJ.

6.1 Características

A ferramenta OZJ visa dar suporte à geração de oráculos para teste de *software* segundo a estratégia descrita no Capítulo 5. O objetivo da ferramenta é apoiar o projetista nas etapas que exigem a sua intervenção e automatizar as demais. De acordo com o descrito na Seção 5.4, a estratégia de mapeamento e a definição das funções de abstração devem ser realizadas pelo projetista e, as etapas relacionadas à tradução das expressões da especificação e à instrumentação do código são realizadas através de procedimentos automáticos.

Nesse contexto, a fim de satisfazer o objetivo delineado, o OZJ realiza as seguintes tarefas durante a geração dos oráculos:

- reconhece as estruturas passíveis de mapeamento na especificação;
- dá suporte à composição do mapeamento, propondo uma notação para sua representação;
- traduz automaticamente as expressões da especificação para Java;
- realiza automaticamente a instrumentação do oráculo.

A seguir, comenta-se cada uma dessas tarefas.

6.1.1 Reconhecimento da Especificação

A estratégia adotada pelo OZJ requer que, necessariamente, todas as estruturas da especificação devam estar mapeadas a estruturas correspondentes na implementação. Portanto, identificar essas estruturas na especificação constitui o primeiro passo da composição do mapeamento. A fim de apoiar o projetista nesta tarefa, o OZJ provê a análise sintática da especificação[7], a qual extrai as estruturas de controle e estruturas de dados passíveis de mapeamento.

Esta tarefa identifica toda definição de classe e de operação, e toda definição de variável de estado, variável de entrada ou variável de saída. No caso das variáveis, identifica-se tanto o identificador como o seu respectivo tipo.

[7] Neste trabalho, considera-se o reconhecimento sintático do subconjunto da linguagem Object-Z - Object-OZJ – descrito na Seção 5.1.1

6.1.2 Notação para Representação do Mapeamento

Para representar o mapeamento de forma sistemática, fez-se necessário definir uma linguagem que o descreva, chamada *OZJ*-mapping. Através de um documento escrito nesta linguagem, representa-se o mapeamento entre as estruturas de dados e de controle da especificação e da implementação. A ferramenta OZJ gera o documento de mapeamento contendo apenas as estruturas da especificação identificadas na tarefa de reconhecimento. Nesse caso, cabe ao projetista complementa-lo com as estruturas da implementação correspondentes. Vale ressaltar que a definição da função de abstração, apresentada no Capítulo 5, também faz parte da composição do mapeamento das estruturas de dados.

A gramática da linguagem *OZJ-Mapping* é apresentada no Anexo 3, segundo a notação de Wirth [WIR 77].

6.1.3 Tradução Automática de Expressões

A ferramenta OZJ é capaz de identificar na especificação as expressões que descrevem o comportamento esperado e traduzi-las para a notação de chamada a procedimentos, conforme apresentada na Seção 5.5. Além de traduzidas, as expressões são classificadas segundo seu tipo – pré-condição, pós-condição, invariante e valor de retorno – tal como apresentado na Seção 5.3.2, e associadas ao método da classe de implementação mapeado ao esquema de operação no qual a expressão está contida. Essa informação é usada durante a instrumentação para determinar o ponto onde a expressão traduzida será inserida no oráculo.

6.1.4 Instrumentação Automática do Oráculo

A instrumentação realizada em OZJ é dividida em duas fases: geração da classe *OracleManager* e a efetiva criação do oráculo.

A classe *OracleManager* agrupa as principais atividades realizadas durante a verificação do oráculo. Esta classe declara como atributos as variáveis que representam o estado abstrato do oráculo. Dessa forma, para cada constante, variável de estado, variável de entrada ou de saída declarada na especificação, OZJ gera um atributo correspondente em *OracleManager*. Além disso, o processo de instrumentação da ferramenta OZJ introduz como métodos, as funções de abstração e os métodos que avaliam as expressões traduzidas da especificação.

O oráculo, propriamente dito, é criado pela instrumentação de uma cópia da classe sob teste. Nesse caso, a instrumentação consiste na declaração de um objeto da classe *OracleManager* como atributo do oráculo, e na introdução de chamadas aos métodos de avaliação das expressões no início e no fim dos métodos da cópia da classe sob teste, tal como definido pela estratégia de verificação apresentada na Seção 5.3.

6.2 Funcionalidades

A utilização da ferramenta OZJ como gerador de oráculo fornece ao projetista as seguintes funcionalidades:

- apoio à composição do mapeamento
- apoio à definição das funções de abstração

- geração automática dos oráculos, a partir da especificação, da implementação e do mapeamento.

A seguir, comenta-se cada uma dessas funcionalidades.

6.2.1 Apoio à Composição do Mapeamento.

Construir o mapeamento através da ferramenta OZJ é vantajoso na medida que a ferramenta indica automaticamente quais estruturas de dados e de controle devem ser obrigatoriamente mapeadas. Tomando o exemplo proposto na Figura 5.2, onde se apresenta a especificação e uma implementação de uma pilha de números inteiros, a ferramenta OZJ gera o documento apresentado na Figura 6.1(a), sob a sintaxe da linguagem OZJ-Mapping. Os "?" indicam valores que devem ser preenchidos pelo projetista. A Figura 6.1(b), mostra o mapeamento completo.

O documento de mapeamento gerado indica a relação entre a classe, os esquemas de inicialização e os esquemas de operação com seus correspondentes na implementação, através das palavras-chaves *class*, *init* e *operation*, respectivamente. O mesmo ocorre para as constantes, as variáveis de estado e as variáveis de entrada e saída, através das palavras-chaves *constant*, *state* e *parameter*, respectivamente. Nota-se que para mapeamentos de dados, define-se ainda a relação entre os tipos das variáveis e constantes através da palavra-chave *type* e identifica-se qual função de abstração é responsável pela captura da informação. O número que as identificam, associado à palavra-chave *abstract*, corresponde ao seu identificador dentro do repositório de funções de abstração, o qual é descrito na Seção 6.2.2.

```
class( StackInt, ? ) {
    state( list, ? ) {
        type( \seq \integer, ? )
        :abstract(?)
    }
    init( init, ? ) {
    }
    operation( push, ? ) {
        parameter( new?, ? ) {
            type( \integer, ? )
            :abstract(?)
        }
    };
    operation( pop, ? ) {
        parameter( top, ? ) {
            type( \integer, ? )
            :abstract(?)
        }
    }
}
```

(a)

```
class( StackInt, StackInteger ) {
    state( list, items ) {
        type( \seq \integer, Vector )
        :abstract(01)
    }
    init( init, StackInteger() ) {
    }
    operation( push, push( Integer item ) ) {
        parameter( new?, item ) {
            type( \integer, Integer )
            :abstract(02)
        }
    };
    operation( pop, pop() ) {
        parameter( top, result ) {
            type( \integer, Integer )
            :abstract(02)
        }
    }
}
```

(b)

FIGURA 6.1 – Documento de mapeamento gerado entre *StackInt* e *StackInteger*

6.2.2 Apoio à Implementação das Funções de Abstração

A ferramenta OZJ gerencia um repositório de funções de abstração. A finalidade desse repositório é permitir a reutilização dessas funções em vários projetos de geração de oráculos. Dessa forma, ao definir um mapeamento de dados, o projetista deve

localizar, no repositório, uma função definida sobre os tipos usados pelo mapeamento em questão. Caso não exista tal função ou seja necessário definir uma função diferente sobre os mesmos tipos, o projetista deve associar ao mapeamento um identificador ainda não relacionado a qualquer função dentro do repositório. Isso, fará que a ferramenta gere o cabeçalho da nova função de abstração e solicite que o restante do corpo da função seja implementado. Tomando-se o exemplo da Figura 6.1(b), supondo que não exista uma função associada ao identificador "01", a ferramenta OZJ gera o código apresentado na Figura 6.2.

```
//:#01 ( \seq \integer, Vector )
public void abstract_#absVarName#( Vector concrete ) {
    Sequence abstractVar = new Sequence();
    // Implement here

    this.#absVarName# = abstractVar;
}
```

FIGURA 6.2 – Código gerado para uma nova função de abstração

O código gerado indica na primeira linha, sob forma de comentário, o identificador numérico da função e os tipos abstrato e concreto do mapeamento, respectivamente. A seguir, assinatura da função declara um parâmetro formal com o tipo concreto – *Vector*. A primeira linha do corpo da função declara uma variável temporária com o tipo abstrato – *Sequence* – a qual deve ser usada para receber o conteúdo da variável concreta. A partir desse modelo, o projetista deve escrever o código que preenche a variável de tipo abstrato com valores correspondentes ao conteúdo da variável concreta. A Figura 6.3 contém um exemplo de implementação dessa função de abstração pronta para ser introduzida no repositório.

Nota-se que o modelo gerado pela ferramenta introduz o marcador *#absVarName#* na definição do nome da função e na última linha de código do modelo gerado. Esse marcador é posteriormente substituído pelo nome da variável abstrata usada no mapeamento.

```
//:#01 ( \seq \integer, Vector )
public void abstract_#absVarName#( Vector concrete ) {
  Sequence abstractVar = new Sequence();
  // Implement here
  ZMTKInteger item;
  for( int i = 0; i < concrete.size(); i++ ) {
    item = new ZMTKInteger( ( (Integer)concrete.elementAt( i ) ).intValue() );
    abstractVar.insert( item );
  }
  this.#absVarName# = abstractVar;
}
```

FIGURA 6.3 – Implementação de função de abstração adicionada ao repositório de funções

O exemplo de função apresentado recebe um objeto da classe *Vector* contendo objetos da classe *Integer* como elementos. Tanto *Vector* como *Integer* são oriundos da biblioteca padrão do Java. Para cada elemento contido no objeto *Vector*, um objeto *ZMTKInteger* é criado e inserido na instância de *Sequence*.

Uma vez definido o corpo da função de abstração, ela é adicionada ao repositório. Quando seu identificador é referenciado em um mapeamento, a função de abstração

correspondente é inserida no corpo de da classe *OracleManager* e o marcador é substituído pelo identificador da variável abstrata do mapeamento. A Figura 6.4 mostra o código da função de abstração gerado para o mapeamento entre a *list* e *items*.

```
//:#01 ( \seq \integer, Vector )
public void abstract_list( Vector concrete ) {
  Sequence abstractVar = new Sequence();
  // Implement here
  ZMTKInteger item;
  for( int i = 0; i < concrete.size(); i++ ) {
    item = new ZMTKInteger( ( (Integer)concrete.elementAt( i ) ).intValue() );
    abstractVar.insert( item );
  }
  this.list = abstractVar;
}
```

FIGURA 6.4 – Exemplo de código de função de abstração inserido na classe *OracleManager*

Nota-se que o marcador *#abstractVarName#* foi substituído pelo identificador da variável abstrata do mapeamento, *list*.

6.2.3 Geração Automática do Oráculo

Uma vez definido o mapeamento entre a especificação e a implementação, o processo de geração de oráculos através da ferramenta OZJ é automático. Ao final do processo, o projetista tem a sua disposição uma nova classe instrumentada com instruções que verificam se o comportamento está de acordo com o especificado.

6.3 Critérios para Composição do Mapeamento

A partir da descrição das funcionalidades previstas para a ferramenta OZJ, observa-se que a principal tarefa do projetista no processo de geração de oráculo é a composição do mapeamento.

O mapeamento é uma tarefa criativa e, nesta abordagem, está baseado no conhecimento do projetista a respeito do significado tanto das estruturas da especificação como das estruturas da implementação. Embora seja uma tarefa criativa e, até certo ponto, com parâmetros subjetivos, é necessário definir alguns critérios que regulam a sua criação. Esses critérios estão fundamentados na semelhança estrutural entre as linguagens de especificação e implementação adotadas para a ferramenta.

O sentido adotado para o mapeamento é sempre da estrutura mais concreta para a estrutura mais abstrata. Dessa forma, as estruturas da especificação são usadas como ponto de referência, indicando o que deve ser mapeado. Isso significa que para completar o mapeamento, todos os itens da especificação devem obrigatoriamente ser representados por uma estrutura correspondente na implementação.

Tendo a especificação definida, a tarefa de compor o mapeamento deve seguir critérios, definidos de acordo com o tipo da estrutura a ser mapeada. A Figura 6.4 ilustra a especificação da classe *Account* para o controle de uma conta bancária, onde é permitido estipular um valor limite para débitos, realizar saques e consultas ao saldo. Uma implementação para a classe *Account* é sugerida com o intuito de ilustrar algumas situações descritas nas próximas seções.

6.3.1 Classes

Uma vez que os conceitos de classe em Object-Z e Java são equivalentes, o mapeamento consiste em assumir que uma classe Java implementa o comportamento descrito na especificação. No que diz respeito à estratégia de verificação, apresentada na seção 5.3, esse mapeamento significa que todas as restrições impostas aos estados da especificação da classe serão aplicadas aos seus estados correspondentes na implementação.

Ao definir um mapeamento, é possível observar ou não questões de hierarquia. A seguir, apresenta-se o procedimento adotado diante das diversas possibilidades envolvendo hierarquias de classe.

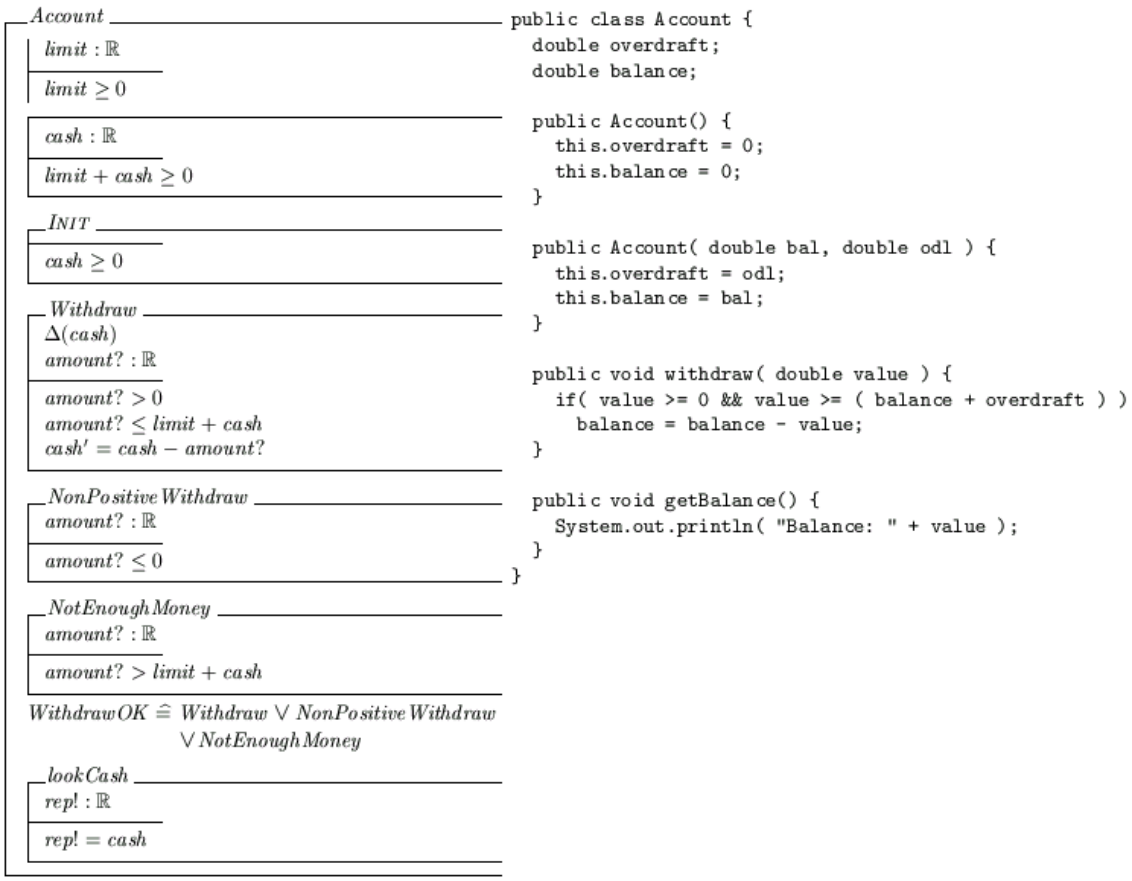


FIGURA 6.5 – Especificação e uma implementação da classe *Account*

- *A especificação não possui super-classe*

Quando uma especificação não é especializada, as suas restrições devem ser aplicadas a uma classe de implementação sem considerar que esta última possui super-classes. Dessa forma, toda a informação necessária para mapear os atributos e as operações da classe de especificação deve ser extraída das estruturas de dados acessíveis à classe de implementação. Isso é equivalente a considerar que a super-classe da classe sob teste está correta. Exemplos típicos deste caso são as classes de usuário que descendem diretamente das classes dos pacotes padrões da linguagem Java. A classe *StackInteger*, apresentada na Figura 5.2 da Seção 5.2, por exemplo, descende diretamente da classe *Object* de Java.

- *A especificação possui super-classe*

Quando a especificação é construída através de uma hierarquia de classes, é possível tirar proveito dessa estrutura se as classes de implementação seguem uma hierarquia correspondente. Por exemplo, supondo que uma especificação E_3 herda características de uma especificação E_2, que por sua vez, herda características da especificação E_1, e supondo que E_1, E_2 e E_3 especificam o comportamento de uma hierarquia de classes de implementação I_1, I_2 e I_3, então é possível compor um mapeamento M_1 de I_1 para E_1, e herdar M_1 para compor o mapeamento M_2 de I_2 para E_2, e assim sucessivamente.

O conceito de herança associado à composição do mapeamento funciona da mesma forma que o conceito do paradigma orientado a objetos, ou seja, ao mapear uma subclasse de uma especificação, aproveita-se os mapeamentos dos atributos herdados, redefine-se os mapeamentos para os atributos redefinidos e define-se novos mapeamentos para os atributos novos. O mesmo procedimento pode ser adotado para operações. Um exemplo desta situação é visto na Figura 5.4 da Seção 5.2, onde o mapeamento feito entre a especificação *StackInt* e a implementação *StackInteger* foi herdado para compor o mapeamento entre a especificação *StackNat* e a implementação *StackNatural*.

6.3.2 Constantes

O conceito de variáveis constantes em Object-Z é idêntico ao conceito presente nas linguagens de programação: variáveis que não podem ser alteradas. Para efeito de mapeamento, entretanto, a estrutura correspondente na implementação não precisa ser necessariamente uma constante Java – definida através do modificador *final*. O importante é que seu valor se mantenha inalterado durante a execução dos testes. Logo, a definição de um mapeamento para uma constante pode ser feita sobre qualquer variável ou estrutura de dados que não se modifique durante o teste.

A Figura 6.5 exemplifica essa situação. A especificação declara uma constante - *limit*, a qual define o limite para débito permitido na conta. Nota-se que a implementação representa esse valor como uma variável - *overdraft*, que recebe seu valor inicial no construtor e não é alterado posteriormente. Nesse caso, *overdraft* tem a mesma semântica que a constante *limit*, e um mapeamento entre essas variáveis pode ser definido mesmo que *overdraft* não seja uma variável constante em Java. A Figura 6.6 exemplifica o mapeamento de constantes e variáveis de estado da classe *Account* segundo a notação OZJ-Mapping.

```
class( Account, Account ) {
   constant( limit, overdraft ) {
      type( \real, double )
      :abstract(03)
   }
   state( cash, balance ) {
      type( \real, double )
      :abstract(03)
   }
```

FIGURA 6.6 – Mapeamento para variáveis constantes e variáveis de estado da classe *Account*

Vale ressaltar que a definição de constantes na especificação acarreta na definição automática de uma assertiva que verifica se seu valor permanece inalterado ao longo da execução do programa.

6.3.3. Variáveis de Estado

Em Object-Z as variáveis de estado são representadas pelas variáveis definidas dentro do esquema de estado. A especificação da classe *Account*, na Fig 6.4, contém apenas uma variável de estado: *cash*.

Para que o oráculo seja capaz de avaliar o resultado das assertivas, é necessário mapear as estruturas de dados da implementação que contenha valores adequados a cada variável de estado da especificação. Em Java representa-se as variáveis de estado da classe pelos atributos de classe. Entretanto, o mapeamento entre as variáveis definidas no esquema de estados e os atributos de classe não é, necessariamente, direto. Isso ocorre porque, por motivos de implementação, é possível que uma variável de estado da especificação tenha como estrutura de dados correspondente na implementação, não um atributo de classe, mas uma outra estrutura, que pode ser resultante, por exemplo, de uma chamada de método ou de um cálculo aritmético. O exemplo da Figura 5.8, contido na Seção 5.2, exemplifica essa situação pelo mapeamento:

$$length \rightarrow list.length().$$

Ou seja, a chamada ao método *length*() da classe *StringBuffer* representa, na implementação, a variável *length* da especificação.

Entretanto, a utilização de chamadas a métodos apresenta um inconveniente, que deve ser tratado com cuidado pelo projetista: métodos que alteram o estado da classe não podem ser usados nos mapeamentos. Utilizar um método em um mapeamento significa que este método será chamado durante a captura das informações. Ou seja, serão realizadas chamadas não previstas na implementação original. Se o método altera o estado da classe durante a captura da informação, fica caracterizado que o oráculo não reproduz fielmente as mudanças de estado previstas na implementação original.

6.3.3 Inicialização

A inicialização do estado de uma especificação de classe em Object-Z é representada através do esquema *INIT*. Este esquema é composto por expressões que impõem as condições iniciais às variáveis de estado. Em Java, as variáveis de estado recebem seus valores iniciais nos construtores. Logo, após a execução do construtor a implementação da classe assume seu estado inicial. Devido a essas semelhanças, os esquemas de inicialização devem ser obrigatoriamente mapeados para o construtor da classe de implementação.

Embora o esquema de inicialização e o construtor exerçam o mesmo papel na especificação e na implementação, Java permite sobrecarga na declaração dos construtores[8] e Object-Z permite a declaração de apenas um esquema de inicialização por classe. Isso gera um problema, pois é necessário verificar as expressões definidas no esquema de inicialização em todos os construtores definidos na implementação.

Esta questão é resolvida mapeando-se todos os métodos construtores para o esquema de inicialização. Isso significa que o estado inicial produzido por cada construtor deve, necessariamente, satisfazer as expressões do esquema de inicialização.

Além disso, eventualmente, os esquemas de inicialização possuem parâmetros de entrada. Nesses casos, deve-se mapear esses parâmetros para alguma estrutura de dados

[8] Em linguagens orientadas a objetos, a sobrecarga de métodos permite a definição de vários métodos com o mesmo nome mas com assinaturas diferentes.

acessível dentro dos construtores. Em geral, tais estruturas são os parâmetros formais do construtor. Na verdade, um dos motivos da utilização de construtores sobrecarregados nas implementações é exatamente permitir uma variação de quantidade e de tipos dos parâmetros formais na inicialização da classe, com o intuito de se obter maior flexibilidade no uso.

O exemplo da Figura 6.5 ilustra o uso de construtores sobrecarregados na implementação, onde um construtor cria uma conta com saldo e limite zero, e outro permite que valores iniciais sejam estipulados. A Figura 6.7 apresenta o mapeamento para este caso.

```
init( init, Account() )  {
}
init( init, Account( double bal, double odl ) {
}
```

FIGURA 6.7 – Mapeamento para construtores sobrecarregados

No trecho de mapeamento da Figura 6.7, os construtores declarados na implementação devem satisfazer a condição definida no esquema de inicialização. Além disso, nota-se que parâmetros formais não são declarados na especificação logo, nesse exemplo, não é necessária a definição de mapeamentos para os parâmetros formais declarados na implementação.

6.3.4 Operações

As operações representam as atividades que alteram o estado da classe ou produzem algum resultado. Object-Z representa as operações através de esquemas de operação e Java através de métodos de classe. Logo, considerando que uma especificação descreve o comportamento de uma determinada implementação, os esquemas de operação são naturalmente mapeados para seus métodos de classe correspondentes.

O mapeamento entre operações é direto se a especificação e a operação satisfazem os requisitos de homogeneidade total, descritos anteriormente na seção 5.4. Assim como nos esquemas de inicialização, os esquemas de operação também podem receber parâmetros de entrada, os quais, também devem ser mapeados para os valores correspondentes na implementação. Além disso, o inconveniente provocado pelos métodos sobrecarregados e pela variação entre os parâmetros formais pode ser tratado nas operações da mesma forma descrita para os esquemas de inicialização.

Os esquemas de operação permitem a utilização das variáveis de saída, que representam os valores que uma operação deve fornecer como resultado. Os valores retornados pelos métodos são correspondentes, em Java, às variáveis de saída. Logo, as variáveis de saída, geralmente, devem ser mapeadas para os valores de retorno. Esta situação ocorre quando a variável de saída descrita no esquema de operação corresponde exatamente ao valor retornado pela instrução *return* da linguagem Java, tal como no exemplo da Figura 5.2. Infelizmente, esta situação ideal nem sempre ocorre e algumas questões devem ser consideradas.

Primeiramente, é possível que um valor de saída descrito na especificação seja implementado não como o valor do retorno da função e sim como um valor gravado em um dispositivo de saída, tal como o monitor, e não é representado, pelas variáveis de estado da classe. Para esta situação, a ferramenta OZJ dispõe de um tipo especial que captura os valores enviados ao dispositivo de saída. Este tipo chama-se *OracleOutputStream* e deve ser considerado como o tipo da implementação no

mapeamento. Embora o uso do tipo *OracleOuputStream* viabilize a verificação de saídas impressas, é possível que a definição da função de abstração se torne difícil, devido à falta de estrutura deste tipo de resposta. A Figura 6.8 ilustra o uso do tipo *OracleOutputStream* em um mapeamento.

```
operation( lookCash, getBalance() ) {
   parameter( rep!, oracleOutput ) {
      type( \real, OracleOutputStream )
      :abstract(04)
   }
}
```

FIGURA 6.8 – Mapeamento utilizando o tipo *OracleOutputStream*

O exemplo da Figura 6.8 apresenta um caso de mapeamento onde o esquema *lookCash*, apresentado na Figura 6.5, especifica que o resultado da operação deve ser igual ao valor da variável *cash*. A implementação dessa operação apenas imprime o valor correspondente no dispositivo de saída, configurando um caso típico para o uso do *OracleOutputStream*. Quando este artifício é usado, automaticamente uma variável chamada *oracleOutput* é criada pelo gerador do oráculo, a qual recebe todas as saídas emitidas pelo método em questão.

Para construir a função de abstração para esse tipo de mapeamento, o projetista deve conhecer o formato da saída emitida de forma a extrair as informações necessárias. A Figura 6.9 apresenta a função de abstração para o exemplo apresentado.

```
//:#04 ( \real, OracleOutputStream )
public void abstract_#absVarName#( OracleOutputStream concrete ) {
    ZMTKReal abstractVar;
    // Implement here
    String line = concrete.getLine( 0 );
    // Get just numeric value
    Double value = new Double( value.substring( 10 ) );
    abstractVar = new ZMTKReal( value );

    this.#absVarName# = abstractVar;
}
```

FIGURA 6.9 – Exemplo de função de abstração utilizando o tipo *OracleOutputStream*

O método *getLine()* da classe *OracleOutputStream* retorna as linhas capturadas segundo a ordem em que elas são impressas no dispositivo de saída.

Outro inconveniente um pouco mais complicado ocorre quando os métodos são implementados com vários pontos de saída, ou seja, existem várias instruções *return* dispostas no interior do método. Essa situação inviabiliza o mapeamento, uma vez que a estratégia de validação, apresentada na Seção 5.3, assume que o estado em que o método deixou a classe será analisado imediatamente depois da última instrução, ou imediatamente antes da instrução *return*, para o caso de funções. Para permitir o mapeamento nesse caso, é necessário que a implementação seja alterada, de forma a conter apenas uma instrução *return*, imediatamente antes do final do método.

6.3.5 Exceções

A especificação do comportamento correto de uma operação nem sempre fornece dados suficientes para a verificação da implementação. Freqüentemente, é necessário que se defina o que ocorre quando valores não previstos são utilizados. Em Object-Z,

uma forma comum para a representação do comportamento correto e para a representação das exceções de uma operação consiste em descrever cada caso em esquemas distintos, onde as expressões que descrevem as exceções são mutuamente exclusivas em relação à expressão que descreve o comportamento correto. Posteriormente, define-se um outro esquema como a disjunção entre o esquema do comportamento correto e os esquemas das exceções.

Nesses casos, o mapeamento deve feito sobre o esquema definido através da disjunção. Dessa forma, a verificação aplicada ao método da implementação será válida se o comportamento detectado estiver de acordo com o esquema que descrever o comportamento correto ou com pelo menos um dos esquemas de exceção.

A especificação apresentada na Figura 6.5 exemplifica essa situação. A operação *Withdraw* descreve o comportamento correto para um saque, impondo duas pré-condições: 1) o valor retirado – *amount* – deve ser positivo e 2) o valor retirado não pode ser maior que a soma do saldo da conta – *cash* – com o limite estipulado – *limit*. Para representar as exceções, dois outros esquemas são declarados: *NonPositiveWithdraw* e *NotEnoughMoney*. Ambos são definidos pela negação das pré-condições 1 e 2, respectivamente. Para representar uma operação que prevê tanto o comportamento correto como as exceções, declara-se a operação *WithdrawOK* pela disjunção de *Withdraw*, *NonPositiveWithdraw* e *NotEnoughMoney*.

Para o exemplo apresentado, o mapeamento deve ser feito apenas entre *WithdrawOK* e o método *withdraw()* da implementação, como mostra a Figura 6.10.

```
operation( WithdrawOk, withdraw( double value ) ) {
   parameter( amount?, value ) {
      type( \real, double )
      :abstract(03)
   }
}
```

FIGURA 6.10 – Mapeamento definido a partir de uma disjunção de esquemas

Nesse caso, no oráculo gerado para a classe *Account* todas as expressões dos esquemas são aplicadas ao método da implementação. Na execução do teste, cada esquema é avaliado separadamente e *WithdrawOK* será válido se o *withdraw* for válido ou se pelo menos um esquema que define exceções for considerado válido. Por exemplo, se o esquema de exceção *NotEnoughMoney* for verdadeiro, conseqüentemente, pela exclusão mútua, *Withdraw* será falso. Lembrando-se que, como o esquema *NotEnoughMoney* não declara alterações de estado[9], logo, para que ele se torne verdadeiro, é necessário que o estado da implementação permaneça inalterado. Dessa forma, se *NotEnoughMoney* for verdadeiro implica que um valor maior que a soma do saldo com o limite da conta foi informado e que o estado da classe não mudou. Ou seja, o saque não ocorreu devido ao saldo insuficiente. O esquema *NonPositiveWithdraw* pode ser analisado de forma semelhante.

6.4 Utilização da Ferramenta OZJ

De acordo com a descrição das características e das funcionalidades, apresentada nas seções 6.1 e 6.2, o uso da ferramenta OZJ apresenta dois pontos de interação com o usuário: 1) a geração do documento de mapeamento, contendo as estruturas da

[9] Alterações de estado são declaradas através do operador Δ

especificação que requerem um mapeamento; 2) e a geração do oráculo propriamente dita, dada uma especificação, uma implementação e o mapeamento.

Nesse contexto, esta seção descreve o perfil necessário ao projetista responsável pela geração do oráculo e os procedimentos que devem ser seguidos para utilização da ferramenta OZJ.

6.4.1 Perfil do Projetista do Oráculo

Ao longo deste trabalho, cita-se a figura do projetista responsável pela geração do oráculo como um componente da equipe de teste que possui conhecimento tanto da estrutura da especificação como da implementação. Esse profissional será chamado nessa seção de projetista do oráculo.

Considerando-se um ambiente de desenvolvimento cuja função do projetista de teste é diferenciada do analista de sistema, do projetista do *software* e dos programadores, deve-se considerar a necessidade de interações entre o projetista do oráculo e aqueles que construíram a especificação. Essas interações são importantes para determinar a semântica correta de cada estrutura contida na especificação. Além disso, em casos onde a especificação não é formal, ou foi formalizada em outra linguagem de especificação, cabe ao projetista do oráculo a tarefa de auxiliar o especificador para que seja construída uma especificação em Object-Z[10] que satisfaça, pelo menos, os requisitos de homogeneidade parcial, definido na Seção 5.4. Assim, para determinar se os requisitos de homogeneidade parcial foram satisfeitos é necessário, também, que o projetista do oráculo interaja com a equipe de programação a fim de determinar a semântica correta das estruturas da implementação.

Quando se considera ambientes de desenvolvimento onde o mesmo profissional acumula funções diferentes, o papel de projetista de oráculo pode ser exercido por qualquer outro profissional envolvido no processo. Muito embora se deva evitar que componentes da equipe de programação participem do teste [MYE 79].

6.4.2 Procedimentos de Utilização

A utilização da ferramenta OZJ segue os passos, sugeridos no diagrama de funcionamento, apresentado na Figura 6.11. A seguir, descreve-se cada um dos passos.

- Criação do projeto de oráculo

  No contexto funcional da ferramenta OZJ, um projeto de oráculo consiste de um conjunto de arquivos que armazenam informações obtidas através do processo de análise sintática e semântica de cada um dos artefatos manipulados durante o processo, ou seja, a especificação, o mapeamento e a implementação. Essas informações são utilizadas também durante a execução do teste para armazenar o resultado da avaliação das assertivas e na geração do histórico de execução.

  Para criar um novo projeto de oráculo, o projetista deve submeter à ferramenta OZJ o nome do novo projeto, a especificação e a sua respectiva implementação, através da seguinte linha de comando:

```
java OZJTool newproject <nome do arquivo do projeto>
                        <nome do arquivo da especificação>
                        <nome do arquivo da implementação>
```

[10] Para o trabalho em questão considera-se, ainda, as restrições definidas pela linguagem Object-OZJ

Para a especificação e a implementação da Figura 6.5, o comando para criação de um projeto chamado "projAccount" seria:

```
java OZJTool newproject projAccount account.tex account.java
```

onde *account.tex* e *account.java* correspondem à especificação e à implementação apresentadas na Figura 6.5.

Após a execução deste comando, o projeto de oráculo chamado *projAccount.ozj* é criado. Observa-se que o arquivo da especificação *account.tex* é um documento Latex, o qual segue a sintaxe desenvolvida pelos criadores da linguagem Object-Z. Maiores detalhes sobre a criação de especificações Object-Z em Latex podem ser encontrados em [WEN 96]

- Criação do documento de mapeamento

  Para criar o documento de mapeamento, o projetista deve executar o seguinte comando:

  ```
  Java OZJTool createmaping <Nome do projeto de oráculo>
  ```

  Esta instrução gera um arquivo contendo as estruturas que requerem mapeamento na especificação.

  Para gerar o documento do mapeamento para o exemplo da Figura 6.5, o comando seria:

  ```
  java OZJTool createmapping projAccount.ozj
  ```

  Este comando cria o arquivo de mapeamento *projAccount.map* como conteúdo apresentado ao longo da Seção 6.2, sem definir as estruturas da implementação relacionadas.

- Composição do mapeamento

  Cabe ao projetista, de acordo com os critérios apresentados na Seção 6.3, completar o documento de mapeamento com as estruturas da implementação correspondentes.

- Criação do oráculo e implementação das funções de abstração

  A criação do oráculo é feita através do comando:

  ```
  java OZJTool createoracle <Nome do Projeto de oráculo>
  ```

  Para o exemplo da Figura 6.4, a comando seria:

  ```
  java OZJTool createoracle projAccount.ozj
  ```

  Este comando cria a classe *OracleManager* e o oráculo. Caso o projetista tenha especificado identificadores de funções de abstração não existentes no repositório, o cabeçalho dessas funções é criado, tal como apresentado na Figura 6.2. Nesse caso, para finalizar a geração do oráculo o projetista deve implementar o corpo das novas funções de abstração.

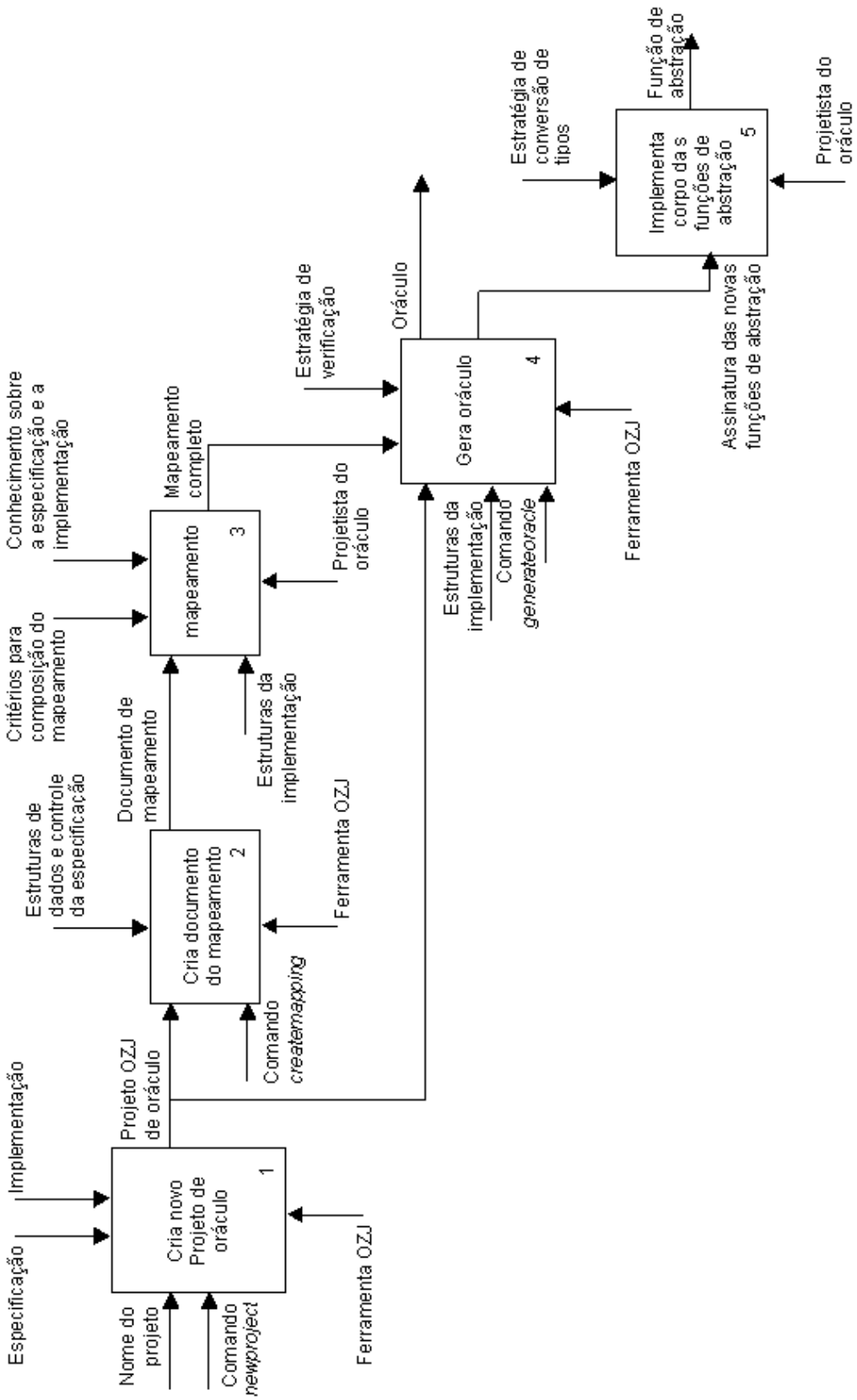


FIGURA 6.11 – Diagrama de funcionamento da ferramenta OZJ

6.5 Considerações sobre a Implementação do Protótipo do OZJ

O protótipo da ferramenta OZJ está implementado em Java, sendo dividido em dois pacotes:

- Pacote OZJ

  Contém as classes que realizam as atividades de verificação, descritas ao longo deste capítulo e do Capítulo 5.

- Pacote ZMTK

  Contém a implementação dos tipos da linguagem Object-Z.

Utiliza-se ainda um pacote Java chamado CPOO (Compilador Parametrizado Orientado a Objetos) [AZE 98], o qual cria reconhecedores a partir de linguagens descritas segundo a notação de Wirth [WIR 77] – também conhecida como BNF estendida. Para este trabalho, as classes do CPOO, originalmente implementadas em C++, foram traduzidas para Java para que pudessem ser usadas internamente pela implementação do OZJ sem problemas de compatibilidade. Através desse pacote, são criados os reconhecedores para a linguagem Object-Z, para a linguagem Java e para a linguagem OZJ-Mapping e o processo de tradução das expressões e a geração das instruções que compõem a instrumentação são realizados.

A seguir, apresenta-se o projeto dos pacotes OZJ e ZMTK.

6.5.1 Pacote OZJ

O pacote OZJ é composto por duas classes: *OZJTool* e *OZJManager*, tal como ilustra o diagrama de classe da Figura 6.12.

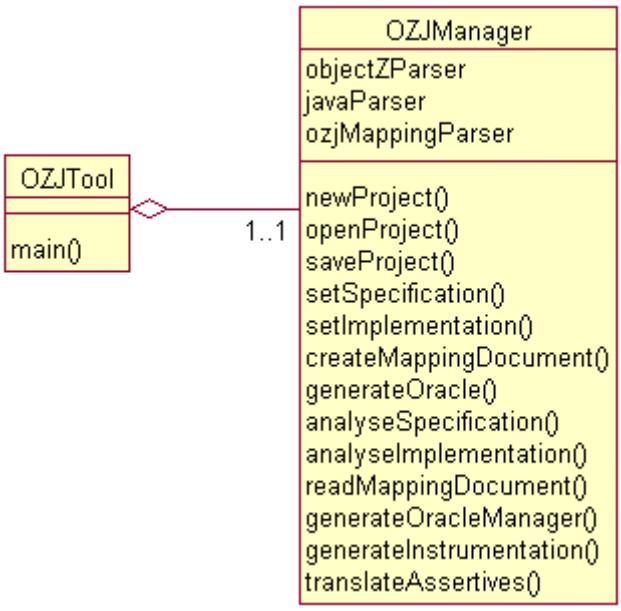


FIGURA 6.12 – Diagrama de classes do pacote OZJ

A classe *OZJTool* é uma aplicação Java cujas funções são receber os parâmetros passados à ferramenta, identificar a função requerida e chamar os métodos da classe *OZJManager* de acordo com a operação requisitada. A instância de *OZJManager*

agregada à ferramenta *OZJTool* realiza todo o trabalho necessário para geração do oráculo.

6.5.2 Pacote ZMTK

O pacote ZMTK consiste na implementação dos tipos da linguagem Object-Z. Uma vez que a linguagem Z representa as entidades do mundo real como conjuntos [SPY 92], a implementação do ZMTK é uma representação da Teoria dos Conjuntos. O diagrama de classes da Figura 6.13 ilustra essa característica.

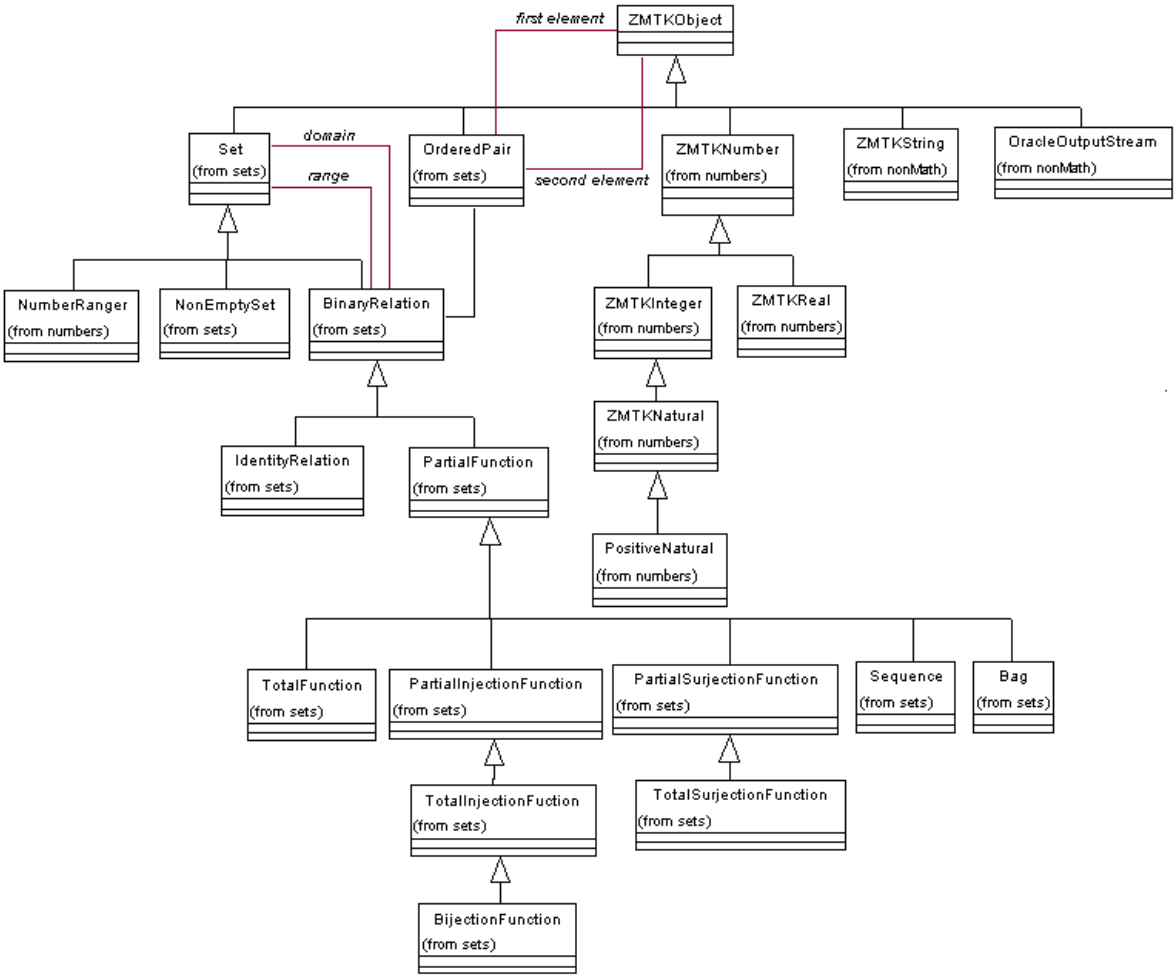


FIGURA 3 – Diagrama de classes do Pacote ZMTK

Pelo diagrama da Figura 6.13, todas as classes descendem da classe abstrata *ZMTKObject*, cuja única função é prover um ancestral comum a todas as demais classes do pacote ZMTK. Seis classes descendem diretamente de *ZMTKObject*: *OrderedPair*, *ZMTKString*, *OracleOutputStream*, Set, ZMTKBoolean e *ZMTKNumber*.

A classe *OrderedPair* é a representação para os pares ordenados, contendo sempre duas instâncias de qualquer descendente de *ZMTKObject*.

A classe *ZMTKString* consiste de uma representação equivalente à classe String de Java. Embora, originalmente, não se defina esse tipo na Teoria dos Conjuntos, define-se esse tipo na linguagem Object-OZJ, uma vez que esta classe facilita a representação abstrata de cadeias de caracteres, as quais são de uso comum nas implementações Java.

A classe *OracleOutputStream* é usada para capturar saídas impressas nos dispositivos de saída. Na prática, objetos dessa classe capturam o conteúdo dos objetos *System.output* de Java, os quais recebem todas as saídas emitidas por aplicações Java. A Seção 6.3.4 apresenta um exemplo de utilização prática dessa classe.

A classe *ZMTKNumber* é outra classe abstrata, da qual descendem as representações para valores numéricos disponíveis na linguagem Z. Para este trabalho definiu-se as operações aritméticas básicas. Os valores numéricos previstos são os números naturais, os números inteiros e os números reais, para os quais são definidas duas subclasses, uma classe para representar os números inteiros, chamada *ZMTKInteger*, e outra para os números reais, chamada *ZMTKReal*. Os números naturais são representados pela *ZMTKNatural*, a qual é uma especialização de *ZMTKInteger* restrita a valores não negativos. Representa-se ainda os números estritamente positivos através da classe *PositiveNatural*, a qual é uma especialização de *ZMTKNatural* restrita aos valores naturais e não negativos.

A classe *Set* representa os conjuntos da linguagem Object-Z e implementa suas operações. *Set* é a classe base para as demais entidades da Teoria dos Conjuntos. A partir dela define-se classes, tais como *BinaryRelation*, que representa as relações binárias como um conjunto de pares ordenados. Seguindo essa linha, definiu-se a classe *PartialFunction*, que representa as funções parciais como uma relação binária onde cada elemento do domínio tem pelo menos um correspondente no contra domínio. *PartialFunction*, por sua vez, é a superclasse de todos os demais tipos de funções, tais como, *TotalFunction* que representa as funções totais e é definida como uma função parcial onde todos os elementos do domínio têm um valor correspondente no contra-domínio; *PartialInjectionFunction* que representa as funções injetoras parciais e é definida como uma função parcial onde cada elemento do domínio tem apenas um correspondente no contra-domínio. O mesmo processo se aplica as demais funções injetoras e sobrejetoras representadas pelas classes *PartialSurjectionFunction*, *TotalSurjectionFunction*, *TotalInjectionFunction* e *BijectionFunction*, de acordo com a hierarquia apresentada na Figura 6.13.

A partir de *PartialFunction*, define-se ainda duas funções especiais, chamadas *Sequence* e *Bag*. A classe *Sequence* relaciona os elementos do contra-domínio com uma seqüência numérica ordenada. A classe *Bag* representa uma seqüência de elementos onde cada elemento do domínio tem no contra-domínio o seu número de ocorrências.

Uma documentação reduzida do pacote ZMTK é apresentada no Anexo 2.

6.6 Resumo do Capítulo

Este capítulo apresentou a ferramenta OZJ através das suas características e funcionalidades. Foram descritos, ainda, o perfil do projetista do oráculo, a forma de utilização da ferramenta a fim de apoiar o projetista na tarefa de geração dos oráculos do oráculo. Apresentou-se, também, resumidamente, o projeto de implementação da ferramenta OZJ e do pacote que implementa os tipos da Linguagem Z. Após a definição da estratégia e da ferramenta de apoio, o próximo capítulo apresenta alguns estudos de casos que ilustram a capacidade de detecção dos oráculos gerados pela ferramenta OZJ.

7 Estudos de Caso

Este capítulo apresenta dois estudos de caso retirados de exemplos clássicos da literatura sobre teste de *software* e sobre especificação formal. Os exemplos ilustram como um projetista de oráculo deve proceder para preparar a especificação formal como fonte de informação para os oráculos e como se deve analisar as informações fornecidas pelo histórico de execução quando erros são detectados. O capítulo está organizado em duas seções: a Seção 7.1 apresenta um exemplo de geração de oráculos para o problema da classificação de triângulos e a Seção 7.2 apresenta um oráculo gerado a partir do problema clássico da especificação de uma agenda telefônica.

7.1 Classificação de Triângulos

O problema da classificação de triângulos segundo o comprimento dos seus lados é um exemplo clássico da literatura sobre teste de *software*. Este exemplo foi primeiramente introduzido por Myers [MYE 79]. Várias abordagens para a geração de casos de teste apresentam exemplos de conjuntos de casos de teste para a verificação de implementações desse problema [STO 93, MEU 95, OFF 99].

A especificação informal do problema, proposta por Myers, é:

"Um programa que lê três valores inteiros, os quais representam o comprimento dos lados de um triângulo, deve responder se o triângulo *é escaleno, isósceles ou eqüilátero."*

A essa especificação acrescenta-se que o programa deve informar se o triângulo é inválido, caso os valores informados não correspondam aos lados de um triângulo.

7.1.1 A Especificação Formal

O primeiro passo para construção do oráculo através da ferramenta OZJ é a criação da uma especificação formal do problema em Object-Z.

A Figura 7.1 apresenta a especificação formal usada como fonte de informação para o oráculo deste estudo de caso. Esta especificação é uma versão orientada a objetos de uma especificação proposta por Stocks [STO 94] para o problema da classificação de triângulos. A versão apresentada nesse trabalho define uma classe *Triangle* e como estado da classe define três variáveis inteiras x, y e z, que representam o comprimento dos lados do triângulo. Duas operações são previstas sobre o estado da classe: *setSides* e *getType*.

A operação *setSides* especifica que os parâmetros de entrada devem ser atribuídos às variáveis de estado e que estes valores devem ser positivos. Esta operação é declarada pela disjunção da operação *setSidesSuccessful*, a qual define o comportamento correto de *setSides*, e *invalidSide*, que especifica o que deve ocorrer em caso de exceção.

A operação *getType* determina o tipo do triângulo de acordo com o número de lados de mesmo comprimento através da disjunção entre as operações *getTypeSuccessful* e *invalidTriangle*. A operação *getTypeSuccessful* determina o tipo do triângulo, se os lados são válidos, e a operação *invalidTriangle* descreve o comportamento esperado quando os lados não correspondem a um triângulo.

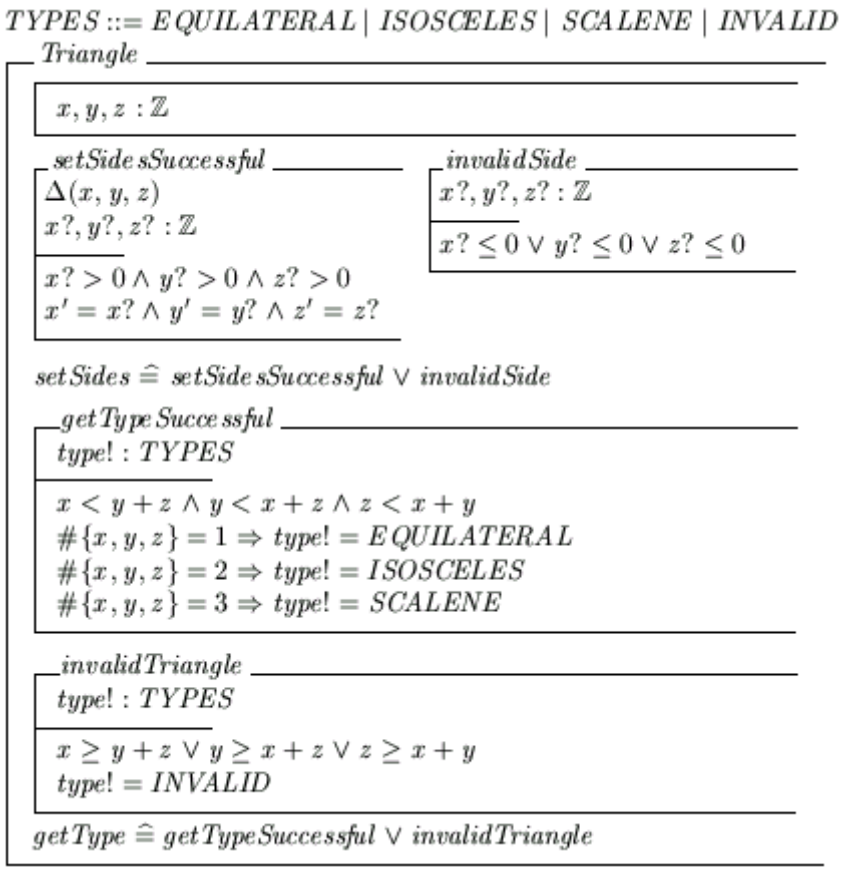


FIGURA 7.1 – Especificação formal do problema do triângulo

Nota-se que a especificação não declara um esquema de inicialização e apenas a operação *setSides* realiza mudanças no estado da classe.

7.1.2 A Implementação e a Composição do Mapeamento

Uma implementação para o problema da classificação de triângulos é apresentada na Figura 7.2. Essa implementação propositalmente não satisfaz à especificação em três situações, as quais são apresentadas na Seção 7.1.3.

Embora exista uma grande homogeneidade entre a especificação e a implementação, para compor um mapeamento para a implementação proposta, uma adaptação é necessária na especificação original.

Uma vez que a declaração do tipo *TYPES* não tem uma representação baseada em tipos primitivos, é necessário remodelar a especificação de forma a representar os tipos de triângulos. Uma alternativa consiste em declarar cada tipo definido em *TYPES* como constantes com valor inteiro, tal como na implementação. A alteração na especificação é vista na Figura 7.3.

Após a eliminação de *TYPES* da especificação, o mapeamento entre as classes de especificação e de implementação é direto e é apresentado na Figura 7.4. Nota-se que devido à redefinição dos tipos de triângulos, também se faz necessário redefinir a declaração da variável de retorno *type!* nas operações *getTypeSuccessful* e *invalidTriangle* como uma variável do tipo inteiro.

```
 1. public class Triangle {
 2.   public final int INVALID = 0;
 3.   public final int SCALENE = 1;
 4.   public final int ISOSCELES = 2;
 5.   public final int EQUILATERAL = 3;

 6.  private int sideX, sideY, sideZ;

 7.  public Triangle() throws Exception {
 8.    sideX = 0;
 9.    sideY = 0;
10.    sideZ = 0;
11.  }
12.  public void setSides( int x, int y, int z ) {
13.    if( x < 0 || y < 0 || z < 0 )
14.      System.out.println( "The sides must be positives" );
15.    else {
16.      sideX = x;
17.      sideY = y;
18.      sideZ = z;
19.    }
20.  }
21.  public int classify() {
22.    int result = INVALID;
23.    if( ( sideX <= sideY + sideZ ) && ( sideY <= sideX + sideZ )
24.        && ( sideZ <= sideX + sideY ) ) {
25.      if( sideX == sideY && sideY == sideZ )
26.        result = EQUILATERAL;
27.      else if( sideX != sideY || sideY != sideZ )
28.        result = SCALENE;
29.      else
30.        result = ISOSCELES;
31.    }
32.    return result;
33.   }
34. }
```

FIGURA 7.2 – Uma implementação para o problema da classificação do triângulo

$$
\begin{array}{|l}
INVALID, EQUILATERAL, ISOSCELES, SCALENE : \mathbb{Z} \\
\hline
INVALID = 0 \\
SCALENE = 1 \\
ISOSCELES = 2 \\
EQUILATERAL = 3
\end{array}
$$

FIGURA 7.3 – Definição dos tipos de triângulos como constantes inteiras

```
class( Triangle, Triangle ) {
    constant( INVALID, INVALID ) {
        type( \integer, int ):abstract(04)
    }
    constant( SCALENE, SCALENE ) {
        type( \integer, int ):abstract(04)
    }
    constant( ISOSCELES, ISOSCELES ) {
        type( \integer, int ):abstract(04)
    }
    constant( EQUILATERAL, EQUILATERAL ) {
        type( \integer, int ):abstract(04)
    }
    state( x, sideX ) {
        type( \integer, int ):abstract(04)
    }
    state( y, sideY ) {
        type( \integer, int ):abstract(04)
    }
    state( z, sideZ ) {
        type( \integer, int ):abstract(04)
    }
    init( init, Triangle() ) {
    }
    schema( setSides, setSides(int x,int y,int z) ) {
        parameter( x?, x ) {
            type( \integer, int ):abstract(04)
        }
        parameter( y?, y ) {
            type( \integer, int ):abstract(04)
        }
        parameter( z?, z ) {
            type( \integer, int ):abstract(04)
        }
    }
    schema( getType, classify() ) {
        parameter( type!, result ) {
            type( \integer, int ):abstract(04)
        }
    }
}
```

FIGURA 7.4 – Mapeamento para classe *Triangle*

Deve-se destacar que embora a especificação não declare um esquema de inicialização, obrigatoriamente o mapeamento deve conter um relacionamento entre a inicialização da especificação e o construtor da classe. Esse mapeamento é importante porque é através da instrumentação aplicada ao construtor que as variáveis abstratas da especificação são inicializadas. A inicialização deve ocorrer mesmo que não exista qualquer verificação a ser feita, como é o caso da classe *Triangle*.

7.1.3 Os Resultados dos Testes

Após a composição do mapeamento, um oráculo para a implementação proposta na Figura 7.2 é criado a partir da especificação formal, segundo a estratégia apresentada nos capítulo 5 e 6.

O oráculo produzido pode ser utilizado em ferramentas que geram casos de teste e *drivers* automaticamente. Entretanto, nesse exemplo, utiliza-se os casos de teste para o problema de classificação de triângulos extraídos de estudos de casos propostos por Meudec [MEU 97] e Stocks [STO 93, STO 94]. Este conjunto de casos de teste é derivado através de diversas estratégias de geração de casos de teste, tais como partição causa-efeito, particionamento de equivalência, análise do valor limite e permutações de valores. O *driver* usado no teste consiste em uma aplicação Java que cria uma instância da classe *Triangle* e faz chamadas aos métodos, passando os valores dos casos de teste como parâmetros.

Durante a execução dos testes, o oráculo gerado para a classe *Triangle* detectou todos os erros introduzidos na implementação. A seguir, descreve-se cada erro detectado:

- 1º Erro

A execução da seqüência de teste *SetSides(1, 1, 1)* → *setSides(0, 0, 0)* detecta um erro na implementação e gera o histórico de execução apresentado na Figura 7.5.

```
=====
test of setSides(int x,int y,int z)
=====
setSides applied to setSides(int x,int y,int z)
-----
setSidesSuccessful or invalidSide -> false

setSidesSuccessful
-----
INVARIANT -> true
PRE CONDITION
   * x? > 0 and y? > 0 and z? > 0 - > false
     x? -> 0
     y? -> 0
     z? -> 0
POS CONDITION -> true
INVARIANT -> true
Result-> false
```

```
invalidSide
-----
INVARIANT -> true
PRE CONDITION -> true
POS CONDITION
   * x = x' - > false
     x -> 1
     x' -> 0
   * y = y' - > false
     y -> 1
     y' -> 0
   * z = z' - > false
     z -> 1
     z' -> 0
INVARIANT -> true
Result -> false
```

FIGURA 7.5 – Histórico de execução para teste da classe *Triangle 1º erro*

A informação apresentada na Figura 7.5 mostra o histórico de execução correspondente à chamada do método *setSides(0, 0, 0)*. O esquema *setSides* não foi satisfeito porque tanto o esquema *setSidesSuccessful* como o esquema *invalidSide* falharam.

O esquema *setSidesSuccessful* falhou porque sua pré-condição requer que os lados do triângulo sejam positivos, o que não é atendido pelos valores 0, 0 e 0. Por outro lado, o esquema *invalidSide* falhou porque determina que se pelo menos um dos valores informados não for positivo, o estado não pode ser atualizado. Nota-se que a mudança de estado ocorreu na implementação observando os valores das variáveis *x*, *y* e *z*, que denotam o estado inicial da operação, e *x'*, *y'*, *z'*, que denotam o estado posterior da operação. As variáveis x, y e z contêm o valor "1" devido à chamada do primeiro método da seqüência – *setSides(1, 1, 1)* – e as variáveis *x'*, *y'* e *z'* contêm o valor "0", caracterizando a mudança do estado.

Analisando-se a implementação, conclui-se que a falha está na instrução da linha 13 da Figura 7.2. A solução consiste em trocar o operador "menor que" (<) pelo operador

"menor ou igual" (≤). Após a correção, a mesma seqüência de teste foi executada e nenhum erro foi detectado.

- 2º Erro

O oráculo para classe *Triangle* detectou outro erro na implementação quando o teste *classify*(10, 4, 10) é executado. A seguir, a Figura 7.6 mostra o histórico de execução gerado.

```
=====
test of classify()
=====
getType applied to classify()
-----
getTypeSuccessful or invalidTriangle

getTypeSuccessful
-----
INVARIANT -> true
PRE CONDITION -> true
RETURN VALUE
   * #{x,y,z} = 2 => type! = ISOSCELES -> false
     x -> 10
     y -> 4
     z -> 10
     type! -> 1
     ISOSCELES -> 2
POS CONDITION -> true
INVARIANT -> true
Result of getTypeSuccessful -> false

invalidTriangle
-----
INVARIANT -> true
PRE CONDITION
   * x >= y + z or y >= x + z or z >= x + y -> false
     x -> 10
     y -> 4
     z -> 10
RETURN VALUE
   * type! = INVALID - > false
     type! -> 1
     INVALID -> 0
POS CONDITION -> true
INVARIANT -> true
Result of invalidTriangle -> false
result of getType -> false
=====================================
```

FIGURA 7.6 - Histórico de execução para teste da classe *Triangle 2º Erro*

O teste do método *classify()* indica que a implementação erra ao classificar o triângulo de lados 10, 4 e 10. De forma semelhante ao 1º erro, falharam tanto o esquema *getTypeSuccessful* como o esquema *invalidTriangle*, acarretando a falha do esquema *getType*.

O esquema *getTypeSuccessful* aponta um erro ocasionado pelo valor da variável *type!*. Como há dois lados iguais no triângulo, o valor esperado para *type!* é 2, que corresponde ao valor da constante *ISOSCELES*. Entretanto, o valor capturado a partir da implementação é 1, que corresponde a um triângulo escaleno. O esquema *invalidTriangle*, por sua vez, falha porque os valores fornecidos são válidos como lados de um triângulo.

Observando-se a implementação proposta na Figura 7.2, percebe-se que a falha ocorre devido a um equívoco na instrução da linha 27, a qual define os triângulos escalenos. O equivoco consiste em considerar a propriedade transitiva para a operação de desigualdade, da mesma forma como se considera a transitividade para a operação de igualdade. Dessa forma, na linha 25, identifica-se um triângulo eqüilátero se x = y e y = z, logo, conclui-se que x = z, e portanto, todos os lados são iguais. Entretanto, se x ≠ y e y ≠ z, não se pode concluir que x ≠ z, ou seja, não se pode concluir que todos os lados são diferentes.

Uma solução para o erro encontrado na linha 27 consiste em reescrever a condição que identifica os triângulos escalenos e isósceles, tal como mostra a Figura 7.7.

```
          ...
27.       else if( sideX == sideY || sideX == sideZ || sideY == sideY )
28.         result = ISOSCELES;
29.       else
30.         result = SCALENE;
          ...
```

FIGURA 7.7 – Uma correção para o erro da classificação de triângulos

Embora a correção apresentada na Figura 7.7 sugira a maneira correta de classificar os triângulos isósceles e escalenos, um terceiro erro foi introduzido com a intenção de simular um erro de digitação do programador.

- 3º Erro

O terceiro erro é detectado pelo oráculo na execução do teste classify(10, 7, 8), gerando o histórico apresentado na Figura 7.8.

```
=====
test of classify()
=====
getType applied to classify()
-----
getTypeSuccessful or invalidTriangle

getTypeSuccessful
-----
INVARIANT -> true
PRE CONDITION -> true
RETURN VALUE
   * #{x,y,z} = 3 => type! = SCALENE -> false
     x -> 10
     y -> 7
     z -> 8
     type! -> 2
     SCALENE -> 1
POS CONDITION -> true
INVARIANT -> true
Result of getTypeSuccessful -> false
```

```
invalidTriangle
-----
INVARIANT -> true
PRE CONDITION
   * x >= y + z or y >= x + z or z >= x + y -> false
     x -> 10
     y -> 7
     z -> 8
RETURN VALUE
   * type! = INVALID -> false
     type! -> 2
     INVALID -> 0
POS CONDITION -> true
INVARIANT -> true
Result of invalidTriangle -> false
result of getType -> false
====================================
```

FIGURA 7.8 - Histórico de execução para teste da classe *Triangle 3º Erro*

A análise do histórico de execução gerado para o 3º erro é semelhante à análise feita para o 2º erro. Nesse caso, o erro ocorre porque um triângulo com três lados diferentes é classificado como isósceles, uma vez que a variável *type!* contém o valor 2, que corresponde à constante *ISOSCELES*. A falha que provocou o 3º erro é um simples erro de digitação na correção apresentada na Figura 7.7. A última comparação entre dois lados deveria ser *sideY == sizeZ* e não *sideY == sideY*. Corrigida a falha, os testes não detectaram outros erros na implementação.

Além disso, vale comentar que além das situações em que erros foram detectados, o oráculo respondeu corretamente aos casos onde a implementação funcionava corretamente. Para a realização do estudo de caso desta seção, foram aplicados os casos de testes apresentados na Tabela 7.1.

TABELA 7.1 – Conjunto de casos de teste usados para o problema da classificação de triângulos

Ordem	*Entrada*	*Resposta*	*Ordem*	*Entrada*	*Resposta*
1	1, 1, 1	Eqüilátero	18	10, 0, 0	Inválido
2	10, 10, 4	Isósceles	19	3, 10, 3	Inválido
3	10, 4, 10	Isósceles	20	0, 10, 3	Inválido
4	4, 10, 10	Isósceles	21	0, 10, 0	Inválido
5	7, 8, 10	Escaleno	22	3, 10, 0	Inválido
6	7, 10, 8	Escaleno	23	3, 3, 10	Inválido
7	8, 7, 10	Escaleno	24	0, 3, 10	Inválido
8	8, 10, 7	Escaleno	25	3, 0, 10	Inválido
9	10, 7, 8	Escaleno	26	0, 0, 10	Inválido
10	10, 8, 7	Escaleno	27	-1,-1,-1	Inválido
11	0, 0, 0	Inválido	28	-1, -1, 1	Inválido
12	1, 1, 0	Inválido	29	-1, 1, -1	Inválido
13	1, 0, 1	Inválido	30	1, -1, -1	Inválido
14	0, 1, 1	Inválido	31	-1, 1, 1	Inválido
15	10, 3, 3	Inválido	32	1, -1, 1	Inválido
16	10, 0, 3	Inválido	33	1, 1, -1	Inválido
17	10, 3, 0	Inválido			

7.2 Agenda Telefônica

O funcionamento de agendas telefônicas é um problema que vem sendo freqüentemente usado para exemplificar o poder de representação de conjuntos em especificações formais Z e Object-Z [DIL 94, SPY 92, FLE 96]. Este exemplo é particularmente interessante como estudo de caso para a ferramenta OZJ por permitir implementações não muito homogêneas em relação à especificação.

A especificação informal para a agenda telefônica usada nesse estudo de caso é:

"Um programa *que gerencie um banco de dados que contém registros de pessoas, às quais podem ser associados a um ou mais números de telefones. Ao banco de dados, também, pode-se inserir e remover registros e associar e remover telefones. A partir do banco de dados deve ser possível localizar registros pela pessoa ou pelo telefone. Além disso, deve-se obedecer à restrição de que um número de telefone pode ser associado a apenas uma única pessoa."*

7.2.1 A Especificação Formal

A especificação formal usada nesse estudo de caso é baseada no exemplo orientado a objetos apresentado por Fletcher e Sajeev [FLE 96]. A Figura 7.9 mostra parte da especificação original da agenda telefônica.

A especificação apresentada na Figura 7.9 representa a agenda telefônica através da classe *PhoneBook*, cujo estado é composto por um conjunto de pessoas – *subscribers* – e por uma função – *telephones* – que associa cada telefone a uma pessoa. Ao estado da classe *PhoneBook*, define-se a restrição de que o contra-domínio da função *telephones* deve estar contido ou ser igual no conjunto *subscribers*.

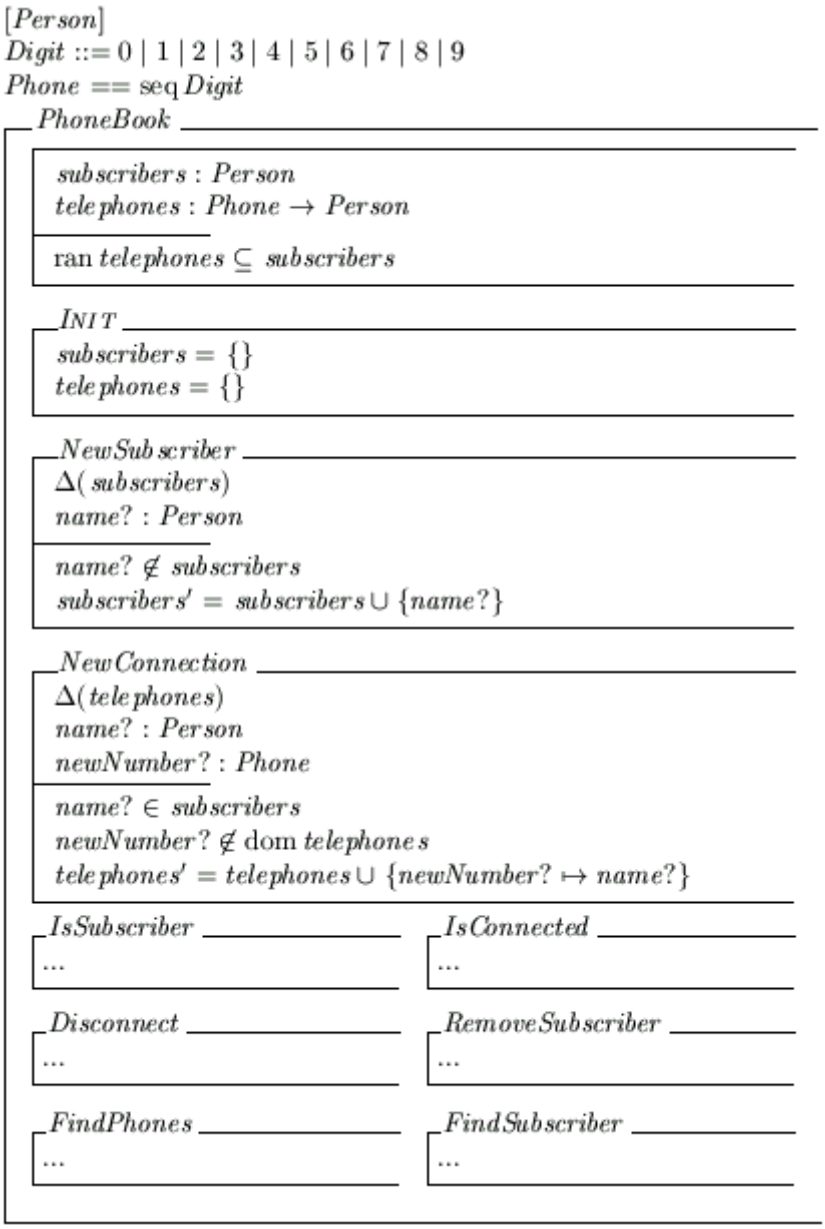


FIGURA 7.9 – Especificação formal para agenda telefônica

A especificação da classe *PhoneBook* define as operações *NewSubscriber*, que descreve o registro de novas pessoas no banco de dados, e *NewConnection*, que mostra como números de telefones devem ser associados às pessoas. O restante da especificação original contém ainda as operações *IsSubscriber* e *IsConnected*, que indicam, respectivamente, se uma determinada pessoa está cadastrada e se um determinado telefone está associado a alguém; *Disconnect* e *RemoveSubscriber*, que mostram como os registros de telefones e pessoas são retirados do banco de dados, e *FindPhones* e *FindPerson*, que descrevem as operações de localização de registros por telefone e por pessoa.

7.2.2 A Implementação e a Composição do Mapeamento

A implementação Java da classe *Agenda*, apresentada em parte na Figura 7.10, foi construída a partir da especificação informal, proposta na Seção 7.3, e da definição das operações permitidas na agenda telefônica. Por não ter sido derivada diretamente da

especificação, a solução apresentada permite exemplificar um caso em que um oráculo é gerado para uma implementação não totalmente homogênea a sua especificação.

```
public class Agenda {

  private Vector items;

  public Agenda() {
    items = new Vector();
  }
  public void insereNome( String novoNome ) {
    ItemAgenda novoItem = new ItemAgenda();
    if( !this.nomeJaExiste( novoNome ) ) {
      novoItem.setNome( novoNome );
      items.add( novoItem );
    } else
      System.out.println( "Nome já cadatrado" );
  }
  public void setaTelefone( String nome,
                            String novoNumero ) {
    ItemAgenda item;
    item = this.getItemPorNome( nome );
    if( item != null )
      item.addTelefone( novoNumero );
  }
  public boolean nomeJaExiste( String nome ) {
    ...
  }
  public boolean telefoneJaExiste( String numero ) {
    ...
  }
  public void removeTelefone( String nome,
                              String numero ) {
    ...
  }
  public void removeNome( String nome ) {
    ...
  }
  public Vector telefonePorNome( String nome ) {
    ...
  }
  public Vector nomePorTelefone( String numero ) {
    ...
  }
  public ItemAgenda getItemPorNome( String nome ) {
    ...
  }
}
```

```
public class ItemAgenda {
  private String nome;
  private Vector telefones;

  public ItemAgenda() {
    nome = "";
    telefones = new Vector();
  }

  public void setNome( String nome ) {
    ...
  }
  public boolean telefoneJaExiste( String numero ) {
    ...
  }
  public void addTelefone( String numero ) {
    ...
  }
  public void removeTelefone( String numero ) {
    ...
  }
  public String getNome() {
    ...
  }
  public String getTelefone( int index ) {
    ...
  }
  public int totalTelefones() {
    ...
  }
}
```

FIGURA 7.10 – Uma implementação para a agenda telefônica

A classe *Agenda* armazena os registros da agenda telefônica em uma lista, implementada pela classe Java *Vector*, e que contém objetos da classe *ItemAgenda*. A classe *ItemAgenda* contém o nome da pessoa agendada associado a um ou mais telefones. Dessa forma, a classe *Agenda* tem apenas uma variável de estado – *items* – e através desta variável implementa o comportamento descrito tanto pelo conjunto *subscriber* como pela função *telephones*.

A exemplo do mapeamento para o problema da classificação de triângulos, para compor o mapeamento entre as classes *Agenda* e *PhoneBook*, é necessário primeiramente reformular os tipos definidos pelos usuários em função dos tipos primitivos da linguagem Z. Seguindo a estrutura da implementação, uma pessoa é representada apenas pelo seu nome, logo, o tipo *Person* pode ser redefinido na especificação com um *String*. O tipo *Phone* declara uma seqüência de *Digits*, o qual corresponde ao intervalo de 0 a 9. A fim de eliminá-los, os tipos *Phones* e *Digits* são redefinidos para uma seqüência de números naturais. Uma invariante é acrescentada ao

estado de *PhoneBook* para restringir os naturais a valores menores ou iguais a 9. A figura 7.11 apresenta, o esquema de estado da especificação reformulada. Nota-se que as variáveis declaradas com o tipo *Person* ou *Phone* também são reformuladas para os novos tipos dos esquemas de operação.

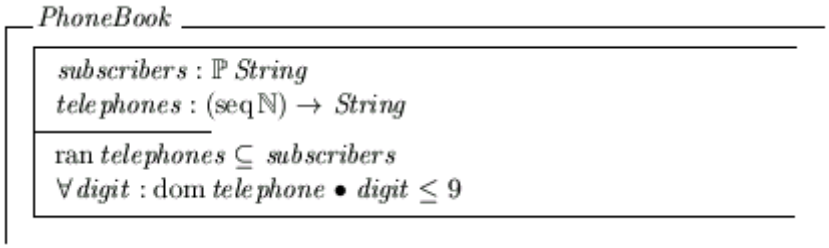


FIGURA 7.11 – Esquema de estado reformulado da classe *Agenda*

Outra alteração necessária na especificação original é o acréscimo de esquemas que descrevem as situações de exceções das operações. A descrição das exceções é importante para a especificação que serve como fonte de informação ao oráculo. Sem as exceções seriam detectados erros em qualquer situação que não corresponda ao domínio definido pelo comportamento bem sucedido da operação. Para a operação *NewSubscriber*, define-se a exceção *NameAlreadySubscribed*, representando a situação de registro de um nome já cadastrado. À operação *NewConnection,* representa duas exceções, *NameNotSubscribed* e *TelefoneAlreadyConnected*, que definem, respectivamente, quando um telefone é associado a um nome não cadastrado e a tentativa de incluir um telefone já existente na agenda. As operações reformuladas para *NewSubscriber* e *NewConnection* são apresentadas na Figura 7.12.

Uma análise da estrutura da implementação mostra que o mapeamento para a estrutura definida na especificação é feito mapeando-se a variável *items* tanto para o conjunto de pessoas – *subscribers* – como para a função de telefones em pessoas – *telephones*. O mapeamento é exemplificado na Figura 7.13

Nesse ponto, faz-se necessária a definição de duas funções de abstração, uma entre os tipos de *items* e *subscribers* e outra entre os tipos de *items* e *telephones*. A implementação criada pelo projetista é apresentada na Figura 7.14.

A função de abstração entre *items* e *subscribers* é relativamente simples e consiste em inserir, em um conjunto, o nome armazenado em cada objeto *ItemAgenda*, contido, por sua vez, na variável *items*. Os objetos inseridos no conjunto são definidos como objetos *ZMTKString*, que correspondem à implementação do tipo String em Object-OZJ.

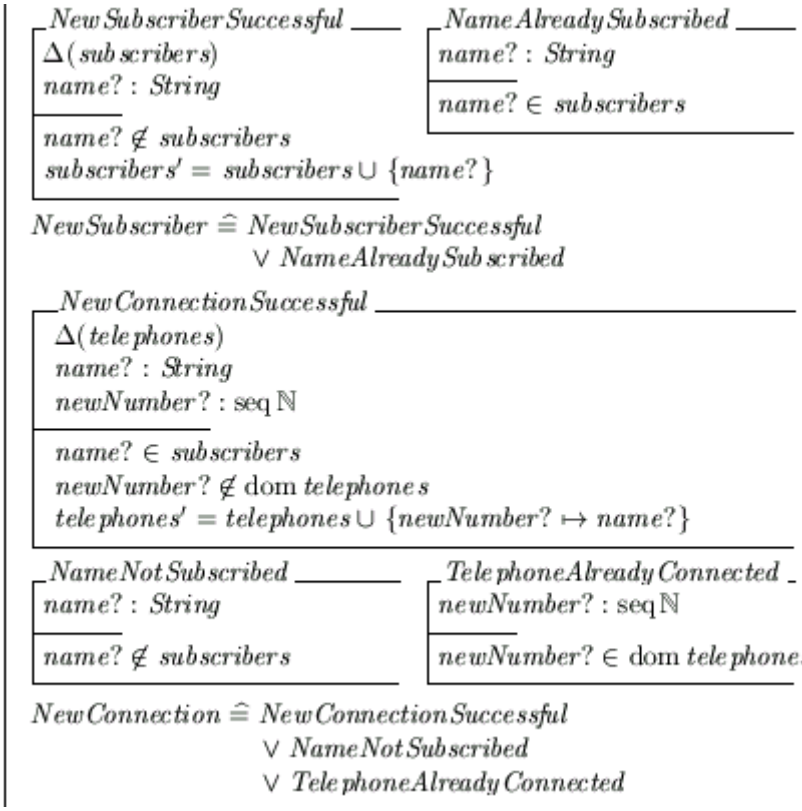


FIGURA 7.12 – Descrição das exceções das operações de PhoneBook

```
class( PhoneBook, Agenda ) {
  state( subscribers, items ) {
    type( \pset \string, Vector )
    :abstract( 05 )
  }
  state( telephones, items ) {
    type( \seq \nat \tfun \string, Vector )
    :abstract( 06 )
  }
```

FIGURA 7.13 – Mapeamento entre o estado de *PhoneBook* e *Agenda*

A função de abstração entre *items* e *telephones* retira da variável *items* a informação necessária para criar os pares ordenados que compõem a função *telephones*. O primeiro elemento do par é gerado como um objeto *Sequence*, preenchido por objetos *ZMTKNatural*. Os objetos *ZMTKNatural* são criados dos caracteres que compõem a *String* Java que representa o telefone. O segundo elemento do par ordenado é obtido criando-se objetos *ZMTKString* a partir do nome recuperado.

```
// Função de abstração para subscribers
public void abstract_subscribers( Vector concrete ) throws Exception {
  Set abstractVar = new Set();
  // Implement here
  ZMTKString item;
  for( int i = 0; i < concrete.size(); i++ ) {
    item = new ZMTKString(((agenda.ItemAgenda) concrete.elementAt(i)).getNome());
    abstractVar.insert(item);
  }
  this.subscribers = abstractVar;
}
// Função de abstração para telephones
public void abstract_telephones( Vector concrete ) throws Exception {
  TotalFunction abstractVar = new TotalFunction();
  // Implement here
  String digits;
  OrderedPair item;
  Sequence first;
  ZMTKString second;
  ZMTKNatural digit;
  agenda.ItemAgenda itemConcrete;
  // Percorre todos os itens do Vector
  for( int i = 0; i < concrete.size(); i++ ) {
    itemConcrete = (agenda.ItemAgenda) concrete.elementAt( i );
    // Recupera nome
    second = new ZMTKString( itemConcrete.getNome() );
    for( int j = 0; j < itemConcrete.totalTelefones(); j++ ) {
      // Recupera fone
      first = new Sequence();
      digits = itemConcrete.getTelefone( j );
      for( int h = 0; h < digits.length(); h++ ) {
        digit = new ZMTKNatural((new Integer(digits.substring(h,h+1))).intValue());
        first.insert( digit );
      }
      // Insere o par ( Telefone, nome ) na função
      item = new OrderedPair( first, second );
      abstractVar.insert( item );
    }
  }
  this.telephones = abstractVar;
}
```

FIGURA 7.14 – Função de abstração entre as variáveis *items* e *subscribers* e entre *items* e *telephones*

7.2.3 Os Resultados do Teste

Os testes aplicados ao oráculo gerado para a classe *Agenda* se mostraram eficientes na detecção de erros. Como exemplo, apresenta-se um erro encontrado no método *setaTelefone()*, em relação ao especificado na operação *NewConnection*, quando é aplicada a seguinte seqüência de casos de testes a uma instância da classe Agenda:

insereNome(“João”)

→ setaTelefone(“11111111”)

→ setaTelefone(“22222222”)

→ insereNome(“Maria”)

→ setaTelefone(“11111111”)

Essa seqüência insere dois nomes – “João” e “Maria” – ao banco de dados, associa dois telefones ao nome “João” e, a seguir, associa um número de telefone já cadastrado – “11111111” – ao nome “Maria”. A execução do método *setaTelefone()* transcorreu normalmente, mas o oráculo detectou uma violação da especificação e gerou o histórico de execução apresentado na Figura 7.15.

O histórico de execução aponta que a pré-condição do esquema *NewConnectionSuccessful* foi violada porque o telefone informado já estava cadastrado; o esquema *TelephoneAlreadyConnected* falhou porque determina que quando um telefone já cadastrado é informado, o estado da classe deve permanecer inalterado, o que não ocorreu. Além disso, o esquema *NameNotSubscribed* falhou porque o nome fornecido como parâmetro estava cadastrado.

Nesse caso, o erro detectado ocorreu devido à atualização do estado, permitindo o registro de um mesmo telefone a nomes diferentes. A correção da falha consiste em incluir uma instrução que verifique se o telefone informado já está cadastrado antes da atualização do estado.

7.3 Resumo do Capítulo

Neste capítulo, foram apresentadas situações reais de uso de oráculos gerados pela ferramenta OZJ. Através de dois exemplos clássicos, este capítulo ilustrou ações que o projetista deve realizar para preparar a especificação formal usada como fonte de informação para o oráculo e a capacidade de detecção de erros dos oráculos. O próximo capítulo conclui esta dissertação e apresenta uma síntese das contribuições, restrições da estratégia e da ferramenta apresentadas e, comenta sobre as considerações sobre possíveis trabalhos futuros.

```
=====
NewConnection applied to setaTelefone(String nome,String novoNumero)
-----
NewConnectionSuccessful or NameNotSubscribed
or TelephoneAlreadyConnected

NewConnectionSuccessful
-----
PRE CONDITION
   * newNumber? \nem \dom telephones - > false
     newNumber? -> < 1, 1, 1, 1, 1, 1, 1, 1 >
     telephones -> { ( < 1, 1, 1, 1, 1, 1, 1, 1 > |-> João ),
                     ( < 2, 2, 2, 2, 2, 2, 2, 2 > |-> João ) }
POS CONDITION -> true
Result of NewConnectionSuccessful -> false

NameNotSubscribed
-----
PRE CONDITION
   * name? \nem subscribers - > false
     name? -> Maria
     subscribers -> { João, Maria }
POS CONDITION
   * telephones = telephones' - > false
     telephones -> { ( < 1, 1, 1, 1, 1, 1, 1, 1 > |-> João ),
                     ( < 2, 2, 2, 2, 2, 2, 2, 2 > |-> João ) }
     telephones' -> { ( < 1, 1, 1, 1, 1, 1, 1, 1 > |-> João ),
                      ( < 2, 2, 2, 2, 2, 2, 2, 2 > |-> João ),
                      ( < 1, 1, 1, 1, 1, 1, 1, 1 > |-> Maria ) }
Result of NameNotSubscribed -> false

TelephoneAlreadyConnected
-----
PRE CONDITION -> true
POS CONDITION
   * telephones = telephones' - > false
     telephones -> { ( < 1, 1, 1, 1, 1, 1, 1, 1 > |-> João ),
                     ( < 2, 2, 2, 2, 2, 2, 2, 2 > |-> João ) }
     telephones' -> { ( < 1, 1, 1, 1, 1, 1, 1, 1 > |-> João ),
                      ( < 2, 2, 2, 2, 2, 2, 2, 2 > |-> João ),
                      ( < 1, 1, 1, 1, 1, 1, 1, 1 > |-> Maria ) }
Result of TelephoneAlreadyConnected -> false
result of NewConnection -> false
=====================================
```

FIGURA 7.15 – Histórico de execução para o teste da classe *Agenda*

8 Conclusão

Estratégias de seleção de casos vêm sendo constantemente estudadas ao longo dos anos e, muito embora tais estratégias tenham se mostrado eficientes na descoberta de erros, a quantidade de testes necessárias para inferir boa qualidade ao *software* ainda é demasiadamente grande. A principal conseqüência decorrente deste inconveniente é o alto custo provocado pela verificação manual dos resultados de cada teste realizado. Soma-se a isso, também, a dificuldade de observação quando o comportamento a ser testado é representado apenas pelo estado interno das implementações.

Os oráculos automáticos vêm sendo desenvolvidos com o objetivo de apoiar a atividade de teste de *software*, viabilizando uma forma de verificação automatizada dos resultados do teste. Um requisito importante para a automatização dos oráculos é a existência de uma fonte de informação confiável e estruturada. Para suprir este requisito as linguagens de especificação formal vêm sendo eficientemente usadas.

Entretanto, alguns aspectos limitam a abrangência de aplicação de algumas abordagens para a geração de oráculos encontradas na literatura. As limitações abordadas neste trabalho são:

1) a relação com o processo de desenvolvimento baseado em especificação formal;
2) a vinculação do oráculo com o processo de seleção de casos de teste.

O restante deste capítulo apresenta na Seção 8.1 as contribuições deste trabalho, na Seção 8.2, as limitações da abordagem proposta e na Seção 8.3, possibilidades de trabalhos futuros.

8.1 Contribuições

Este trabalho apresentou um estudo sobre diversas abordagens para geração de oráculos a partir de especificações formais. Este estudo teve como objetivo estabelecer quais os fatores que influenciam na aplicabilidade de geradores de oráculos e quais são os aspectos fundamentais abordados pelas pesquisas anteriores. A partir do estudo de abordagens precedentes, foi possível delinear a estrutura geral da estratégia para a geração de oráculos proposta neste trabalho

Nesse contexto, a principal contribuição deste trabalho foi propor uma estratégia semi-automática[11] que aumenta a abrangência de utilização dos geradores de oráculos. A partir das limitações estabelecidas pelo estudo de outras abordagens, foi proposta uma abordagem que agrega as seguintes características:

1) independência do processo de desenvolvimento, ou seja, um oráculo que pode ser aplicado a qualquer implementação desenvolvida sob qualquer processo de desenvolvimento e;
2) independência do processo de seleção de casos de teste, ou seja, um oráculo que pode ser gerado a partir de uma descrição genérica do comportamento do

[11] Nota-se que o processo de geração do oráculo é semi-automático, entretanto, a verificação do comportamento realizada pelo oráculo é automática, logo, o oráculo gerado pode ser classificado como oráculo automático.

software, e não especificamente para casos de teste determinados no processo de seleção.

Dentro deste contexto, com o objetivo de apresentar uma proposta aplicada, fez-se uma opção por linguagens de especificação e de implementação largamente utilizadas, tanto em ambientes acadêmicos como em ambientes comercias. Estas linguagens são Object-Z e Java.

Outra contribuição apresentada neste trabalho, decorrente da estratégia proposta, é a definição da sintaxe da linguagem *OZJ-Mapping* para a representação do mapeamento entre a especificação e a implementação. Essa linguagem considera tanto o relacionamento entre estruturas de controle como o relacionamento entre as estruturas de dados e, embora, neste trabalho, seja aplicada apenas às linguagens Object-Z e Java, a linguagem *OZJ-Mapping* pode ser usada de uma forma geral para representar o mapeamento entre outras linguagens orientadas a objetos.

Ainda em decorrência da estratégia proposta, este trabalho estabeleceu critérios para a composição do mapeamento. Embora, o mapeamento seja uma tarefa criativa, os critérios definidos partem da semelhança estrutural entre a linguagem de especificação e a linguagem de implementação e impõem restrições que regulam e auxiliam o projetista.

Complementando a estratégia proposta neste trabalho, foi desenvolvido um protótipo da ferramenta OZJ que implementa a maioria das características descritas nos capítulos anteriores. Dentre as características previstas na estratégia, não foram incluídas no protótipo a geração de oráculos para hierarquias de classes e a capacidade de verificação de predicados que contêm quantificadores. As demais características estão implementadas, e através delas, foi possível a realização dos estudos de casos apresentados no Capítulo 6 onde pode ser realizada uma validação preliminar das funcionalidades implementadas.

O protótipo implementado provê as seguintes funcionalidades:

1) auxilia o projetista na criação do projeto de geração de oráculos;
2) gera automaticamente o documento de mapeamento com as estruturas de controle e as estruturas de dados da especificação;
3) fornece suporte ao projetista na implementação das funções de abstração, manipulando um repositório onde são armazenadas funções pré-existentes e gerando automaticamente o cabeçalho de novas funções.
4) gera automaticamente o oráculo a partir da especificação formal, da implementação e do mapeamento de uma classe.

8.2 Limitações

Através do estudo da estratégia apresentada neste trabalho e do projeto de implementação da ferramenta OZJ, constatara-se algumas limitações tanto na estratégia proposta como na ferramenta implementada. A seguir, comenta-se as limitações detectadas:

- *Dependência da correta realização das tarefas do projetista*

O funcionamento correto da ferramenta OZJ depende fortemente da capacidade do projetista de realizar corretamente as tarefas sob sua responsabilidade. Conforme visto nos capítulos anteriores, o projetista atua diretamente na especificação do

comportamento desejado, na composição do mapeamento e na implementação das funções de abstração. Embora erros de sintaxe cometidos pelo projetista possam ser detectados pelos reconhecedores da linguagem de especificação, da linguagem de implementação e do mapeamento, erros de caráter semântico não são detectados. Erros semânticos podem levar o oráculo a emitir respostas erradas. Cita-se como erros semânticos o uso de uma especificação que não descreve corretamente o comportamento desejado, o mapeamento de estruturas que não têm o mesmo significado e a implementação incorreta da conversão de tipos nas funções de abstração.

Esse inconveniente é, na verdade, o custo a ser pago para aumentar a abrangência no uso dos oráculos para implementações não derivadas diretamente da estrutura de especificações formais.

- *Restrições à verificação*

A uso da estratégia apresentada neste trabalho e, conseqüentemente, o uso da ferramenta OZJ, limita-se ao aspecto funcional das implementações. Nesse contexto, a verificação provida pela ferramenta está restrita ao poder de representação da linguagem de especificação adotada.

Outro fator limitador é que a verificação está restrita ao "comportamento observável" do estado da implementação. Dessa forma, a aplicação do oráculo está restrita àquelas implementações cujo comportamento está completamente representado pelo valor dessas variáveis internas. Essa característica impede o uso dos oráculos gerados pela ferramenta OZJ às operações que realizam alterações em SGBD's, por exemplo.

- *Efeito colateral da instrumentação*

Dentre os efeitos colaterais da instrumentação, a sobrecarga de execução pode ser considerada um fator limitado-se, à medida que impõe restrições à quantidade de informação manipulada durante a verificação.

8.3 Trabalhos Futuros

O estudo desenvolvido neste trabalho abre algumas possibilidades para melhorias e extensões. A seguir, cita-se algumas:

- *Geração de oráculos para hierarquias de classes e geração de assertivas com quantificadores*

  Embora prevista na estratégia, essas características não foram implementadas no protótipo apresentado neste trabalho. Entretanto, planeja-se implementá-las no futuro.

- *Tratamento de exceção durante o reconhecimento da especificação e da implementação*

  Uma melhoria interessante ao protótipo da ferramenta OZJ, apresentado neste trabalho, seria um tratamento mais elaborado sobre as exceções do processo de geração do oráculo. Cita-se, como exemplo, o uso de variáveis no mapeamento que não existem na especificação. Esse tipo de exceção, embora tratado na versão atual, requer um tratamento mais elaborado de forma a dar informações mais precisas ao projetista.

- *Extensão do pacote ZMTK*

  A implementação de características adicionais ao pacote ZMTK é uma extensão interessante ao protótipo proposto neste trabalho por aumentar o poder de representação da linguagem de especificação e, conseqüentemente, aumentar o poder de verificação do oráculo.

- *Desenvolvimento de uma interface gráfica para a composição do mapeamento*

  A versão desenvolvida da ferramenta OZJ gera um documento de mapeamento na forma de um arquivo texto, o qual é atualizado com as estruturas da implementação pelo projetista em um editor de texto qualquer. Para gerar esse documento, o reconhecimento sintático da especificação e da implementação identifica todas as estruturas de controle e as estruturas de dados. Essa informação pode ser usada não só para gerar o documento de mapeamento como para dar suporte a uma interface que apóie o projetista na composição do mapeamento, permitindo que as estruturas da implementação possam ser associadas graficamente às estruturas da especificação. O produto final dessa interface seria o documento de mapeamento completo.

- *Desenvolvimento de um ambiente para gerenciar o processo de geração de oráculos*

  O processo de geração de oráculos é composto por três passos: a criação do projeto, a geração do documento de mapeamento e a geração do oráculo. No decorrer deste processo, normalmente são necessárias correções na especificação e no mapeamento, o que acarreta a necessidade de manipulação desses arquivos. Além disso, para a aplicação dos testes no oráculo, comumente é necessário fazer uma cópia da implementação original e substituí-la pelo oráculo. Com o intuito de apoiar o projetista, essas tarefas podem ser realizadas automaticamente dentro de um ambiente que englobe as funcionalidades implementadas até então para a ferramenta OZJ.

- *Extensão a uma ferramenta para geração de casos de teste*

  Uma extensão bastante interessante à ferramenta OZJ seria a definição de uma ferramenta para a geração de casos de teste através do uso inverso da conversão de valores concretos em valores abstratos. Como exemplo, cita-se a abordagem para geração de casos de teste proposta por Stocks [STO 93], que gera casos de teste descritos em linguagem Z. Esses valores abstrato, podem ser transformados em valores concretos e aplicados automaticamente ao oráculo gerado pela ferramenta OZJ.

Anexo 1 Gramática da Linguagem Object-OZJ

```
Specification   =   Paragraph
Paragraph       =   [ TypeDefs ]
                    "\begin" "{" "class" "}" "{" Name "}"
                            [ Visibility ]
                            [ "\inherit" Namelist "\endinherit" ]
                            [ AxiomaticDefinition ]
                            [ StateSchema ]
                            [ InitialSchema ]
                            [ Operations ]
                    "\end" "{" "class" "}"
TypeDefs        =   EqualDelc { "," EqualDecl }
Visibility      =   "\filter" "(" NameList ")"
LocalDefs       =   LocalDef { Sep LocalDef }
LocalDef        =   TypeDef
                |   AxDef
AxDef           =   DeclPart [ "\ST" predicate ]
                |   Predicate
TypeDef         =   [ NameList ]
                |   ColonDecl
StateSchema     =   "\begin" "{" "state" "}" AxDef "\end" "{" "state" "}"
InitialSchema   =   "\begin" "{" "init" "}" AxDef "\end" "{" "init" "}"
Operations      =   Operation [ Operations ]
Operation       =   "\begin" "{" "op" "}" "{" Name "}"
                            [ "\delta" "(" NameList ")" ] [ AxDef ]
                    "\end" "{" "op" "}"
                |   Name "\sdef" Name { ( "\land" | "\lor" ) Name }
NameList        =   Name { "," Name }
Sep             =   ";" | "\\"
Predicate       =   BasicPred ( [ Sep Predicate ]
                              | "\iff" Predicate
                              | "\implies" Predicate
                              | "\lor" Predicate
                              | "\land" Predicate )
```

```
BasicPred     =   ( "\forall" | "\exists" | "\exists_1" ) SchemaText "\dot" Predicate
              |   "\lnot" Predicate
              |   Expression
              |   "true"
              |   "false"
              |   "(" Predicate ")"
SchemaText    =   DeclPart [ "|" Predicate ]
              |   Predicate
DeclPart      =   Declaration { Sep Declaration }
Declaration   =   ColonDecl
              |   EqualDecl
ColonDecl     =   ( "Name" | "ZVar" ) { , ( "Name" | "ZVar" ) } ":" Expression
EqualDecl     =   ( "Name" | "ZVar" ) "==" Expression
Expression    =   BasicExpr ( ( InGen | InFun | InRel ) Expression
                          | { "," Expression }
                          | PostFun
                          | "(" Expression ")"
                          | "\limg" Expression"\rimg"
                          | "\lseq" Expression "\rseq"
                          | "\lbag" Expression "\rbag" )
BasicExpr     =   ( Name | ZVar ) | ( Integer | Real ) | "\nat" | "\Integer"
              |   "\Real" | "\Bool" | PreGen Expression | PreFun Expression
              |   "\emptyset" | "\lseq" "\rseq" | "\lbag" "\rbag" | "(" Expression ")"
InGen         =   "\rel" | "\pfun" | "\tfun" | "\pinj" | "\ting" | "\psur" | "\tsur" | "\bij"
              |   "\cross" | "\map" | ".."
PreGen        =   "\pset" | "\seq" | "\bag"
InFun         =   InFun1
              |   InFun2
              |   InFun3
InFun1        =   "+" | "-" | "\union" | "\difer" | "\cat" | "\buni" | "\bagminus"
InFun2        =   "*" | "\div" | "\mod" | "\inter" | "\ires" | "\sres" | "\bagscale"
InFun3        =   "\dres" | "\rres" | "\dsub" | "\rsub" | "\fovr" | "\subbag"
InRel         =   "=" | "\neq" | "\nem" | "\subs" | "\subset" | "<" | "\leq" | "\geq" | ">"
              |   "\in" |
PostFun       =   "\inv" | "\tcl" | "\rtcl"
PreFun        =   "\dom" | "\ran" | "\#" | "\first" | "\second | "\head" | "\tail" | "\front"
              |   "\last"
Name          =   Letter { Letter | Digit } { Stroke }
```

Integer	=	Digit {Digit}
Real	=	Digit { Digit } [“.” Digit { Digit }]
Letter	=	“A” | … | “Z” | “a” | … | “z”
Stroke	::=	“‘” | “?” | “!”
Digit	::=	“0” | … | “9”

Anexo 2 Documentação do Pacote ZMTK

A seguir, apresenta-se uma documentação técnica das classes do pacote ZMTK.

2.1 Pacote ZMTK

O objetivo do pacote ZMTK é criar uma representação através de classes para os tipos da linguagem Z utilizados neste trabalho. Como em muitas situações, certas operações manipulam operadores de uma forma genérica, sem precisar determinar a qual tipo especificamente eles pertencem, a única classe do pacote ZMTK é a classe *ZMTKObject*.

O pacote ZMTK é composto ainda pelos pacotes *numbers*, *non-math*, *logics* e *sets*, os quais são comentados nas próximas seções deste anexo.

2.1.1 Classe ZMTK.ZMTKObject

ZMTKObject tem a função de prover um ancestral comum a todas as demais classes do pacote ZMTK de forma a viabilizar a implementação de operações que manipulam parâmetros sem considerar tipo.

Herança
`java.lang.Object`

Construtor
`ZMTKObject``()`

Métodos	

`abstract public boolean`	**`EqualityByAttribute`**`( ZMTKObject object )`
	`Define se o parâmetro "object" é igual, ou seja, se ele tem atributos com o mesmo valor que a instância corrente.`
	`Toda instância de classe pertencente ao ZMTK deve implementar este método.`

2.2 Pacote ZMTK.numbers

O pacote *numbers* contém as classes que representam os valores numéricos do ZMTK.

2.2.1 Classe ZMTK.numbers.ZMTKNumber

ZMTKNumber é uma classe abstrata que representa os números de uma maneira geral. Sua única função é servir de ancestral comum as demais classes que representam tipos numéricos específicos.

Herança

java.lang.Object
  |
  +--ZMTK.ZMTKObject

2.2.2 Classe ZMTK.numbes.ZMTKInteger

Representa os números inteiros.

Herança

java.lang.Object
  |
  +--ZMTK.ZMTKObject
  |
  +--ZMTK.numbers.Number

Atributos	
private int	**value** Contém o valor do número inteiro.

Construtor
ZMTKInteger(int value) Cria um novo ZMTKInteger a partir do valor da variável *value*. Parâmetros

Métodos	
protected boolean	**newNumberOK**(int number) Retorna "true" se o valor do paramêtro *number* é um número inteiro válido. Para a classe ZMTKInteger, retorna sempre "true", mas é redefinida pelas suas subclasses.

public boolean	**equalityByAtribute**(ZMTKObject parm) Implementação do método abstrato definido em ZMTKObject. Retorna "true" se o valor da instância corrente for igual ao valor do parâmetro *parm*. Caso contrário, retorna "false".
public ZMTKInteger	**addition**(ZMTKInteger another) Cria um novo objeto ZMTKInteger pela soma do valor da instância corrente ao valor do parâmetro *another*.
public ZMTKInteger	**subtration**(ZMTKInteger another) Cria um novo objeto ZMTKInteger pela subtração do valor da instância corrente do valor do parâmetro *another*.
public ZMTKInteger	**multiplication**(ZMTKInteger another) Cria um novo objeto ZMTKInteger pela multiplicação do valor da instância corrente com o valor do parâmetro *another*.
public ZMTKInteger	**division**(ZMTKInteger another) Cria um novo objeto ZMTKInteger pela divisão do valor da instância corrente com o valor do parâmetro *another*. O valor retornado corresponde ao resultado da divisão inteira.
public void	**set**(int newValue) Altera o valor do objeto pelo valor do parâmetro *newValue*.
public int	**value()** Retorna valor corrente do objeto sob seu tipo primitivo correspondente em Java.
public String	**toString**() Retorna a representação do inteiro na forma de cadeia de caracteres.

Métodos Herdados da classe java.lang.Object
clone, finalize, getClass, notify, notifyAll, wait
Métodos Herdados da classe ZMTK.ZMTKObject
equalityByAttribute

2.2.3 Classe ZMTK.numbers.ZMTKNatural

Representa os números naturais.

Herança

java.lang.Object
  |
  +--ZMTK.ZMTKObject
  |
  +--ZMTK.numbers.Number
  |
  +--ZMTK.number.ZMTKInteger

Construtor
ZMTKNatural(int value) Cria um novo ZMTKNatural a partir do valor da variável *value*.

Métodos	
protected boolean	**newNumberOK**(int number) Redefinição da operação newNumberOK de ZMTKInteger. Retorna "true" se o valor do paramêtro *number* é um número natural válido, ou seja, um valor maior ou igual ao número "0".

Métodos Herdados da classe java.lang.Object
clone, finalize, getClass, notify, notifyAll, wait
Métodos Herdados da classe ZMTK.ZMTKInteger
equalityByAttribute, addition, subtraction, multiplication, division, set, value, toString

2.2.4 Classe ZMTK.numbers.PositiveNatural

Representa os números naturais estritamente positivos.

Herança

java.lang.Object
  |
  +--ZMTK.ZMTKObject
  |
  +--ZMTK.numbers.Number
  |
  +--ZMTK.numbers.ZMTKInteger
  |
  +--ZMTK.numbers.PositiveNatural

Construtor
ZMTKNatural(int value) Cria um novo ZMTKNatural a partir do valor da variável *value*.

Métodos	
protected boolean	**newNumberOK**(int number) Redefinição da operação newNumberOK de ZMTKNatural. Retorna "true" se o valor do paramêtro *number* é um número natural válido, ou seja, um valor maior que "0".

Métodos Herdados da classe java.lang.Object
clone, finalize, getClass, notify, notifyAll, wait
Métodos Herdados da classe ZMTK.ZMTKInteger
equalityByAttribute, addition, subtraction, multiplication, division, set, value, toString

2.2.5 Classe ZMTK.numbes.ZMTKReal

Representa os números reais.

Herança

java.lang.Object
  |
  +--ZMTK.ZMTKObject
  |
  +--ZMTK.numbers.Number

Atributos	
private double	**value** Contém o valor do número real.

Construtor
ZMTKInteger(double value) Cria um novo ZMTKReal a partir do valor da variável *value*.

Métodos	
Public boolean	**equalityByAtribute**(ZMTKObject parm) Implementação do método abstrato definido em ZMTKObject, retorna "true" se o valor da instância corrente for igual ao valor do parâmetro *parm*, mesmo que *parm* seja uma de suas subclasses. Caso contrário, retorna "false".

public ZMTKReal	**addition**(ZMTKNumber another) Cria um novo objeto ZMTKReal pela soma do valor da instância corrente ao valor do parâmetro *another*.
public ZMTKReal	**subtration**(ZMTKNumber another) Cria um novo objeto ZMTKReal pela subtração do valor da instância corrente do valor do parâmetro *another*.
public ZMTKReal	**multiplication**(ZMTKNumber another) Cria um novo objeto ZMTKReal pela multiplicação do valor da instância corrente com o valor do parâmetro *another*.
public ZMTKReal	**division**(ZMTKNumber another) Cria um novo objeto ZMTKReal pela divisão do valor da instância corrente com o valor do parâmetro *another*.
public void	**set**(int newValue) Altera o valor do objeto pelo valor do parâmetro *newValue*.
public double	**value**() Retorna valor corrente do objeto sob o tipo primitivo correspondente em Java.
public String	**toString**() Retorna a representação do número real na forma de cadeia de caracteres.

Métodos Herdados da classe java.lang.Object
clone, finalize, getClass, notify, notifyAll, wait
Métodos Herdados da classe ZMTK.ZMTKObject
equalityByAttribute

2.3 Pacote ZMTK.sets

O pacote sets representa a Teoria dos Conjuntos. Ele implementa os tipos básicos da linguagem Z e as principais operações sobre esses tipos.

2.3.1 Classe ZMTK.sets.OrderedPair

Representa um par ordenado dentro do pacote ZMTK.

Herança

java.lang.Object
  |
  +--ZMTK.ZMTKObject

Atributos	
private Vector	**first** É o primeiro elemento do par ordenado.

private Vector	**second** É o segundo elemento do par ordenado.

Construtor
OrderedPair(ZMTKObject f, ZMTKObject s) Cria um par ordenado que contém duas outras instâncias quaisquer da classe ZMTKObject.

Métodos	
public boolean	**equalityByAttribute**(ZMTKObject parm) Implementação do método abstrato definido em ZMTKObject, retorna "true" se os primeiros e os segundos elementos da instância corrente e do par ordenado *parm* são iguais, caso contrário, retorna "false"
public ZMTKObject	**first**() Retorna o primeiro elemento do par ordenado.
public ZMTKObject	**second**() Retorna o segundo elemento do par ordenado.
public String	**toString**() Retorna uma representação do par ordenado sobre a forma de uma cadeia de caracteres.

Métodos Herdados da classe java.lang.Object
clone, finalize, getClass, notify, notifyAll, wait

2.3.2 Classe ZMTK.sets.Set

A classe *Set* representa um conjunto. No contexto deste trabalho, um conjunto consiste de uma determinada quantidade de instâncias diferentes da classe *ZMTKObject* sem ordem definida.

Herança

java.lang.Object
  |
  +--ZMTK.ZMTKObject

Atributos	
private Vector	**container** Contém os elementos do conjunto, ou seja, as instâncias de ZMTKObject.

Construtor
Set() Cria um conjunto vazio.
Set(ZMTKObject item) Cria um novo conjunto que contém o elemento *item*.

Métodos	
protected ZMTKBoolean	**newElementOK**(ZMTKObject element) Retorna "true" se o parâmetro *element* é um elemento válido para o conjunto, ou seja, se ele não é duplicado, caso contrário, retorna "false". Este método é chamado internamente pelo método *insert* para determinar se o novo elemento pode fazer parte do conjunto. As subclasses da classe *Set* redefinem este método para acrescentar as restrições adequadas a elas.
public boolean	**equalityByAttribute**(ZMTKObject parm) Implementação do método abstrato definido em ZMTKObject, retorna "true" se todos os elementos de *parm* estão contidos na instância corrente, caso contrário, ou se *parm* não for um conjunto, retorna "false"
public ZMTKBoolean	**equality**(Set another) Realiza a mesma função de *equalityByAttribute*, mas retorna um ZMTKBoolean. A razão desse método é que apenas métodos que retornem objetos ZMTKBoolean ou ozj.OZJResult podem ser usados nas funções de avaliação do oráculo. Dessa forma, as chamadas ao método *equalityByAttribute* são feitas apenas internamente aos métodos das classes do pacote ZMTK.
public ZMTKBoolean	**emptySet()** Retorna "true" se o conjunto está vazio, caso contrário, retorna "false".
public ZMTKInteger	**cardinality**() Retorna o número de elementos do conjunto.
public Set	**union**(Set another) Cria um novo conjunto pela união dos elementos da instância corrente e do parâmetro *another*. Elementos duplicados no dois conjuntos são incluídos apenas uma vez.
public Set	**difference** (Set another) Cria um novo conjunto pela diferença dos elementos da instância corrente com os elementos de parâmetro *another*.
public Set	**intersection**(Set another) Cria um novo conjunto pela intersecção dos elementos da instância corrente e do parâmetro *another*.

public ZMTKBoolean	**subset**(Set another) Retorna "true" se todos os elementos do conjunto another estão contidos na instância corrente. Caso contrário, retorna "false".
public ZMTKBoolean	**properSubset**(Set another) Retorna "true" se todos os elementos do conjunto another estão contidos na instância corrente e os conjuntos são diferentes. Caso contrário, retorna "false".
public ZMTKBoolean	**membership**(ZMTKObject element) Retorna "true" se existe no conjunto um outro elemento igual, segundo os termos do método *equalityByattribute*. Caso contrário, retorna "false".
public ZMTKBoolean	**noMembership**(ZMTKObject element) Realiza a operação inversa de *membership*.
public void	**insert**(ZMTKObject element) Insere um novo elemento não duplicado no conjunto. Esta não é uma operação da teoria dos conjuntos mas foi definida para facilitar a inserção de novos elementos nos conjuntos. Este método é chamado pelo construtor para inserir os elementos iniciais especificados como parâmetro.
public ZMTKObject	**item**(int index) Recupera um elemento do conjunto de acordo com a ordem em que ele foi inserido. Esta não é uma operação da teoria dos conjuntos mas foi definida para facilitar a recuperação de elementos nos conjuntos.
public String	**toString**() Retorna uma representação do conjunto sobre a forma de uma cadeia de caracteres.

Métodos Herdados da classe java.lang.Object
clone, finalize, getClass, notify, notifyAll, wait

2.3.3 Classe ZMTK.sets.RangeNumber

Esta classe representa um conjunto finito cujos elementos são os números contidos em um intervalo numérico inteiro.

Herança

java.lang.Object
  |
  +--ZMTK.ZMTKObject
  |
  +--ZMTK.sets.Set

Atributos	
private ZMTKInteger	**begin** Representa o menor número – inicio - do intervalo.
private ZMTKInteger	**end** Representa o maior número – fim - do intervalo.

Construtor
NumberRange(ZMTKInteger b, ZMTKInteger e) Cria um intervalo que cujo início é *b* e o fim e *e*.

Métodos	
protected ZMTKBoolean	**newElementOK**(ZMTKObject element) Redefinição do método newElementOK da classe Set. Retorna "true" se o parâmetro *element* é um elemento válido para o intervalo, ou seja, se ele está contido no intervalo definido, caso contrário, retorna "false".
public ZMTKInteger	**begin**() Retorna o primeiro elemento do intervalo
public ZMTKInteger	**end**() Retorna o último elemento do intervalo

Métodos Herdados da classe java.lang.Object
clone, finalize, getClass, notify, notifyAll, wait
Métodos Herdados da classe ZMTK.sets.Set
equalityByAttribute, emptySet, equality, cardinality, union, difference, intersection, subset, properSubset, membership, noMembeship, insert, item, toString

2.3.4 Classe ZMTK.sets.BinaryRelation

Esta classe representa uma relação binária. A forma de representação adotada considera uma relação binária como um conjunto de pares ordenados, portanto, a classe *BinaryRelation* é um uma subclasse da classe Set que admite apenas objetos *OrderedPair* como elementos.

Herança

java.lang.Object
  |
  +--ZMTK.ZMTKObject
  |
  +--ZMTK.sets.Set

Atributos	
private Set	**domain** É o domínio da relação binária e consiste do conjunto formado pelos primeiros elementos dos pares ordenados.
private Set	**image** É o contra-domínio da relação binária e consiste do conjunto formado pelos segundos elementos dos pares ordenados.

Construtor
BinaryRelation() Cria uma relação binária vazia.
BinaryRelation(Set dom, Set ran, Set elements) Cria uma relação onde são especificados os elementos iniciais do domínio, os elementos iniciais do contra-domínio e os pares ordenados. Observa-se que os pares ordenados contidos em *elements* devem estar de acordo com os elementos do domínio *dom* e do contra-domínio *ran*.
BinaryRelation(Set elements) Cria uma relação binária a partir do conjunto de pares ordenados *elements* especificado. O domínio e o contra-domínio são a partir dos primeiros e segundos elementos dos pares, respectivamente.

Métodos	
protected ZMTKBoolean	**newElementOK**(ZMTKObject element) Redefinição do método newElementOK da classe Set. Retorna "true" se o parâmetro *element* é um elemento válido para a relação binária, ou seja, se o parâmetro é uma instância de *OrderedPair*, caso contrário, retorna "false".
public Set	**domain**() Retorna o domínio da relação binária.
public Set	**range**() Retorna o contra-domínio da relação binária.
public BinaryRelation	**union**(BinaryRelation another) Cria uma nova relação binária pela união dos elementos da instância corrente com os elementos do parâmetro *another*.
public BinaryRelation	**intersection**(BinaryRelation another) Cria uma nova relação binária pela intersecção dos elementos da instância corrente com os elementos do parâmetro *another*.

public BinaryRelation	**difference**(BinaryRelation another) Cria uma nova relação binária pela diferença dos elementos da instância corrente com os elementos do parâmetro *another*.
public BinaryRelation	**overriding**(BinaryRelation another) Cria uma nova relação binária pela união dos elementos da instância corrente com os elementos do parâmetro *another*, entretanto, quando há duplicatas entre os elementos da instância corrente e os elementos do parâmetro *another*, descarta-se o elemento da instância corrente e inclui-se os elementos do parâmetro *another*.
public Set	**image**(Set setX) Cria um conjunto que contem os elementos correspondentes no contra-domínio da relação binária em relação aos elementos do parâmetro *SetX*.
public BinaryRelation	**domainRestriction**(Set another) Cria uma nova relação binária pela que contém apenas os pares ordenados cujo primeiro elemento pertence ao conjunto *another*.
public BinaryRelation	**rangeRestriction**(Set another) Cria uma nova relação binária pela que contém apenas os pares ordenados cujo segundo elemento pertence ao conjunto *another*.
public BinaryRelation	**domainAntiRestriction**(Set another) Cria uma nova relação binária que contém apenas os pares ordenados cujo primeiro elemento não pertence ao conjunto *another*.
public BinaryRelation	**relationalInversion**() Cria uma nova relação binária pela inversão do domínio e do contra-domínio.
public BinaryRelation	**rangeAntiRestriction**(Set another) Cria uma nova relação binária pela que contém apenas os pares ordenados cujo segundo elemento não pertence ao conjunto *another*.
public OrderedPair	**pair**(int index) Retorna um par ordenado da relação de acordo com sua ordem de entrada indicada pelo parâmetro *index*.
public void	**insert**(ZMTKObject element) Redefinição da operação *insert* da classe Set. Insere um par ordenado não duplicado à relação binária. Esta não é uma operação da teoria dos conjuntos mas foi definida para facilitar a inserção de novos elementos.

Métodos Herdados da classe Java.lang.Object
clone, finalize, getClass, notify, notifyAll, wait
Métodos Herdados da classe ZMTK.sets.Set
equalityByAttribute, emptySet, equality, cardinality, union, difference, intersection, subset, properSubset, membership, noMembeship, item, toString

2.3.5 Classe ZMTK.sets.IdentifyRelation

Representa a relação de identidade. Uma relação de identidade consiste de uma relação binária onde todos os seus pares ordenados têm o primeiro elemento igual ao segundo.

Herança

java.lang.Object
  |
  +--ZMTK.ZMTKObject
  |
  +--ZMTK.sets.Set
  |
  +--ZMTK.sets.BinaryRelation

Construtor
IdentifyRelation(Set elements) Cria uma relação de identidade a partir dos elementos especificados em *elements*.

Métodos	
public void	**insert**(ZMTKObject element) Redefinição da operação *insert* da classe BinaryRelation. Insere um par ordenado não duplicado à relação de identidade. O novo elemento consiste de um par ordenado onde tanto o primeiro como o segundo elemento são iguais a *element*.

Métodos Herdados da classe Java.lang.Object
clone, finalize, getClass, notify, notifyAll, wait
Métodos Herdados da classe ZMTK.sets.Set
equalityByAttribute, emptySet, equality, cardinality, union, difference, intersection, subset, properSubset, membership, noMembeship, item, toString

Métodos Herdados da classe ZMTK.sets.BinaryRelation
newElementOK, domain, range, union, intersection, difference, overriding, image, domainRestriction, domainRestriction, rangeRestriction, domainAntiRestriction, rangeAntiRestriction, pair

2.3.6 Classe ZMTK.sets.PartialFunction

Esta classe representa uma função parcial. A implementação desse tipo considera uma função parcial como uma relação binária onde cada elemento no domínio é possível associar apenas um elemento no contra-domínio.

Herança

java.lang.Object
  |
  +--ZMTK.ZMTKObject
  |
  +--ZMTK.sets.Set
  |
  +--ZMTK.sets.BinaryRelation

Construtor
PartialFunction() Cria uma função parcial vazia.
PartialFunction(Set elements) Cria uma função parcial a partir dos elementos especificados em *elements*.
PartialFuncion(Set dom, Set ran, Set elements) Cria uma função parcial onde são especificados os elementos iniciais do domínio, os elementos iniciais do contra-domínio e os pares ordenados. Observa-se que os pares ordenados contidos em *elements* devem estar de acordo com os elementos do domínio *dom* e do contra-domínio *ran*.

Métodos	
protected ZMTKBoolean	**newElementOK**(ZMTKObject element) Redefinição do método newElementOK da classe BinaryRelation. Retorna "true" se o parâmetro *element* é um elemento válido para a função parcial, ou seja, se ele é uma instância de *OrderedPair* e se o primeiro elemento do par já não está associado a outro elemento do contra-dominio. Caso contrário, retorna "false". Este método é chamado pelos construtores e pelo método *insert*.
public ZMTKObject	**f**(ZMTKObject x) Retorna o elemento que corresponde à imagem do parâmetro *x* na função.

Métodos Herdados da classe Java.lang.Object
clone, finalize, getClass, notify, notifyAll, wait
Métodos Herdados da classe ZMTK.sets.Set
equalityByAttribute, emptySet, equality, cardinality, union, difference, intersection, subset, properSubset, membership, noMembeship, item, toString
Métodos Herdados da classe ZMTK.sets.BinaryRelation
insert, domain, range, union, intersection, difference, overriding, image, domainRestriction, domainRestriction, rangeRestriction, domainAntiRestriction, rangeAntiRestriction, relationalInversion, pair

2.3.7 Classe ZMTK.sets.TotalFunction

Representa uma função total, ou seja, uma função onde todos os elementos do domínio devem ser relacionados a um elemento do contra-domínio.

Herança

java.lang.Object
  |
  +--ZMTK.ZMTKObject
  |
  +--ZMTK.sets.Set
  |
  +--ZMTK.sets.BinaryRelation
  |
  +--ZMTK.sets.PartialFunction

Construtor
TotalFunction() Cria uma função total vazia.
TotalFunction (Set elements) Cria uma função total a partir dos elementos especificados em *elements*.
TotalFunction (Set dom, Set ran, Set elements) Cria uma função total onde são especificados os elementos iniciais do domínio, os elementos iniciais do contra-domínio e os pares ordenados. Observa-se que os pares ordenados contidos em *elements* devem estar de acordo com os elementos do domínio *dom* e do contra-domínio *ran*.

Métodos	
protected ZMTKBoolean	**newElementOK**(ZMTKObject element) Redefinição do método *newElementOK* da classe *PartialFunction*. Retorna "true" se o parâmetro *element* é um elemento válido para a função total, ou seja, se ele é uma instância de *OrderedPair*, se o primeiro elemento do par já não está associado a outro elemento do contra-domínio e se todos os elementos do domínio estão associados a um elemento no contra-domínio. Caso contrário, retorna "false". Este método é chamado pelos construtores e pelo método *insert*.

Métodos Herdados da classe Java.lang.Object
clone, finalize, getClass, notify, notifyAll, wait
Métodos Herdados da classe ZMTK.sets.Set
equalityByAttribute, emptySet, equality, cardinality, union, difference, intersection, subset, properSubset, membership, noMembeship, item, toString
Métodos Herdados da classe ZMTK.sets.BinaryRelation
insert, domain, range, union, intersection, difference, overriding, image, domainRestriction, domainRestriction, rangeRestriction, domainAntiRestriction, rangeAntiRestriction, pair
Métodos Herdados da classe ZMTK.sets.PartialFunction
f

2.3.8 Classe ZMTK.sets.PartialInjectionFunction

Representa as funções injetoras parciais. Esse tipo de função é implementado como uma função parcial onde todos os elementos do domínio têm está associado com apenas um elemento do contra-domínio.

Herança

```
java.lang.Object
  |
  +--ZMTK.ZMTKObject
       |
       +--ZMTK.sets.Set
            |
            +--ZMTK.sets.BinaryRelation
                 |
                 +--ZMTK.sets.PartialFunction
```

Construtor
PartialInjectionFunction()
Cria uma função parcial injetora vazia.
PartialInjectionFunction(Set elements)
Cria uma função parcial injetora a partir dos elementos especificados em *elements*.
PartialInjectionFunction(Set dom, Set ran, Set elements)
Cria uma função parcial injetora onde são especificados os elementos iniciais do domínio, os elementos iniciais do contra-domínio e os pares ordenados. Observa-se que os pares ordenados contidos em *elements* devem estar de acordo com os elementos do domínio *dom* e do contra-domínio *ran*.

Métodos	
protected	**newElementOK**(ZMTKObject element)
ZMTKBoolean	Redefinição do método *newElementOK* da classe *PartialFunction*. Retorna "true" se o parâmetro *element* é um elemento válido para a função parcial injetora, ou seja, se ele é uma instância de *OrderedPair*, se o primeiro elemento do par já não está associado a outro elemento do contra-domínio e se todos os elementos do contra-domínio têm apenas um elemento correspondente associado no domínio. Caso contrário, retorna "false". Este método é chamado pelos construtores e pelo método *insert*.

Métodos Herdados da classe Java.lang.Object
clone, finalize, getClass, notify, notifyAll, wait
Métodos Herdados da classe ZMTK.sets.Set
equalityByAttribute, emptySet, equality, cardinality, union, difference, intersection, subset, properSubset, membership, noMembeship, item, toString
Métodos Herdados da classe ZMTK.sets.BinaryRelation
insert, domain, range, union, intersection, difference, overriding, image, domainRestriction, domainRestriction, rangeRestriction, domainAntiRestriction, rangeAntiRestriction, relationalInversion, pair
Métodos Herdados da classe ZMTK.sets.PartialFunction
f

2.3.9 Classe ZMTK.TotalInjectionFunction

Representa as funções injetoras totais. Essa função é implementada como funções injetoras parciais acrescidas da restrição de que todos os elementos do domínio deve ter um elemento correspondente no contra-domínio.

Herança

java.lang.Object
  |
  +--ZMTK.ZMTKObject
  |
  +--ZMTK.sets.Set
  |
  +--ZMTK.sets.BinaryRelation
  |
  +--ZMTK.sets.PartialFunction
  |
  +--ZMTK.sets.PartialInjectionFunction

Construtor
TotalInjectionFunction() Cria uma função total injetora vazia.
TotalInjectionFunction(Set elements) Cria uma função total injetora a partir dos elementos especificados em *elements*.
TotalInjectionFunction(Set dom, Set ran, Set elements) Cria uma função total injetora onde são especificados os elementos iniciais do domínio, os elementos iniciais do contra-domínio e os pares ordenados. Observa-se que os pares ordenados contidos em *elements* devem estar de acordo com os elementos do domínio *dom* e do contra-domínio *ran*.

Métodos	
protected ZMTKBoolean	**newElementOK**(ZMTKObject element) Redefinição do método *newElementOK* da classe *PartialInjectionFunction*. Retorna "true" se o parâmetro *element* é um elemento válido para a função parcial injetora, ou seja, se ele é uma instância de *OrderedPair*, se o primeiro elemento do par já não está associado a outro elemento do contra-domínio, se todos os elementos do domínio têm um e apenas um elemento correspondente associado no contra-domínio. Caso contrário, retorna "false". Este método é chamado pelos construtores e pelo método *insert*.

Métodos Herdados da classe Java.lang.Object
clone, finalize, getClass, notify, notifyAll, wait
Métodos Herdados da classe ZMTK.sets.Set
equalityByAttribute, emptySet, equality, cardinality, union, difference, intersection, subset, properSubset, membership, noMembeship, item, toString
Métodos Herdados da classe ZMTK.sets.BinaryRelation
insert, domain, range, union, intersection, difference, overriding, image, domainRestriction, domainRestriction, rangeRestriction, domainAntiRestriction, rangeAntiRestriction, pair
Métodos Herdados da classe ZMTK.sets.PartialFunction
f

2.3.10 Classe ZMTK.BijectionFunction

Representa as funções bijetoras, ou seja, funções que são totais, injetoras e sobrejetoras.

Herança

java.lang.Object
  |
  +--ZMTK.ZMTKObject
  |
  +--ZMTK.sets.Set
  |
  +--ZMTK.sets.BinaryRelation
  |
  +--ZMTK.sets.PartialFunction
  |
  +--ZMTK.sets.PartialInjectionFunction
  |
  +--ZMTK.sets.TotalInjectionFunction

Construtor
BijectionFunction() Cria uma função bijetora vazia.
BijectionFunction(Set elements) Cria uma função bijetora a partir dos elementos especificados em *elements*.
BijectionFunction(Set dom, Set ran, Set elements) Cria uma função bijetora onde são especificados os elementos iniciais do domínio, os elementos iniciais do contra-domínio e os pares ordenados. Observa-se que os pares ordenados contidos em *elements* devem estar de acordo com os elementos do domínio *dom* e do contra-domínio *ran*.

Métodos	
Protected ZMTKBoolean	**newElementOK**(ZMTKObject element) Redefinição do método *newElementOK* da classe *TotalInjectionFunction*. Retorna "true" se o parâmetro *element* é um elemento válido para a função parcial injetora, ou seja, se ele é uma instância de *OrderedPair*, se o primeiro elemento do par já não está associado a outro elemento do contra-domínio, se todos os elementos do domínio estão associados a apenas um elemento do contra-domínio e todos os elementos do contra-domínio têm um correspondente no domínio. Caso contrário, retorna "false". Este método é chamado pelos construtores e pelo método *insert*.

Métodos Herdados da classe Java.lang.Object
clone, finalize, getClass, notify, notifyAll, wait
Métodos Herdados da classe ZMTK.sets.Set
equalityByAttribute, emptySet, equality, cardinality, union, difference, intersection, subset, properSubset, membership, noMembeship, item, toString
Métodos Herdados da classe ZMTK.sets.BinaryRelation
insert, domain, range, union, intersection, difference, overriding, image, domainRestriction, domainRestriction, rangeRestriction, domainAntiRestriction, rangeAntiRestriction, relationalInversion, pair
Métodos Herdados da classe ZMTK.sets.PartialFunction
f

2.3.11 Classe ZMTK.sets.PartialSurjectionFunction

Representa as funções sobrejetoras parciais. Esse tipo de função é implementado como uma função parcial onde todos os elementos do contra-domínio têm uma associação com algum elemento do domínio.

Herança

```
java.lang.Object
  |
  +--ZMTK.ZMTKObject
       |
       +--ZMTK.sets.Set
            |
            +--ZMTK.sets.BinaryRelation
                 |
                 +--ZMTK.sets.PartialFunction
```

Construtor

PartialSurjectionFunction()

Cria uma função parcial sobrejetora vazia.

PartialSurjectionFunction (Set elements)

Cria uma função parcial sobrejetora a partir dos elementos especificados em *elements*.

PartialSurjectionFunction (Set dom, Set ran, Set elements)

Cria uma função parcial sobrejetora onde são especificados os elementos iniciais do domínio, os elementos iniciais do contra-domínio e os pares ordenados. Observa-se que os pares ordenados contidos em *elements* devem estar de acordo com os elementos do domínio *dom* e do contra-domínio *ran*.

Métodos

protected ZMTKBoolean	**newElementOK**(ZMTKObject element) Redefinição do método *newElementOK* da classe *PartialFunction*. Retorna "true" se o parâmetro *element* é um elemento válido para a função parcial sobrejetora, ou seja, se ele é uma instância de *OrderedPair*, se o primeiro elemento do par já não está associado a outro elemento do contra-domínio e se todos os elementos do domínio têm um elemento correspondente associado no contra-domínio. Caso contrário, retorna "false". Este método é chamado pelos construtores e pelo método *insert*.

Métodos Herdados da classe Java.lang.Object

clone, finalize, getClass, notify, notifyAll, wait

Métodos Herdados da classe ZMTK.sets.Set
equalityByAttribute, emptySet, equality, cardinality, union, difference, intersection, subset, properSubset, membership, noMembeship, item, toString
Métodos Herdados da classe ZMTK.sets.BinaryRelation
insert, domain, range, union, intersection, difference, overriding, image, domainRestriction, domainRestriction, rangeRestriction, domainAntiRestriction, rangeAntiRestriction, pair
Métodos Herdados da classe ZMTK.sets.PartialFunction
f

2.3.12 Classe ZMTK.TotalInjectionFunction

Representa as funções injetoras totais. Essa função é implementada como funções injetoras parciais acrescidas da restrição de que todos os elementos do contra-domínio têm um elemento correspondente no domínio.

Herança

java.lang.Object
  |
  +--ZMTK.ZMTKObject
  |
  +--ZMTK.sets.Set
  |
  +--ZMTK.sets.BinaryRelation
  |
  +--ZMTK.sets.PartialFunction
  |
  +--ZMTK.sets.PartialSurjectionFunction

Construtor
TotalInjectionFunction() Cria uma função total sobrejetora vazia.
TotalInjectionFunction(Set elements) Cria uma função total sobrejetora a partir dos elementos especificados em *elements*.
TotalInjectionFunction(Set dom, Set ran, Set elements) Cria uma função total sobrejetora onde são especificados os elementos iniciais do domínio, os elementos iniciais do contra-domínio e os pares ordenados. Observa-se que os pares ordenados contidos em *elements* devem estar de acordo com os elementos do domínio *dom* e do contra-domínio *ran*.

Métodos	
protected ZMTKBoolean	**newElementOK**(ZMTKObject element) Redefinição do método *newElementOK* da classe *PartialSurjectionFunction*. Retorna "true" se o parâmetro *element* é um elemento válido para a função parcial sobrejetora, ou seja, se ele é uma instância de *OrderedPair*, se o primeiro elemento do par já não está associado a outro elemento do contra-domínio, se todos os elementos do domínio têm um elemento correspondente associado no contra-domínio domínio. Caso contrário, retorna "false". Este método é chamado pelos construtores e pelo método *insert*.

Métodos Herdados da classe Java.lang.Object
clone, finalize, getClass, notify, notifyAll, wait
Métodos Herdados da classe ZMTK.sets.Set
equalityByAttribute, emptySet, equality, cardinality, union, difference, intersection, subset, properSubset, membership, noMembeship, item, toString
Métodos Herdados da classe ZMTK.sets.BinaryRelation
insert, domain, range, union, intersection, difference, overriding, image, domainRestriction, domainRestriction, rangeRestriction, domainAntiRestriction, rangeAntiRestriction, relationalInversion, pair
Métodos Herdados da classe ZMTK.sets.PartialFunction
f

2.3.13 Classe ZMTK.sets.Sequence

Representa uma lista ordenada na linguagem Z. Um *Sequence* consiste de uma função onde os elementos do domínio são uma seqüência numérica ininterrupta de inteiros e o contra-domínio é composto de qualquer tipo de elemento.

Herança

java.lang.Object
  |
  +--ZMTK.ZMTKObject
  |
  +--ZMTK.sets.Set
  |
  +--ZMTK.sets.BinaryRelation
  |
  +--ZMTK.sets.PartialFunction

Construtor
Sequence() Cria uma seqüência vazia.
Sequence(Set elements) Cria uma seqüência a partir dos elementos especificados em *elements*.
Sequence(ZMTKObject item) Cria uma seqüência com um elemento.

Métodos	
protected ZMTKBoolean	**newElementOK**(ZMTKObject element) Redefinição do método *newElementOK* da classe *PartialFunction*. Retorna "true" se o parâmetro *element* é um elemento válido para a seqüência, ou seja, se ele é uma instância de *OrderedPair*, se o primeiro elemento do par já não está associado a outro elemento do contra-domínio e o novo elemento respeita seqüência numérica do domínio. Caso contrário, retorna "false". Este método é chamado pelos construtores e pelo método *insert*.
public Sequence	**concatenation**(Sequence another) Cria uma nova seqüência pela concatenação da seqüência corrente com o parâmetro *another*, nesta ordem.
public Sequence	**reverse**() Cria uma nova seqüência pela inversão da ordem dos elementos da seqüência corrente.
public ZMTKObject	**head**() Retorna o primeiro elemento da seqüência.

public ZMTKObject	**last**() Retorna o último elemento da seqüência.
public Sequence	**tail**() Retorna a seqüência corrente sem o primeiro elemento.
public Sequence	**front**() Retorna a seqüência corrente sem o último elemento.
public Sequence	**extraction**(Set setX) Cria uma nova seqüência composta pelos elementos relacionados aos valores do conjunto de índices *setX*.
public Sequence	**filtering**(Set setY) Cria uma nova seqüência composta pelos elementos pertencentes à seqüência corrente e ao conjunto *setY*, na mesma ordem da seqüência corrente.
public boolean	**prefix**(Sequence another) Retorna "true" se os primeiros elementos da seqüência *another* são os mesmos da seqüência corrente, caso contrário, retorna "false".
public boolean	**suffix**(Sequence another) Retorna "true" se os últimos elementos da seqüência *another* são os mesmos da seqüência corrente, caso contrário, retorna "false".
public boolean	**in**(Sequence another) Retorna "true" se a seqüência *another* está contida na seqüência corrente.
public Bag	**items**() Cria um Bag a partir do elementos da seqüência corrente.
public void	**insert**(ZMTKObject element) Insere um novo elemento na seqüência. O parâmetro element é inserido como o segundo elemento de um par ordenado onde o primeiro elemento é calculado somando-se um ao índice o último elemento.

Métodos Herdados da classe Java.lang.Object
clone, finalize, getClass, notify, notifyAll, wait
Métodos Herdados da classe ZMTK.sets.Set
equalityByAttribute, emptySet, equality, cardinality, union, difference, intersection, subset, properSubset, membership, noMembeship, item, toString
Métodos Herdados da classe ZMTK.sets.BinaryRelation
insert, domain, range, union, intersection, difference, overriding, image, domainRestriction, domainRestriction, rangeRestriction, domainAntiRestriction, rangeAntiRestriction, relationalInversion, pair

Métodos Herdados da classe ZMTK.sets.PartialFunction
f

2.3.14 Classe ZMTK.sets.Bags

Bags é uma função onde o domínio é formado por um conjunto de elementos quaisquer e o contra-domínio é formado por um número natural que represente a quantidade de ocorrências do primeiro elemento.

Herança

java.lang.Object
  |
  +--ZMTK.ZMTKObject
  |
  +--ZMTK.sets.Set
  |
  +--ZMTK.sets.BinaryRelation
  |
  +--ZMTK.sets.PartialFunction

Construtor
Bags() Cria um *bags* vazio.
Bags(Set elements) Cria um bags a partir dos elementos especificados em *elements*.
Bags(ZMTKObject item) Cria um bags com um elemento.

Métodos	
public ZMTKNatural	**count**(ZMTKObject element) Retorna o valor associado ao elemento *element*.
public Bags	**scalling**(ZMTKNatural factor) Cria um novo *bags* com os elementos do *bags* corrente associados aos seus respectivos números naturais multiplicados pelo parâmetro *factor*.
public ZMTKBoolean	**membership**(ZMTKObject element) Retorna "true" se *element* está contido no *bags* e está associado a um valor maior que zero.
public ZMTKBoolean	**subbag**(Bags another) Retorna "true" se o *bags another* está contido no *bags* corrente, ou seja, se todos os elementos associados a valores maiores que zero em *another*, existem e são associados a valores maiores que zero no *bags* corrente.

public Bags	**union**(Bags another) Cria um novo *bags* composto pelos elementos do *bags* corrente e de *another*, sendo que os elementos comuns têm seus valores somados.
public Bag	**difference**(Bags another) Cria um novo bags composto pelos elementos do bags corrente
public String	**toString**() Retorna uma representação do Bags sobre a forma de uma cadeia de caracteres.
public void	**insert**(ZMTKObject element) Insere um novo elemento no Bags. O parâmetro *element* é inserido como o primeiro elemento de um par ordenado onde o segundo elemento é calculado como "1" se *element* não existe no Bags, ou somando-se "1" ao seu índice, se o *element* já existe.

Métodos Herdados da classe Java.lang.Object
clone, finalize, getClass, notify, notifyAll, wait
Métodos Herdados da classe ZMTK.sets.Set
equalityByAttribute, emptySet, equality, cardinality, union, difference, intersection, subset, properSubset, membership, noMembeship, item
Métodos Herdados da classe ZMTK.sets.BinaryRelation
domain, range, union, intersection, difference, overriding, image, domainRestriction, domainRestriction, rangeRestriction, domainAntiRestriction, rangeAntiRestriction, relationalInversion, pair
Métodos Herdados da classe ZMTK.sets.PartialFunction
f

2.4 Pacote ZMTK.nonMath

O pacote *nonMath* contém entidades presentes na linguagem de especificação mas não são originários de entidades matemáticas.

2.4.1 Classe ZMTK.nonMath.ZMTKString

Representa uma cadeia de caracteres.

Herança

java.lang.Object
  |
  +--ZMTK.ZMTKObject

Atributos	
private String	**value** É a cadeia de caracteres.

Construtor
ZMTKString() Cria uma cadeia de caracteres vazia.
ZMTKString(String v) Cria uma cadeia de caracteres a partir de uma String Java.

Métodos	
public boolean	**equalityByAttribute**(ZMTKObject parm) Implementação do método abstrato definido em ZMTKObject, retorna "true" se todos os caracteres da cadeia corrente e do parâmetro *parm* são iguais e estão na mesma ordem, caso contrário, retorna "false"
public void	**set**(String v) Seta a cadeia de caracteres a partir do parâmetro *v*.
public ZMTKString	**value**() Retorna a cadeia de caracteres corrente.

Métodos Herdados da classe java.lang.Object
clone, finalize, getClass, notify, notifyAll, wait

2.4.2 Classe ZMTK.nonMath.OracleOutputStream

OracleOutputStream representa um tipo especial que captura todas as impressões nos dispositivos de saída.

Herança

java.lang.Object
  |
  +--ZMTK.ZMTKObject

Atributos	
private Vector	**Output** Contém os valores impressos nos dispositivos de saída.
private boolean	**isStandard** É um flag que indica se as impressões estão ou não sendo capturadas.
private PrintStream	**oraOutput** É um contêiner onde as saídas impressas são temporariamente armazenadas.
private PrintStream	**systemOutput** É um contêiner que substitui o local onde as saídas são impressas.

Construtor
OracleOutputStream()

Métodos	
public boolean	**equalityByAttribute**(ZMTKObject parm) Implementação do método abstrato definido em ZMTKObject, retorna "true" se todas as saídas impressas são iguais, caso contrário, retorna "false"
public void	**setOracleAsStandardOutput**() Captura o dispositivo de saída e começa a receber todas as saídas impressas.
public boolean	**isStandard**() Retorna "true" se o dispositivo de saída atual é o dispositivo padrão. Caso o dispositivo esteja capturado pelo *OracleOutputStream*.
public void	**getOutputBuffered**() Preenche a instância corrente como as saídas capturadas.
public String	**getOutputLine**(int line) Retorna uma linha capturada de acordo com a sua ordem de captura – *line*.

public int	**size**() Retorna o número de linhas capturadas.
public void	**setSystemAsStandardOutput**() Devolve o controle do dispositivo de saída ao sistema.
public void	**flush**() Limpa o contêiner de linhas capturadas.

Métodos Herdados da classe java.lang.Object
clone, finalize, getClass, notify, notifyAll, wait

Anexo 3 Gramática da Linguagem OZJ-Mapping

```
Mapping OZJ          =    ClassMap "{"
                                HierarchyMap
                                ConstantMaps
                                StateVariablesMaps
                                InitMap
                                OperationMaps
                          "}"
ControlMapping       =    "class(" ControlMapping ")"
MappingClass         =    Identifier "," Identifier
HierarchyMap         =    "hierarchy(" parentList ")"
ParentList           =    ClassName { "," ClassName }
ConstantMaps         =    ConstantMap { ";" ConstantMap }
ConstantMap          =    "constant(" DataMapping ")" "{"
                              TypeMap AbstractFunctionID
                          "}"
DataMapping          =    Zdentifier "," JavaIdentifier
TypeMap              =    "type(" MappingType ")"
MappingType          =    ZType "," JavaType
AbstractFunctionID   =     ":abstract(" Number ")"
StateVariablesMaps   =    StateVariableMap { ";" StateVariableMap }
StateVariableMap     =    "state(" DataMapping ")" "{"
                                TypeMap AbstractFunctionID
                          "}"
InitMap              =    "init(" "init" "," JavaIdentifier ")" "{"
                              ParameterMaps
                          "}"
ParameterMaps        =    ParameterMap { ";" ParameterMap }
ParemeterMap         =    "parameter(" DataMapping ")" "{"
                              TypeMap AbstractFunctionID
                          "}"
OperationMaps        =    OperationMap ";" OperationMap
OperationMap         =    "operation(" ControlMapping ")" "{"
                                ParameterMaps
                          "}"
ClassName            =    Identifier
```

Identifier	=	Letter { Letter | Number | _ }
ZIdentifier	=	Identifier ("?" | "!" | "'")
JavaIdentifier	=	Identifier
ZType	=	"\" Letter { Letter }
JavaType	=	Identifier
Letter	=	"A" | ... | "Z" | "a" | ... | "z"
Number	=	"0" | ... | "9"

Bibliografia

[ABR 96] ABRIAL, J.-R. **The B-Book**. Cambridge: Cambridge University Press, 1996.

[AMM 98] AMMANN, P. E.; BLACK, P. E.; MARJUSKI, W. Using Model Check to Generate Tests for Specifications. In: IEEE INTERNATIONAL CONFERENCE OF FORMAL ENGINEERING METHODS, 1998. **Proceedings...** Brisbane: ICFEM, 1998

[ANT 2000] 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, Jan. 2000.

[AZE 98] AZEVEDO, E. E. **Uma Proposta para Utilização de Orientação a Objetos para a Construção de Compiladores Parametrizados**. 1998. Trabalho de Conclusão (Bacharelado em Ciência da Computação) – CBCC, Universidade Federal do Pará, Belém.

[BER 91] BERNOR, G.; GAUDEL, M.-C.; MARRE, B. Software Testing based on Formal Specifications: a Theory and a Tool. **Software Engineering Journal**, [S.l.], v.6, n.6, p.387-405, Nov. 1991.

[BIE 92] BIEMAN, J. M.; YIN, H. Design for Software Testability Using Automated Oracles. In INTERNATIONAL TEST CONFERENCE, 1992. **Proceedings...** Baltimore:ITC, 1992.

[BIN 94] BINDER, R. V. **The FREE Approach to Testing Object-Oriented Software:** An Overview. Chicago:Robert Binder Systems Consulting, 1994. (Technical Report RBSC-94-003).

[CAL 96] CALLAHAN, J.; EASTERBROOK S.; SCHNEIDER, F. Automated Software Testing Using Model-Checking. In: SPIN WORKSHOP, 1996. **Proceedings...** Rutger:[s.n.], 1996.

[CHA 96] CHANG, R.; RICHARDSON, D. J. Structural Specification-based Testing with ADL. **Software Engineering Notes**, New York, v.21, n.3, p.62-70, May 1996. Trabalho apresentado no International Symposium on Software Testing and Analysis, ISSTA, 1996, San Diego.

[CHA 99] CHANG, J.; RICHARDSON, D. Structural Specification-based Testing: Automated Support and Experimental Evaluation**.** **Software Engineering Notes**, New York, v.14, n.6, 1999.

[CLA 96] CLARKE, E. M.; WING, J. M. Formal Methods State of The Art and Future Directions. **ACM Computing Surveys**, n.28, v.4, p.626-643, Dec. 1996.

[DIL 94] DILLER, A. **Z**: An Introduction to Formal Methods. Chichester: Jonh Wisley Sons, 1994.

[DON 97] DONAT, M. Automating Formal Specification-Based Testing. TAPSOFT'97, 1997. In: INTERNATIONAL JOINT CONFERENCE ON THEORY AND PRACTICE OF SOFTWARE DEVELOPMENT, TAPSOFT, 7., 1997. **Theory and Practice of Software Development**:

Proceedings. Berlin:Springer-Verlag, 1997. p.833-847. (Lecture Notes on Computer Science, v.1214).

[FIE 89] FIEDLER, S. P. Object-Oriented Unit Testing. **Hewlet-Packard Journal**, Palo Alto, v.40, n.2, p.69-75, Apr. 1989.

[FLE 96] FLETCHER, R.; SAJEEV, A. S. M. A Framework for Testing Object Oriented Software Using Formal Specifications. In: ADA-EUROPE INTERNATIONAL CONFERENCE ON RELIABLE SOFTWARE TECHNOLOGIES. **Reliable Software Technologies**: proceedings. Mountreux: Springer-Verlag, 1996. p.159-170. (Lecture Notes in Computer Science, v.1008)

[GAN 81] GANNON, J.; McMULLIN, P.; HAMLET, R. Data-Abstraction Implementation, Specification and Testing. **ACM Transactions on Programming Languages and Systems**, New York, v.3, n.3, p.211-223, July 1981.

[GAU 95] GAUDEL, M-C. Advantages and Limits of Formal Approaches for Ultra-High Dependability. In: RANDELL, B. **Predictably Dependable Computing Systems**. Berlin:Springer-Verlag, 1995. p.241-251.

[GOO 75] GOODENOUGH, J. B.; GERHART, S. L. Towards a theory of test data selection. **IEEE Transactions on Software Engineering**, New York, v.2, n.1, p.156-173, 1975.

[GOR 91] GORLICK, M. M. The Fliter Record: An Architetural Aid for System Monitoring. In: WORKSHOP ON PARALLEL AND DISTRIBUTED DEBBUGING, 1991. **Proceeding…**. [S.l.:s.n], 1994.

[HAM 91] HAMLET, D. Software Quality, Software Process and Software Testing. **Advances in Computer**, [S.l.], n.41, p.191-229, 1995.

[HAR 92] HARROLD, M. J.; McGREGOR, J. D.; FITZPATRICK, Kevin J. Incremental Testing of Object-Oriented Class Structures. In: INTERNATIONAL CONFERENCE OF SOFTWARE ENGINEERING, 14., 1992. **Proceedings…**. Melbourne: ACM Press, 1992. p.98-79.

[HAY 86] HAYES, I. J. Specification Directed Module Testing. **IEEE Transactions on Software Engineering**, New York, v.1, n.12, p.124-133, Jan. 1986.

[HEI 96] HEITMEYER, C.; JEFFORDS, R.; LABAW, B. Automated Consistency Checking of Requirements Specifications. **ACM Transactions on Software Engineering and Methodology**, New York, v.3, n.5, p.231-261, July 1996.

[HOA 69] HOARE, C. A. R. An Axiomatic Basis for Computer Programming. **Communcations of ACM**, New York, v.10, n.12, p.576-583, Oct. 1969.

[HOA 72] HOARE, C. A. R. Proof of Correctness of Data Representations. **Acta Informatica**, Berlin, v.1, n.4, p. 271-281, Feb.1972.

[HOF 91] HOFFMAN, D. M.; STROOPER, P. Automated Module Testing in Prolog. **IEEE Transactions on Software Engineering**, New York, v.9, n.17, p.934-942, Sept. 1991.

[HOF 97] HOFFMAN, D. M.; STROOPER, P. A. A Methodology and Framework for Automated Class Testing. Software Practice and Experience. **IEEE Transactions on Software Engineering**, New York, v.5, n.27, p.573-597, May 1997.

[HÖR 95] HÖRCHER, Hans-Martin. Improving Software Testing Using Z Specifications, In: ANNUAL Z USER MEETING, 9., 1995. **Proceedings…**. Limeck: [s.n.], 1995.

[JON 90] JONES, C. B. **Systematic Software Development Using VDM**. 2nd ed. Englewood Chfts:Prentice Hall, 1990.

[KUN 95] KUNG, D. et al. Developing an Object-Oriented Software Testing and Maintenance Enviroment. **Communications of ACM**, New York, v.38, n.10, p.75-87, Oct. 1995.

[LAM 2000] LAMSWEERDS, A. V. Formal Specification: a Roadmap, In: INTERNATIONAL CONFERENCE OF SOFTWARE ENGINEERING, 2000. **Proceedings…** Atlantic City:[s.n.], 2000.

[MAR 98] MARTINS, E. **Verificação e validação**. Campinas: Instituto de Computação, UNICAMP, 1998.

[McD 97] McDONALD, J.; STROOPER, P.; MURRAY, L. Translating Object-Z Specifications to Object-Oriented Test Oracles, In: ASIAN-PACIFIC SOFTWARE ENGINEERING CONFERENTE, 1997. **proceedings…**. Hong Kong:[s.n.], 1997.

[McG 96] McGREGOR, J. D. Testing Object-Oriented Components, In: EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, 1996. **Proceedings…**. Berlin:[s.n.], 1996.

[MEU 97] MEUDEC, C. **Automatice Generation of Software Tests From Formal Specifications**. 1997. PhD Thesis – The Queen's University of Belfast, Belfast.

[MEY 92] MEYER, B. **Eiffel The Language**. Hertfordshire:Prentice-Hall, 1992.

[MIK 95] MIKK, E. Compilation of Z Specifications into C for Automatic Test Result Evaluation, In: Z USER MEETING, 1995. **Proceedings…**. Berlin:[s.n.], 1995.

[MOS 91] MOSLEY, Daniel J. **The Handbook of MIS Application Software Testing**. New Jersey: Yourdon Press, 1991.

[MYE 79] MYERS, G. **The Art of Software Testing**. New York: Wiley-Interscience, 1979.

[OFF 99] OFFUTT, A., LIU. S. Generating Test Data From SOFL Specification. **The Journal of Systems and Software**, [S.l.], v.1, n.49, p.49-62, Dec. 1999

[OMA 96] O'MALLEY, Owen. **A Model of Specification-based Test Oracles**. 1996. PhD Thesis - University of California, Irvine.

[OST 88] OSTRAND, T. J.; BALCER, M. J. The Category-Partition Method for Specifying and Generating Functional Testes. **Communications of the ACM**, New York, v.6, n.31, p.676-686, June 1988.

[PAR 95] PARNAS, D. L, MADEY, J. Functional Documentation for Computer Systems. **Science of Computer Programming**, [S.l.], v.25, n.1, p.41-61, May 1995.

[PET 98] PETER, D. K.; PARNAS, D. L. Using Test Oracles Generated from Program Documentation. **IEEE Transactions on Software Engineering**, New York, v.3, n.24, p.161-173, Mar. 1998.

[PRA 83] PRATHER, R. E. Theory of Program Testing – An Overview. **The Bell Technical Journal**, [S.l.], v.10, n.62, June 1983.

[RAP 85] RAPPS, S.; WEYUKER, E. J. Selecting Software Test using Data Flow Information. **IEEE Transactions on Software Engineering**, New York, v.4, n.11, p.367-375, Apr. 1985.

[REI 85] REISIG, W. **Petri Nets:** An Introduction. Berlin:Springer-Verlag, 1985.

[RIC 89] RICHARDSON, D. J.; O'MALLEY, O.; TITTLE, C. Approaches to Specification-based Testing. **Software Engineering Notes**, New York, v.8, n.14, p.86-96, Dec. 1989.

[RIC 92] RICHARDSON, D. J.; O'MALLEY, O.; LEIF, S. Specification-based Test Oracle for Reactive Systems. In: INTERNATIONAL CONFERENCE ON SOFTWARE TESTING, 14., 1992. **Proceedings…** Melborne:ICSE, 1992.

[RIC 94] RICHARDSON, D. J. TAOS: Testing with Analysis and Oracle Support. In: INTERNATIONAL SYMPOSIUM ON SOFTWARE TESTING AND ANALYSIS, 1994. **Proceedings…** Seatle:[s.n.], 1994.

[ROS 94] 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.

[RUS 93] 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.

[SIN 97] SINGH, H.; CONRAD, M.; SADEGHIPOUR, S. Test Case Desing Based on Z on the Classification-Tree Method, In: INTERNATIONAL CONFERENCE OF FORMAL ENGINEERING METHODS, 1997. **Proceedings...** Hiroshima: [s.n.], 1997.

[SMI 92] SMITH, M. D.; ROBSON, D. J. A Framework for Testing Object-Oriented Programs. **Journal of Object-Oriented Programming**, [S.l.], v.5, n.3, p.45-54, June 1992.

[SPY 92] SPYLEY, Mike. **The Z Notation:** A Reference Manual. 2nd ed. Hemel Hempstead:Prentice-Hall, 1992.

[STO 93] STOCKS, P. A. **Applying Formal Methods to Software Testing**. 1993. PhD Thesis - The University of Queesland, Brisbane.

[STO 94] STOCKS, P. A.; CARRINGTON, D. A tale of two paradigms: Formal methods and Software Testing. In: ENGLISH Z USER MEETING, 8., 1994. **Proceedings…** Cambridge:[s.n.], 1994.

[WEN 96] WENDY, J. **A Type Checker for Object-Z**.Queensland, Software Verification Research Centre, The University of Queensland, 1996. (Technical Report 96-24). Disponível em: <ftp://svrc.it.uq.edu.au/pub/techreports/tr96-24.ps>. Acesso em: nov. 2000.

[WEY 80] WEYUKER, E. J.; OSTRAND, T. J. Theories of Program Testing and Applications of Revealing Subdomains. **IEEE Transactions on Software Engineering**, New York, v.6, n.3, p.236-246, June 1980.

[WEY 86] WEYUKER, E. J. Axiomatizing Software Test Data Engineering. **IEEE Transactions on Software Engineering**, New York, v.16, n.2, p.121-128, Feb. 1986.

[WEY 91] WEYUKER, E. J.; JENG, B. Analyzing Partition Testing Strategies. **IEEE Transactions of Software Engineering**, New York, v.2, v.17, p.703-711, July 1991.

[WHI 80] WHITE, J. L.; COHEN, A. E. A Domain Strategy for Computer Program Testing. **IEEE Transactions on Software Engineering**, New York, v.6, n.3, p.249-257, May 1980.

[WIR 77] WIRTH, N. What Can We Do About the Unnecessary Diversity of Notation for Syntactic Definitions? **Communications of the ACM**, New York, v.20, n.11, p.822-823, Nov. 1977.