Universidade Federal do Rio Grande do Norte
Departamento de Informática e Matemática Aplicada
PPGSC - Programa de Pós Graduação em Sistemas e Computação

Geração de Casos de Teste a partir de Especificações B

Fernanda Monteiro de Souza

Orientadora: Anamaria Martins Moreira

Natal
Dezembro de 2009

Fernanda Monteiro de Souza

Geração de Casos de Teste a partir de Especificações B

Orientadora: Anamaria Martins Moreira

Dissertação apresentada ao Departamento de Informática e Matemática Aplicada da Universidade Federal do Rio Grande do Norte, como requisito para o cumprimento da disciplina DIM0000 - Dissertação de Mestrado.

Natal
Dezembro de 2009

Catalogação da Publicação na Fonte. UFRN / SISBI / Biblioteca Setorial Especializada do Centro de Ciências Exatas e da Terra – CCET.

Souza, Fernanda Monteiro de.

Geração de casos de teste a partir de especificações B / Fernanda Monteiro de Souza. – Natal, 2010.

100 f. : il.

Orientador: Anamaria Martins Moreira.

Dissertação (Mestrado) – Universidade Federal do Rio Grande do Norte. Centro de Ciências Exatas e da Terra. Departamento de Informática e Matemática Aplicada. Programa de Pós-Graduação em Sistemas e Computação.

1. Testes de programas – Dissertação. 2. Especificações formais B – Dissertação. 3. Testes baseados em modelos – Dissertação. I. Moreira, Anamaria Martins. II. Título.

Agradecimentos

Agradeço imensamente e principalmente à Anamaria, minha orientadora, que desde a graduação “atura” minhas mudanças de cidade, meus esquecimentos, minha falta de disponibilidade em algumas horas e meus prolongamentos de prazos. A sua atuação foi fundamental para a conclusão dessa dissertação.

Nos bastidores desta dissertação e desde sempre comigo, agradeço a meus pais, Alba e Fernando, e a minha irmã Leila. E nesta última década e mais alguns poucos anos, agradeço ao meu marido, Adelmo Júnior, pelo companheirismo de todas as horas.

Resumo

Com o crescente aumento da complexidade dos sistemas de *software*, há também um aumento na preocupação com suas falhas. Essas falhas podem causar prejuízos financeiros e até prejuízos de vida. Sendo assim, propomos neste trabalho a minimização de falhas através de testes em *software*s especificados formalmente. A conjunção de testes e especificações formais vem ganhando força na academia principalmente através dos TBM (Testes Baseados em Modelos).

O desenvolvimento de *software* a partir de especificações formais, quando todo o processo de refinamento é feito rigorosamente, garante que o que está especificado será implementado na aplicação. Sendo assim, a implementação gerada a partir destas especificações iria retratar fielmente o que estaria especificado. Mas nem sempre a especificação é refinada até o nível de implementação e geração de código, e nesses casos os testes gerados a partir da especificação tendem a encontrar falhas. Adicionalmente, a geração dos chamados "testes inválidos", ou seja, testes que exercitem cenários da aplicação que não foram tratados na especificação, complementa mais significativamente o processo de desenvolvimento formal.

Sendo assim, neste trabalho é proposto um método para geração de testes a partir de especificações formais B. Este método foi estruturado em pseudo-código. O método se baseia na sistematização das técnicas de testes caixa preta da análise do valor limite, particionamento de equivalência, bem como da técnica dos pares ortogonais. O método foi aplicado em uma especificação B e foram geradas máquinas B de teste que geram casos de teste independentes de linguagem de implementação. Com o intuito de validação do método, os casos de teste foram transformados manualmente em casos de teste do JUnit e a aplicação, criada a partir da especificação B, e desenvolvida em Java foi testada. Foram encontradas falhas com a execução dos casos de teste JUnit.

Palavras-chave: Testes, Especificações Formais, Testes Baseados em Modelos.

Abstract

With the increasing complexity of software systems, there is also an increased concern about its faults. These faults can cause financial losses and even loss of life. Therefore, we propose in this paper the minimization of faults in software by using formally specified tests. The combination of testing and formal specifications is gaining strength in searches mainly through the MBT (Model-Based Testing).

The development of software from formal specifications, when the whole process of refinement is done rigorously, ensures that what is specified in the application will be implemented. Thus, the implementation generated from these specifications would accurately depict what was specified. But not always the specification is refined to the level of implementation and code generation, and in these cases the tests generated from the specification tend to find fault. Additionally, the generation of so-called "invalid tests", ie tests that exercise the application scenarios that were not addressed in the specification, complements more significantly the formal development process.

Therefore, this paper proposes a method for generating tests from B formal specifications. This method was structured in pseudo-code. The method is based on the systematization of the techniques of black box testing of boundary value analysis, equivalence partitioning, as well as the technique of orthogonal pairs. The method was applied to a B specification and B test machines that generate test cases independent of implementation language were generated. Aiming to validate the method, test cases were transformed manually in JUnit test cases and the application, created from the B specification and developed in Java, was tested. Faults were found with the execution of the JUnit test cases.

Keywords: Testing, Formal Specification, Model Based Testing.

Índice de Figuras

Índice de Tabelas

Sumário

Capítulo 1 Introdução

Sistemas computacionais estão crescendo cada vez mais em complexidade. Por causa desse crescimento há uma maior preocupação com o gerenciamento de falhas. Por **falha** (Myers, 2004) entende-se um comportamento diferente do comportamento que foi especificado e que deve ser observado no sistema em execução. As falhas podem causar perdas em dinheiro e até perdas de vidas humanas (Rosenblum, 1996). Neste trabalho serão apresentadas duas formas de minimização dessas falhas, utilizadas em conjunto: os testes de *software* e as especificações formais. Os testes têm por objetivo encontrar falhas com o *software* já construído, enquanto que especificações formais buscam minimizá-las nas fases iniciais do desenvolvimento do *software*. Ambas as técnicas apresentam vantagens e limitações, bem como um alto grau de complementaridade, o que faz com que o seu uso conjunto possa contribuir significativamente para a qualidade dos resultados. É nessa linha de ação que se encaixam atualmente as pesquisas sobre testes baseados em modelos (TBM) (Third Workshop on Model Based Testing, 2007): a cooperação da atividade de testes com a atividade de modelagem (gráfica e/ou formal).

À medida em que testes são executados e falhas são identificadas e corrigidas, espera-se que haja uma diminuição da quantidade de falhas remanescentes no *software*. Contudo, há limitações na atividade de testes. Uma dessas limitações é que a identificação de uma falha indica um teste bem projetado no que diz respeito à precisão na identificação dos dados e casos de teste e sua conseqüente aplicação no exercício do *software*, mas o contrário não significa que o *software* seja perfeito (Kaner, Falk, & Nguyen, 1999). O que se busca é a identificação da maior quantidade possível de falhas distintas, aplicando-se técnicas de projeto de teste de *software*. Isso porque a combinação de todas as possibilidades de dados de entrada para identificar falhas pode tornar o teste inviável, dependendo da quantidade de variáveis envolvidas no teste.

Como a especificação dos requisitos servem como entrada para o projeto dos testes, esta especificação deve estar escrita de maneira adequada. Por isso, um bom desenvolvimento de testes de *software* deve partir de uma especificação clara e concisa de seus requisitos (Loveland, 2005). Para isto podem ser utilizadas especificações formais na engenharia do *software*. As especificações formais, que objetivam minimizar falhas em sistemas computacionais e reduzir erros nas fases iniciais do desenvolvimento do *software*, possuem sintaxe e semântica baseadas em fundamentos matemáticos (Pressman, 2005). Um **erro** é definido por (Myers, 2004) como uma interpretação errônea dos requisitos que faz com que a especificação do *software* seja feita incorretamente. Especificações formais buscam eliminar a má interpretação dos requisitos que a ambigüidade da linguagem natural proporciona,

diminuindo, assim, a quantidade de erros no *software*. Com a utilização de especificações formais há um maior custo nas fases de especificação e análise do sistema que irá ser compensado pela qualidade do produto e pela minimização de retrabalhos (Pressman, 2005).

Para partir de especificações e chegar à execução dos testes e análise dos resultados, surgiram os TBM (Testes Baseados em Modelos). TBM atualmente são bastante discutidos no meio acadêmico e vêm ganhando interesse na indústria de *software*. Em TBM, os testes são derivados de um modelo do sistema (Conrad, 2002) e o processo de testes inclui todas as atividades dos testes de *software*, com exceção do planejamento: projeto de testes, execução e análise dos resultados. Mesmo TBM não incluindo a etapa de planejamento, este planejamento deve ser desempenhado anteriormente.

Neste trabalho foi desenvolvido um método que, a partir do modelo de sistema especificado formalmente em linguagem B, gera casos de teste independentes de linguagem. Sendo assim, buscamos excluir erros que a linguagem natural poderia introduzir, especificando *software* formalmente, e encontrar algumas falhas que porventura ainda estivessem no *software* desenvolvido com a aplicação dos testes. Sabe-se que, quando a aplicação é desenvolvida a partir de sua especificação, ela retratará fielmente o que foi especificado. Dessa forma, o que não necessariamente pode estar especificado e conseqüentemente implementado são situações inválidas, ou seja, casos que quebrariam o funcionamento correto da aplicação. Esta é uma motivação para o uso de TBM e conseqüentemente para o desenvolvimento deste trabalho. Também, o método proposto é aplicável às aplicações que partiram de especificações B, mas sem que estas especificações fossem refinadas até a geração de código. Outra motivação diz respeito à manutenção dos testes: se porventura a especificação do *software* for modificada, os testes são alterados de maneira automática.

Para o desenvolvimento do método proposto, foi feito um estudo de caso de uma aplicação de bilhetagem eletrônica para transporte coletivo urbano, desenvolvida em Java. O método foi estruturado em pseudo-código e foi aplicado para teste desta aplicação, desenvolvida a partir da modelagem em B. O método resultou em casos de teste gerados manualmente no JUnit. Estes casos de teste foram executados e algumas falhas foram encontradas.

1.1. Objetivo do trabalho

Este trabalho tem como objetivos:

1. O desenvolvimento de um método, que foi estruturado em pseudo-código, para a definição de casos de teste em B. Esses casos de teste são desenvolvidos a partir de especificações B de uma aplicação e são independentes de linguagem de implementação. Cada caso de teste B será

animado e resultará em combinações de dados de teste de entrada. Os resultados esperados são obtidos a partir de animações da especificação do *software* sob teste;

2. Implementação e execução do código de teste em JUnit, a partir das entradas e saídas esperadas, para teste da aplicação sob teste. Estas saídas esperadas são comparadas com as saídas do sistema sob teste e assim o caso de teste recebe o status de passou ou falhou. A execução dos testes em JUnit tem o objetivo de validação do método proposto no tocante à descoberta de falhas no sistema.

1.2. Metodologia

Para o desenvolvimento deste trabalho, já foi desenvolvido um trabalho anterior (Souza, 2005). Foi feito em (Souza, 2005) um estudo de caso em uma aplicação de bilhetagem eletrônica (Gomes, 2005) para geração de testes para uma máquina B. Este trabalho anterior focou na escolha das técnicas a serem utilizadas na geração dos casos de teste. Foram escolhidas duas delas: o particionamento de equivalência e a análise do valor limite. Decidiu-se também selecionar, para a escolha dos dados, apenas a pré-condição da operação e partes do invariante da máquina B. As partes selecionadas do invariante eram apenas aquelas que continham variáveis que eram manipuladas pela pré-condição da operação a ser testada. Como a escolha de dados de testes feita naquele trabalho não aconteceu de maneira sistemática, não foi verificada a necessidade de selecionar também cláusulas do invariante que continham informações a respeito daquelas cláusulas já selecionadas. Também naquele trabalho, o tratamento de quais cláusulas selecionar e o tratamento das dependências entre as variáveis estavam sendo feitas de maneira não sistemática e sem a adoção de nenhuma técnica. Neste contexto, tínhamos como pendentes o tratamento das cláusulas a serem selecionadas, a seleção de cláusulas a mais do invariante e o tratamento da dependência entre variáveis, além da sistematização do que já tinha sido levantado.

Sendo assim, para se atingir o objetivo do presente trabalho, a seguinte metodologia foi seguida:

1. Estudo de como era possível gerar dados de teste para variáveis relacionadas:
    a. Estudo das restrições para as variáveis relacionadas;
    b. Estudo de como as restrições deveriam ser combinadas:
        i. Geração de uma ferramenta;
        ii. Estudo da técnica dos pares ortogonais;
2. Estudo de ferramenta para resolução de restrições;
3. Formalização do método;
4. Geração de casos de teste em B;
5. Geração de casos de teste no JUnit.

A partir das pendências encontradas em (Souza, 2005), no presente trabalho foi estudado como seria possível gerar casos de teste para variáveis das especificações B que tinham algum relacionamento. Para isso foi preciso estudar as restrições que uma variável impunha em outras e como essas restrições poderiam ser manipuladas para serem combinadas. Para fazer essas combinações foi estudada a técnica dos pares ortogonais, que será apresentada no próximo capítulo, e também foi gerada uma ferramenta.

De posse das combinações das restrições, foi estudado como poderíamos fazer para que obtivéssemos dados que satisfizessem essas restrições, de forma que não fosse necessário ao usuário estudar alguma outra linguagem para expressar as restrições. Como já vínhamos trabalhando com B, o propósito era expressar as restrições na própria linguagem B. Sendo assim encontramos a ferramenta ProB (Leuschel & Butler, 2003). Os detalhes dessa ferramenta, bem como a comparação com outras, serão apresentados nos próximos capítulos.

Com a utilização do ProB é possível expressar restrições na pré-condição de operações B, fazer animações em cada operação, e essas animações resultam em combinações de dados de teste para cada operação sob teste especificada em uma máquina B. Sendo assim era possível, a partir de operações B sob teste, gerar operações B de teste.

A partir dos dados resultantes das animações do ProB foram implementados, de forma manual, casos de teste no JUnit para testar a aplicação, que foi implementada em Java. Vale ressaltar que a aplicação em Java foi gerada a partir da máquina abstrata B aqui chamada máquina sob teste* (que contém a especificação da aplicação) e o código JUnit foi gerado manualmente a partir da máquina B de teste (que contém as operações de teste). A utilização do JUnit se deu por esta ferramenta já dispor de métodos definidos para comparação de resultados esperados com resultados obtidos da aplicação que está sendo testada e também por ser a ferramenta mais utilizada no teste de sistemas Java.

Os detalhes dessa metodologia poderão ser vistos no Capítulo 3.

*OBS.: Quando se vir o termo “máquina sob teste” favor interpretar como “Máquina do sistema sob teste”.

1.3. Estrutura do documento

Capítulo 2: Fundamentação Teórica: Este capítulo mostra uma revisão teórica sobre os assuntos abordados neste trabalho. Já que o presente trabalho tem o objetivo de gerar casos de teste a partir de especificações B, é mostrada inicialmente uma revisão teórica sobre os testes de *software*, mostrando TBM e técnicas de projeto de casos de teste. Logo após é mostrada uma seção sobre o método e a notação B e uma sobre Lógica Proposicional. Também será apresentado brevemente o JUnit para a geração de testes.

Capítulo 3: Geração de Casos de Teste a partir de Especificações B: Este capítulo mostra o método desenvolvido para a geração de combinações de dados e casos de teste a partir de especificações B. Para ilustração do método, a cada etapa é também mostrado um exemplo.

Capítulo 4: Trabalhos Relacionados: Neste capítulos serão apresentados os trabalhos mais próximos ao presente trabalho, comparando-se os métodos propostos.

Capítulo 5: Apresentação e Análise dos Resultados: Neste capítulo são apresentados os principais resultados encontrados através do método proposto e sua relevância para a área.

Capítulo 5: Conclusões: Neste capítulo é feito um balanço dos resultados alcançados e um resumo sobre os principais trabalhos futuros sugeridos. São mostradas ainda as principais dificuldades encontradas no decorrer do desenvolvimento do presente trabalho.

Capítulo 2 Fundamentação Teórica

Este capítulo mostra uma revisão teórica sobre os assuntos abordados neste trabalho. Será mostrada inicialmente uma revisão teórica sobre os testes de *software* (seção 2.1.), mostrando TBM (Testes Baseados em Modelos) e técnicas de projeto de casos de teste. As especificações que serão utilizadas como entrada para a geração dos testes são escritas em linguagem B, por isso será mostrada uma seção sobre o método e a notação B (seção 2.2.). Por fim será feita uma introdução ao JUnit, que será utilizado para teste da aplicação.

2.1. Testes de *Software*

Teste é uma sub-atividade de engenharia de *software*, com ciclo de vida paralelo ao ciclo de desenvolvimento de *software* e que tem o objetivo de medir e melhorar a qualidade do *software* que está sendo testado (Myers, 2004). Os testes tentam quebrar o funcionamento correto do sistema. Essa quebra de funcionamento deve acontecer no ambiente de desenvolvimento de projeto para que o mínimo possível de falhas chegue ao cliente.

Testes podem ser desempenhados de duas maneiras (Myers, 2004): de forma ad-hoc, onde o testador tenta quebrar o sistema através de testes executados aleatoriamente e sem nenhum planejamento; e de maneira planejada, onde o testador faz um planejamento do que testar. De maneira planejada pode-se analisar os resultados dos testes, bem como analisar porcentuais de cobertura e eficiência. Quando testes são feitos de maneira planejada, eles devem acompanhar todo o ciclo do desenvolvimento do *software*, incluindo atividades de planejamento, projeto, execução e análise dos resultados.

O planejamento dos testes determina em que momentos, no processo de desenvolvimento de *software*, as demais atividades de teste (projeto, execução e análise dos resultados) devem ser desempenhadas (Black, 2003). Deve ser estimada a cobertura aceitável dos testes.

O projeto dos testes faz a especificação dos casos de teste que foram planejados na etapa de planejamento (Black, 2003). Esses casos de teste são passos necessários para que seja verificado um resultado. Dessa forma, um caso de teste precisa conter a ação a ser executada, os dados de entrada e eventualmente os dados de saída, bem como os resultados esperados. Esse comportamento esperado deve ser comparado ao comportamento verificado do sistema, para que se registre um resultado obtido, que deve passar se o resultado obtido for igual ao resultado esperado e falhar caso contrário. Devem

existir casos de teste válidos e inválidos. Os válidos são aqueles que exercitam o *software* em seu fluxo normal de execução e os inválidos executam o *software* em seus fluxos de exceção. Agrupamentos de casos de teste formam as suítes de teste. Os casos de teste projetados devem encontrar a maioria dos defeitos, utilizando o mínimo de esforço. Para isto são usadas técnicas de projeto de casos de teste.

As técnicas de projeto de casos de teste são classificadas em tipo Caixa Preta ou Caixa Branca (Pressman, 2005). O projeto do tipo Caixa Preta enxerga o *software* com uma visão externa, ou seja, não enxerga detalhes de implementação. Testes Caixa Preta são testes de alto nível, pois para geração de casos de testes temos como artefato de entrada a especificação do sistema. De outro lado existem os testes Caixa Branca. Os testes Caixa Branca focam nas estruturas internas do sistema. Sendo assim, como artefatos de entrada para este tipo de projeto de casos de teste, temos o próprio código do sistema. Detalhes sobre testes caixa preta serão vistos mais adiante. Testes caixa branca podem ser vistos em (Myers, 2004).

A análise dos resultados de teste verifica, entre outras coisas, a quantidade de casos de teste que falharam e que passaram e ainda se a cobertura dos casos de teste projetados está aceitável. O critério de aceitação desta cobertura deve ter sido definido no planejamento dos testes (Black, 2003).

Para tratar de todas as atividades de teste, com exceção do planejamento, surgiram os TBM, os quais serão descritos a seguir.

2.1.1. Model Based Testing (TBM – Testes Baseados em Modelos)

TBM surgiram com o intuito de tratar das atividades de teste de maneira sistemática, exceto do planejamento. TBM atualmente são bastante discutidos no meio acadêmico e vêm ganhando interesse na indústria de *software*. Em TBM, os casos de teste são derivados de uma especificação do sistema (Conrad, 2002).

Em TBM, os testes podem ser gerados sistematicamente a partir de especificações. Essas especificações servem como entrada para a definição de casos de teste para testar um determinado sistema. Se a geração dos testes for feita de maneira automática, qualquer modificação na especificação do sistema será refletida rapidamente nos casos de teste. No entanto, existem algumas dificuldades para aplicação de TBM, como mencionado por (Bertolino, 2004): um distanciamento entre o nível de abstração de um modelo e o nível de abstração de casos de testes.

TBM são compostos por diversas atividades. As principais são (Lindholm, 2006):

- **Construção do modelo**: Para a definição de TBM, diversos tipos de modelos têm sido utilizados. Por exemplo, diagramas UML com OCL, como nos trabalhos de (Barbosa, 2005) (Briand & Labiche, 2001). No presente trabalho são utilizadas especificações B, como nos trabalhos de

(Legeard, Peureux, & Utting, 2002) (Bernard, Legeard, Luck, & Peureux, 2004) (Ambert, et al., 2002). O uso de especificações formais leva a um melhor entendimento do sistema, além de propiciar meios automatizáveis para a identificação de especificações incompletas ou inconsistentes, fazendo com que as demais fases de um TBM partam de uma especificação menos propícia a erros. Já que o uso de especificações formais tende a diminuir ambigüidades que a linguagem natural poderia gerar, a geração automática de testes é privilegiada. A linguagem de especificação formal B será a utilizada no presente trabalho e pode ser vista em detalhes na seção 2.2.

- **Definição do critério de derivação de casos de teste a partir do modelo e técnica de projeto de casos de teste**. Deve-se primeiramente definir a técnica no projeto dos casos de teste. Pode-se utilizar técnicas de casos de teste do tipo Caixa Branca (aquelas onde o código do sistema é visível ao testador) ou técnicas de casos de teste do tipo Caixa Preta (onde apenas a especificação é considerada – o código não é visível).

  O critério de derivação define a cobertura dos testes. No contexto de (Lindholm, 2006), por exemplo, que gera casos de teste a partir de máquinas de estados finitos (FSA – *Finite State Automaton*), são considerados dois tipos de cobertura. A primeira é quando a suite de teste alcança cada estado do modelo pelo menos uma vez. A segunda é quando a suite de teste alcança cada transição de estados pelo menos uma vez. Evidentemente, a segunda é mais rigorosa, gerando um maior número de testes, dado que todas as combinações de estados dois a dois são alcançadas pelo menos uma vez. No entanto, em muitos casos, mesmo o critério de cobertura por estado pode ser inviável, dependendo do tipo de modelo com que se trabalha. Outro critério de cobertura é o baseado em análise de partições. Para isso utiliza-se a técnica de particionamento de equivalência (seção 2.1.2.1). Este critério evita a construção total de uma FSA, diminuindo a possibilidade de explosão de casos de teste gerados quando baseados em estados.

- **Geração dos casos de teste e suite de teste**: Os critérios de derivação dos testes definidos no item anterior são aplicados. A partir desta interação, manual ou automática, os casos de teste são gerados e agrupados em suites de teste.

- **Adaptação dos casos de teste**: Os casos de teste gerados no item anterior podem, por partirem de um modelo abstrato, ter um nível de abstração não aplicável aos testes do sistema. Caso isso ocorra, é necessário fazer uma adaptação dessa suíte para o nível concreto do sistema a ser testado.

- **Execução dos casos de teste**: Os casos de teste são finalmente executados. Nesta etapa podem acontecer duas situações diferentes quanto aos resultados: pode-se ter resultados de forma automática ou o resultado pode ser observado pelo próprio testador. Resultados automáticos são

mais custosos de construção, mas em compensação são mais facilmente reusados e facilitam a geração de relatórios indicando o sucesso ou a falha do sistema (uma vez que estes relatórios são feitos automaticamente). Os resultados podem ser analisados ao final da execução de todos os casos de teste. Esta análise serve como base para verificar se os testes estão sendo eficientes ou não. Pode-se então optar por definir um novo modelo e, a partir deste, gerar casos de teste, modificar o modelo ou parar os testes.

As técnicas de projeto de casos de teste serão apresentadas na próxima seção, bem como o método e a notação B na seção 2.2. e uma visão geral da lógica de predicados na seção 2.3. O método proposto neste trabalho passa por todas as atividades citadas menos a de construção do modelo.

2.1.2.Técnica de Projeto de Caso de Teste do Tipo Caixa Preta (Black Box)

Também conhecido como teste funcional, o teste caixa preta foca nas funcionalidades do sistema. Um testador fornece as entradas a um componente ou ao sistema e examina uma saída. Os defeitos investigados pelos testes Caixa Preta estão em uma classe de defeitos que os testes Caixa Branca podem eventualmente encontrar, mas que não os têm como foco. Esses defeitos, de acordo com (Pressman, 2005) são:

- Funções incorretas ou ausentes;
- Defeitos de interface;
- Defeitos de desempenho;
- Defeitos de inicialização e término.

Neste trabalho focaremos na identificação de funções incorretas. Trataremos os demais tipos de defeitos como instâncias de funções incorretas: se uma função é ausente, ou tem defeitos de interface entre componentes ou tem defeitos de inicialização ou término, as saídas encontradas serão diferentes das saídas esperadas, caracterizando uma função incorreta. Defeitos de desempenho não foram considerados neste trabalho, já que variáveis como tempo de resposta não estão sendo medidas.

Existem algumas técnicas de teste caixa preta, entre elas o particionamento de equivalência e a análise do valor limite, bem como os pares ortogonais, que são o foco do nosso trabalho. O particionamento de equivalência e análise do valor limite e os pares ortogonais serão apresentados a seguir.

2.1.2.1. Particionamento de Equivalência

O particionamento de equivalência considera que conjuntos de dados podem ser agrupados em partições e que, utilizando-se um elemento de cada partição para teste, esse elemento identifica a mesma falha que qualquer outro da mesma partição. Dessa forma, um elemento de uma partição é o representante daquela partição.

A técnica do particionamento de equivalência é baseada na premissa de que entradas e saídas de um componente ou de um sistema completo podem ser separadas em partições (Ross, 1998). O particionamento de equivalência resulta num particionamento do domínio de entrada do *software* a ser testado. As saídas também podem ser particionadas (Burstein, 2003). Para o melhor entendimento dessa técnica será explicado a seguir o que é uma partição ou classe de equivalência.

> *"Partição é um conceito matemático básico, geralmente associado com teoria dos conjuntos. Formalmente, uma partição de um conjunto S é a divisão desse conjunto em subconjuntos G1, G2, ..., Gn."* (Tian, 2005)

O particionamento deve satisfazer às seguintes propriedades (Tian, 2005):

- Os subconjuntos resultantes são mutuamente exclusivos, ou seja, se um elemento existe em um subconjunto ele não pode existir em nenhum outro;
- Os subconjuntos são coletivamente exaustivos, ou seja, eles coletivamente cobrem ou incluem todos os elementos do conjunto original.

Os subconjuntos resultantes do particionamento são chamados partições ou classes de equivalência.

A técnica de particionamento de equivalência tem as seguintes vantagens (Burstein, 2003):

- Elimina a necessidade de testes exaustivos;
- Proporciona a seleção de um subconjunto de entradas com uma maior probabilidade de encontrar falhas a partir das classes de equivalência do que testes ad-hoc, que serão mostradas mais à frente;
- Proporciona a cobertura de um vasto domínio de entradas/ saídas com um pequeno subconjunto selecionado a partir de classes de equivalência.

O critério de cobertura desse método, segundo (Pressman, 2005), diz que cada partição deve ser considerada ao menos uma vez no teste. Já (Ross, 1998) afirma que deve ser considerada cada partição exatamente uma vez no teste. Ora, se um valor de uma partição representa toda a partição, ou seja, se ele revela uma falha no teste, todos os demais valores dessa partição revelariam a mesma falha, não é necessário considerar mais de um valor de teste para esta partição. Sendo assim a idéia de (Ross, 1998) é a mais aplicável e será a utilizada neste trabalho.

Somente serão considerados mais de um dado para uma mesma partição quando utilizada a técnica da análise do valor limite, que será explicado mais à frente. Dessa forma, o seu critério de cobertura considera um dado por partição, evitando a explosão da quantidade de casos de teste gerados quando se utiliza cobertura de transições de estados baseados na construção de uma FSA total.

Os passos para o particionamento de equivalência são:

1. Decompor o programa em funções;
2. Identificar as variáveis que determinam o comportamento de cada função;
3. Identificar as restrições sobre cada variável;
4. Particionar cada restrição em classes de equivalência (válidas e inválidas);
5. Selecionar dados de teste cobrindo as classes válidas e inválidas de todas as restrições de todas as variáveis;

A seguir será apresentado como podemos encontrar as classes de equivalência.

Identificando as Classes de Equivalência e Dados destas Classes

As classes de equivalência são identificadas para cada restrição de cada variável. Autores diferentes ditam regras diferentes para a classificação de equivalência sobre as restrições de cada variável. Segundo (Tian, 2005) existem as seguintes classes de equivalência:

1. **Um conjunto**: relação de ser membro de um conjunto ou não, como em $x \in S$. As partições são identificadas como os elementos que pertençam a um conjunto e os elementos que não pertençam;
2. **Condições booleanas**: condições vistas através da lógica de predicados. As partições de equivalência são identificadas de forma a tornar os predicados verdadeiros ou falsos;
3. **Condições numéricas**: com variáveis que usam operadores de comparação numérica como =, ≠, <, >, ≤ e =. Em (Tian, 2005) mostra-se somente que são condições do tipo $x < 0$ onde teríamos duas classes de equivalência, uma que é $x < 0$ e a outra $x = 0$.

Outros autores como (Myers, 2004) (Burstein, 2003) (Pressman, 2005) classificam as classes de equivalência assim:

1. **Intervalo de valores**: selecione uma classe de equivalência válida que cubra o intervalo, bem como duas classes de equivalência inválidas, uma acima e uma abaixo do intervalo;

2. **Um valor pertencente a um conjunto**: selecione uma classe de equivalência válida que contém todos os membros do conjunto e uma classe inválida com todos os valores não pertencentes ao conjunto.

3. **Condição "deve ser" (must be)**: aplicado a algum predicado. Dessa forma, selecione uma classe de equivalência válida que satisfaça a condição (predicado) e uma classe inválida que não satisfaça a condição (predicado).

4. **Uma quantidade de valores**: selecione uma classe de equivalência válida que contém a quantidade válida de dados e duas classes inválidas (uma com nenhum valor e outra com mais valores que a quantidade correta). Como exemplo, se temos que uma pessoa só pode comprar um total de cinco mercadorias em uma loja, os possíveis valores para teste seriam cinco (valor válido) e zero e seis (valores inválidos).

Para (Myers, 2004) (Burstein, 2003) (Pressman, 2005) as quatro classificações de equivalência englobam as três feitas por (Tian, 2005), o qual, na condição numérica, manipula os operadores =, ≠, <, >, ≤ e =. Entendemos que esses operadores indicam classes de equivalência diferentes (<, >, ≤ e = simbolizam uma condição numérica, mas que pode ser refinada a um valor pertencente a um intervalo, já que traz a idéia de intervalo). Já para os operadores = e ≠, entendemos que são mais propícios à classe de uma quantidade de valores, se tivermos valores inteiros. Caso não tenhamos inteiros, podemos classificar como uma condição booleana. Por exemplo, x=5 seria classificado como uma quantidade de valores e x="Maria" como uma condição booleana. Dessa forma, partiremos da classificação de (Myers, 2004) (Burstein, 2003) (Pressman, 2005) neste trabalho.

Neste trabalho fizemos algumas manipulações de conjuntos, com cláusulas do tipo $A \subseteq B$. Nessas cláusulas temos que A pertence ao conjunto dos subconjuntos de B. Dessa forma podemos tratar a quantidade de elementos do conjunto dos subconjuntos de B como um intervalo que varia da menor cardinalidade dos subconjuntos de B até a maior cardinalidade (que na verdade é igual à de B). Por exemplo, se B = {1, 2, 3}, A poderia tomar um desses valores: A={1}, A={2}, A={3}, A={1, 2}, A={1,3}, A={2,3}, A={1,2,3}. Então quaisquer desses conjuntos poderiam ser escolhidos para testar a cláusula $A \subseteq B$.

Como dados inválidos teremos um valor não pertencente ao conjunto das partes de B (que se encaixa na classificação de um valor pertencente a um conjunto). Esse tipo de restrição, que engloba características de um intervalo e um valor pertencente a um conjunto receberá o nome de **Um Conjunto**.

Neste trabalho serão utilizadas as classificações de **um intervalo, uma condição booleana, um valor pertencente a um conjunto e um conjunto**. Entendemos que uma quantidade de valores pode ser manipulada através da classificação de um intervalo.

Após a identificação das classes de equivalência, os dados de teste devem ser selecionados. Dados devem ser identificados para cada restrição. Para isto, as restrições impostas para uma mesma variável, se existir mais de uma restrição, devem ser consideradas. Com os dados de teste identificados, esses dados devem ser combinados.

Um exemplo para ilustrar a identificação de classes de equivalência e escolha dos valores pode ser vista a seguir:

Um endereço de e-mail deve conter 5 caracteres seguidos de um arroba(@) e depois mais 5 caracteres.

Teremos neste caso uma classificação de equivalência para os 5 primeiros caracteres, uma para o arroba e uma para os 5 últimos caracteres.

- Para os 5 primeiros e 5 últimos temos uma classificação para uma quantidade de valores, que tem como classe válida uma sequência com 5 caracteres e como classes inválidas menor que 5 valores e mais que 5 valores;
- Para o arroba teremos uma condição booleana que tem como classe válida ser igual ao arroba e como classe inválida ser diferente.

Os dados de teste para essas classes de equivalência são os seguintes:

- Para os 5 primeiros e 5 últimos: classe válida = abcde, classe inválida = abcd e classe inválida = abcdef (mais de 5 caracteres). Os dados foram escolhidos aleatoriamente. Somente foi considerada a quantidade de caracteres;
- Para o arroba: classe válida = @, classe inválida = & (escolhido aleatoriamente).

Algumas combinações dos dados de teste podem ser vistas a seguir:

abcde@abcde (combinação válida)

abcdef@abcde (combinação inválida)

abcd@abcde (combinação inválida)

abcde@abcd (combinação inválida)

...

Neste trabalho iremos considerar a existência de somente um dado inválido por vez. Isto servirá para auxiliar a tarefa de depuração. Isso porque, caso primeiramente o *software* tenha sido testado com todos os dados válidos e o teste tenha passado e logo após seja inserido uma combinação de dados inválida e uma falha seja encontrada, sabemos ao certo qual dado provocou tal falha.

Particionamento de equivalência é uma abordagem apropriada quando a entrada é definida em termos de valores discretos. Elas podem ser melhoradas quando usadas em conjunto com testes de fronteiras (técnica de análise do valor limite), quando se tem a idéia de intervalo para uma variável.

2.1.2.2. Análise do Valor Limite

É uma variação da técnica de particionamento de equivalência que foca nos limites (quando houver) do domínio das variáveis ou condições sobre uma variável (Burstein, 2003). A análise do valor limite tem como diretriz que alguns valores dentro de uma partição têm maior probabilidade de revelar falhas em testes que outros. Esta técnica trabalha com os limites de variáveis que representam intervalos. Note que neste momento, a idéia de classes de equivalência é deixada de lado. A nomenclatura classes de equivalência e partições de equivalência ainda será utilizada, mas com um sentido diferente (somente para o agrupamento de valores – a escolha dos dados será feita de maneira diferente).

Para o uso dessa técnica, é primeiramente feito o particionamento de equivalência nas entradas do sistema (Burstein, 2003) (Ross, 1998) (Tian, 2005), sem a identificação dos dados de teste. Os autores (Myers, 2004) (Pressman, 2005) também falam em particionamento das saídas. Com a obtenção das classes de equivalência são definidos os valores limites, ou seja, as classes de equivalência são definidas, mas a escolha dos valores para testes é feita de maneira diferente, de acordo com (Burstein, 2003) (Myers, 2004):

- **Intervalo de valores**: é necessário selecionar dois valores válidos: o maior valor do intervalo (limite superior); o menor valor do intervalo (limite inferior). (Tian, 2005) fala ainda em tomar mais um valor válido, um valor intermediário, entre o limite superior e inferior. É preciso também selecionar dois valores inválidos: um valor maior que o maior valor válido (logo acima); um valor menor que o menor valor válido (logo abaixo);
- **Uma quantidade máxima de valores**: os valores válidos são o número mínimo válido de valores e máximo de valores. Os inválidos são logo abaixo do mínimo e logo acima do máximo. Embora (Myers, 2004) chame essa classificação de Uma Quantidade de valores, aqui será chamada de uma quantidade **máxima** de valores, para que não seja confundida com a classificação do particionamento de equivalência. Lá se tem uma quantidade exata de valores e aqui se tem uma quantidade máxima, por isso uma quantidade de valores do particionamento de equivalência não tem a idéia de limites;
- **Um conjunto ordenado**: os valores a serem focados nos testes são o primeiro elemento e o último elemento do conjunto ordenado. As referências não trazem informações sobre valores inválidos para teste, mas podemos propor que sejam utilizados elementos do conjunto fora da

ordem para valores inválidos. Note que esta classificação não era definida no particionamento de equivalência e que somente surgiu com a análise do valor limite.

Perceba que “um intervalo” pode ser definido de duas formas, tanto a do particionamento de equivalência quanto na análise do valor limite (as classes de equivalência identificadas são as mesmas, o que difere são os valores escolhidos para teste).

Neste trabalho estamos considerando somente a classificação de um intervalo, pois entendemos que todas as classificações mostradas anteriormente podem ser trabalhadas como um intervalo.

Vale salientar que tanto o particionamento de equivalência quanto a análise do valor limite não tratam de dependências entre as variáveis. Neste trabalho, adicionalmente, foram tratadas as dependências (isto será mostrado no Capítulo 3).

2.1.2.3.Teste Funcional Sistemático

O teste funcional sistemático (Maldonado, 2007) combina as técnicas de particionamento de equivalência e a análise do valor limite. Ele age nas entradas e saídas de especificações, inicialmente aplicando o particionamento de equivalência e posteriormente a análise do valor limite. O seu critério de cobertura requer ao menos dois dados de teste por partição.

O teste funcional sistemático é aplicado sobre algumas condições. As principais delas são:

- Valores numéricos: para valores discretos deve-se testar todos os valores e para intervalo de valores devem ser testadas as bordas e um valor intermediário;
- Tipos de valores diferentes e casos especiais: casos especiais são campos vazios e o valor zero;
- Valores ilegais: o mesmo que dados inválidos;
- Intervalos variáveis: considera relacionamento entre variáveis de intervalo. Isso acontece quando o valor de uma variável de intervalo varia de acordo com restrições impostas por outra variável.

(Maldonado, 2007) é o único autor estudado que menciona o relacionamento entre variáveis, embora também não mostre como os problemas de relacionamentos entre variáveis são resolvidos.

2.1.2.4.Pares Ortogonais

Quando se fala em combinação de dados de teste podemos fazê-las de duas formas: um dado de cada partição deve aparecer somente uma vez no teste ou cada dado de cada partição deve ser combinado com todos os dados das demais partições. Nestas duas soluções temos algumas desvantagens. Para a primeira temos que algumas falhas poderiam ser descobertas em combinações que não foram utilizadas

em teste. Para a segunda temos uma grande quantidade de combinações. Sendo assim, uma solução intermediária são os Pares Ortogonais.

Pares Ortogonais é uma técnica de teste Caixa Preta inspirada na matemática (Craig & Jaskiel, 2002). Os pares ortogonais nada falam de como escolher os dados de teste, mas sim como combinar os dados de teste escolhidos. Dessa forma é necessário aplicar outra técnica de teste para escolha desses dados.

A combinação desses dados será mostrada em uma tabela (Figura 2.1) onde a primeira coluna indica a quantidade de combinações de dados de teste existentes. A primeira linha indica o número de variáveis, que são quatro no exemplo (1, 2, 3 e 4 – que podem ser observadas na primeira linha). Na tabela pode-se perceber que cada variável tem três valores de teste (1, 2 e 3 – valores encontradas abaixo de cada variável). As linhas e colunas internas indicam as combinações de dados necessárias. A Figura 2.1 a seguir mostra um exemplo dessa técnica.

	1	**2**	**3**	**4**
1	1	1	1	1
2	1	2	2	2
3	1	3	3	3
4	2	1	2	3
5	2	2	3	1
6	2	3	1	2
7	3	1	3	2
8	3	2	1	3
9	3	3	2	1

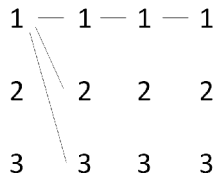
Figura 2.1 - Tabela com os pares ortogonais

Para este exemplo temos 9 combinações. A quantidade de combinações varia de acordo com a quantidade de variáveis e a quantidade de valores para cada variável. Pode-se perceber aqui que se combinássemos todos os elementos de todas as variáveis, teríamos um total de 81 combinações (3^4, onde 3 é a quantidade de valores para cada variável e 4 é a quantidade de variáveis). Temos assim uma diminuição de combinações bastante significativas com pares ortogonais.

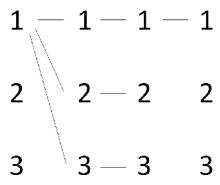
Esta técnica garante a combinação de todos os valores de variáveis se analisadas duas a duas, ou seja, sempre que pegarmos duas variáveis cada combinação de seus valores estará presente em algum caso de teste. Por exemplo, o valor 2 da variável 3 aparece combinado com o 1, 2 e 3 da variável 1 nos testes 2, 4 e 9, respectivamente; com 1, 2 e 3 da variável 2 nos testes 4, 2 e 9 respectivamente e com 1, 2 e 3 da variável 4 nos testes 9, 2 e 4 respectivamente. Esta técnica não garante que todas as combinações de dados de teste escolhidos sejam originadas, ou seja, falhas que eventualmente sejam descobertas por combinações específicas de variáveis podem não ser encontradas com os pares ortogonais.

Para a combinação mostrada na tabela anterior é utilizada a metodologia a seguir. Vamos ilustrar isso com uma matriz onde cada coluna contém os elementos de uma variável. Cada combinação será ilustrada pela ligação dos elementos de cada variável através de linhas.

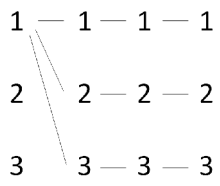
Fixamos então o primeiro elemento da primeira variável e ele é então combinado aos demais da segunda variável:



Passa-se então para os elementos da segunda variável. Como o “1” já está combinado ao “1” da terceira variável, deve-se agora combinar o “2” da segunda variável com o elemento da terceira que ainda não foi combinado. Combinado, deve-se então combinar o próximo elemento da segunda variável com o próximo elemento da terceira variável, ou seja, o “3” com o “3”, respectivamente:



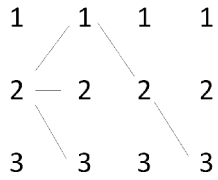
Findado isso, pulamos para a próxima variável, e as seguintes combinações são geradas:



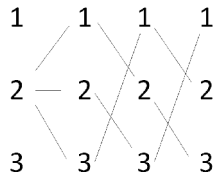
O segundo elemento da primeira variável deve agora ser fixado e ele deve ser combinado a todos da segunda variável.

Para fazer a combinação com os demais elementos das demais variáveis, precisa-se verificar qual o elemento que cada elemento de cada variável estava fixado na combinação anterior. Por exemplo, o elemento “1” da segunda variável, na combinação anterior, estava combinado ao elemento “1” da terceira variável. Assim, na combinação presente, o “1” da segunda variável será combinado ao “2” da

terceira variável. Neste raciocínio, o elemento “2” da terceira variável irá se combinar com o elemento “3” da quarta variável, resultando assim na combinação 2, 1, 2, 3, que é a combinação da quarta linha da Figura 2.1.



Todas as combinações para o segundo elemento da primeira variável podem ser vistas a seguir:



Isto também deve ser procedido para combinar o terceiro elemento da primeira variável. Em resumo, esta técnica dita que para a primeira variável fixa-se cada elemento um a um e este é combinado a todos os demais da segunda variável. A partir da segunda variável em diante, pega-se cada elemento e combina-se este ao elemento da próxima variável, elemento este que é o sucessor do combinado anteriormente.

Neste trabalho, a técnica dos pares ortogonais é uma alternativa dada pelo método proposto para combinação dos dados de teste. Sendo assim, as técnicas de particionamento de equivalência e análise do valor limite são utilizadas na identificação das classes de equivalência e escolha dos dados de teste e os pares ortogonais podem ser utilizados na combinação dos dados.

2.2. O Método e a Notação B

No contexto deste trabalho, a especificação alvo para a derivação dos casos de teste será descrita em linguagem B. Por isso, o método e a notação B serão apresentados a seguir.

O método B (Abrial, 1996) é um método de desenvolvimento formal de *software* que apóia o desenvolvimento de código a partir de especificações formais descritas na notação específica do método: a notação B. A notação e o método foram derivados de pesquisas anteriores na área, como as linguagens Z (Spivey, 1992) e VDM (Sheppard, 1995) e o cálculo de refinamentos de Hoare (Hoare, 1969).

Especificações B se baseiam na lógica de predicados, na teoria dos conjuntos e na aritmética inteira, complementados pelo chamado cálculo de substituições generalizadas, que permite que especificações tenham uma aparência de programação imperativa, facilitando o seu uso. Isto é visto no exemplo ainda neste capítulo.

Uma especificação em B é estruturada em módulos que são classificados de acordo com seus níveis de abstração. O nível mais alto é a máquina abstrata (MACHINE, que tem como dados conjuntos, relações, funções e predicados). As máquinas podem ser refinadas em refinamentos (REFINEMENT), opcionalmente, e então em implementações (IMPLEMENTATION). É a partir das implementações que é gerado o código de uma aplicação.

Após a especificação das máquinas é necessário provar sua consistência. Em seguida, cada passo do refinamento deve ser provado correto quanto à sua máquina ou refinamento precedente. As implementações são verificadas também quanto à conformidade com um gerador de código para uma linguagem particular e, se for o caso, o código do programa é gerado. Assumindo a correção do gerador de código, o código gerado pode ser certificado como satisfazendo as propriedades da especificação abstrata original. Pode-se, alternativamente, implementar código a partir da máquina abstrata, não fazendo refinamentos.

Uma máquina B contém duas partes principais: uma definição do domínio de estados e as operações disponíveis. As máquinas podem ainda conter cláusulas auxiliares (parâmetros, constantes e asserções), mas elas são menos importantes no contexto deste trabalho. A discussão do presente trabalho será restrita a um núcleo de cláusulas de especificação de máquinas.

A especificação dos componentes de estado de uma máquina é encontrada nas cláusulas VARIABLES (variáveis) e INVARIANT(invariante). Estas cláusulas enumeram os componentes de estado (variáveis) e definem restrições sobre os valores (tipos e outras propriedades) que eles podem assumir (invariante). O invariante é um predicado que deve ser verdade antes e ao final da execução de cada operação. Basicamente, se V denota as variáveis de estado de uma máquina, o invariante é um predicado sobre V. Outra cláusula relevante é a INITIALISATION. Esta atribui os valores iniciais às variáveis. Esses valores iniciais devem tornar o invariante válido.

Adicionalmente temos a cláusula OPERATIONS, onde são declaradas as operações. As operações manipulam os estados das variáveis. Todas as verificações realizadas em um processo de desenvolvimento têm a intenção de verificar que nenhum estado inválido será alcançado enquanto as operações de uma máquina forem usadas como especificadas. Para a especificação da inicialização, bem como das operações, B oferece um conjunto de substituições: construções com estilo de linguagens imperativas e regras de tradução que definem sua semântica como sendo o efeito que elas têm sobre os valores das variáveis às quais elas são aplicadas.

Uma substituição bastante importante para este trabalho é a PRE-THEN-END, que pode ser usada para especificar alguma pré-condição que a definição da operação assume para funcionar adequadamente, ou seja, especifica os valores válidos como entrada da operação. É a partir dessa informação, juntamente com a descrição das variáveis de estado contida no invariante da máquina, que serão definidos os dados de testes para cada operação especificada.

A seguir, através de um exemplo extraído de (Gomes, 2005), serão apresentadas as cláusulas de B descritas anteriormente. Para maior entendimento do exemplo, a seguir são mostrados alguns símbolos de B:

Tabela 2.1 - Símbolos de B

|	Tal que
:	Pertence
/:	Não pertence
&	And
= >	Implica
/=	Diferente
||	Paralelismo (as partes que estão separadas por este símbolo ocorrem em paralelo)
:=	Recebe
!	Existe

O exemplo corresponde a uma aplicação de bilhetagem eletrônica oferecida a usuários de *Transport*e coletivo através de um cartão inteligente. Para melhor compreensão do exemplo reproduzimos a seguir a especificação informal que serviu de base para o desenvolvimento da especificação abstrata B:

> *O valor correspondente à passagem é dependente da categoria do cartão. A categoria é armazenada no cartão, bem como o total em créditos, a hora e a data em que os dados foram manipulados. A categoria pode ser inteira, estudante ou gratuita. O usuário poderá inserir créditos no cartão, desde que o cartão seja das categorias inteira ou estudante. A diferenciação do valor da passagem entre inteira e estudante é feita no ato da compra do crédito, pois o cartão armazena somente a quantidade inteira de créditos e não o valor em dinheiro. O cartão inteiro exige um valor em dinheiro para a compra de créditos maior que o valor em dinheiro para a compra do crédito do cartão estudante. Caso o cartão seja da categoria gratuita, o usuário poderá passar o cartão na leitora de cartões e nenhum crédito será descontado. Após a operação de inserção do cartão na leitora de cartões, identificação da categoria do cartão, débito de 1 crédito no cartão (para os cartões inteiro e estudante), atualização da data e hora, o usuário tem o acesso ao ônibus liberado.*

A máquina abstrata desenvolvida para esta especificação é a máquina *Transport*. Esta máquina está associada à lógica de negócios da especificação e define as variáveis de estado e as operações que modelam as funcionalidades da aplicação a serem implementadas no cartão. Parte dessa máquina pode ser vista na Tabela 2.2:

Tabela 2.2 – Máquina Transport

```
1  MACHINE Transport
2  SEES Transport_Constants
3  DEFINITIONS
4       DATE ==
5            day, month, year | day : 1..31 & month : 1..12 & year : NAT1 &
6            (day = 31 => month : 1, 3, 5, 7, 8, 10, 12) &
7            (month = 2 => (day <= 28 or (day = 29 &
8                 ((year mod 400 =0) or (year mod 4 = 0 &
9                 year mod 100 /= 0)))))
10 VARIABLES
11      balance, card_type, time, date
12 INVARIANT
13      balance : NAT &
14      time : (0..23) * (0..59) &
15      date : DATE &
16      card_type : CARD_TYPES &
17      (card_type = gratutious_card => balance = 0)
18 INITIALISATION
19      balance := 0 ||
20      time := 0 |-> 0 ||
21      date := 1 |-> 1 |-> 1 ||
22      card_type :: CARD_TYPES
23 OPERATIONS
24      addCredit (cr) =
25           PRE
26                cr : NAT1 &
27                card_type /= gratuitous_card &
28                balance + cr <= MAXINT
29           THEN
30                balance := balance + cr
31           END
```

```
32
33        END
```

Tabela 2.3 – Máquina Transport_Constants

```
34  MACHINE Transport_Constants
35  SETS CARD_TYPES = {entire_card, student_card, gratutious_card}
```

Transport é declarada no início da máquina pela cláusula MACHINE (linha 1). Neste caso, a máquina não contém parâmetros.

Transport aponta para outra máquina, a *Transport_Constants (*Tabela 2.3*).* Isto acontece através da cláusula SEES (linha 2). Esta cláusula faz com que *Transport* possa acessar o conteúdo de *Transport_Constants* mas não alterá-lo.

Neste caso, *Transport_Constants* (linha 32) traz o conjunto *CARD_TYPES* (linha 33) que é o conjunto que define os tipos possíveis de cartões que podem ser: cartão inteiro (*entire_card*), estudante (student_card) ou gratuito (*gratutious_card*). A introdução do conjunto é feita na cláusula SETS (linha 33).

Continuando na máquina *Transport*, temos a cláusula DEFINITIONS (linha 3). Esta cláusula relaciona um nome a uma expressão. Dessa forma, toda vez que a palavra DATE (linha 4) for usada na máquina, ela será substituída pela expressão correspondente.

A cláusula VARIABLES (linha 10) define o conjunto de variáveis globais (variáveis de estado) contidas na máquina. Estas variáveis guardam respectivamente o saldo do cartão (*balance*), o tipo do cartão (*card_type*), uma hora (*time*) e uma data (*date*) a ser usada para registro da última utilização do cartão.

Logo após é feita a tipagem das variáveis e também declaradas algumas restrições que precisam ser satisfeitas. Isto é apresentado na cláusula INVARIANT (linha 12).

Neste exemplo temos algumas cláusulas definidas. Embora tenhamos definições de tipos para as variáveis, nenhuma delas tem uma tipagem primitiva. Isto porque cláusulas como $balance : NAT$ tem além de tipagem, alguma restrição, ou seja, NAT é um tipo derivado. Isso porque a tipagem primitiva para esta variável seria $balance : INT$, que é o único tipo primitivo de B. Os demais tipos, como o tipo NAT já são restrições sobre inteiros, ou seja, NAT já restringe o domínio dos inteiros para valores maiores ou iguais a zero. Neste trabalho, cláusulas do tipo $x : S$ serão consideradas como cláusulas de tipagem, pois o tipo de balance pode ser inferido a partir dele, embora saibamos que esta não é uma tipagem primitiva.

Então, o invariante indica que o saldo do cartão deve ser um número natural, que a hora varia de 0 a 23 e os minutos de 0 a 59, a data é definida na cláusula DEFINITIONS (a data recebe um valor definido

em DEFINITIONS) e o tipo do cartão é definido pelo conjunto *CARD_TYPES*. O invariante impõe uma restrição adicional: caso o cartão seja do tipo gratuito o saldo deve ser igual a zero (*linha 17*).

Na cláusula INITIALISATION (linha 18) as variáveis recebem seus valores iniciais. O saldo do cartão sempre começa com zero, a hora com zero hora e zero minuto, a data como (1,1,1 – correspondente a 1 de janeiro do ano 0001) e o tipo do cartão pode ser inicialmente de qualquer tipo (dentre os tipos definidos).

OPERATIONS (linha 23) traz as operações da máquina. As operações têm um nome e podem ou não ter valores de entrada e saída. A *Transport* contém cinco operações de atualização (*addCredit, debitCredit, changeCardType, setTime, setDate*) e quatro operações de consulta (*getCardType, getBalance, getTime, getDate*). Aqui somente a operação *addCredit* foi mostrada. A operação *addCredit* (linha 24) tem um valor de entrada *(cr) e* não tem valores de saída. *addCredit* armazena uma quantidade de créditos no cartão igual a *cr*. A sua pré-condição diz que *card_type* não pode ser gratuito (*card type / = gratuitous card*) e que o valor a ser adicionado no cartão *(cr)* tem que ser maior ou igual a 1 e também que somado ao valor já existente no cartão *(balance)* não pode extrapolar o limite máximo da linguagem *(MAXINT)* para que a operação de adicionar possa ser executada com segurança. Todas as operações podem ser vistas em detalhes no Apêndice.

Após a especificação da máquina abstrata podem-se fazer refinamentos (REFINAMENT) e logo após implementações (IMPLEMENTATIONS). Estes não serão apresentados aqui, pois neste trabalho somente geraremos testes a partir das máquinas abstratas (MACHINES). Detalhes de refinamentos e implementações podem ser vistos em (Abrial, 1996).

2.2.1. Tipagem das variáveis abstratas na linguagem B

A partir de uma especificação B, o método proposto gera casos de teste em linguagem B. Para esta geração foi necessário estudar a sintaxe de cláusulas que restringem os valores das variáveis da máquina B sob teste para que, a partir dessas cláusulas, fossem definidos os critérios de escolha dos dados de teste. Dentre estas cláusulas estão as cláusulas de tipagem, que terão sua sintaxe mostrada nesta seção.

A sintaxe para variáveis abstratas encontradas no manual de B (Clearsy Systems Engineering) é a seguinte:

Convenção em *Typing_abstract_data*

- Símbolos terminais estão entre aspas duplas;
- Um elemento não terminal é mostrado em itálico;

- x ::= y representa a produção da gramática sendo que x é um não terminal e y é uma sequência de itens desta gramática concatenados;
- x|y representa o item x ou o item y;
- x^{+y} significa n instâncias de x, separadas por y, onde n maior ou igual a 1;

Neste trabalho será considerado que todas as variáveis e constantes são tipadas desta forma:

Typing_abstract_data ::=
Ident "∈" *Expression*$^{+"\times"}$
| *Ident* "⊂" *Expression*
| *Ident* "⊆" *Expression*
| *Ident*$^{+","}$ "=" *Expression*$^{+","}$

As notações de *Typing_abstract_data* têm a seguinte convenção:

Significado dos termos:

- Símbolos terminais: são palavras reservadas da linguagem. Os símbolos terminais de *Typing_abstract_data* são ∈, ⊂, ⊆, ×, e =,
- Identificador (Ident): é uma sequência de letras, números ou o caracter subescrito "_". O primeiro caracter deve ser uma letra.

Neste trabalho, a partir das construções de tipagem da linguagem B, fizemos uma classificação quanto às classes de equivalência utilizadas. Aqui foram identificadas as classes de um valor pertencente a um conjunto, um conjunto e um valor booleano. Algumas particularidades quanto a classificação de equivalência serão apresentadas no próximo capítulo.

A semântica de *Typing_abstract_data* é a seguinte (sempre a variável abstrata a ser tipada está do lado esquerdo das expressões. O lado direito são expressões previamente tipadas):

- Para a primeira expressão (*Ident* "∈" *Expression*$^{+"\times"}$) temos semanticamente que *Ident* é uma variável **pertencente a um conjunto** (um elemento de um conjunto), conjunto este que é mostrado em *Expression* ou *Expression* $^{\times}$ *Expression* (produto cartesiano de conjuntos);
- Para as expressões *Ident* "⊂" *Expression* e *Ident* "⊆" *Expression* temos que *Ident* é uma variável que é **um conjunto**, já que ⊂ e ⊆ são operações sobre conjuntos. *Expression* é outro conjunto

(previamente definido). Dessa forma temos do lado esquerdo da expressão um conjunto que está sendo tipado e do lado direito um conjunto previamente tipado.

- Para a última expressão *Ident*$^{+", "}$ "=" *Expression*$^{+", "}$ temos do lado esquerdo uma lista de variáveis e do lado direito uma lista de valores previamente tipados. Neste caso, cada variável do lado esquerdo leva o tipo do valor correspondente do lado direito. Existem outras expressões com = que não são cláusulas de tipagem. Dessa forma, as cláusulas que são de tipagem são aquelas em que a variável do lado esquerdo ainda não foi tipada (está sendo tipada ali). Aqui temos a classificação de **um valor booleano**.

A tipagem definida foi para variáveis abstratas, ou seja, variáveis definidas nas máquinas abstratas de B (MACHINE), onde ainda não se tem certas informações de como os dados serão representados na linguagem ao qual o código deve ser gerado.

Para a geração de casos de teste, pode ser necessária ainda a manipulação das especificações de modo que, por exemplo, partes irrelevantes não sejam consideradas. Uma das formas de fazer essa manipulação é utilizando conceitos da lógica proposicional. Apesar de B ser derivado da lógica de predicados, neste trabalho precisaremos apenas de informações obtidas através da lógica proposicional. Por isto estas informações serão mostradas na próxima seção.

2.3. Lógica Proposicional

A linguagem consistindo de variáveis proposicionais e dos operadores lógicos chama-se a Lógica Proposicional ou Cálculo Proposicional (Bedregal & Acióly, 2002). A lógica proposicional trata de proposições. Uma proposição é uma afirmação declarativa que assume um valor verdadeiro (V) ou falso (F). Proposições podem ser atômicas, quando não possuem conectivos lógicos. A linguagem da lógica proposicional, como qualquer outra linguagem, é construída de dimensões sintáticas e semânticas.

A sintaxe da linguagem é constituída de fórmulas, que são um conjunto de seqüências de símbolos. Os símbolos da lógica proposicional constituem o alfabeto da linguagem (Bedregal & Acióly, 2002). Esses símbolos são:

- Variáveis proposicionais ou símbolos proposicionais: símbolos usados para representar proposições simples. Serão usados p, q, r e s, com ou sem índices (0, 1, 2, . . .). Um símbolo proposicional ou sua negação é denominado literal;
- Conectivos lógicos: símbolos usados para representar os tipos de composição. Agrupam proposições formando novas proposições. Podemos contar com cinco conectivos na lógica proposicional:

- Negação ($\neg$): para negar o valor de uma proposição;
- Conjunção ($\wedge$): Uma conjunção é formada por duas ou mais proposições ligadas pelo conectivo "e" ($\wedge$). Chama-se conjunção de duas proposições p e q a proposição representada por "p e q", cujo valor lógico é a verdade (V) quando as proposições p e q são ambas verdadeiras e falsidade (F) nos demais casos.
- Disjunção ($\vee$): Duas ou mais proposições ligadas pelo conectivo "ou" ($\vee$). Chama-se disjunção de duas proposições p e q a proposição representada por "p ou q", cujo valor lógico é a verdade (V) quando ao menos uma das proposições p e q é verdadeira e falsidade (F) quando as proposições p e q são ambas falsas.
- Condicional ou implicação ($\rightarrow$): a partir de duas proposições podemos obter uma terceira que vai ser verdadeira se e somente se a primeira proposição é falsa ou a segunda é verdadeira;
- Bi-condicional ou bi-implicação ($\leftrightarrow$): a partir de duas proposições podemos obter uma terceira que será verdadeira se e somente se ambas as proposições tiverem o mesmo valor verdade.

- Símbolos auxiliares: abre e fecha parênteses.

Uma fórmula (ou sentença ou fórmula bem formada – fbf) tem algumas propriedades, que são:

- Toda variável proposicional é uma fórmula (denominada fórmula atômica ou átomo);
- Se α é uma fórmula, então ($\neg\alpha$) é uma fórmula;
- Se α e β são fórmulas, então ($\alpha \wedge \beta$), ($\alpha \vee \beta$), ($\alpha \rightarrow \beta$), e ($\alpha \leftrightarrow \beta$), são fórmulas;
- Só são fórmulas as palavras que podem ser obtidas como mostrado nos tópicos anteriores.

Na lógica proposicional, diferentes fórmulas podem ter o mesmo valor verdade. Essas fórmulas podem ser transformadas em uma única fórmula. Para isto as fórmulas podem ser escritas através de Formas Normais. Existem dois tipos de formas normais: a Forma Normal Conjuntiva (FNC) e a Forma Normal Disjuntiva (FND). A FNC será utilizada neste trabalho, por isso será detalhada a seguir. A FND pode ser vista em detalhes em (Bedregal & Acióly, 2002).

Uma fórmula α está na FNC quando α é uma conjunção $\beta_1 \wedge \beta_2 \wedge ... \wedge \beta_n$, (n = 1) em que cada β_i ($1 \leq i \leq n$) é uma disjunção de literais ou um literal, ou seja, β_i é da forma $p_1 \vee \neg p_2 \vee p_m$ (m =1).

A fórmula α está na FNC se e somente se:

- Contém como conectivos apenas $\vee$, $\wedge$, $\neg$ não contendo nenhuma ocorrência de conectivos $\rightarrow$ ou $\leftrightarrow$;

- $\neg$ só opera sobre proposições atômicas (não tem alcance sobre $\vee$, $\wedge$);
- Não aparecem negações sucessivas ($\neg\neg$);
- $\vee$ não tem alcance sobre $\wedge$, ou seja, não existe expressão do tipo: $p \vee (q \wedge r)$.

Uma fórmula na FNC é um conjunto de Cláusulas. Uma cláusula é uma disjunção de literais, ou seja, é um conjunto de literais que estão relacionados pelo operador de disjunção.

Caso a fórmula não esteja na FNC, podemos transformá-la seguindo os seguintes passos:

- Aplicação das Leis de eliminação (para eliminação dos conectivos $\leftrightarrow$ e $\rightarrow$)

  $\neg \quad p \leftrightarrow q = (p \rightarrow q) \wedge (p \rightarrow q) = (\neg p \vee q) \wedge (\neg q \vee p)$ (eliminação da bi-implicação)

  $\neg \quad p \rightarrow q = \neg p \vee q$ (eliminação da implicação)

- Aplicação da Lei da eliminação da negação

  $\neg(\neg h) = h$

- Aplicação das Leis de De Morgan

  $\neg(p \vee q) = \neg p \wedge \neg q$

  $\neg(p \wedge q) = \neg p \vee \neg q$

- Aplicação das Leis distributivas:

  $f \vee (g \wedge h) = (f \vee g) \wedge (f \vee h)$

A lógica proposicional expressa bem a negação, a conjunção e a disjunção, mas não consegue expressar sentenças como existe e para todo, e especificações formais precisam desse poder de expressão. Especificações formais precisam definir variáveis e precisam de quantificadores da lógica de predicados ($\exists$ e $\forall$). Nisto verifica-se que as especificações formais precisam de uma lógica com maior poder de expressão que a lógica proposicional, que verifique argumentos sobre propriedades e relações entre indivíduos e elementos.

A lógica de predicados então pode ser vista como uma linguagem formal com recursos para formalizar situações em teoria dos conjuntos (funções, relações e elementos dos conjuntos). Contudo, como já mencionado anteriormente, a lógica proposicional é suficiente para fazer algumas manipulações na especificação B para a geração de casos de teste. Por isto ela será utilizada neste trabalho.

Essas manipulações, neste trabalho, são feitas, as classes de equivalência são identificadas e um conjunto de restrições precisam ser resolvidas para a escolha dos dados de teste. Para resolução dessas restrições podemos utilizar Programação de restrições, que será apresentada a seguir.

2.4. Programação de restrições

Programação de restrições, diferentemente de programação funcional e imperativa, é um paradigma declarativo no qual os programas ou problemas são especificados usando um conjunto de restrições ao invés de um algoritmo. Programadores não precisam descrever como resolver as restrições, somente precisam especificá-las. Um solucionador de restrições irá fazer isso.

Um bom exemplo desse paradigma é computação lógica. Este cenário reflete a seguinte idéia da computação: “Humanos especificam o problema e o computador o resolve”. Este paradigma é utilizado em um conjunto de aplicações tais como planejamento e cronograma.

Alguns conceitos sobre programação de restrições são apresentados a seguir:

Restrição:

Diminuição do domínio válido das variáveis. Por exemplo, se temos uma variável booleana, cujo domínio pode ser 0 ou 1, uma restrição pode fazer com que essa variável somente assuma valor 1.

Soluções:

É uma valoração das variáveis das restrições (a partir do domínio das restrições) que satisfaz a valoração de todas as restrições ao mesmo tempo. Duas restrições são equivalentes se elas têm o mesmo conjunto de soluções.

Satisfação de restrições:

Oriundo de pesquisas da inteligência artificial, o problema da satisfação de restrições (*CSP – Constraint Satisfaction Problem*) é um problema onde pode acontecer:

- Um conjunto de variáveis finitas;
- Uma função que mapeia cada variável para um domínio finito;
- Um conjunto finito de restrições.

Cada restrição restringe a combinação de valores que um conjunto de variáveis pode assumir simultaneamente. A tarefa é encontrar uma solução ou todas as soluções.

CSP é um problema combinatorial que pode ser resolvido através de buscas. Pode existir um algoritmo trivial que resolve tal problema ou pode não ser encontrada solução alguma. O algoritmo CSP gera todas as possíveis combinações de valores e então testa se uma dada combinação de valores satisfaz todas as

restrições ou não. Este algoritmo pode levar um tempo grande para executar. Por isso as pesquisas em satisfação de restrições se concentram em encontrar algoritmos que resolvam os problemas mais eficientemente, pelo menos para um subconjunto de problemas.

Neste sentido um resolvedor de restrições fornece valores para variáveis de restrições. Os valores dados a essas variáveis devem satisfazer ao conjunto de restrições.

Temos ainda animadores de especificações. Os animadores, dado uma especificação e dados de entrada, são geradas saídas compatíveis com a especificação.

No contexto de resolvedores de restrições e animadores encontra-se **ProB** (Leuschel & Butler, 2003), uma ferramenta de animação e *model checking* para o método B. Utilizando CSP está seu animador que foi desenvolvido em Tcl/ Tk e SICStus Prolog 3.10, utilizando-se assim de Prolog para a representação de termos. O animador de ProB tem ferramentas visuais que apresentam os resultados das animações. ProB suporta animações aleatórias com operações escolhidas. O número de buscas é definido pelo usuário. Este animador suporta operações não determinísticas e animação de máquinas B. Em suma, através do seu resolvedor de restrições, o ProB fornece ao usuário combinações válidas de dados de entrada. O usuário então pode escolher um subconjunto desses dados e assim animar a máquina sob teste e verificar os resultados obtidos nesta máquina. Como o ProB fornece dados de entrada para a máquina através do seu resolvedor de restrições mas esses dados são obtidos aleatoriamente, nosso trabalho propõe uma forma de escolher os dados mais relevantes de acordo com as técnicas de teste caixa preta, fazendo maiores restrições nos domínios das variáveis (o que será detalhado no próximo capítulo).

As transições de estados podem ser visualizadas pelo usuário. Para isso a ferramenta é integrada ao pacote graphviz. ProB desempenha animações simbólicas além de animações convencionais.

Algumas vantagens adicionais do ProB são:

- Suporta integração entre múltiplas máquinas B (de arquivos diferentes);
- Verifica se a operação animada quebra o invariante;
- Suporta construções do tipo x::NAT, não precisando explicitar os limites do tipo NAT.

Neste trabalho o ProB foi utilizado para encontrar dados de teste, animando as restrições impostas por cada operação de teste. Essas operações de testes são criadas a partir das restrições sobre as variáveis de cada operação da máquina sob teste. Para se testar a aplicação, de posse dos dados de teste gerados pelo ProB, neste trabalho geramos casos de teste concretos com o JUnit, pois nossa aplicação foi desenvolvida em Java. O JUnit será apresentado na próxima seção.

2.5. JUnit

O JUnit (JUnit.org) é um *framework* de código aberto e na atualidade um padrão para testes de aplicações Java. Foi desenvolvido por Kent Beck (Beck, 2004) e Erich Gamma (Gamma, 1994).

O JUnit tem uma API (*Application Programming Interface*) que facilita a criação de código para a automação de testes com apresentação dos resultados. Sem a utilização do JUnit para teste de aplicações Java, geralmente os testadores utilizam o comando *println()* do Java para observar os resultados. A utilização do *println()* não tem o conceito explícito de falha ou sucesso de um teste, ou seja, o testador precisa verificar se aquele dado de saída da aplicação é o correto ou não.

Para um maior entendimento do *framework*, a seguir será apresentado um exemplo de código de teste.

O código de teste a seguir será explicado através de referência às suas linhas.

```
1.      // Define a subclass of TestCase
2.      public class StringTest extends TestCase {

3.      // Create fixtures
4.      protected void setUp(){ /* run before */}
5.      protected void tearDown(){ /* after */ }

6.      // Add testing methods
7.      public void testSimpleAdd() {
8.      String s1 = new String("abcd");
9.      String s2 = new String("abcd");
10.     assertTrue("Strings not equal", s1.equals(s2));
11.     }

12.     // Could run the test in batch mode
13.     public static void main(String[] args){
14.     junit.textui.TestRunner.run (suite ());
15.         }
16.     }
```

- Inicialmente a classe de testes criada deve herdar da classe *TestCase* (linha 2);
- Os métodos *setUp* e *tearDown* são executados, respectivamente, antes e depois de cada caso de teste (linhas 4 e 5).
- Os métodos de teste são definidos. No nosso exemplo temos apenas um (linhas de 7 a 11). Um padrão adotado por codificadores para JUnit é o método com o prefixo "test", seguido do nome do método sob teste (linha 7). Neste caso o método que está sendo testado é o simpleAdd();
- Temos as asserções que são os componentes que determinam o sucesso ou falha do teste. Asserções são comparações entre um valor esperado e um valor obtido na execução da aplicação sob teste. No nosso exemplo a asserção está na linha 10. O *assertTrue()* compara os dois

parâmetros recebidos devolve um booleano que revela se a aplicação passou no teste (*true*) ou falhou no teste (*false*);

- Opcionalmente se define um método *main()* para execução de casos de teste em modo *standalone* (linhas 13 a 15);

O passou ou falhou da aplicação é apresentado na interface JUnit de acordo com a figura que segue:

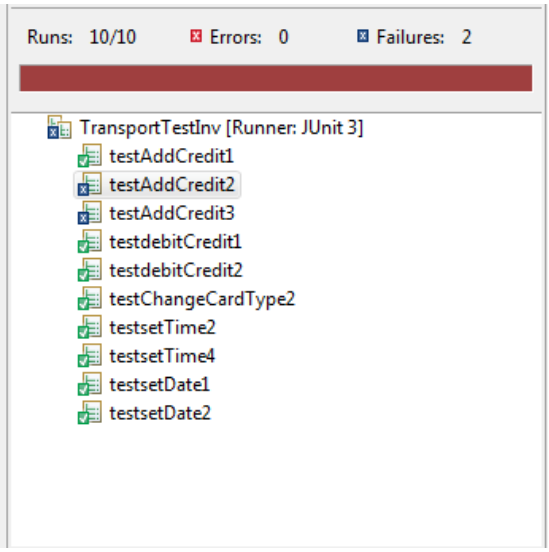


Figura 2.2 - Tela de resultados do JUnit (com casos de teste falhos)

Na Figura 2.2 podemos observar em *Runs* a quantidade de casos de teste executados. Temos nesta figura 0 erros (*Errors*) e 2 falhas (*Failures*). Os erros dizem respeito às exceções não tratadas e pegas pelo JUnit. As falhas trazem quantos casos de teste tem o resultado esperado diferente do resultado observado na operação sob teste. Devido às falhas observadas, foi apresentada uma barra vermelha. Se tivéssemos todos os casos de teste passados, esta barra seria verde.

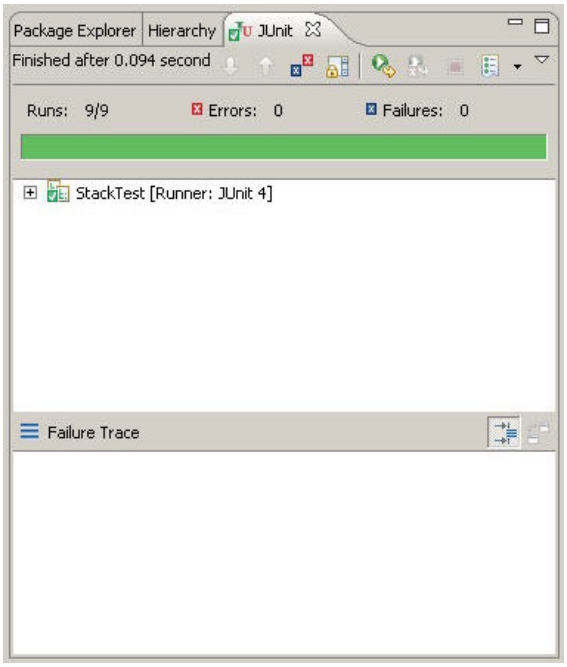
1. 

Figura 2.3 - Tela com casos de teste passados

Logo abaixo da barra vermelha ou verde temos todas as classes de teste e todos os métodos de teste que foram executados e contendo um marcador para caso tenham passado ou apresentem falhas.

Com a apresentação dos assuntos relacionados findada, podemos agora mostrar o trabalho proposto propriamente dito: a geração dos casos de teste a partir de especificações B. Isto será feito em seguida, no próximo capítulo.

Capítulo 3 Geração de Casos de Teste a partir de Especificações B

Neste trabalho foi desenvolvido um método para TBM (Teste Baseado em Modelos) onde o modelo é uma máquina B. A partir desta são geradas combinações de dados de teste abstratos para cada uma de suas operações. Para isto, utilizam-se as técnicas de teste caixa preta do particionamento de equivalência, pares ortogonais e análise do valor limite nas variáveis de entrada.

O método sistematiza os passos para o particionamento de equivalência (seção 2.1.2.1), que nas referências consultadas trazem a identificação de dados de teste de maneira não sistemática. Embora os passos para o particionamento de equivalência tenham sido utilizados, no momento da escolha dos dados de teste, para condições de variáveis relacionadas a intervalo, a análise do valor limite foi utilizada para alguns níveis de cobertura propostos. O método proposto pode ser visualizado a seguir (Figura 3.1).

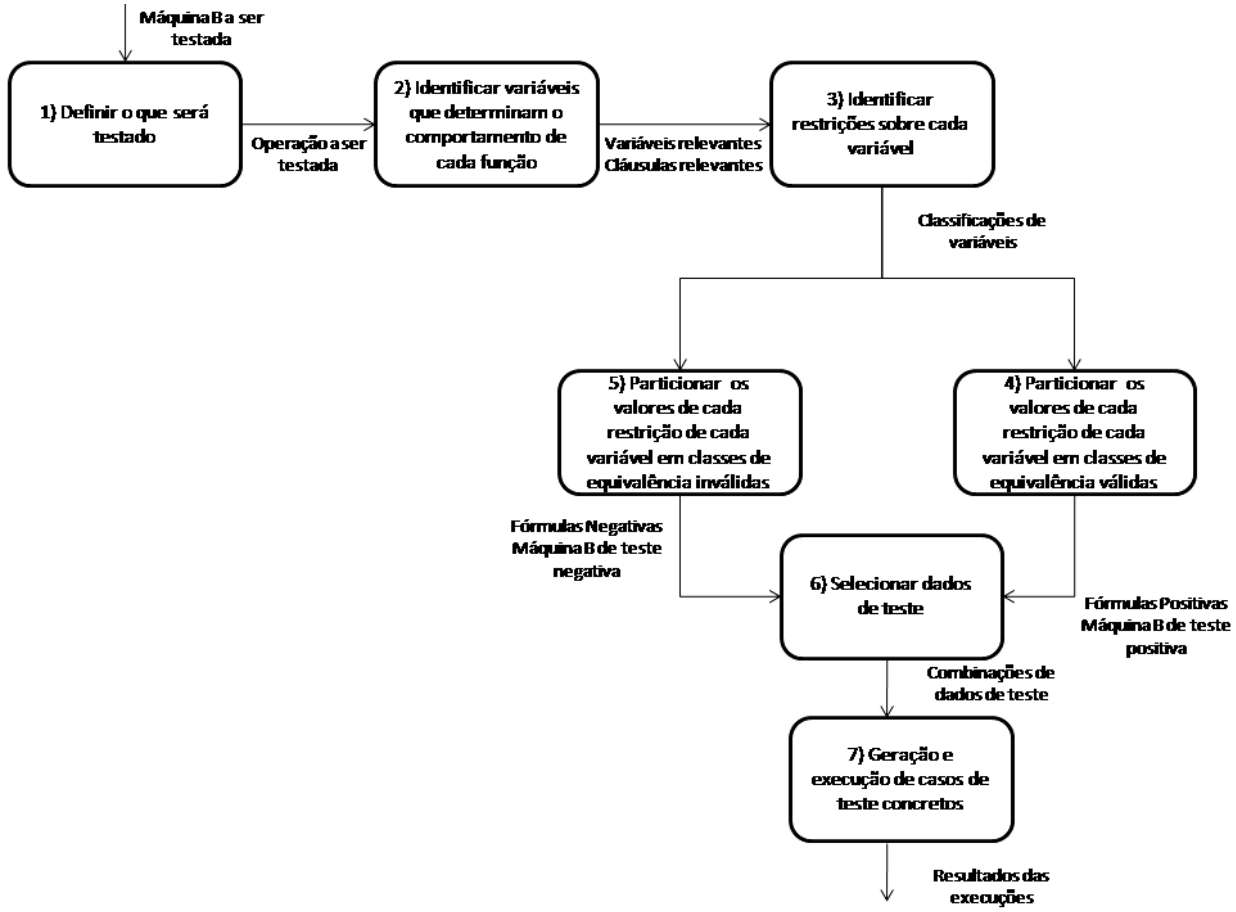


Figura 3.1 – Etapas do método de geração de casos de teste

OBS.: O método para geração de casos de teste tem como entrada uma máquina abstrata B na FNC. É necessário que a máquina esteja na FNC para que as cláusulas sejam manipuladas com uma quantidade limitada de quantificadores, facilitando a aplicação do método.

1. **Definir o que será testado**: O *software*, para ser testado, deve ser dividido em funções. Estas funções são as operações em B. O método proposto é aplicado a cada operação que o usuário deseje testar.

2. **Identificar variáveis que determinam o comportamento de cada função**: Para cada operação sob teste, são identificadas as variáveis que determinam seu comportamento (seção 3.1.1). Dessa forma estas variáveis são escolhidas (variáveis relevantes), bem como as cláusulas que trazem informações a seu respeito (cláusulas relevantes).

3. **Identificar restrições sobre cada variável**: Logo após são identificadas as restrições sobre cada variável (a partir das cláusulas relevantes) (seção 3.1.2). As restrições de tipagem, que são apresentadas pelas cláusulas de tipagem (um subconjunto das cláusulas relevantes), classificam cada variável relevante segundo o particionamento de equivalência proposto no capítulo anterior: intervalo, pertencente a um conjunto, um conjunto e um valor booleano. Sendo assim, a tipagem das variáveis é utilizada para a classificação de cada uma delas, mas no momento da animação, são consideradas também as demais restrições para a escolha dos dados de teste. Por exemplo, se temos uma tipagem como x : INT, teremos x como um intervalo. Se existe ainda uma outra restrição onde x < 10, para a escolha dos dados de teste, colocaríamos que, por exemplo, x=11 é um dado inválido. Também se x está relacionado com y, por exemplo em x <> y, quando x assumir 1, y não poderá também assumi-lo. Em suma, a classificação de acordo com o particionamento de equivalência é feita por variável e a escolha de cada dado para uma determinada variável é feita considerando-se todas as restrições sobre aquela variável. Maiores detalhes serão vistos ao longo deste capítulo.

4. **Particionar os valores de cada restrição de cada variável em classes de equivalência válidas**: A partir da identificação das restrições sobre cada variável, mostrada no item anterior, são identificadas as classes de equivalência válidas (seção 3.1.3). Essas classes de equivalência válidas são caracterizadas por fórmulas que são verdadeiras para todos os valores da classe. Essas fórmulas são utilizadas para identificar os dados que devem ser escolhidos para representar a classe. Essas fórmulas são combinadas de maneira a identificarem combinações de dados válidos e operações B de teste são montadas. Estas combinações podem ser feitas de acordo com três níveis de cobertura diferentes, conforme detalhado a seguir, ficando a escolha de um dos níveis a critério do usuário. Adicionalmente o conteúdo exato dessas fórmulas leva em consideração não apenas a descrição das classes como o problema de dependência entre variáveis, também descrito a seguir. Para fazer combinações foi criado um programa onde o

usuário entra com as fórmulas e este gera como saída um arquivo texto com as combinações. Este programa faz a combinação de todas as fórmulas entre si. Para cada operação sob teste deve existir uma quantidade de operações de teste de acordo com a quantidade de combinações de condições sobre os valores das variáveis relevantes de teste (casos de teste). Todas as operações de teste decorrentes das operações sob teste constam em uma máquina B de teste (máquina de teste válida – que originará combinações de dados de teste válidas).

5. **Particionar os valores de cada restrição de cada variável em classes de equivalência inválidas**: Também deve ser gerada uma máquina de teste inválida – para combinações de dados de teste inválidas (seção 3.1.4). Algumas cláusulas relevantes são negadas (as cláusulas que não são de tipagem) e assim operações de teste são montadas. Entendemos que buscar dados fora do domínio válido traria um erro de compilação e o teste não poderia ser efetuado. Da forma proposta, o erro de compilação não existe e se consegue testar a robustez do código que implementa a operação sob teste. Existe uma máquina de teste B inválida para cada máquina B que se deseja testar. Essa máquina de testes contém todas as operações com combinação de condições que levem à geração de dados inválidos.
6. **Selecionar dados de teste**: Com as máquinas de teste geradas (máquinas inválida e válida) é hora da descoberta das combinações de dados de teste (seção 3.1.5). Para isto, são feitas animações nas máquinas de teste (com a utilização de um solucionador de restrições). Para estas animações utilizamos a ferramenta ProB. Sendo assim, ao final de cada animação o ProB fornece os dados de teste abstratos que satisfazem às restrições das operações de teste. Para obtenção dos resultados esperados são feitas animações na máquina de teste, para testes válidos. Para os inválidos o testador deve identificar a saída esperada.
7. **Geração e Execução de casos de teste concretos**: Devem ser agora definidos casos de teste para a aplicação (seção 3.1.6). Neste trabalho os casos de teste foram criados em JUnit para testar a aplicação em Java, mas o método proposto pode ser utilizado em qualquer outra ferramenta. É criado um caso de teste no JUnit para cada combinação da dados de teste encontrada. Sendo assim, pode haver operação do sistema sob teste com vários casos de teste. O JUnit fornece os resultados do testes (passou ou falhou) que podem ser visualizados através de cores (verde para casos aprovados e vermelho para casos reprovados).

Os passo 4, 5 e 6 estão direcionados ao uso de ProB, mas as condições correspondentes a cada caso de teste são fórmulas lógicas que poderiam ser fornecidas a outras ferramentas de solução de restrições ou mesmo serem resolvidas pelo desenvolvedor de teste. Neste último caso, a ajuda seria somente a identificação dos casos de teste e das condições que tem que ser satisfeitas pelos dados em cada caso.

O problema da dependência entre variáveis na geração de testes

Para a geração dos dados de teste, nos deparamos com variáveis que tem seus domínios relacionados, ou seja, para que tenhamos um valor específico para uma variável, o valor de outra ou de outras variáveis devem ser considerados.

Uma das formas de dependências entre variáveis que foram tratadas neste trabalho foram de variáveis do tipo intervalo. Outros tipos de variáveis também tinham dependência, mas estas dependências eram resolvidas diretamente pelo ProB. Já com intervalos que se relacionavam como para dados de teste, deveríamos selecionar os limites dos intervalos, ou seja, o domínio válido para teste era minimizado. Sendo assim, as chances de duas variáveis de intervalo relacionadas, quando os valores limites de ambas eram procurados, não terem valores satisfatíveis, era alta, conforme será explicado na Análise dos Resultados. Para uma maior explicação, observemos o exemplo a seguir.

Considere que temos duas variáveis numa especificação, balance e cr, dois intervalos. Esse exemplo já foi mostrado no Capítulo 2 e será utilizado ao longo deste capítulo para exemplificação do método proposto.

- Balance : NAT
- Cr: NAT1

Essas duas variáveis são relacionadas entre si na especificação da máquina B através da fórmula:

- Balance + Cr <= MAXINT

Os valores limites absolutos válidos a serem buscados para estas variáveis, são os mínimos (balance=0 e Cr=1) e os máximos (balance= MAXINT e Cr=MAXINT). Dessa forma teríamos as seguintes combinações de dados de teste para estas variáveis, por enquanto sem considerar as dependências:

- Combinação 1: balance=0 e Cr=1
- Combinação 2: balance=0 e Cr=MAXINT
- Combinação 3: balance=MAXINT e Cr=1
- Combinação 4: balance=MAXINT e Cr=MAXINT

Neste exemplo, quando tentamos maximizar as duas variáveis ao mesmo tempo, teremos balance = MAXINT e Cr = MAXINT (Combinação 4). Dessa forma a cláusula Balance + Cr <= MAXINT se torna insatisfatível. Também torna Balance + Cr <= MAXINT insatisfatível a combinação 3. Assim conseguimos minimizar as duas variáveis e maximizar somente Cr (na combinação 2).

Pensando nisso, ao invés de buscarmos os valores absolutos diretamente, buscamos os limites das variáveis incluindo as cláusulas dependentes. Essas cláusulas dependentes fazem com que tenhamos

limites absolutos em alguns casos e relativos quando os absolutos não são possíveis. Assim teremos a seguintes combinações para descoberta de dados de teste:

- Combinação 1: balance = maior possível, cr = maior possível;
- Combinação 2: balance = maior possível, cr=menor possível;
- Combinação 3: balance = menor possível, cr = maior possível;
- Combinação 4: balance = menor possível, cr = menor possível.

Nestas novas combinações, o valor máximo para balance é MAXINT-1, já que o valor mínimo para cr = 1. Isso é fornecido pela combinação 3. O limite superior de balance é um limite relativo, mas o seu inferior continua sendo 0 e os limites inferior e superior de cr são os limites absolutos (1 e MAXINT).

Níveis de cobertura

Os níveis de cobertura diferem na escolha dos dados de teste válidos, enquanto que os inválidos são escolhidos da mesma forma para os três níveis. Vamos discutir primeiramente a escolha dos dados válidos e ao final a escolha dos dados inválidos. Vale ressaltar que os três níveis podem ser aplicados para geração de testes de qualquer aplicação, mas alguns são mais indicados em algumas situações que os outros. A Tabela 3.1 mostra uma análise comparativa entre os níveis de cobertura, considerando quais níveis são mais indicados em cada situação para os dados válidos:

Tabela 3.1 – Relacionamento entre os níveis de cobertura

Nível	Particionamento de equivalência	Análise do valor limite	Dependência entre duas variáveis	Dependência entre mais de duas variáveis
1				
2				
3				

Nível 1

O nível mais básico, nível 1, estabelece a cobertura pelo particionamento de equivalência. O critério de cobertura do particionamento de equivalência pode ser visto em detalhes no Capítulo 2. Neste nível, um dado de teste é encontrado para cada classe de equivalência positiva identificada. A classe negativa será tratada a parte e pode ser vista na seção “Escolha dos dados inválidos”. Este nível é mais adequado, quando não se tem variáveis de intervalo, pois não considera que os limites são mais relevantes para encontrar falhas, e quando se deseja executar apenas um teste básico na especificação.

Nível 2

O nível 2 faz combinações através da técnica dos pares ortogonais. Neste nível estamos considerando, para variáveis de intervalo, dados de teste que sigam o critério de cobertura da análise do valor limite e para os demais tipos de variáveis, o critério do particionamento de equivalência. Podem acontecer aqui a escolha de dados nos limites absolutos e quando estes não podem ser alcançados, os limites relativos são encontrados.

No nível 2, duas variáveis serão sempre combinadas uma a uma, mas não se pode garantir que três ou mais tenham todas as suas combinações para teste. Dessa forma, pode acontecer de uma condição que relacione três ou mais variáveis não acontecer, como por exemplo, uma que minimize três variáveis ao mesmo tempo.

Nível 3

O nível mais robusto, nível 3, busca o critério de cobertura da análise do valor limite (Capítulo 2) para as variáveis classificadas como um intervalo, sendo que os limites gerados podem ser os limites absolutos ou relativos. Para as demais classificações de variáveis, a cobertura se dá pelo particionamento de equivalência.

Este nível de cobertura é indicado para especificações com fortes dependências entre variáveis e também para especificações com variáveis de limite.

A escolha dos dados inválidos

A parte comum dos três níveis diz respeito aos dados inválidos. Tanto para o particionamento de equivalência quanto para a análise do valor limite, segundo (Myers, 2004) os dados inválidos são escolhidos fora do domínio válido da variável. Se tivéssemos, por exemplo, uma variável do tipo x: Y onde Y é um conjunto qualquer, se escolhêssemos um dado inválido para teste z onde z/ : Y (neste caso buscando a cobertura por particionamento de equivalência), e passássemos z para x, isto poderia levar a um erro de compilação da máquina de teste, já que x receberia um valor fora de seu domínio válido se o domínio coincide com o tipo na linguagem de implementação.

Sendo assim, foram gerados dados de teste inválidos a partir da negação das restrições da máquina alvo de teste que não contêm informações de tipagem. Numa combinação de dados de teste somente teremos,

no máximo, um dado inválido. Seguindo o exemplo dado anteriormente, os dados de teste inválidos para balance e Cr são aqueles que tornam a restrição *not(balance+Cr<=MAXINT)* verdadeira, ou seja, os dados negativos serão escolhidos negando-se a cláusula *balance+Cr<=MAXINT*. Isto será explicado detalhadamente ao longo deste capítulo. A geração dos dados de teste pelo ProB neste caso trará várias combinações de teste. Cabe ao usuário escolher somente uma e assim garantir a cobertura por particionamento de equivalência.

Com o método proposto, caso não se tenha restrições de não tipagem, não são gerados dados de teste inválidos para a máquina alvo de teste.

3.1. O Método Proposto

O método de geração de casos de teste será apresentado, através de notação algorítmica, em etapas. Para cada etapa serão apresentadas, inicialmente, as entradas e saídas, logo após o algoritmo para identificação das saídas e ao final a exemplificação do algoritmo. As diferenças entre os níveis de cobertura serão ressaltados ao longo do algoritmo.

As etapas serão mostradas e ao seu final haverá um exemplo de sua aplicação na máquina *Transport*. Os detalhes desta máquina podem ser vistos no Capítulo 2. A seguir ela será apresentada novamente para facilitar o entendimento do método. O exemplo será feito na operação *addCredit*. Também será apresentada a máquina *Transport_Constants*, que contém um conjunto utilizado por *Transport*. A aplicação gerada a partir dessa máquina, e que encontra-se no Apêndice, foi implementada a partir da máquina abstrata, ou seja, refinamentos não foram procedidos.

Tabela 3.2 – Reapresentação das máquinas B Transport e Transport_Constants

```
1  MACHINE Transport
2  SEES Transport_Constants
3  DEFINITIONS
4       DATE ==
5            day, month, year | day : 1..31 & month : 1..12 & year : NAT1 &
6            (day = 31 => month : 1, 3, 5, 7, 8, 10, 12) &
7            (month = 2 => (day <= 28 or (day = 29 &
8                 ((year mod 400 =0) or (year mod 4 = 0 &
9                 year mod 100 /= 0)))))
10 VARIABLES
11      balance, card_type, time, date
12 INVARIANT
13      balance : NAT &
14      time : (0..23) * (0..59) &
15      date : DATE &
16      card_type : CARD_TYPES &
```

```
17      (not(card_type = gratutious_card) or balance = 0)
18 INITIALISATION
19      balance := 0 ||
20      time := 0 |-> 0 ||
21      date := 1 |-> 1 |-> 1 ||
22      card_type :: CARD_TYPES
23 OPERATIONS
24      addCredit (cr) =
25          PRE
26              cr : NAT1 &
27              card_type /= gratuitous_card &
28              balance + cr <= MAXINT
29          THEN
30              balance := balance + cr
31      END
```

```
1  MACHINE Transport_Constants
2  SETS CARD_TYPES = {entire_card, student_card, gratutious_card}
```

3.1.1. Identificar variáveis e cláusulas que determinam o comportamento de cada operação

- **Entradas**: as `Variáveis, o Invariante e a Pré-condição da especificação B da operação`
- **Saídas**: os conjuntos `Variáveis_relevantes e Proposição_resultante`

Esta etapa é comum aos três níveis de cobertura. Nesta seção serão manipuladas três estruturas de uma máquina B: Variáveis, Invariante e Pré-condição da operação que se deseja testar. Com isto tem-se como resultado dois conjuntos: `Variáveis_relevantes` e `Proposição_resultante`.

O conjunto `Variáveis_relevantes` contém variáveis que devem ser consideradas para teste da operação em questão. Estas são as variáveis da pré-condição e as variáveis globais da máquina que se relacionam com as variáveis da pré-condição. Com a escolha dessas duas estruturas de B busca-se identificar dados de teste que satisfazem e não satisfazem tanto o invariante da máquina quanto a pré-condição da operação que se deseja testar. As variáveis do corpo da operação não estão sendo tratadas e isto é sugestão de trabalhos futuros. Sabe-se que a negação do invariante, quando simula-se uma máquina B, não é interessante, já que se o invariante está quebrado de início, nada se pode afirmar dos resultados posteriores. Neste trabalho estamos considerando o invariante mesmo assim, pois nada se sabe da forma como o *software* será implementado, ou seja, se o programador fará seu código robusto o suficiente para capturar exceções provocadas, por exemplo, por dados incorretos preenchidos pelo

usuário. Dessa forma, nosso método se preocupa em gerar dados abstratos inválidos para as variáveis de estado da máquina B.

O conjunto `Proposição_resultante` é originado das cláusulas da pré-condição da operação, como também das cláusulas do invariante que trazem informações a respeito das variáveis globais que se relacionam com as variáveis da pré-condição. Existem ainda alguns conjuntos auxiliares. O algoritmo para obtenção dos dois conjuntos resultantes pode ser visto na Tabela 3.3 que segue.

Tabela 3.3 – Algoritmo da seção 3.1.1

```
1. Inicializar variáveis:

     Variáveis_relevantes = variáveis da pré-condição da operação;

     Proposição_resultante = cláusulas da pré-condição da operação;

     Claúsulas_invariante_relevantes = Cláusulas do invariante que contêm algum elemento
          de Variáveis_relevantes;

     Variáveis_pendentes = Variáveis contidas em Claúsulas_invariante_relevantes menos
          Variáveis_relevantes;

2. Enquanto Variáveis_pendentes for diferente de vazio:

     Variáveis_relevantes = Variáveis_relevantes união Variáveis_pendentes;

     Cláusulas_invariante_relevantes = Cláusulas_invariante_relevantes união cláusulas do
          invariante que contêm algum elemento de Variáveis_relevantes;

     Variáveis_pendentes = Variáveis contidas em Claúsulas_invariante_relevantes menos
          Variáveis_relevantes;

3. Proposição_resultante = Proposição_resultante união Cláusulas_invariante_relevantes.
```

Temos como conjuntos auxiliares `Cláusulas_invariante_relevantes` e `Variáveis_pendentes`. O conjunto `Cláusulas_invariante_relevantes` contém as cláusulas que contém elementos de `Variáveis_relevantes`.

O conjunto `Variáveis_pendentes` identifica as variáveis globais que se relacionam com `Variáveis_relevantes`. Assim, se temos alguma variável global que se relaciona com a variável da pré-condição da operação por outra variável global (relacionamento indireto), esta também será considerada para teste. Então, todas as cláusulas que contêm elementos de `Variáveis_pendentes`

devem ser adicionadas ao conjunto `Cláusulas_invariante_relevantes` (sempre que for identificado um novo elemento para `Variáveis_pendentes`, deve-se adicionar elementos a `Cláusulas_invariante_relevantes`).

EXEMPLO

- Setar os valores iniciais:

  Variáveis_relevantes = {cr, card_type, balance};

  Proposição_resultante = {cr: NAT1,
  card_type /= gratutious_card,
  balance+Cr <= MAXINT};
  Claúsulas_invariante_relevantes =
  {balance: NAT,
  card_type: CARD_TYPES,
  (not(card_type = gratutious_card) or balance = 0)}

  Variáveis_pendentes = { }

- **Proposição_resultante** = {cr: NAT1,
  card_type /= gratutious_card,
  balance + Cr <= MAXINT, balance:
  NAT, card_type: CARD_TYPES,
  (not(card_type = gratutious_card) or balance = 0)}

3.1.2. Identificar as restrições sobre cada variável

- **Entradas**: `Variáveis_relevantes e Proposição_resultante`
- **Saídas**: `Classificações , Cláusulas_Tipagem e Cláusulas_Não_Tipagem`

Esta etapa também é comum aos três níveis de cobertura. Nesta seção serão manipulados os dois conjuntos oriundo da seção anterior: `Variáveis_relevantes` e `Proposição_resultante`. O conjunto resultante é denominado `Classificações`. Para que este seja formado é necessário identificar quais elementos de `Proposição_resultante` se adéquam à sintaxe da construção da linguagem B *Typing_abstract_data* (Clearsy Systems Engineering), mostrada no Capítulo 2. Esta é a sintaxe B para cláusulas de tipagem de máquinas abstratas.

Dessa forma, cada elemento de `Classificações` é uma tripla contendo: o elemento de `Variáveis_relevantes` (variável que está sendo tipada), o elemento de

Proposição_resultante (tipagem da variável) e a classificação de equivalência em que a tipagem se enquadra. As cláusulas que foram para Classificações são postas em Clausulas_Tipagem, pois são as cláusulas que trazem informações sobre os tipos das variáveis e as demais vão para Cláusulas_Não_Tipagem. O algoritmo pode ser visto a seguir (Tabela 3.4):

Tabela 3.4 – Algoritmo da seção 3.1.2

1. Clausulas_Não_Tipagem = Proposição_Resultante;

2. Para cada elemento de Variáveis_relevantes:

2.1. Para cada elemento de Proposição_Resultante:

2.1.1. Se elemento de Proposição_Resultante é do tipo: (elemento de Variáveis_relevantes) "∈" Expression+"×" e se Expression é do tipo NAT ou Z ou NAT1 ou do tipo x..y, onde x e y são inteiros, então:

Classificações **=**

Classificações união

{(elemento de Variáveis_relevantes,

elemento de Proposição_Resultante,

"intervalo")};

Clausulas_Não_Tipagem **=**

Clausulas_Não_Tipagem –

{elemento de Proposição_Resultante};

2.1.2.Senão (elemento de Proposição_Resultante é do tipo: (elemento de Variáveis_relevantes) "∈" Expression+"×") então

Classificações **=**

Classificações união

{(elemento de Variáveis_relevantes,

elemento de Proposição_Resultante,

"pertencente a um conjunto")};

Clausulas_Não_Tipagem **=**

Clausulas_Não_Tipagem –

{elemento de Proposição_Resultante};

2.1.3.**Se elemento Proposição_Resultante é do tipo: (elemento de Variáveis_relevantes) "⊂" Expression então:**

Classificações **=**

Classificações união

{(elemento de Variáveis_relevantes,

elemento de Proposição_Resultante,

“conjunto”)};

Clausulas_Não_Tipagem **=**

Clausulas_Não_Tipagem –

{elemento de Proposição_Resultante};

2.1.4. **Se elemento Proposição_Resultante é do tipo: (elemento de Variáveis_relevantes) "⊆" Expression então:**

Classificações **=**

Classificações união

{(elemento de Variáveis_relevantes,

elemento de Proposição_Resultante,

“conjunto”)}

Clausulas_Não_Tipagem **=**

Clausulas_Não_Tipagem –

{elemento de Proposição_Resultante};

2.1.5. **Se elemento Proposição_Resultante é do tipo: (elemento de Variáveis_relevantes) +", " "=" Expression+", " então:**

Classificações **=**

Classificações união

{(elemento de Variáveis_relevantes,

elemento de Proposição_Resultante,

“booleano”)}

Clausulas_Não_Tipagem **=**

Clausulas_Não_Tipagem –

{elemento de Proposição_Resultante};

3. **Cláusulas_Tipagem = Proposição_Resultante – Cláusulas_Não_Tipagem**

Neste método, além de cláusulas de tipagem explícita, tratamos ainda de cláusulas de tipagem implícita, como as da forma $x \in 1..10$, como no item 2.1.1. Todos os trabalhos pesquisados (ver capítulo de Trabalhos relacionados), somente trabalham como tipagens explícitas, ditando assim que seus métodos só funcionam se a especificação for escrita com este tipo de estilo.

No presente trabalho, caso a tipagem seja implícita, podemos perder algumas informações importantes para a geração de testes. No exemplo anterior, o método proposto não geraria o dado x=11 e então poderíamos estar deixando passar uma possível falha. Como nosso método interpreta tipagens implícita e explícita, para maior precisão do método, o usuário deve utilizar tipagem explícita.

EXEMPLO

Entradas:

- **Variáveis_relevantes** = {cr, card_type, balance};
- **Proposição_resultante** = {cr: NAT1, card_type /= gratutious_card, balance + Cr <= MAXINT, balance: NAT, card_type: CARD_TYPES, (not(card_type = gratutious_card) or balance = 0)}

1. Cláusulas_Não_Tipagem = {cr: NAT1, card_type /= gratutious_card, balance + Cr <= MAXINT,balance: NAT, card_type: CARD_TYPES, (not(card_type = gratutious_card) or balance = 0)}

2. Para cada elemento de Variáveis_relevantes (elemento 1 = Cr):

2.1. Para cada elemento de Proposição_Resultante (elemento 1 = cr: NAT1):

2.1.1. Se elemento de Proposição_Resultante é do tipo: (elemento deVariáveis_relevantes) "∈" *Expression*$^{+"\times"}$ e se Expression é do tipo NAT ou Z ou

2.1.2. NAT1 ou do tipo x..y, onde x e y são inteiros, então:

Classificações =
Classificações união
{(cr, cr: NAT1, “intervalo”)}

Classificações =
{(cr, cr: NAT1, “intervalo”)}

Cláusulas_Não_Tipagem =
{card_type /= gratutious_card,
balance + Cr <=MAXINT,
balance: NAT,
card_type: CARD_TYPES,
(not(card_type = gratutious_card)or balance = 0)}

2. Para cada elemento de Variáveis_relevantes (elemento 2 = card_type):

2.2. Para cada elemento de Proposição_Resultante (1 a 3 sem alterações e elemento4 =

card_type: CARD_TYPES):

2.2.1. Se elemento de Proposição_Resultante é do tipo: (elemento deVariáveis_relevantes) "∈" Expression+"×" então:

Classificações =
{(cr, cr: NAT1, “intervalo”),
{(card_type, card_type:CARD_TYPES, “pertence a um conjunto”)}

Cláusulas_Não_Tipagem =

{card_type /= gratutious_card,

balance + Cr <=MAXINT,

balance: NAT,

(not(card_type = gratutious_card) or balance = 0)}

2. Para cada elemento de Variáveis_relevantes (elemento 3 = balance):

2.1. Para cada elemento de Proposição_Resultante (1 e 2 sem alterações e elemento3 = balance: NAT):

2.1.1. Se elemento de Proposição_Resultante é do tipo: (elemento deVariáveis_relevantes) "∈" *Expression*$^{+"\times"}$ então:

Classificações =
Classificações união
{(balance, balance: NAT,“intervalo”)}

Classificações =

{(cr, cr: NAT1, “intervalo”),

{(card_type, card_type:CARD_TYPES, “pertence a um conjunto”),

(balance, balance: NAT,“intervalo”)}

Cláusulas_Não_Tipagem =

{card_type /= gratutious_card,

balance + Cr <=MAXINT,

(not(card_type = gratutious_card) or balance = 0)}

3. Cláusulas_Tipagem = {cr: NAT1, balance: NAT, card_type: CARD_TYPES}

OBS.: O processamento para cláusulas que não satisfaziam as condições não foram mostrados.

3.1.3.Particionar os valores de cada restrição de cada variável em classes de equivalência válidas

- Entradas: `Classificações, Variáveis_relevantes`
- Saídas: `Combinações_de_Fórmulas e máquina de teste para identificação de dados válidos`

A partir do conjunto `Classificações`, as fórmulas para geração de casos de teste para cada operação devem ser identificadas. Isso originará o conjunto `Fórmulas`, que é um conjunto auxiliar nesta etapa. Neste conjunto existirá um elemento para cada variável. Cada um destes elementos contém a variável e as fórmulas que gerarão dados válidos para teste de acordo com a tipagem de cada variável. As fórmulas podem ser diferentes de acordo com o nível de cobertura escolhido. Cada cláusula de cada variável é combinada às cláusulas das demais variáveis, formando o conjunto `Combinações_de_Fórmulas`. Estas combinações são feitas da mesma forma para os níveis 1 e 3 e é diferente no nível 2. Para cada combinação deve ser gerada uma operação de teste, onde cada uma dessas operações de teste dará origem a um caso de teste.

Para esta seção será criada uma função Class() que recebe uma variável de Variáveis_Relevantes e retorna sua classificação, por exemplo, Class(cr) = “intervalo”.

Outra função auxiliar é a Tipagem() que recebe uma variável de Variáveis_Relevantes e retorna a sua cláusula de tipagem, por exemplo, Tipagem(cr) = cr: NAT1.

Tabela 3.5 – Algoritmo da seção 3.1.3

```
1. Para cada elemento de Variáveis_Relevantes:

    1.1.1. Se o elemento for classificado como um intervalo (Class(e) = “intervalo”)
           então:

               (nível 1) Fórmulas = Fórmulas ∪ {(e, Tipagem(e))}

               (níveis 2 e 3) Fórmulas =
                     Fórmulas ∪
                     {(e, Tipagem(e)),
                     (!b. ((b: intervalo da variável &
                          b /= Classificações(n, 1)) => Classificações(n, 1) < b)),
                     (!b. ((b: intervalo da variável &
                          b /= Classificações(n, 1))=> Classificações(n, 1) > b)))}

    1.1.2. Se o elemento for Class(e) = “pertence a um conjunto” ou Class(e) =
           “conjunto” ou Class(e) = “booleano” então:

               (níveis 1, 2 e 3) Fórmulas = Fórmulas ∪ {(e, Tipagem(e))}.
```

2. Para cada elemento de Fórmulas:

    2.1.1. (nível 1 e 3) Combinar cada cláusula de teste com as cláusulas das demais variáveis. Cada combinação é um elemento de Combinações_de_Fórmulas;

    2.1.2. (nível 2) Combinar as cláusulas de teste de acordo com a técnica dos pares ortogonais. Cada combinação é um elemento de Combinações_de_Fórmulas;

3. (níveis 1, 2 e 3) Para cada elemento de Combinações_de_Fórmulas gerar um caso de teste na máquina de teste (cada elemento de Combinações_de_Fórmulas é um caso de teste), onde cada operação de teste terá o nome da operação a ser testada seguido do nome "Teste" seguido de um seqüencial que é correspondente a posição do elemento de Combinações_de_Fórmulas, de forma que:

    Cada elemento de Variáveis_relevantes é um parâmetro da operação;

    Para cada elemento de Variáveis_relevantes deve haver um parâmetro de saída da operação;

    Cada elemento de Combinações_de_Fórmulas deve ser posto na pré-condição da operação, separados por "&" com os elementos de Proposição_Resultante;

    Colocar os elementos de Cláusulas_Não_Tipagem que tem relacionamento com as variáveis de intervalo dentro do domínio da variável b, na primeira parte da implicação, trocando a variável que está buscando o limite pela variável auxiliar b;

    Os parâmetros de entrada devem ser passados, no corpo da operação, aos parâmetros de saída.

Colocar os elementos de Cláusulas_Não_Tipagem no corpo da fórmula para busca de limites faz com que se obtenha os limites relativos de cada variável. Sendo assim, caso os limites absolutos não sejam obtidos pela influência de alguma outra cláusula ou algumas outras cláusulas, estas serão incluídas na fórmula para obtenção dos dados de teste e esta nova fórmula somente gera dados que satisfazem todas as demais cláusulas relevantes para teste da operação.

Nesta etapa pode-se notar que as fórmulas identificadas para as classes de equivalência de "pertencente a um conjunto", "conjunto" e "booleano" são idênticas. Conservaram-se as classes de equivalência para que ficasse consistente com as citadas no capítulo anterior, mas nada impede das variáveis serem classificadas em somente "um intervalo" e "um valor booleano" (um valor booleano incluiria também um valor pertencente a um conjunto e um conjunto).

Para combinar as fórmulas foi feito um programa onde o usuário insere as fórmulas e o programa dá como saída um arquivo texto onde cada linha é uma combinação. Este programa é utilizado somente

para os níveis de cobertura 1 e 3, não sendo evoluído ainda para a forma de combinação do nível 2. Isto porque a maneira de combinação das fórmulas de teste é igual para os níveis 1 e 3. Sendo assim, para o nível 2, o incremento do programa é uma proposta de trabalho futuro.

EXEMPLO

Entradas:

- **Classificações** = {(cr, cr: NAT1, “intervalo”), {(card_type, card_type: CARD_TYPES, “pertence a um conjunto”), (balance, balance: NAT, “intervalo”)}
- **Variáveis_relevantes** = {Cr, card_type, balance}

1. Para cada elemento de Variáveis_Relevantes (cr):

   1.1.1. Se o elemento for classificado como um intervalo (Class(cr) = “intervalo”) então:

   (nível 1) Fórmulas = {} ∪ {(cr, (cr: NAT1))}

   (níveis 2 e 3) Fórmulas = {} ∪
   {(cr, (cr: NAT1),
   (!b. ((b: NAT1 & b /= cr) => cr < b)),
   (!b. ((b: NAT1 & b /= cr)=> cr > b)))}

1. Para cada elemento de Variáveis_Relevantes (card_type):

   1.1.1. Se o elemento for Class(card_type) = “pertence a um conjunto” então:

   (nível 1)
   Fórmulas =
   {(cr, (cr: NAT1))} ∪
   {(card_type, (card_type: CARD_TYPES))}

   Fórmulas =
   {(cr, (cr: NAT1)),
   (card_type, (card_type: CARD_TYPES))}

   (níveis 2 e 3)
   Fórmulas =
   {(cr, (cr: NAT1),
   (!b. ((b: NAT1 & b /= cr) => cr < b)),
   (!b. ((b: NAT1 & b /= cr)=> cr > b)))}
   ∪
   {(card_type, (card_type: CARD_TYPES))}

Fórmulas =
{(cr, (cr: NAT1),
(!b. ((b: NAT1 & b /= cr) => cr < b))),
(!b. ((b: NAT1 & b /= cr)=> cr > b)),
(card_type, (card_type: CARD_TYPES)))}

1. Para cada elemento de Variáveis_Relevantes (balance):

1.1.1. Se o elemento for classificado como um intervalo (Class(balance) = "intervalo") então:

(nível 1)

Fórmulas =
{(cr, (cr: NAT1)), (card_type, (card_type: CARD_TYPES)))} ∪
{(balance, (balance: NAT))}

Fórmulas =
{(cr, (cr: NAT1)),
(card_type, (card_type: CARD_TYPES)),
(balance, (balance: NAT)))}

(níveis 2 e 3)

Fórmulas =
{(cr, (cr: NAT1),
(!b. ((b: NAT1 & b /= cr) => cr < b)),
(!b. ((b: NAT1 & b /= cr)=> cr > b))),
(card_type, (card_type: CARD_TYPES))}
∪
{(balance, (balance: NAT),
(!b. ((b: NAT & b /= balance) => balance < b)),
(!b. ((b: NAT & b /= balance)=> balance > b)))}

Fórmulas =
{(cr, (cr: NAT1),
(!b. ((b: NAT1 & b /= cr) => cr < b)),
(!b. ((b: NAT1 & b /= cr)=> cr > b)))),
(card_type, (card_type: CARD_TYPES)),
(balance, (balance: NAT),
(!b. ((b: NAT & b /= balance) => balance < b)),
(!b. ((b: NAT & b /= balance)=> balance > b)))}

2. Para cada elemento de Fórmulas:

    2.1.1. Combinar cada cláusula de teste com as cláusulas das demais variáveis. Cada combinação é um elemento de Combinações_de_Fórmulas.

As combinações podem ser vistas na tabela a seguir, onde cada linha da tabela é um elemento de Combinações_de_Fórmulas. As combinações são mostradas para o nível 1 (Tabela 3.6), nível 2 (Tabela 3.7) e nível 3 (Tabela 3.8).

Tabela 3.6 – Combinações para o nível 1

Id	Cr	Card_type	Balance
1	cr: NAT1	card_type: CARD_TYPES	balance: NAT

Tabela 3.7 – Combinações para o nível 2

Id	Cr	Card_type	Balance
1	cr: NAT1	card_type: CARD_TYPES	balance: NAT
2	(!b. ((b: NAT1 & b /= cr) => cr < b))	card_type: CARD_TYPES	(!b. ((b: NAT & b /= balance) => balance < b))
3	(!b. ((b: NAT1 & b /= cr) => cr < b))	card_type: CARD_TYPES	(!b. ((b: NAT & b /= balance)=> balance > b))
4	(!b. ((b: NAT1 & b /= cr)=> cr > b))	card_type: CARD_TYPES	(!b. ((b: NAT & b /= balance) => balance < b))
5	(!b. ((b: NAT1 & b /= cr)=> cr > b))	card_type: CARD_TYPES	(!b. ((b: NAT & b /= balance)=> balance > b))

(nível 3)

Tabela 3.8 – Combinações para o nível 3

Id	Cr	Card_type	Balance
1	cr: NAT1	card_type: CARD_TYPES	balance: NAT
2	(!b. ((b: NAT1 & b /= cr) => cr < b))	card_type: CARD_TYPES	(!b. ((b: NAT & b /= balance) => balance < b))
3	(!b. ((b: NAT1 & b /= cr) => cr < b))	card_type: CARD_TYPES	(!b. ((b: NAT & b /= balance)=> balance > b))
4	(!b. ((b: NAT1 & b /= cr)=> cr > b))	card_type: CARD_TYPES	(!b. ((b: NAT & b /= balance) => balance < b))
5	(!b. ((b: NAT1 & b /= cr)=> cr > b))	card_type: CARD_TYPES	(!b. ((b: NAT & b /= balance)=> balance > b))

Note que o nível 2 se tornou igual ao nível 3 já que *card_type* tem somente uma fórmula. Assim, todas as combinações de *balance* e *cr* estão presentes.

3. Para cada elemento de Combinações_de_Fórmulas gerar um caso de teste na máquina de teste. Será mostrado apenas uma operação de teste, que equivale ao terceiro elemento de Combinações_de_Fórmulas para o nível 3 ou a terceira linha da tabela anterior. Essa operação será chamada de addCreditTest3 (Tabela 3.9).

Cada elemento de Variáveis_relevantes é um parâmetro da operação:

Tabela 3.9 – Parâmetros do método de teste da máquina de teste

addCreditTest3(cr, card_type, balance)

Para cada elemento de Variáveis_relevantes deve haver um parâmetro de saída da operação (Tabela 3.10):

Tabela 3.10 – Assinatura do método de teste da máquina de teste

cr_saida, card_type_saida, balance_saida ← **addCreditTest3**(cr, card_type, balance)

Cada elemento de Combinações_de_Fórmulas deve ser posto na pré-condição da operação, separados por "&". Também devem ser postos os elementos de Proposição_Resultante que ainda não foram adicionados (Tabela 3.11):

Tabela 3.11 – Pré-condição da operação de teste da máquina de teste (somente as cláusulas de tipagem)

```
cr_saida, card_type_saida, balance_saida ← addCreditTest3(cr, card_type, balance) =
   PRE
      cr:NAT1 & card_type:CARD_TYPES & balance:NAT &
      (!b. ((b: NAT1 & b /= cr) => cr < b)) &
      (!b. ((b: NAT & b /= balance) => balance > b)) &
      card_type /= gratutious_card &
      balance + Cr <= MAXINT &
      (not(card_type = gratutious_card) or balance = 0)
```

Adicionalmente deve-se colocar os elementos de Cláusulas_Não_Tipagem que tem relacionamento com as variáveis de intervalo dentro do domínio da variável b, na primeira parte da implicação, de acordo com a Tabela 3.12:

Tabela 3.12 - Pré-condição completa da operação da máquina de teste

```
cr_saida, card_type_saida, balance_saida ← addCreditTest3(cr, card_type, balance) =
   PRE
      cr:NAT1 & card_type:CARD_TYPES & balance:NAT &
      (!b. ((b: NAT1 & balance + b <= MAXINT & b/= cr) => cr < b))&
      (!b. ((b: NAT & b + cr <= MAXINT & b/= balance & (not(card_type = gratutious_card)
               or b = 0))=> balance > b)) &
```

```
            balance + cr <= MAXINT &
            card_type /= gratutious_card &
            (not(card_type = gratutious_card) or balance = 0)
```

Os parâmetros de entrada devem ser passados, no corpo da operação, aos parâmetros de saída, conforme Tabela 3.13:

Tabela 3.13 - Inserção dos parâmetros de saída no corpo da operação de teste

```
    cr_saida, card_type_saida, balance_saida ← addCreditTest3(cr, card_type, balance) =
       PRE
           cr:NAT1 & card_type:CARD_TYPES & balance:NAT &
           (!b. ((b: NAT1 & balance + b <= MAXINT & b/= cr) => cr < b))&
           (!b. ((b: NAT & b + cr <= MAXINT & b/= balance & (not(card_type = gratutious_card)
or b = 0))=> balance > b)) &
           balance + cr <= MAXINT &
           card_type /= gratutious_card &
           (not(card_type = gratutious_card) or balance = 0)
       THEN
           cr_saida := cr || card_type_saida := card_type || balance_saida := balance
       END
```

3.1.4.Particionar os valores de cada restrição de cada variável em classes de equivalência inválidas

- Entradas: `Cláusulas_Tipagem`, `Cláusulas_Não_Tipagem`, `Variáveis_relevantes`
- Saídas: `Combinações_Negativas` e máquina de teste para obtenção dos dados inválidos

A seleção das fórmulas para identificação de dados de teste inválidos é comum aos três níveis de cobertura. Nesta etapa serão utilizados os conjuntos `Clausulas_Tipagem` e `Cláusulas_Não_Tipagem` identificados nas seções anteriores.

Da mesma forma que na seção 3.1.3, esta etapa gerará casos de teste na máquina de teste negativa, só que agora esses casos de teste servirão para identificar casos de teste inválidos. Sendo assim devemos seguir o seguinte algoritmo:

Tabela 3.14 - Algoritmo da seção 3.1.4

1. Combinar os elementos de Cláusulas_Não_Tipagem entre si, negando um elemento em cada combinação, resultando no conjunto Combinações_Negativas;

2. Para cada elemento de Combinações_Negativas gerar um caso de teste na máquina de teste, onde cada operação de teste terá o nome da operação a ser testada seguido do nome "Teste" seguido de um seqüencial que é correspondente a posição do elemento de Combinações_Negativas, de forma que:

   Cada elemento de Variáveis_relevantes é um parâmetro da operação;

   Para cada elemento de Variáveis_relevantes deve haver um parâmetro de saída da operação;

   Cada elemento de Combinações_Negativas deve ser posto na pré-condição da operação, separados por "&" com os elementos de Cláusulas_Tipagem;

   Os parâmetros de entrada devem ser passados, no corpo da operação, aos parâmetros de saída.

EXEMPLO

Entradas:

- **Variáveis_relevantes** = `{Cr, card_type, balance}`
- **Cláusulas_Tipagem =** `{cr: NAT1, balance: NAT, card_type: CARD_TYPES}`
- **Cláusulas_Não_Tipagem** = `{card_type /= gratutious_card, balance + Cr <= MAXINT, (card_type = gratutious_card => balance = 0)}`

1. Combinar os elementos de Cláusulas_Não_Tipagem (Tabela 3.15) entre si, negando um elemento em cada combinação, resultando no conjunto Combinações_Negativas;

Tabela 3.15 - Combinação dos elementos de Clausulas_Não_Tipagem

1	**Not**(card_type /= gratutious_card)	balance + Cr <= MAXINT	(not(card_type = gratutious_card) or balance = 0)
2	card_type /= gratutious_card	**Not**(balance + Cr <= MAXINT)	(not(card_type = gratutious_card) or balance = 0)
3	card_type /= gratutious_card	balance + Cr <= MAXINT	**Not**((not(card_type = gratutious_card) or balance = 0))

A numeração da primeira coluna indica o n-ésimo elemento de Combinações_Negativas.

2. Para cada elemento de Combinações_Negativas gerar um caso de teste na máquina de teste, onde cada operação de teste terá o nome da operação a ser testada seguido do nome "Teste" seguido de um seqüencial que é correspondente a posição do elemento de Combinações_Negativas, de forma que:

   Cada elemento de Variáveis_relevantes é um parâmetro da operação;

   Para cada elemento de Variáveis_relevantes deve haver um parâmetro de saída da operação;

Cada elemento de Combinações_Negativas deve ser posto na pré-condição da operação, separados por “&” com os elementos de Cláusulas_Tipagem;

Os parâmetros de entrada devem ser passados, no corpo da operação, aos parâmetros de saída.

Mostrando a combinação 1 de Combinações_Negativas, teremos o seguinte caso de teste:

```
Cr_saida, balance_saida, card_types_saida ← addCreditTeste1 (Cr, balance, card_types)=
PRE
    cr: NAT1 &  balance: NAT & card_type: CARD_TYPES &
    Not(card_type /= gratutious_card) &
    balance + Cr <= MAXINT &
   (not(card_type = gratutious_card) or balance = 0)
THEN
    cr_saida := cr || card_type_saida := card_type || balance_saida := balance
  END
```

As máquinas de teste (positivas e negativas), bem como as combinações de dados geradas podem ser vistas no Apêndice.

3.1.5.Selecionar dados de teste abstratos

Após a geração das operações de teste devemos animar as máquinas de teste no ProB (utilizando seu solucionador de restrições). O ProB então fornece várias combinações de dados que satisfazem às restrições impostas pelas variáveis de cada operação de teste.

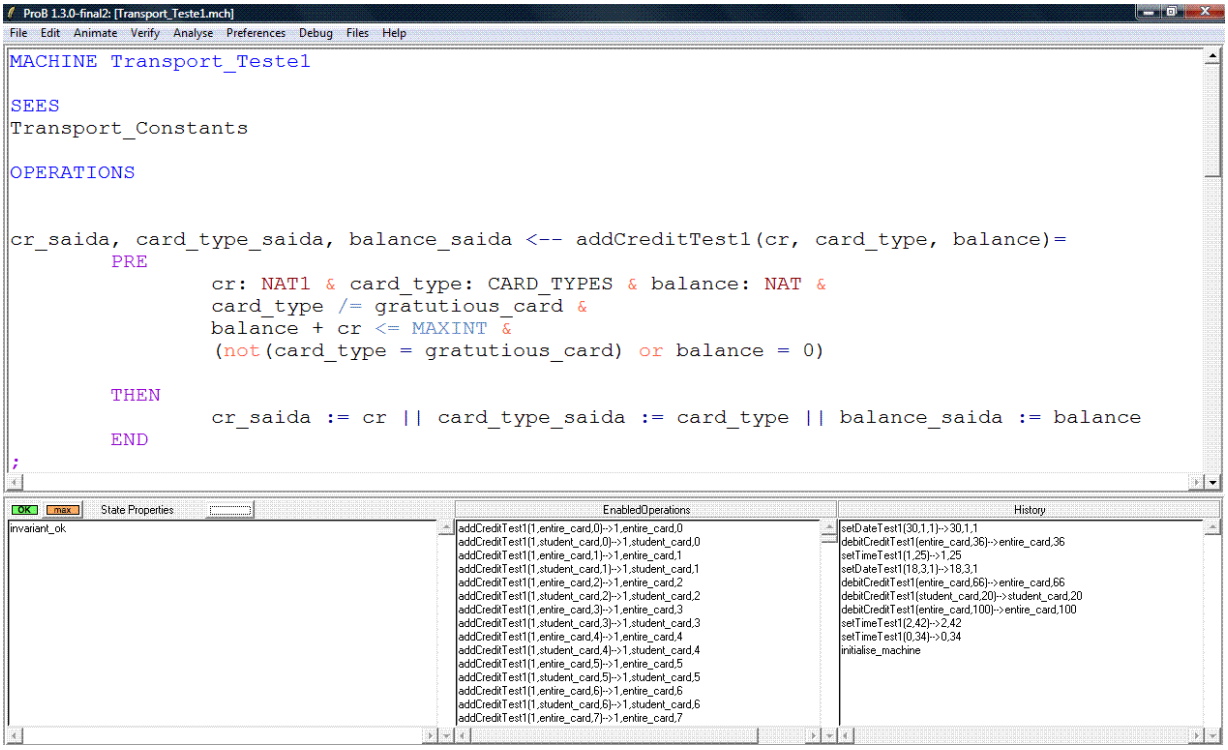


Figura 3.2 - Resultados do ProB

Vamos explicar agora as partes relevantes da figura acima: na parte superior do ProB está a especificação B que foi animada, na parte inferior esquerda informações sobre a quebra ou não do invariante, na parte inferior central, o resultado da animação da máquina (Enabled Operations – ou seja, as operações escolhidas pelo usuário para serem animadas).

Com o ProB podemos configurar algumas variáveis, como por exemplo, os valores máximo e mínimos de tipos da variáveis de intervalo. Assim os dados obtidos, no caso de trabalharmos com inteiros por exemplo, já são dados concretos. Em contra-partida, se trabalhamos com uma variável de conjunto em B, como por exemplo, x = {a, b, c}, esses dados somente serão dados concretos na implementação da máquina de teste e conseqüentemente somente serão dados de teste concretos após refinamentos da máquina de teste.

No exemplo de *Transport* foi utilizado *byte* para a tipagem dos tipos para a máquina de teste. Dessa forma, na máquina de teste o valor de MAXINT foi configurado para 128. A utilização de bytes foi feita para agilizar o tempo de animação.

Algumas operações têm mais de uma combinação de dados de teste como saída, ou seja, mais de uma combinação de dados satisfazem as restrições da operação de teste. Deve-se escolher apenas uma combinação para cada operação de teste, nestes casos, aleatoriamente. Com a escolha aleatória, pode

acontecer de se escolher duas combinações iguais para casos de teste diferentes (casos de teste da máquina de teste).

Por exemplo, para a operação *addCredit3* foram geradas 556 combinações, das quais algumas são mostradas no apêndice. A título de exemplo, duas delas são mostradas a seguir:

Combinação 1: cr=1, balance=126, card_type= entire_card

Combinação 2: cr=1, balance=126, card_type= student_card

Aleatoriamente foi escolhida a combinação 1.

3.1.6.Implementar e executar casos de teste concretos

Foi gerado, de forma manual, a implementação da máquina *Transport* (ver Apêndice) e casos de teste com JUnit com os dados obtidos com a animação da máquina de teste.

A máquina *Transport* foi inicialmente especificada em B e foi gerado código em Java Card (Gomes, 2005). Neste trabalho, este código foi transformado em código Java por facilidade de manipulação. Os casos de teste foram desenvolvidos em JUnit, sendo um caso de teste JUnit para cada operação de teste da máquina B de teste. A transformação da máquina B de teste para o código de teste foi feita de maneira não sistematizada, ou seja, o desenvolvedor cria a classe de teste.

Na Tabela 3.16 pode-se ver o método de teste em JUnit para a operação de teste addCredit3 do nível 3 de cobertura, para os dados de teste escolhidos:

Tabela 3.16 - Caso de teste gerado no JUnit

```
public void testAddCredit3() {
      t.setBalance((byte)126);
      t.setCardType((byte)Constants.ENTIRE_CARD);
      t.addCredit((byte)1);

      Assert.assertEquals(t.getBalance(), 127);
}
```

O *Assert.assertEquals* dá o resultado do teste (passou ou falhou). Note que os dados de entrada são fornecidos pela animação na máquina de teste gerada. Já os resultados esperados, neste caso o 127, foram obtidos a partir de animações na máquina sob teste.

3.2. Considerações Finais

Neste capítulo foi mostrado um método para geração de casos de teste a partir de especificações em B. Para geração desse método foi utilizado como estudo de caso uma aplicação de bilhetagem eletrônica para transporte coletivo urbano. Consideramos para o método que as informações mais relevantes para teste seriam a pré-condição da operação e o invariante da máquina.

Com a escolha das estruturas de B a serem trabalhadas para a geração de testes, algumas fórmulas são encontradas. Essas fórmulas precisam ser combinadas de maneira que cada combinação diga respeito a uma condição de teste, ou seja, uma situação específica a ser testada. Sendo assim, para especificações com muitas fórmulas, essa combinação levaria um tempo considerável para ser findada. Foi desenvolvida então uma pequena aplicação onde o testador digita as fórmulas e a aplicação as combina. Essa aplicação não foi ainda implementada para suportar a combinação de fórmulas de acordo com os pares ortogonais (nível 2 de cobertura definido).

Solucionado o problema da morosidade para combinar fórmulas, as operações de teste são geradas e animadas na ferramenta ProB. Daí surgiu um problema: muitas dessas operações de teste não resultavam em dados de teste, ou seja, as fórmulas combinadas entre si eram insatisfatíveis. Foi percebido que na maioria das vezes que isto acontecia, era devido a restrições sobre variáveis de limites, ou seja, para que se pudesse pegar o limite máximo de uma variável, por exemplo, x=MAXINT, isto fazia com que outras variáveis relacionadas a x, como por exemplo, y>x + 1, tivesse um resultado insatisfatível por y não poder ser maior que MAXINT na linguagem B.

Sendo assim foi pensado na idéia de limites relativos para as variáveis. No exemplo acima, como x se relaciona com y, o limite superior absoluto de x nunca seria um dado de teste que satisfizeaae às duas restrições. Com limites relativos outros dados de teste para x e y são procurados de modo que x é o maior possível e depois y é o maior possível. Os limites relativos nos dão agora resultados para todas as combinações de fórmulas que são oriundas das especificações sob teste.

São gerados dados de teste abstratos e em seguida concretos para cada operação sob teste. Os dados de teste concretos podem ser postos no JUnit, juntamente com o resultado da animação da máquina sob teste (que nos fornece os resultados esperados) e assim o JUnit nos fornece os resultados de teste (passou ou falhou) para cada caso de teste.

Todos os resultados obtidos e comparações com trabalhos anteriormente desenvolvidos, bem como nossas principais contribuições podem ser vistas nos próximos capítulos.

Capítulo 4 Trabalhos Relacionados

Atualmente existem inúmeros trabalhos sendo desenvolvidos na área de Testes Baseados em Modelos (MBT). Um trabalho inicial na geração de testes a partir de UML e OCL é o (Briand & Labiche, 2001). Já em (Barbosa, 2005) é apresentada a ferramenta SPACES que suporta um método para geração, execução e análise dos resultados de testes funcionais para componentes de *software*. (Barbosa, 2005) gerou testes a partir de análise de partições, incrementando assim o trabalho de (Briand & Labiche, 2001). A idéia principal da SPACES é aproveitar artefatos desenvolvidos ao longo do desenvolvimento do *software*, os diagramas UML com restrições OCL, gerando testes automaticamente e, dessa forma, não gerar trabalho adicional para o testador. Os casos de testes identificados pela ferramenta podem ser selecionados no teste ou não pelo testador, de acordo com sua experiência. Um conjunto de cenários forma um grafo, onde cada vértice tem uma probabilidade de ser visitado. O usuário pode alterar essa probabilidade se quiser que aquele cenário seja testado mais vezes, com mais dados de entrada. A validação do método foi feita ao longo do desenvolvimento de um *software* livre. Este *software* consiste em um sistema WEB responsável pelo gerenciamento centralizado das reservas de uma cadeia de hotéis.

Na linha de análise de partições, outros trabalhos significativos, que mais evoluíram e que se assemelham com o presente trabalho por gerarem testes a partir de especificações B são os trabalhos de (Legeard & Peureux, 2001) (Legeard, Peureux, & Utting, 2002) (Bouquet, Legeard, & Peureux, 2002) (Ambert, et al., 2002). Nestes são apresentados um método e uma ferramenta para teste caixa preta, utilizando as técnicas de particionamento de equivalência e análise do valor limite, a partir de especificações Z e B. O método foi definido utilizando como estudo de caso um *software* para *Smart Cards*. A ferramenta se chama BZ-TT (*B and Z Testing Tool*). Ela é baseada em resolução de restrições. Os trabalhos que originaram a BZ-TT compararam os testes especificados manualmente com os testes gerados pela ferramenta e mostraram que os automatizados cobriram cerca de 85% dos manuais gerados por testadores experientes. A ferramenta gerou também cerca de 50% de testes que não foram especificados manualmente. Sendo assim, esses testes a mais têm chances de detectar diferentes falhas, logicamente que dependendo da qualidade desses testes.

Entraremos agora em detalhes dos métodos destas duas ferramentas – SPACES e BZ-TT, falando basicamente de como elas tratam as valores limites e as dependências entre variáveis, de como o critério de cobertura e dados de teste válidos e inválidos são definidos, bem como a definição do resultado esperado.

Valores Limites e Dependências entre Variáveis

Dos trabalhos citados, SPACES foi o primeiro a se preocupar com seleção de valores limites. Para isto a seleção de dados de teste deveria ser feita de forma manual pelo testador. Automaticamente somente era possível a seleção de dados aleatórios obtidos com o auxílio da ferramenta Dresden OCL (Dresden-OCL, Último acesso em: 20/ 02/ 2010), que interpreta restrições de especificações OCL. Já no que diz respeito a B, (Legeard & Peureux, 2001) (Legeard, Peureux, & Utting, 2002) (Bernard, Legeard, Luck, & Peureux, 2004) (Ambert, et al., 2002) foram as primeiras pesquisas encontradas a desenvolverem testes a partir da análise do valor limite. No presente trabalho, os dados limites são encontrados sistematicamente, se assemelhando a SPACES e a automatização é citada como trabalho futuro.

Nos trabalhos relacionados, bem como neste, são definidos casos de teste onde as variáveis de estado e/ ou os parâmetros da operação são levados a valores limites em suas faixas. O que difere os dois trabalhos do nosso é referente à análise de estruturas internas ao corpo das operações, como por exemplo, estruturas condicionais na definição das partições. Este trabalho, por restrições de tempo, não fez este tratamento e isto está sugerido como trabalho futuro.

Por outro lado, as duas ferramentas apresentadas não tratam de valores limites dependentes, o que é uma importante contribuição do presente trabalho. Neste trabalho, o método gera dados de teste para qualquer combinação de fórmulas das operações de testes, o que os trabalhos relacionados não fazem.

Critério de Cobertura

O critério de cobertura do BZ-TT considera todos os estados da máquina e todas as variáveis de entrada das operações com valores limites. Já o do SPACES possibilita a interação do usuário na escolha dos estados, o que faz com que se tenha uma maior flexibilidade, não se tendo um critério de cobertura fixo. Nosso trabalho tem um critério de cobertura intermediário, já que oferece três opções de cobertura, mas não deixa livre como o SPACES. A vantagem de não se ter apenas um critério de cobertura como foi proposto no nosso método é que um testador pode, a partir de características da sua especificação, encontrar dados de teste de forma que se teste mais ou menos a robustez do sistema. Já se ter um critério de cobertura não definido pode se tornar insatisfatório quando o testador não é experiente. Ele pode escolher um subconjunto de testes não significativos o bastante para a aplicação em questão. No nosso trabalho, mesmo o testador não sabendo investigar as características do *software* para que se escolha o nível de cobertura adequado, ele pode optar pelo nível mais robusto e assim ter uma maior segurança quanto aos resultados do teste. Sendo assim, propomos um critério de cobertura intermediário entre as duas ferramentas citadas.

Definição dos Dados de Teste

Tanto SPACES quanto nosso trabalho, utilizam solucionadores de restrições já desenvolvidos (para domínios booleanos finitos) para definição de dados de teste. Para BZ-TT foi desenvolvido um solucionador de restrições próprio para resolver características particulares de especificações B, manipulando restrições sobre conjuntos e relações. No presente trabalho foi utilizada a ProB, que é adequado às construções de B já que foi desenvolvido especificamente para animar especificações B. Esta ferramenta, em detrimento das demais apresentadas, tem a facilidade de modularidade (utilização de máquinas relacionadas em arquivos diferentes) e reconhece tipagem implícita. Nas outras ferramentas, os limites deveriam ser explicitamente especificados. Também reconhece os valores máximos e mínimos que são inseridos na animação pelo usuário. Isso nos traz a vantagem da diminuição do *gap* entre a especificação de alto nível e a linguagem de código quando colocamos os limites das variáveis como o mesmo valor dos limites das variáveis na linguagem de código. No entanto os três trabalhos apresentam o problema comum a MBT relativo ao *gap* existente entre dados abstratos e concretos quando a distância entre os tipos de dados de especificação e o de implementação é maior do que simplesmente limites.

Escolha de Dados de Entrada Válidos e Inválidos

Em BZ-TT a definição dos casos de teste válidos e inválidos é feita automaticamente. Os válidos são aqueles em que tanto os estados de fronteira quanto os parâmetros da operação com dados de fronteira contêm dados válidos para teste. Os casos de teste inválidos são aqueles que têm os estados de fronteira válidos e os parâmetros da operação com dados inválidos. Não são testadas variáveis globais com valores inválidos. Já a ferramenta SPACES define dados válidos automaticamente, mas os inválidos devem ser inseridos pelo usuário. Sendo assim, nosso trabalho, em detrimentos dos outros dois trabalhos apresentados, define dados inválidos para variáveis globais. Sabemos que com variáveis globais com valores inválidos de teste, na especificação B, as obrigações de provas não serão satisfeitas, já que o invariante será quebrado, mas quando estivermos diante de um programa executável, não é garantido, por exemplo, que o usuário insira como dado de entrada para uma variável global um valor dentro do domínio válido da variável. Isso é coberto no nosso trabalho, onde atentamos para a robustez do *software*. A definição de dados inválidos de teste é uma contribuição importante do presente trabalho.

Definição dos Resultados Esperados

No que diz respeito a resultados esperados para dados válidos, os três trabalhos os definem com animações das especificações do sistema sob teste. Quanto aos dados inválidos, os resultados esperados devem ser postos pelo testador. Nos três trabalhos, esses resultados esperados, assim como os dados de

entrada, correspondentes a cada caso de teste, são dados abstratos, que podem precisar ser refinados para a definição dos dados concretos de teste. Os casos de teste, em SPACES bem como no nosso trabalho, são gerados em JUnit e em BZ-TT não encontramos esta informação.

Capítulo 5 Apresentação e Análise dos Resultados

Neste capítulo serão apresentadas as principais contribuições deste trabalho, bem como os resultados obtidos.

5.1. Principais contribuições

Diante dos trabalhos relacionados podemos considerar as seguintes contribuições do presente trabalho:

- **Sistematização da identificação das restrições das variáveis** através do particionamento de equivalência e análise do valor limite, bem como a identificação das classes de equivalência a partir dessas restrições. Nas referências (Black, 2003) (Burstein, 2003) (Craig & Jaskiel, 2002) (Myers, 2004) (Pressman, 2005), as técnicas de teste caixa preta citadas são descritas sem nenhuma sistematização. Já em (Legeard, Peureux, & Utting, 2002) (Bernard, Legeard, Luck, & Peureux, 2004) (Ambert, et al., 2002) não é mostrado como as restrições são identificadas;
- **Dependências entre variáveis**: no caso de geração de testes através da análise de valor limite em que as restrições especificadas relacionam o valor de duas variáveis, ou seja, onde o valor de uma delas tem influência sobre os valores válidos da outra, nossa proposta evita que casos de teste deixem de ser gerados, identificando valores máximos e mínimos relativos às condições conjuntas;
- **A geração de casos de teste considerando apenas as variáveis globais relevantes**, diferentemente de (Legeard, Peureux, & Utting, 2002) (Bernard, Legeard, Luck, & Peureux, 2004) (Ambert, et al., 2002) que geram testes para todas as variáveis globais, independentemente se as variáveis influenciam na operação a ser testada;
- **Níveis de cobertura**: neste trabalho o testador pode escolher entre três níveis de cobertura. O nível mais apropriado depende da quantidade de casos de teste que se deseja (se o testador deseja testes mais ou menos robustos), da técnica de teste caixa preta que deseja utilizar (particionamento de equivalência e análise do valor limite) e das dependências entre as variáveis. Os trabalhos estudados apresentam apenas uma opção de nível de cobertura;

- **Escolha dos dados de teste válidos e inválidos**: neste trabalho, os dados válidos são levados aos limites, tanto para variáveis globais quanto para parâmetros das operações. Os dados de teste para as classes de equivalência inválidas são escolhidos de modo a quebrar o invariante.

O principal ponto pendente do presente trabalho é a não consideração de estruturas além da pré-condição das operações e do invariante, ou seja, o comportamento especificado para a operação sob teste não foi considerado na definição das classes de equivalência. Também não foi automatizada a maior parte do método (com exceção da obtenção dos dados de teste e das combinações das restrições de teste). Com relação à sistematização, as partes sistematizadas são apresentadas pelas caixas claras e as não sistematizadas são apresentadas pelas caixas escuras da Figura 5.1. As partes mecanizadas, que utilizam respectivamente ProB e JUnit, são as caixas 6 e 7.

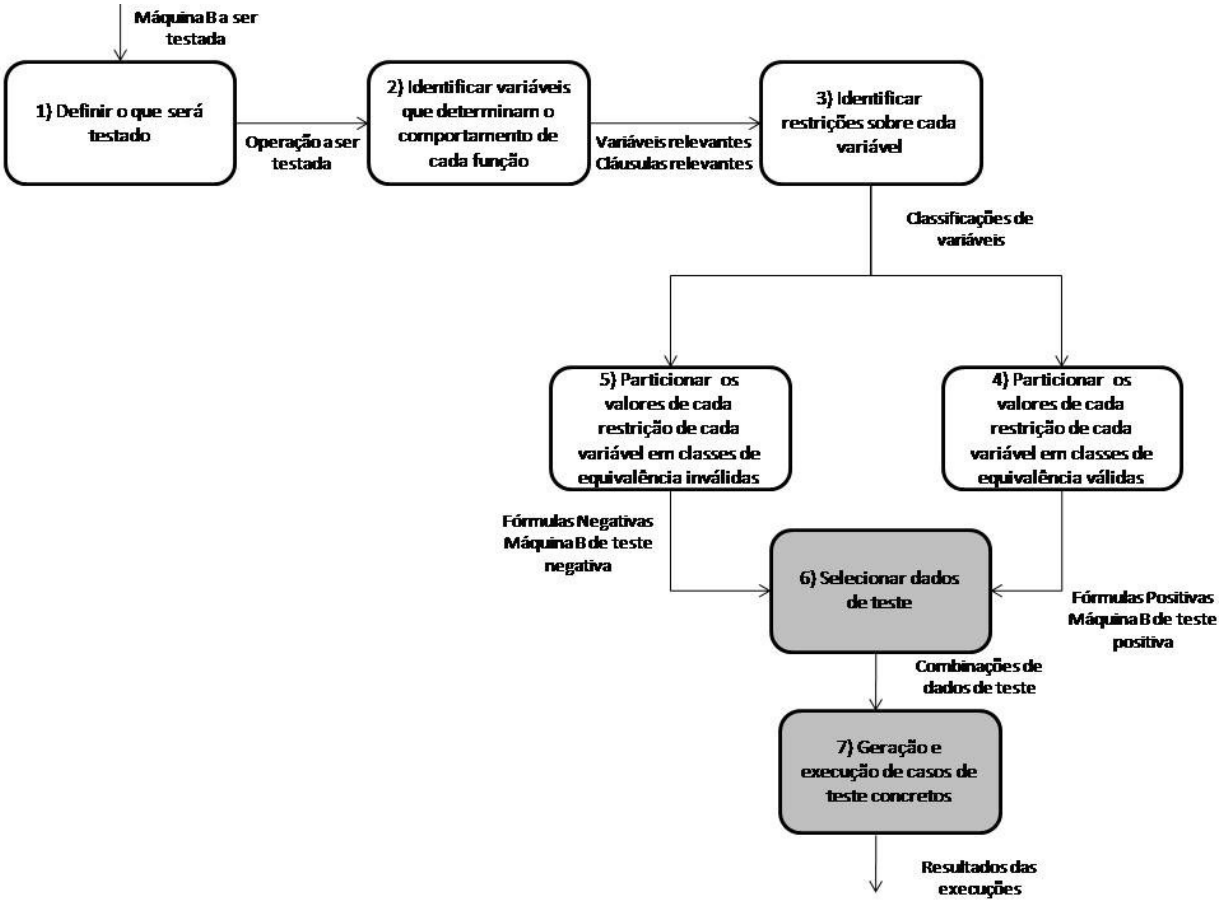


Figura 5.1 - Partes sistematizadas do método

Vale ressaltar que na caixa 6 o que não foi sistematizado foi a escolha dos dados de teste dentre os dados gerados automaticamente pelo solucionador de restrições do ProB (por limitações de tempo).

5.2. Resultados obtidos – Estudo de Caso

Com relação ao estudo de caso, foram obtidos os seguintes números (ver Apêndice para detalhes):

- Quantidade de operações de teste positivas (em B) para o nível 1: 5;
- Quantidade de operações de teste positivas (em B) para o nível 2: 21;
- Quantidade de operações de teste positivas (em B) para o nível 3: 25;
- Quantidade de operações de teste (em B) inválidas: 13.

Todas as operações positivas tiveram combinações de dados de teste como resposta da animação. Cabe ao testador escolher somente uma combinação de dados para cada operação de teste.

Nas operações válidas, comparando os três níveis de cobertura, tivemos uma proporção de cinco vezes mais operações de teste entre os níveis 1 e 3 e de 4,2 vezes do nível 1 para o 2. Veja que do nível 2 para o três a quantidade de operações de teste é próxima. Isso se deve ao fato de que as variáveis, em sua maioria, têm apenas dois valores de teste, um válido e um inválido. Como explicado no capítulo anterior, o nível 2 de cobertura só faz uma redução significativa da quantidade de combinações de dados de teste para variáveis com várias restrições para teste combinadas a outras também com muitas restrições.

Já para as operações inválidas, as animações não tiveram resultados. Essas são aquelas que o conjunto de restrições é insatisfatível. Foram 3 operações de teste insatisfatíveis, ou seja, cerca de 23% das operações de teste. Foram elas *changeCardTypeTest1*, *setTimeTest1* e *setTimeTest3*, da máquina de Testes Inválidos (ver Apêndice).

Foram construídos casos de teste com JUnit a partir dos dados de teste obtidos nas animações. Isso foi feito de maneira não sistemática. Sendo assim existiu um caso de teste no JUnit para cada operação de teste da máquina de teste, somente para as operações satisfatíveis. Isso significa que para as operações da máquina inválida insatisfatíveis não tivemos casos de teste no JUnit.

Comparando estes resultados dos testes válidos (quantidade de operações de teste satisfatíveis) com os resultados obtidos no estudo de caso que não considerava os limites relativos temos conforme Tabela 5.1:

Tabela 5.1 – Comparação entre os resultados dos testes válidos

Nível	Estudo de caso sem os limites relativos (EC1) – quantidade de operações satisfatíveis	Estudo de caso com os limites relativos (EC2) – quantidade de operações satisfatíveis	Quantidade de operações de teste geradas (a mesma quantidade nos dois estudos de caso)
N1	5	5	5
N2	14	21	21

N3	18	25	25

A quarta coluna traz o número de operações geradas para a definição dos casos de teste. A quantidade de operações é a mesma quando olhamos um mesmo nível. O que difere são o número de operações satisfatíveis e insatisfatíveis de cada estudo de caso.

Sendo assim, a comparação entre porcentagem de operações de teste satisfatíveis para os estudos de caso EC1 e EC2, por nível, pode ser visto na Figura 5.2:

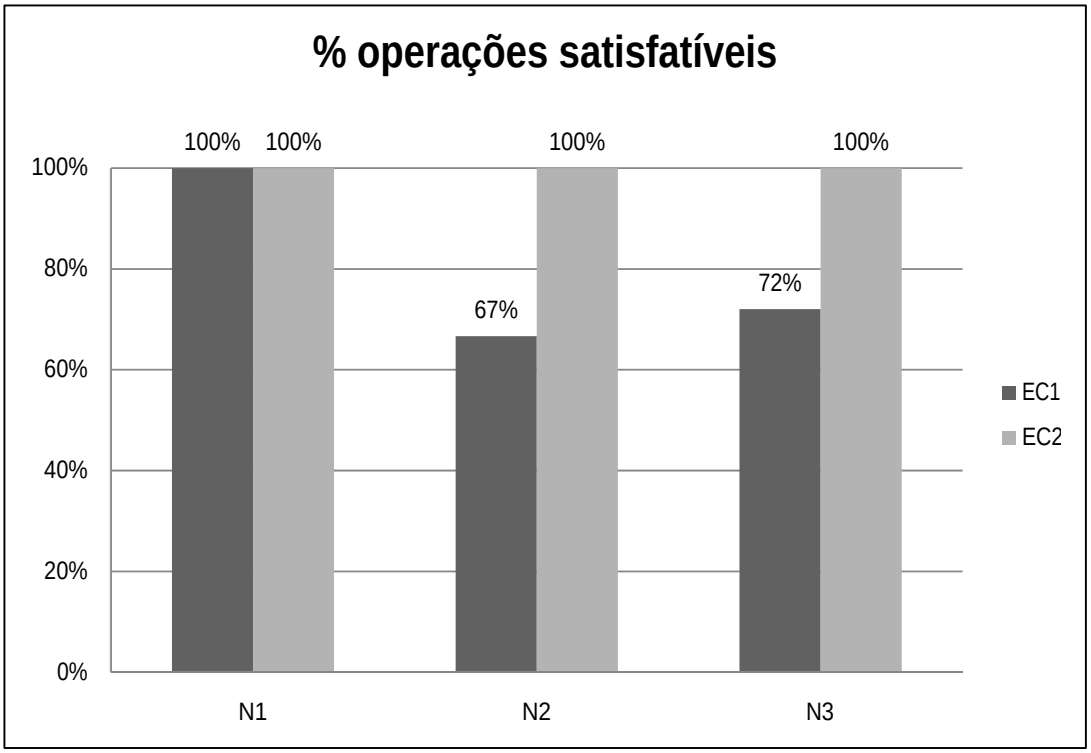


Figura 5.2 – Comparação entre as porcentagens de operações de testes insatisfatíveis

Para os testes inválidos os limites relativos não fizeram diferença na quantidade de operações de teste satisfatíveis.

Com relação aos resultados dos testes do JUnit, tivemos um total de 100% de casos de teste válidos passados (para todos os níveis de cobertura). Para testes inválidos, de um total de 10 casos de teste gerados, tivemos 2 casos de teste falhos (20%). Como o código gerado aqui partiu da máquina abstrata, , ou seja, foi implementado manualmente, o codificador deixou de considerar uma cláusula da operação e isso ocasionou as falhas. A cláusula não considerada foi a `cr + balance <= MAXINT`. Os casos de teste que falharam são os seguintes:

```
public void testAddCredit2() {
            t = new Transport();
            t.setBalance((byte)127);
            t.setCardType((byte)Constants.ENTIRE_CARD);
            t.addCredit((byte)2);

            Assert.assertEquals(t.getBalance(), 127);
      }

      public void testAddCredit3() {
            t = new Transport();
            t.setBalance((byte)126);
            t.setCardType((byte)Constants.STUDENT_CARD);
            t.addCredit((byte)3);

            Assert.assertEquals(t.getBalance(), 126);

      }
```

A tela do JUnit para os casos de teste inválidos rodados pode ser visualizada na Figura 5.3.

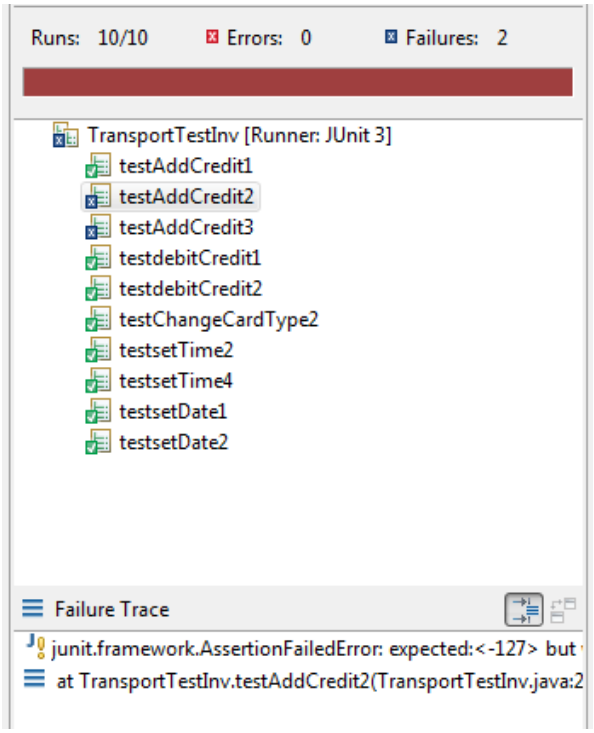


Figura 5.3 - Resultado da execução dos casos de teste no JUnit

Essa quantidade de falhas identificadas diz respeito a trechos da especificação que não foram postos no código gerado, isso porque a máquina abstrata não foi refinada até o nível de geração de código. Isto pode ser visualizado na Figura 5.4.

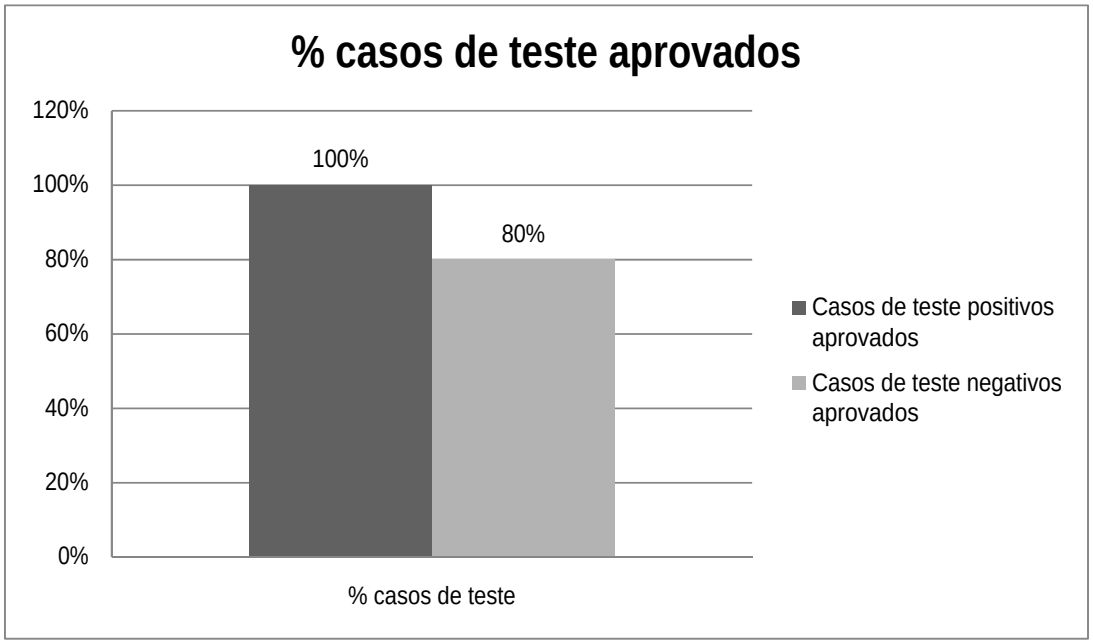


Figura 5.4 - Porcentagem de casos de teste aprovados

Sendo assim conclui-se que mesmo com a utilização de especificações formais para redução de erros no início do desenvolvimento do *software*, testes ainda são necessários ao final para encontrar possíveis falhas escapadas.

Para as pendências citadas na Figura 5.1, a seguir, pode-se visualizar uma arquitetura proposta.

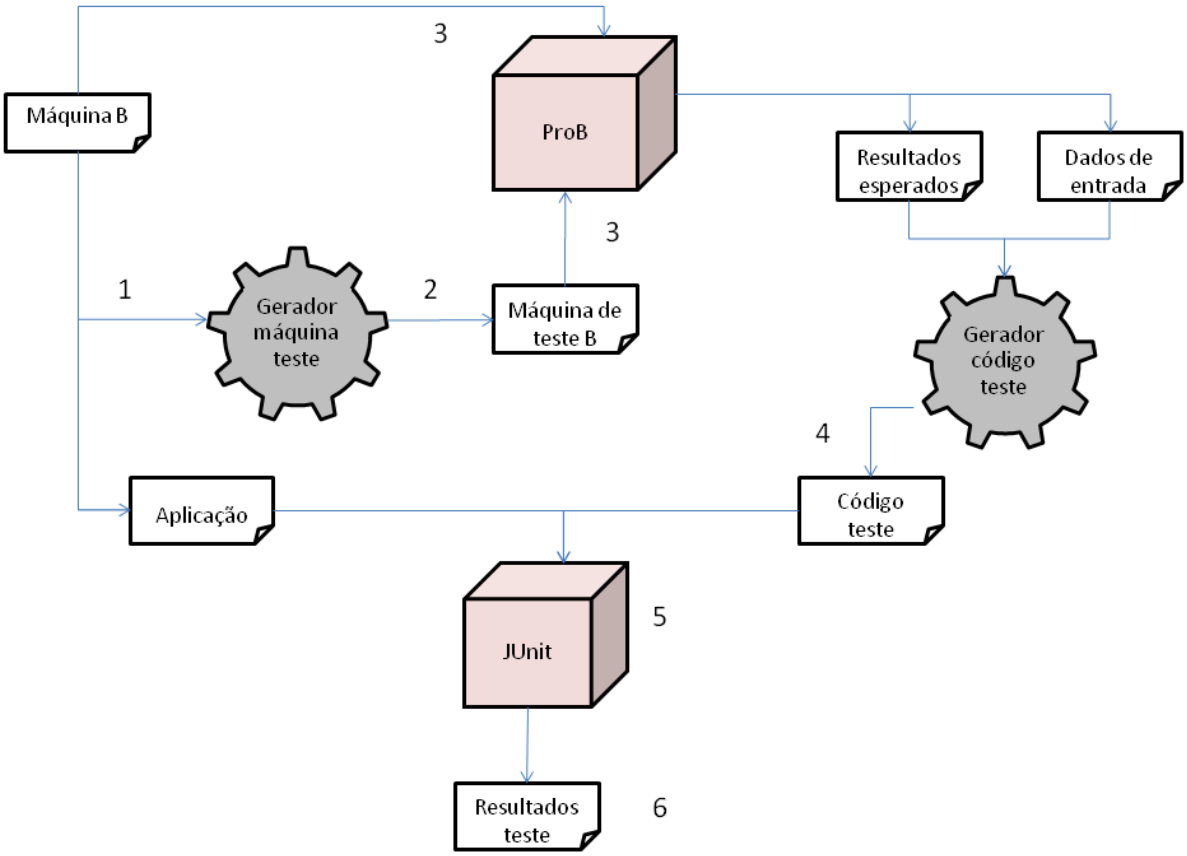


Figura 5.5 - Arquitetura proposta para trabalhos futuros

1. A máquina B deve ser passada no Gerador da máquina de teste. A partir de agora chamaremos a máquina B de máquina sob teste;
2. A máquina sob teste, após a aplicação do Gerador da máquina de teste, originará a Máquina de teste B;
3. As duas máquinas serão animadas pelo ProB resultando nos resultados esperados do teste, a partir de animações da máquina sob teste, e dados de entrada para os testes, a partir de animações da máquina de teste. Os resultados esperados para dados de entrada inválidos devem ser inseridos pelo testador;
4. De posse dos resultados esperados e dados de entrada para as operações a serem testadas, um Gerador de código de teste deve ser aplicado, resultando no código de teste que é um código JUnit. Esta parte não foi sistematizada pelo método proposto neste trabalho;
5. O código deve então ser executado pelo JUnit, comparando-se com as resultados obtidos com a execução da aplicação.

Os resultados de teste, ou seja, os testes que passaram e falharam, serão então mostrados pelo JUnit.

Capítulo 6 Considerações Finais

Os testes de *software* utilizados juntamente com especificações formais são uma parceria para o aumento da qualidade de um *software*. Com o surgimento dos TBM (Testes Baseados em Modelos), essa parceria vem se tornando cada vez mais difundida. Neste trabalho os testes são gerados a partir de especificações formais em B. O critério de cobertura de geração de testes adotado é baseado em análise de partições, com a utilização da técnica de teste caixa preta do particionamento de equivalência, bem como da análise do valor limite.

A aplicação do particionamento de equivalência em TBM pode ser otimizado quando se escolhe, para uma partição com idéia de limites, dados nas bordas dos domínios dessa variável. Isto faz com que o critério de escolha de dados dentro das partições não mais seja feito de forma aleatória. Neste trabalho sistematizamos as técnicas de particionamento de equivalência e análise do valor limite, o que não é facilmente encontrado na literatura.

O método proposto analisa uma máquina B, extrai dela as condições para a escolha de dados de teste para teste de cada operação. Elas são então combinadas e animadas no ProB, resultando em dados de teste. Esses dados de teste, após transformados em dados concretos (manualmente), são postos como entrada dos casos de teste em JUnit e o resultado esperado é obtido através de animações da máquina sob teste (também essas animações trazem dados abstratos que devem ser transformados em concretos e então utilizados aqui).

Os casos de teste gerados são independentes de linguagem de implementação. São gerados casos de teste válidos (casos que contêm apenas dados válidos), e casos de teste inválidos (casos de teste com dados inválidos). Os casos de teste são gerados para cada operação da máquina sob teste. Para isto, primeiramente é preciso identificar e selecionar as variáveis e cláusulas relevantes para teste de cada operação. Isto porque, ao invés do que foi feito por (Ambert, et al., 2002) (Bernard, Legeard, Luck, & Peureux, 2004) (Legeard, Peureux, & Utting, 2002), que consideravam para a escolha de dados de teste todas as variáveis da pré-condição da operação e todas as variáveis do invariante da máquina, no método proposto, apenas as variáveis direta ou indiretamente relacionadas às variáveis da pré-condição da operação são consideradas, selecionando somente as cláusulas do invariante que restringem o valor dessas variáveis. Com isso o animador trabalha com menos condições para a busca de dados de teste e assim o tempo de processamento é menor.

As cláusulas relevantes para teste são cláusulas de tipagem e cláusulas relacionadas a estas de tipagem. Nos trabalhos citados anteriormente, todas as tipagens deveriam ser feitas explicitamente, ou seja, uma situação onde $x \in S$ deveria ser posta e não uma situação do tipo $x \in 1..10$, que tem uma condição adicional à tipagem. No nosso trabalho, a tipagem não precisa ser explícita, já que o ProB consegue

interpretar esse tipo de cláusula como x sendo do tipo Z e tendo a restrição que x varia de 1 a 10. Assim sendo, não existem estilos especiais de escrita de especificação a serem adotados para que o nosso método proposto funcione.

Neste trabalho são propostos três níveis de cobertura de teste (todos os trabalhos pesquisados têm apenas um nível de cobertura): o nível 1 que estabelece a cobertura por particionamento de equivalência; o nível 2 que oferece cobertura através do particionamento de equivalência, análise do valor limite e as combinações de condições de teste são feitas através dos pares ortogonais; o nível 3 que trabalha também com particionamento de equivalência, análise do valor limite e as combinações de condições de testes são feitas como um produto cartesiano das condições. Todos os níveis trabalham com limites relativos quando absolutos não são possíveis.

Aqui também é tratada a geração de dados de teste para variáveis dependentes. Isso é feito utilizando a idéia de limites relativos, quando os limites absolutos não são satisfatíveis. Com isso obtivemos a geração de combinações de casos de teste para todas as operações de teste positivas especificadas. Comparando com um estudo de caso anterior feito sem a utilização de limites relativos, somente com limites absolutos, passamos de 67% de condições de teste satisfatíveis no nível 2 de cobertura com a análise do valor limite para 100%. Também no nível 3 passamos a 100% um total de 72% de condições satisfatíveis. O nível 1 não teve diferença nos dois estudos de caso, já que não é utilizada a análise do valor limite. Para dados de teste inválidos ainda temos um total de 23% de operações insatisfatíveis, o que não conseguimos reduzir no segundo estudo de caso. Casos de teste gerados no JUnit e rodados, tivemos ainda 20% de falhas encontradas nas operações negativas. Essas falhas encontradas dizem respeito a condições que existiam na máquina abstrata e que não existiam no código sob teste. As operações positivas não geraram falhas.

Principais utilizações deste trabalho

O estudo de caso utilizado provou que mesmo com a utilização de especificações formais, os testes ainda são capazes de evidenciar falhas. Sendo assim, este trabalho é interessante quando se quer:

- Verificação de robustez dos sistemas desenvolvidos, e refinados até a geração de código, a partir de especificações formais. Isto porque nem sempre todos os casos inválidos são tratados na especificação e conseqüentemente na aplicação. O método proposto consegue descobrir combinações de dados a partir das pré-condições da aplicação que podem não ter sido tratadas na implementação da aplicação e assim quebrar seu o funcionamento correto.
- Verificação de correção dos sistemas desenvolvidos, e não refinados até a geração de código, a partir de especificações formais. O nosso método, conforme mostrado no estudo de caso deste

trabalho, encontra problemas inseridos pelos codificadores que geram código manualmente a partir da especificação da máquina abstrata B;

- Verificar e validar um sistema que foi especificado formalmente em B. Isto porque, quando se tem uma especificação B, após todo o método ser automatizado, toda a geração de casos de teste e também a geração de relatórios de teste que mostram a qualidade do sistema, é feito somente com um comando do testador. Em outras palavras, para se testar um código feito a partir de uma especificação em B não seria preciso um esforço adicional do testador, a não ser um comando que gera resultados de qualidade do *software*, e para isso ainda o testador pode escolher entre um dos três níveis de cobertura propostos e mais ainda, nenhum estilo de escrita da especificação em B precisa ser adotado.
- Padronização dos testes por testadores diferentes e com níveis diferentes de experiência. Com o método sistematizado, testadores não tão experientes podem testar aplicações da mesma forma que um testador que conheça as técnicas de geração de testes, fazendo assim com que uma equipe de desenvolvimento de *software* trabalhe de maneira uniforme quanto aos testes. Com o método ainda não automatizado, o testador tem todo o passo a passo para a geração de seus dados e casos de teste. Sendo assim ele pode aplicar o método na sua especificação B e gerar testes para todo o seu *software*.

Principais limitações do método

Algumas limitações do método proposto são as seguintes:

- Não tratamento de condições encontradas no corpo das operações: se tivermos, por exemplo, condições tratadas por estruturas condicionais, estas condições não necessariamente serão testadas. O teste apenas vai acontecer se tivermos esta mesma condição na pré-condição da operação.
- Redundância nos dados de teste: com a seleção de combinações de dados de teste de maneira aleatória dentre as combinações geradas por uma operação de teste, operações de teste diferentes e com objetivos diferentes, eventualmente tem como resultado a mesma combinação de teste.

Trabalhos futuros sugeridos

Alguns trabalhos futuros sugeridos são:

- Automatizar as combinações das fórmulas para o nível 2 de cobertura: atualmente o combinador só está definido para os níveis 1 e 3;

- Seleção de dados de teste, de forma automática, entre as combinações de dados que resultam das animações do ProB. Atualmente, fórmulas diferentes podem ser animadas e terem algumas combinações iguais. Escolhendo-se uma combinação de cada fórmula aleatoriamente, pode acontecer de serem escolhidas as mesmas combinações, fazendo com que tenhamos redundância de dados. Propõe-se, como trabalho futuro, buscar uma heurística para esta escolha.
- Sistematização da passagem da especificação B de teste para código JUnit. Neste trabalho esta passagem foi feita de maneira não sistemática;
- Refinamento da máquina de teste de acordo com o refinamento da máquina sob teste. O método propõe, mas isto não foi feito no estudo de caso. Isto é importante para que a máquina para teste também seja refinada e dessa forma, o *gap* entre a especificação de alto nível e os casos de teste concretos sejam diminuídos;
- Desenvolvimento de outros estudos de caso, de maior complexidade, para que possamos aprimorar o método e tenhamos uma informação mais aprimorada da quantidade média de operações de teste geradas por operação a ser testada e quantidade média de operações inválidas insatisfatíveis.

Outra pendência, e a julgada mais importante é a automatização de todo o método e das demais pendências listadas.

Bibliografia

Abdurazik, A., & Offutt, J. (2000). Using UML Collaboration Diagrams for Static Checkig and Test Generation, York - UK. *In the Third International Conference on the Unified Modellin Language (UML00)* .

Abrial, J. (1996). *The B Book: Assigning Programs to Meanings.* Cambridge University Press.

Ambert, F., Bouquet, F., Chemin, F., Guenaud, S., Legeard, B., Peureux, F., et al. (2002). BZ-TT: A Tool-Set for Test Generation from Z and B using Constraint Logic Programming. *Proceedings of the CONCUR'02 Workshop on Formal Approaches to Testing of Software (FATES'02)* .

Barbosa, D. (2005). *Um Método Automático para Teste Funcional para Verificação de Componentes.* Dissertação apresentada a UFCG (Universidade Federal de Campina Grande).

Beck, K. (2004). *Extreme Programming Explained: Embrace Change.* Addison-Wesley Professional.

Bedregal, B., & Acióly, B. (2002). *Lógica para a ciência da computação.* Natal.

Benattou, M., Bruel, J., & Hameurlain, N. (2002). Generating Test Data from OCL Specification, France. *Internal Research Report R2I-02-01* .

Bernard, E., Legeard, B., Luck, X., & Peureux, F. (2004). Generation of Test Sequences from Formal Specifications: GSM 11-11 Standard Case-Study, France. *Rapport de Recherche n° RR 2004-16* .

Bertolino, A. (2004). Towards Anti-Models-Based Testing - Florence - Italy - June 28 - July 1. *DNS:International Conference on Dependable Systems and Networks* .

Black, R. (2003). *Critical Testing Process: Plan, Prepare, Perform, Perfect.* Addison Wesley.

Bouquet, F., Legeard, B., & Peureux, F. (April de 2002). CLPS-B - A Constraint Solver for B. *Proceedings of the conference of Tools and Algorithms for the Construction and Analysis of Systems* .

Briand, L., & Labiche, Y. (2001). A UML-based Aproach to System Testing. *Lecture Notes in Computer Science* , pp. 60-70.

Burstein, I. (2003). *Pratical Software Testing.* Artech House Publishers Boston.

Clearsy Systems Engineering. (s.d.). B Language Reference Manual, versão 1.8.5.

Conrad, M. (2002). Graph Transformations for Model-Based Testing. *Proceedings of Modellierung - GI-Lecture Notes in Informatics* , pp. 39-50.

Craig, R., & Jaskiel, S. (2002). *Systematic Software Testing.* Artech House.

Dresden-OCL. (Último acesso em: 20/02/2010). http://dresden-ocl.sourceforge.net.

Gamma, E. e. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison - Wesley professional computing series.

Gomes, B. (2005). *Geração de código Java a partir de Especificações Formais Elaboradas em Linguagem B.* Trabalho de Final de Curso (graduação) apresentado ao Departamento de Informática e Matemática Aplicada - DIMAP-UFRN.

Hartmann, J., Vieira, M., & Ruder, A. (2004). UML-based Test Generation and Execution, Berlin. *In Proceedings of the 21st Workshop on Software Test, Analisys and Verification* .

Hoare, C. (1969). *An Axiomatic Basis for Computer Programming.* ACM.

JUnit.org. (s.d.). JUnit - http://www.junit.org. *última visita em 18/01/2009* .

Kaner, C., Falk, J., & Nguyen, H. (1999). *Testing Computer Software.* New York: Wiley.

Legeard, B., & Peureux, F. (November de 2001). Generation of functional test sequences from B formal specifications - Presentation and industrial case-study. *Proceedings of the 16th International Conference on Automated Software Engineering (ASE'01)* .

Legeard, B., Peureux, F., & Utting, M. (2002). Automated Boundary Testing from B and Z. *FME - LNCS 2391* , pp. 21-40.

Legeard, B., Peureux, F., & Utting, M. (July de 2002). Automated boundary testing from Z and B. *Int. Conf. on Formal Methods Europe, FME'02, volume 2391 of LNCS, pages 21–40* .

Leuschel, M., & Butler, M. (September de 2003). ProB: A model-Checker for B. *FM 2003: 12th International FME Symposium* .

Lindholm, J. (2006). *Model Based Testing.* Relatório apresentado à University of Helsinki, Department of Computer Science - Helsinki.

Loveland, S. (2005). *Software Testing Techniques: Finding the Defects that Matter.* Massachusetts: Charles River Media.

Maldonado, D. (2007). *Introdução aos Testes de Software.* Campus.

Myers, G. (2004). *The Art of Software Testing.* New Jersey: John Wiley & Sons.

Offutt, J., & Abdurazik, A. (1999). Generating Tests from UML Specifications, Fort Collins. *In Proceedings of the Second IEEE International Conference on the Unified Modellig Language (UML99)* , pp. 416-429.

Peleska, J., & Siegel, M. (1996). From Testing Teory to Test Driver Implementation, England. *FME: Industrial Benefit and Advances in Formal Methods* , pp. 538-556.

Philipps, J., Pretschner, A., Slotosch, O., Aiglstorfer, E., Kriebel, S., & Scholl, K. (2003). Model-Based Test Case Generation for Smart Cards. *Eletronic Notes in Theoretical Computer Science , 80*.

Pressman, I. (2005). *Engenharia de Software.* Pearson Education do Brasil.

Rosenblum, D. (1996). Formal Methods and Testing: Why the state of art is not the state of pratic, San Francisco - California -United States/ 16-18 october. *Proceedings of the 4th ACM SIGSOFT symposium on Foundations of software engineering* , p. 64.

Ross, K. (1998). *Pratical Guide to Software Testing.* Ross and Associates Pty Ltd.

Salas, P., & Aichernig, B. (2005). *Automatic Test Case Generation for OCL: A Mutation Approach.* Macau: UNU-IIST Report No. 321.

Satpathy, M., Leuschel, M., & Butler, M. (2004). ProTest: An Automatic Test Environment for B Specifications, Barcelona - Spain. *In: International workshop on Model Based Testing, co-located with ETAPS 2004* .

Sheppard, D. (1995). *An Introduction to Formal Specification with Z and VDM.*

Souza, F. (2005). *Geração de Dados de Teste a Partir de Especificações feitas em Linguagem B.* Relatório final de curso (graduação) apresentado ao DIMAp - UFRN/Natal.

Spivey, J. (1992). *The Z Notation: A reference Manual.* Prentice-Hall International Series in Computer Science.

Third Workshop on Model Based Testing. (2007). *Eletronic Notes in Theorical Computer Science - LNCS* , pp. 1-126.

Tian, J. (2005). *Software Quality Engineering: Testing, Quality Assurance and Quantifiable Improvement.* John Wiley and Sons.

APÊNDICE

Máquinas B de Teste (somente addCredit)

Nível 1

```
MACHINE Transport_Teste1


SEES

Transport_Constants


OPERATIONS



cr_saida, card_type_saida, balance_saida <-- addCreditTest1(cr, card_type, balance)=

	PRE

		cr: NAT1 & card_type: CARD_TYPES & balance: NAT &

		card_type /= gratutious_card &

		balance + cr <= MAXINT &

		(not(card_type = gratutious_card) or balance = 0)


	THEN

		cr_saida := cr || card_type_saida := card_type || balance_saida := balance

	END

END
```

Níveis 2 e 3

```
MACHINE Transport_Teste2

SEES

Transport_Constants


OPERATIONS


cr_saida, card_type_saida, balance_saida <-- addCreditTest1(cr, card_type, balance)=

	PRE

		cr: NAT1 & card_type: CARD_TYPES & balance: NAT &
```

```
		card_type /= gratutious_card &
		balance + cr <= MAXINT &
		(not(card_type = gratutious_card) or balance = 0)

	THEN
		cr_saida := cr || card_type_saida := card_type || balance_saida := balance
	END
	;


cr_saida, card_type_saida, balance_saida <-- addCreditTest2(cr, card_type, balance)=
	PRE
		cr:NAT1 & card_type:CARD_TYPES & balance:NAT &
		(!b. ((b: NAT1 & b /= cr & balance + b <= MAXINT) => cr < b)) &
		(!b. ((b: NAT & b /= balance & b + cr <= MAXINT & (not(card_type = gratutious_card) or
b = 0)) => balance < b)) &
		card_type /= gratutious_card &
		balance + cr <= MAXINT &
		(not(card_type = gratutious_card) or balance = 0)

	THEN
		cr_saida := cr || card_type_saida := card_type || balance_saida := balance
	END
	;


cr_saida, card_type_saida, balance_saida <-- addCreditTest3(cr, card_type, balance)=
	PRE
		cr:NAT1 & card_type:CARD_TYPES & balance:NAT &
		(!b. ((b: NAT1 & balance + b <= MAXINT & b/= cr) => cr < b))&
		(!b. ((b: NAT & b + cr <= MAXINT & b/= balance & (not(card_type = gratutious_card) or
b = 0))=> balance > b)) &
		balance + cr <= MAXINT &
		card_type /= gratutious_card &
		(not(card_type = gratutious_card) or balance = 0)
```

```
        THEN
                cr_saida := cr || card_type_saida := card_type || balance_saida := balance
        END
        ;


cr_saida, card_type_saida, balance_saida <-- addCreditTest4(cr, card_type, balance)=
        PRE
                cr:NAT1 & card_type:CARD_TYPES & balance:NAT &
                (!b. ((b: NAT1 & b /= cr & balance + b <= MAXINT)=> cr > b)) &
                (!b. ((b: NAT & b /= balance & b + cr <= MAXINT & (not(card_type = gratutious_card) or
b = 0)) => balance < b))&
                card_type /= gratutious_card &
                balance + cr <= MAXINT &
                (not(card_type = gratutious_card) or balance = 0)

        THEN
                cr_saida := cr || card_type_saida := card_type || balance_saida := balance
        END
        ;

cr_saida, card_type_saida, balance_saida <-- addCreditTest5(cr, card_type, balance)=
        PRE
                cr:NAT1 & card_type:CARD_TYPES & balance:NAT &
                (!b. ((b: NAT1 & b /= cr & balance + b <= MAXINT)=> (cr > b)))&
                (!b. ((b: NAT & b /= balance & b + cr <= MAXINT & (not(card_type = gratutious_card) or
b = 0))=> (balance > b))) &
                card_type /= gratutious_card &
                balance + cr <= MAXINT &
                (not(card_type = gratutious_card) or balance = 0)

        THEN
                cr_saida := cr || card_type_saida := card_type || balance_saida := balance
        END
END
```

Máquina para dados inválidos

```
MACHINE Transort_Teste_Invalidos

SEES
 Transport_Constants

OPERATIONS

cr_saida, card_type_saida, balance_saida <-- addCreditTest1(cr, card_type, balance)=
        PRE
                cr: NAT1 & card_type: CARD_TYPES & balance: NAT &
                not(card_type /= gratutious_card) &
                balance + cr <= MAXINT &
                (not(card_type = gratutious_card) or balance = 0)

        THEN
                cr_saida := cr || card_type_saida := card_type || balance_saida := balance
        END
;

cr_saida, card_type_saida, balance_saida <-- addCreditTest2(cr, card_type, balance)=
        PRE
                cr: NAT1 & card_type: CARD_TYPES & balance: NAT &
                card_type /= gratutious_card &
                not(balance + cr <= MAXINT) &
                (not(card_type = gratutious_card) or balance = 0)

        THEN
                cr_saida := cr || card_type_saida := card_type || balance_saida := balance
        END
;

cr_saida, card_type_saida, balance_saida <-- addCreditTest3(cr, card_type, balance)=
        PRE
```

```
		cr: NAT1 & card_type: CARD_TYPES & balance: NAT &

		card_type /= gratutious_card &

		balance + cr <= MAXINT &

		not(not(card_type = gratutious_card) or (balance = 0))


	THEN

		cr_saida := cr || card_type_saida := card_type || balance_saida := balance

	END

END
```

Aplicação em Java

```
public class Transport implements Constants{

	private byte balance;
	private byte cardType;
	private byte hour;
	private byte min;
	private byte day;
	private byte month;
	private byte year;

	public Transport(){
		balance = 0;
		cardType = Constants.ENTIRE_CARD;
		hour = 0;
		min = 0;
		day = 1;
		month = 1;
		year = 1;
	}

	public final void addCredit(byte cr){
		if ((cr >= 1) /*&& (cr + balance <= Constants.MAXINT)*/){ /* a
implementação original não tinha essa parte do teste das somas*/
			if(cardType == Constants.ENTIRE_CARD||
			   cardType == Constants.STUDENT_CARD){
				balance = (byte)(balance + cr);
			   }
			else{
				System.out.println("Tipo de cartão incorreto. Operação
não realizada.");
			}
		}
		else{
			System.out.println("Crédito incorreto. Operação não
realizada.");
		}
	}

	public final void debitCredit(){
		if (balance > 0){
			if(cardType == Constants.ENTIRE_CARD ||
```

```
                  cardType == Constants.STUDENT_CARD){
                        balance = (byte)(balance - 1);
                  }
                  else{
                        System.out.println("Tipo de cartão incorreto. Operação
não realizada.");
                  }
            }
            else{
                  System.out.println("Crédito insuficiente. Operação não
realizada.");
            }
      }

      public final void changeCardType(byte newCardType){
            if (balance != 0){
                  System.out.println("Crédito é diferente de zero. Operação não
realizada.");
            }
            else{
                  if(newCardType == Constants.ENTIRE_CARD){
                        cardType = Constants.ENTIRE_CARD;
                  }
                  else if (newCardType == Constants.STUDENT_CARD){
                        cardType = Constants.STUDENT_CARD;
                  }
                  else if (newCardType == Constants.GRATUTIOUS_CARD){
                        cardType = Constants.GRATUTIOUS_CARD;
                  }
                  else {
                        System.out.println("Tipo de cartão inválido. Operação não
realizada.");
                  }
            }
      }

      public final void setTime(byte h, byte m){
            if((h>=0 && h<=23) && (m>=0 && m<=59)){
                  hour = h;
                  min = m;
            }
            else{
                  System.out.println("Hora inválida. Operação não realizada.");
            }
      }

      public final void setDate(byte d, byte m, byte y){
            boolean valid = false;
            if(y>=0){
                  if (m==1 || m==3 || m==7 || m==8 || m==10 || m==12){
                        if(d>=1 && d<=31){
                              valid = true;
                        }
                  }
                  else{
                        if(m==4 || m==6 || m==9 || m==11){
                              if(d>=1 && d<=30){
                                    valid = true;
                              }
                        }
                        else{
```

```
                        if(m==2){
                              if(d>=1 && d<=28){
                                    valid = true;
                              }
                              else{
                                    if(d==29 && (y % 400 == 0)){
                                          valid = true;
                                    }
                              }
                        }
                  }
            }
            if (valid){
                  day = d;
                  month = m;
                  year = y;
            }
            else{
                  System.out.println("Data inválida. Operação não realizada.");
            }
      }


      public final byte getDay(){
            return day;
      }

      public final byte getMonth(){
            return month;
      }

      public final byte getYear(){
            return year;
      }

      public final byte getCardType(){
            return cardType;
      }

      public final void setCardType(byte cardType){
            this.cardType = cardType;
      }

      public final short getBalance(){
            return balance;
      }

      public final void setBalance(byte balance){
            this.balance = balance;
      }

      public final byte getHour(){
            return hour;
      }

      public final byte getMin(){
            return min;
      }

}
```

```
public interface Constants {
	final static int MAXINT = 127;
	final static byte ENTIRE_CARD = 1;
	final static byte STUDENT_CARD = 2;
	final static byte GRATUTIOUS_CARD = 3;

}
```

Dados de Teste Gerados pelo ProB

Nível 1	Nível 2	Nível 3	Inválidos
addCreditTest1(1,entire_card,0)-->1,entire_card,0	addCreditTest2(1,entire_card,0)-->1,entire_card,0	addCreditTest2(1,entire_card,0)-->1,entire_card,0	addCreditTest1(1,gratutious_card,0)-->1,gratutious_card,0
addCreditTest1(1,student_card,0)-->1,student_card,0	addCreditTest2(1,student_card,0)-->1,student_card,0	addCreditTest2(1,student_card,0)-->1,student_card,0	addCreditTest1(2,gratutious_card,0)-->2,gratutious_card,0
addCreditTest1(1,entire_card,1)-->1,entire_card,1	addCreditTest3(1,entire_card,126)-->1,entire_card,126	addCreditTest3(1,entire_card,126)-->1,entire_card,126	addCreditTest1(3,gratutious_card,0)-->3,gratutious_card,0
addCreditTest1(1,student_card,1)-->1,student_card,1	addCreditTest3(1,student_card,126)-->1,student_card,126	addCreditTest3(1,student_card,126)-->1,student_card,126	addCreditTest1(4,gratutious_card,0)-->4,gratutious_card,0
addCreditTest1(1,entire_card,2)-->1,entire_card,2	addCreditTest4(127,entire_card,0)-->127,entire_card,0	addCreditTest4(127,entire_card,0)-->127,entire_card,0	addCreditTest1(5,gratutious_card,0)-->5,gratutious_card,0
addCreditTest1(1,student_card,2)-->1,student_card,2	addCreditTest4(127,student_card,0)-->127,student_card,0	addCreditTest4(127,student_card,0)-->127,student_card,0	addCreditTest1(6,gratutious_card,0)-->6,gratutious_card,0
addCreditTest1(1,entire_card,3)-->1,entire_card,3	addCreditTest5(1,entire_card,126)-->1,entire_card,126	addCreditTest5(1,entire_card,126)-->1,entire_card,126	addCreditTest1(7,gratutious_card,0)-->7,gratutious_card,0
addCreditTest1(1,student_card,3)-->1,student_card,3	addCreditTest5(1,student_card,126)-->1,student_card,126	addCreditTest5(1,student_card,126)-->1,student_card,126	addCreditTest1(8,gratutious_card,0)-->8,gratutious_card,0
addCreditTest1(1,entire_card,4)-->1,entire_card,4	addCreditTest5(2,entire_card,125)-->2,entire_card,125	addCreditTest5(2,entire_card,125)-->2,entire_card,125	addCreditTest1(9,gratutious_card,0)-->9,gratutious_card,0
addCreditTest1(1,student_card,4)-->1,student_card,4	addCreditTest5(2,student_card,125)-->2,student_card,125	addCreditTest5(2,student_card,125)-->2,student_card,125	addCreditTest1(10,gratutious_card,0)-->10,gratutious_card,0
addCreditTest1(1,entire_card,5)-->1,entire_card,5	addCreditTest5(3,entire_card,124)-->3,entire_card,124	addCreditTest5(3,entire_card,124)-->3,entire_card,124	addCreditTest1(11,gratutious_card,0)-->11,gratutious_card,0
addCreditTest1(1,student_card,5)-->1,student_card,5	addCreditTest5(3,student_card,124)-->3,student_card,124	addCreditTest5(3,student_card,124)-->3,student_card,124	addCreditTest1(12,gratutious_card,0)-->12,gratutious_card,0
addCreditTest1(1,entire_card,6)-->1,entire_card,6	addCreditTest5(4,entire_card,123)-->4,entire_card,123	addCreditTest5(4,entire_card,123)-->4,entire_card,123	addCreditTest1(13,gratutious_card,0)-->13,gratutious_card,0
addCreditTest1(1,student_card,6)-->1,student_card,6	addCreditTest5(4,student_card,123)-->4,student_card,123	addCreditTest5(4,student_card,123)-->4,student_card,123	addCreditTest1(14,gratutious_card,0)-->14,gratutious_card,0
addCreditTest1(1,entire_card,7)-->1,entire_card,7	addCreditTest5(5,entire_card,122)-->5,entire_card,122	addCreditTest5(5,entire_card,122)-->5,entire_card,122	addCreditTest1(15,gratutious_card,0)-->15,gratutious_card,0
debitCreditTest1(student_card,1)-->student_card,1	debitCreditTest2(entire_card,1)-->entire_card,1	debitCreditTest2(entire_card,1)-->entire_card,1	addCreditTest2(1,entire_card,127)-->1,entire_card,127
debitCreditTest1(entire_card,2)-->entire_card,2	debitCreditTest2(student_card,1)-->student_card,1	debitCreditTest2(student_card,1)-->student_card,1	addCreditTest2(1,student_card,127)-->1,student_card,127
debitCreditTest1(student_card,2)-->student_card,2	debitCreditTest3(entire_card,127)-->entire_card,127	debitCreditTest3(entire_card,127)-->entire_card,127	addCreditTest2(2,entire_card,126)-->2,entire_card,126
debitCreditTest1(entire_card,3)-->entire_card,3	debitCreditTest3(student_card,127)-->student_card,127	debitCreditTest3(student_card,127)-->student_card,127	addCreditTest2(2,student_card,126)-->2,student_card,126
debitCreditTest1(student_card,3)-->student_card,3	setTimeTest1(0,0)-->0,0	setTimeTest1(0,0)-->0,0	addCreditTest2(2,entire_card,127)-->2,entire_card,127
debitCreditTest1(entire_card,4)-->entire_card,4	setTimeTest1(0,1)-->0,1	setTimeTest1(0,1)-->0,1	addCreditTest2(2,student_card,127)-->2,student_card,127
debitCreditTest1(student_card,4)-->student_card,4	setTimeTest1(0,2)-->0,2	setTimeTest1(0,2)-->0,2	addCreditTest2(3,entire_card,125)-->3,entire_card,125
debitCreditTest1(entire_card,5)-->entire_card,5	setTimeTest1(0,3)-->0,3	setTimeTest1(0,3)-->0,3	addCreditTest2(3,student_card,125)-->3,student_card,125
debitCreditTest1(student_card,5)-->student_card,5	setTimeTest1(0,4)-->0,4	setTimeTest1(0,4)-->0,4	addCreditTest2(3,entire_card,126)-->3,entire_card,126
debitCreditTest1(entire_card,6)-->entire_card,6	setTimeTest1(0,5)-->0,5	setTimeTest1(0,5)-->0,5	addCreditTest2(3,student_card,126)-->3,student_card,126
debitCreditTest1(student_card,6)-->student_card,6	setTimeTest1(0,6)-->0,6	setTimeTest1(0,6)-->0,6	addCreditTest2(3,entire_card,127)-->3,entire_card,127
debitCreditTest1(entire_card,7)-->entire_card,7	setTimeTest1(0,7)-->0,7	setTimeTest1(0,7)-->0,7	addCreditTest2(3,student_card,127)-->3,student_card,127
debitCreditTest1(student_card,7)-->student_card,7	setTimeTest1(0,8)-->0,8	setTimeTest1(0,8)-->0,8	addCreditTest2(4,entire_card,124)-->4,entire_card,124
debitCreditTest1(entire_card,8)-->entire_card,8	setTimeTest1(0,9)-->0,9	setTimeTest1(0,9)-->0,9	addCreditTest2(4,student_card,124)-->4,student_card,124
debitCreditTest1(student_card,8)-->student_card,8	setTimeTest1(0,10)-->0,10	setTimeTest1(0,10)-->0,10	addCreditTest2(4,entire_card,125)-->4,entire_card,125
setTimeTest1(0,0)-->0,0	setTimeTest1(0,11)-->0,11	setTimeTest1(0,11)-->0,11	debitCreditTest1(gratutious_card,1)-->gratutious_card,1
setTimeTest1(0,1)-->0,1	setTimeTest1(0,12)-->0,12	setTimeTest1(0,12)-->0,12	debitCreditTest1(gratutious_card,2)-->gratutious_card,2
setTimeTest1(0,2)-->0,2	setTimeTest1(0,13)-->0,13	setTimeTest1(0,13)-->0,13	debitCreditTest1(gratutious_card,3)-->gratutious_card,3
setTimeTest1(0,3)-->0,3	setTimeTest1(0,14)-->0,14	setTimeTest1(0,14)-->0,14	debitCreditTest1(gratutious_card,4)-->gratutious_card,4
setTimeTest1(0,4)-->0,4	setDateTest1(31,1,1)-->31,1,1	setDateTest1(31,1,1)-->31,1,1	debitCreditTest1(gratutious_card,5)-->gratutious_card,5
setTimeTest1(0,5)-->0,5	setDateTest1(31,3,1)-->31,3,1	setDateTest1(31,3,1)-->31,3,1	debitCreditTest1(gratutious_card,6)-->gratutious_card,6
setTimeTest1(0,6)-->0,6	setDateTest1(31,5,1)-->31,5,1	setDateTest1(31,5,1)-->31,5,1	debitCreditTest1(gratutious_card,7)-->gratutious_card,7
setTimeTest1(0,7)-->0,7	setDateTest1(31,7,1)-->31,7,1	setDateTest1(31,7,1)-->31,7,1	debitCreditTest1(gratutious_card,8)-->gratutious_card,8
setTimeTest1(0,8)-->0,8	setDateTest1(31,8,1)-->31,8,1	setDateTest1(31,8,1)-->31,8,1	debitCreditTest1(gratutious_card,9)-->gratutious_card,9
setTimeTest1(0,9)-->0,9	setDateTest1(31,10,1)-->31,10,1	setDateTest1(31,10,1)-->31,10,1	debitCreditTest1(gratutious_card,10)-->gratutious_card,10
setTimeTest1(0,10)-->0,10	setDateTest1(31,12,1)-->31,12,1	setDateTest1(31,12,1)-->31,12,1	debitCreditTest1(gratutious_card,11)-->gratutious_card,11
setTimeTest1(0,11)-->0,11	setDateTest1(1,2,1)-->1,2,1	setDateTest1(1,2,1)-->1,2,1	debitCreditTest1(gratutious_card,12)-->gratutious_card,12
setTimeTest1(0,12)-->0,12	setDateTest1(2,2,1)-->2,2,1	setDateTest1(2,2,1)-->2,2,1	debitCreditTest1(gratutious_card,13)-->gratutious_card,13
setTimeTest1(0,13)-->0,13	setDateTest1(3,2,1)-->3,2,1	setDateTest1(3,2,1)-->3,2,1	debitCreditTest1(gratutious_card,14)-->gratutious_card,14
setTimeTest1(0,14)-->0,14	setDateTest1(4,2,1)-->4,2,1	setDateTest1(4,2,1)-->4,2,1	debitCreditTest1(gratutious_card,15)-->gratutious_card,15
setDateTest1(1,1,1)-->1,1,1	setDateTest1(5,2,1)-->5,2,1	setDateTest1(5,2,1)-->5,2,1	debitCreditTest2(entire_card,0)-->entire_card,0
setDateTest1(2,1,1)-->2,1,1	setDateTest1(6,2,1)-->6,2,1	setDateTest1(6,2,1)-->6,2,1	debitCreditTest2(student_card,0)-->student_card,0

Casos de teste gerados no JUnit

Nível 1

```
import junit.framework.Assert;
import junit.framework.TestCase;


public class TransportTest1 extends TestCase {

	Transport t;

	public TransportTest1(String n){
		super(n);
		t = new Transport();
	}

	public void testAddCredit1() {
		t.setBalance((byte)4);
		t.setCardType((byte)Constants.ENTIRE_CARD);
		t.addCredit((byte)1);

		Assert.assertEquals(t.getBalance(), 5);
	}

}
```

Níveis 2 e 3

```
import junit.framework.Assert;
import junit.framework.TestCase;


public class TransportTest2 extends TestCase {

	Transport t;

	public TransportTest2(String n){
		super(n);
		t = new Transport();
	}

	public void testAddCredit1() {
		t.setBalance((byte)4);
		t.setCardType((byte)Constants.ENTIRE_CARD);
		t.addCredit((byte)1);

		Assert.assertEquals(t.getBalance(), 5);
	}

	public void testAddCredit2() {
		t.setBalance((byte)0);
		t.setCardType((byte)Constants.STUDENT_CARD);
		t.addCredit((byte)1);

		Assert.assertEquals(t.getBalance(), 1);
	}
```

```
	public void testAddCredit3() {
		t.setBalance((byte)126);
		t.setCardType((byte)Constants.ENTIRE_CARD);
		t.addCredit((byte)1);

		Assert.assertEquals(t.getBalance(), 127);
	}

	public void testAddCredit4() {
		t.setBalance((byte)0);
		t.setCardType((byte)Constants.ENTIRE_CARD);
		t.addCredit((byte)127);

		Assert.assertEquals(t.getBalance(), 127);
	}

	public void testAddCredit5() {
		t.setBalance((byte)63);
		t.setCardType((byte)Constants.STUDENT_CARD);
		t.addCredit((byte)64);

		Assert.assertEquals(t.getBalance(), 127);
	}

}
```

Dados inválidos

```
import junit.framework.Assert;
import junit.framework.TestCase;


public class TransportTestInv extends TestCase {

	Transport t;

	public TransportTestInv(String n){
		super(n);
	}

	public void testAddCredit1() {
		t = new Transport();
		t.setBalance((byte)0);
		t.setCardType((byte)Constants.GRATUTIOUS_CARD);
		t.addCredit((byte)9);

		Assert.assertEquals(t.getBalance(), 0);
	}

	public void testAddCredit2() {
		t = new Transport();
		t.setBalance((byte)127);
		t.setCardType((byte)Constants.ENTIRE_CARD);
		t.addCredit((byte)2);

		Assert.assertEquals(t.getBalance(), 127);
	}
```

```
	public void testAddCredit3() {
		t = new Transport();
		t.setBalance((byte)126);
		t.setCardType((byte)Constants.STUDENT_CARD);
		t.addCredit((byte)3);

		Assert.assertEquals(t.getBalance(), 126);
	}

}
```

Livros Grátis

(http://www.livrosgratis.com.br)

Milhares de Livros para Download:

Baixar livros de Literatura
Baixar livros de Literatura de Cordel
Baixar livros de Literatura Infantil
Baixar livros de Matemática
Baixar livros de Medicina
Baixar livros de Medicina Veterinária
Baixar livros de Meio Ambiente
Baixar livros de Meteorologia
Baixar Monografias e TCC
Baixar livros Multidisciplinar
Baixar livros de Música
Baixar livros de Psicologia
Baixar livros de Química
Baixar livros de Saúde Coletiva
Baixar livros de Serviço Social
Baixar livros de Sociologia
Baixar livros de Teologia
Baixar livros de Trabalho
Baixar livros de Turismo