Ernesto Cid Brasil de Matos

BETA: Uma ferramenta para geração de testes de unidade a partir de especificações B

Natal/RN

2012

Ernesto Cid Brasil de Matos

BETA: Uma ferramenta para geração de testes de unidade a partir de especificações B

Dissertação apresentada ao Programa de Pós-graduação em Sistemas e Computação do Departamento de Informática e Matemática Aplicada da Universidade Federal do Rio Grande do Norte, como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação

Orientadora: Prof[a]. Dr[a]. Anamaria Martins Moreira

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

Natal/RN

2012

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

Matos, Ernesto Cid Brasil de.

Beta: uma ferramenta para geração de testes de unidade a partir de especificações B / Ernesto Cid Brasil de Matos. – Natal, RN, 2012.

123 f. ; il.

Orientadora: Profa. Dra. 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. Engenharia de Software – Dissertação. 2. Métodos formais – Método B – Dissertação. 3. Testes de software – Dissertação. 4. Testes baseados em modelos – Dissertação. 5. Controle de qualidade – Dissertação. I. Moreira, Anamaria Martins. II. Título.

RN/UF/BSE-CCET CDU 004.41

BETA: uma ferramenta para geração de testes de unidade a partir de especificações B

Ernesto Cid Brasil de Matos

Dissertação de Mestrado aprovada em 10 de fevereiro de 2012 pela banca examinadora composta pelos seguintes membros:

Profa. Dra. Anamaria Martins Moreira (orientadora) ..... DIMAp/UFRN

Profa. Dra. Patricia Duarte de Lima Machado ..... DSC/UFCG

Profa. Dra. Roberta de Souza Coelho ..... DIMAp/UFRN

Aos meus pais, Ernesto e Naltide.

Agradecimentos

À minha família, por todo o carinho e apoio, mesmo estando longe de casa.

Aos amigos Dann, Simone, Ítalo, Edmilson, Pablo e Chicão, por fazerem a distância de casa parecer menor.

À minha orientadora, Anamaria, pela orientação durante este trabalho e por me mostrar o que é ser um professor de verdade.

À CAPES e ao INES, pelo suporte financeiro.

"You got to make the back of the fence that nobody will see just as good looking as the front of the fence. That will show that you're dedicated to making something perfect."

— Paul Jobs, pai adotivo de Steve Jobs

Resumo

Métodos formais e testes são ferramentas para obtenção e controle de qualidade de software. Quando utilizadas em conjunto, elas provêem mecanismos para especificação, verificação e detecção de falhas de um software. Apesar de permitir que sistemas sejam matematicamente verificados, métodos formais não são suficientes pra garantir que um sistema esteja livre de defeitos, logo, técnicas de teste de software são necessárias para completar o processo de verificação e validação de um sistema. Técnicas de Testes Baseados em Modelos permitem que testes sejam gerados a partir de outros artefatos de software como especificações e modelos abstratos. Ao utilizarmos especificações formais como base para a criação de testes, podemos gerar testes de melhor qualidade pois estas especificações costumam ser precisas e livres de ambiguidade. Fernanda Souza (2009) propôs um método para definir casos de teste a partir de especificações do Método B. Este método utilizava informações do invariante de uma máquina e das pré-condições de uma operação para definir casos de teste positivos e negativos para tal operação, através de técnicas baseadas em particionamento em classes de equivalência e análise de valor limite. No entanto, a proposta de 2009 não incluía automação e possuía algumas deficiências conceituais como, por exemplo, não se encaixar exatamente em uma classificação de critérios de cobertura bem definida. Iniciamos nosso trabalho com um estudo de caso que aplicou o método a um exemplo de especificação B proveniente da indústria. A partir deste estudo obtivemos subsídios para o aperfeiçoá-lo. Em nosso trabalho aperfeiçoamos o método proposto, reescrevendo e adicionando características para torná-lo compatível com uma classificação de testes utilizada pela comunidade. O método também foi melhorado para suportar especificações estruturadas em vários componentes, utilizar informações sobre o comportamento da operação durante a criação de casos de teste e utilizar novos critérios de cobertura. Além disso, implementamos uma ferramenta para automatizá-lo e o submetemos a estudos de caso mais complexos.

Palavras-chave: Engenharia de Software; Método B; Testes de Software; Testes Baseados em Modelos; Testes de Unidade.

Abstract

Formal methods and software testing are tools to obtain and control software quality. When used together, they provide mechanisms for software specification, verification and error detection. Even though formal methods allow software to be mathematically verified, they are not enough to assure that a system is free of faults, thus, software testing techniques are necessary to complement the process of verification and validation of a system. Model Based Testing techniques allow tests to be generated from other software artifacts such as specifications and abstract models. Using formal specifications as basis for test creation, we can generate better quality tests, because these specifications are usually precise and free of ambiguity. Fernanda Souza (2009) proposed a method to define test cases from B Method specifications. This method used information from the machine's invariant and the operation's precondition to define positive and negative test cases for an operation, using equivalent class partitioning and boundary value analysis based techniques. However, the method proposed in 2009 was not automated and had conceptual deficiencies like, for instance, it did not fit in a well defined coverage criteria classification. We started our work with a case study that applied the method in an example of B specification from the industry. Based in this case study we've obtained subsidies to improve it. In our work we evolved the proposed method, rewriting it and adding characteristics to make it compatible with a test classification used by the community. We also improved the method to support specifications structured in different components, to use information from the operation's behavior on the test case generation process and to use new coverage criterias. Besides, we have implemented a tool to automate the method and we have submitted it to more complex case studies.

Keywords: Software Engineering; B Method; Software Testing; Model Based Testing; Unit Testing.

Sumário

Lista de Figuras

Lista de Tabelas

1 Introdução

Software está presente em todos os aspectos de nossa vida atualmente. Seja na cafeteira que utilizamos para fazer nosso café pela manhã, nos leitores de cartão dos ônibus que utilizamos para chegar ao trabalho ou na universidade, no telefone celular que carregamos no bolso ou nos aparelhos de TV que ligamos à noite para acompanhar as notícias do dia, software está em todo lugar.

Algo tão essencial às nossas vidas não pode ser desenvolvido de qualquer maneira. Desde a crise do software no final década de sessenta [DIJKSTRA, 1972] – quando o poder computacional e a complexidade dos programas desenvolvidos começaram a crescer rapidamente – a comunidade de engenheiros de software se preocupa em desenvolver metodologias que visam aumentar a qualidade dos sistemas desenvolvidos, respeitando restrições de tempo e orçamento.

Tendo isto em mente, grande parte destas metodologias de desenvolvimento de software reservam uma fração considerável do tempo de um projeto para atividades de validação e verificação. Cerca de 50% do tempo e dos custos de um projeto de software estão relacionados à atividades de testes [MYERS, 2011]. É através destas atividades que procura-se garantir que um software atende às expectativas do usuário e ocasiona o mínimo possível de falhas durante sua execução.

Em Engenharia de Software existem várias ferramentas para se desenvolver sistemas visando qualidade. Este trabalho propõe aliar duas destas ferramentas: Métodos Formais e Testes de Software.

Métodos formais [FMWIKI, 2011] utilizam conceitos matemáticos na especificação e construção de sistemas. Os modelos matemáticos gerados por esta abordagem passam por sequências de provas e verificações, para garantir a consistência e robustez de um sistema. O processo de desenvolvimento formal de um sistema costuma ser trabalhoso, e pode implicar em altos custos para um projeto. Por isso, estas técnicas são normalmente utilizadas no desenvolvimento de sistemas críticos e em partes críticas de sistemas, onde

o investimento é necessário para que, por exemplo, vidas não sejam postas em risco.

Em nosso trabalho, utilizamos o Método B [ABRIAL, 1996] como método formal. O Método B é uma metodologia para especificação, validação e construção de sistemas. Usando este método, sistemas podem ser especificados através de máquinas de estado abstratas especificadas utilizando conceitos de matemática inteira, lógica de primeira ordem e teoria dos conjuntos. Estas máquinas geram obrigações de provas que devem ser comprovadas para garantir a consistência do modelo. O método possui ainda um mecanismo de refinamento, no qual máquinas podem ser refinadas até um nível algorítmico, que pode ser traduzido em código em uma linguagem de programação.

Testes de Software consistem em executar um programa com o objetivo de detectar defeitos [MYERS, 2011]. É através de testes que procuramos validar se um sistema está de acordo com os requisitos levantados, e se o mesmo funciona como esperado, sem levantar falhas durante a execução. Uma falha ocorre quando, devido a um defeito, o software é levado a um estado ilegal que ocasiona um erro durante a execução do código [AMMANN; OFFUTT, 2008].

A atividade de testes costuma se estender por todo o processo de desenvolvimento, sendo mais intensa nas etapas finais do processo, após a implementação do sistema ou de partes dele. Quando defeitos no software são encontrados nestes estágios do ciclo de desenvolvimento, o custo para correção destes costuma ser alto. Comparado com o custo necessário para solucionar um defeito durante as etapas iniciais do ciclo de desenvolvimento, o custo para solucionar o mesmo defeito em etapas de integração de componentes pode chegar até dez vezes mais, enquanto corrigi-lo após o lançamento pode custar trinta vezes mais [NIST, 2002].

Uma prática comum a uma maioria dos processos de desenvolvimento é a utilização de modelos abstratos para auxiliar a construção de sistemas. Estes modelos permitem que engenheiros de software tenham uma visão conceitual do sistema ainda nas etapas iniciais do ciclo de desenvolvimento, possibilitando que defeitos sejam encontrados e corrigidos rapidamente.

Técnicas de Testes Baseados em Modelos (TBM) [UTTING *et al.*, 2011] permitem que testes sejam derivados a partir destes modelos (ou outros artefatos de software), tornando possível a criação de testes para o software antes mesmo de sua implementação. Outra vantagem do uso destas técnicas é que a geração dos testes costuma ser automática, o que reduz custos de desenvolvimento.

Este trabalho utiliza técnicas de TBM para geração de testes de unidade a partir de especificações B. Evoluímos uma abordagem de testes inicialmente proposta em [SOUZA, 2009], capaz de gerar casos de teste positivos e negativos a partir de máquinas abstratas B, utilizando técnicas baseadas em particionamento em classes de equivalência e análise de valor limite. Melhoramos esta abordagem reescrevendo-a para que ela ficasse de acordo com uma classificação de testes bem definida, adicionamos outras técnicas para melhorar o particionamento das classes de equivalência, elaboramos um processo para a criação de oráculos e adaptamos a abordagem para que ela trabalhe com especificações estruturadas em vários componentes. Além disso, desenvolvemos uma ferramenta capaz de automatizá-la e a submetemos à estudos de caso mais complexos.

1.1 Motivação

Atualmente existe o esforço das comunidades de métodos formais e testes de software para integrar ambas disciplinas. Como pode ser visto no Capítulo 3 sobre trabalhos relacionados, existem várias tentativas de gerar testes a partir de especificações formais. Apesar da quantidade de trabalhos, ainda existem vários problemas em aberto sobre o assunto que precisam ser estudados, tais como, geração de oráculos de teste, mapeamento entre dados abstratos e concretos, problemas sobre especificações não-determinísticas do comportamento de operações, entre outros.

Apesar de permitirem que um sistema seja verificado com rigor matemático, métodos formais não são suficientes para garantir que ele esteja livre de defeitos. Desta forma, testes de software podem complementar especificações formais pois provêem mecanismos para detectar falhas, explorando possíveis defeitos inseridos no software durante a implementação.

Como especificações formais costumam descrever os requisitos do software de maneira rigorosa, geralmente precisa e livre de ambiguidades, elas podem ser utilizadas como base para a criação de casos de teste de qualidade, possivelmente melhores do que testes criados a partir de especificações informais.

Projetar bons testes de software é uma tarefa difícil e que requer muito tempo e esforço de engenheiros de teste. Um bom conjunto de testes é aquele capaz de detectar o maior número de defeitos utilizando poucos casos de teste. Ao provermos uma ferramenta capaz de gerar casos de teste automaticamente, podemos reduzir consideravelmente o esforço necessário para desenvolver estes testes, reduzindo os custos de um projeto.

Outra vantagem que a geração de testes a partir de especificações formais traz diz respeito a casos em que métodos formais não são seguidos a risca. Muitas vezes, devido restrições de tempo e orçamento, métodos formais são utilizados apenas para níveis mais abstratos de especificação e a implementação do software é feita a parte. Desta forma, testes gerados a partir da especificação podem ajudar a manter a coesão entre a especificação e a implementação, verificando se o que foi implementado está de acordo com o que foi especificado.

Nosso método também é capaz de gerar casos de teste negativos. Este tipo de teste utiliza dados de teste que desrespeitam os requisitos impostos aos dados de entrada do software. Testes negativos são importantes para avaliar a robustez e segurança de sistemas críticos, já que usuários mal-intencionados costumam explorar este tipo de falha.

Outros pontos positivos do nosso trabalho em relação aos trabalhos relacionados dizem respeito a maneira como conceitos de teste de software são empregados e ao suporte ferramental para a abordagem. Dentre os trabalhos encontrados na bibliografia, a grande maioria não utiliza conceitos e classificações de testes bem definidas, além de possuírem pouco ou nenhum suporte ferramental. Nosso trabalho procura cobrir estas deficiências, apresentando uma abordagem bem definida e baseada em conceitos de testes de software bem estabelecidos. Também provemos uma ferramenta que suporta a abordagem, automatizando o processo de geração de especificações de teste.

Com este trabalho estabelecemos uma base para que outros projetos sejam desenvolvidos. Inicialmente focamos na geração de teste de unidade mas ainda é possível expandir a abordagem para que ela suporte outros níveis de teste como, por exemplo, geração de testes de sistema a partir de especificações em *Event-B* [ABRIAL *et al.*, 2010].

1.2 Objetivos

1.2.1 Objetivos Gerais

Este trabalho tem o objetivo de aperfeiçoar uma abordagem para geração de casos de teste a partir de especificações B proposta inicialmente por [SOUZA, 2009], além de desenvolver uma ferramenta capaz de automatizá-la e avaliá-la através de estudos de caso.

Iniciamos nossos trabalhos com um estudo de caso que aplicava a abordagem proposta inicialmente a um exemplo da indústria de metrôs [MATOS *et al.*, 2010]. Este estudo de caso nos proveu os subsídios necessários para propor melhorias a abordagem e nos

confirmou que, para que seu uso fosse viável em um projeto comercial, seria necessário desenvolver uma ferramenta que a suportasse. Por outro lado, o estudo nos provou que a abordagem pode ser utilizada por um usuário sem formação em métodos formais.

1.2.2 Objetivos Específicos

Como objetivos específicos deste trabalho tivemos:

- *Avaliar a abordagem inicialmente proposta:* no trabalho desenvolvido por [SOUZA, 2009], os exemplos utilizados como estudo de caso foram simples e não foram suficientes para avaliar a abordagem e revelar possíveis deficiências que ela possuísse. Em nosso trabalho submetemos a abordagem a estudos de caso mais complexos, com especificações baseadas em sistemas existentes na indústria, que nos proveram subsídios para melhorá-la, além de corrigir deficiências que foram encontrados em sua primeira versão.

- *Aperfeiçoar a abordagem utilizando classificações e conceitos de teste de software bem estabelecidos:* da maneira que a abordagem foi descrita inicialmente, era difícil apontar como os conceitos de teste de software eram empregados durante o processo. Foi necessário reescrevê-la e adaptá-la para que eles se tornassem mais claros e bem embasados. Após estas alterações, a abordagem ficou de acordo com conceitos de particionamento do espaço de entrada e com critérios de seleção de dados de entrada definidos em [AMMANN; OFFUTT, 2008].

- *Modificar a abordagem para que ela suporte especificações componentizadas:* na abordagem original e em grande parte dos trabalhos relacionados, os métodos de geração de teste restringem que a especificação se resuma a apenas um componente (ou máquina, no caso do Método B). Este tipo de restrição não reflete as especificações que encontramos na indústria, logo, precisamos evoluir a abordagem para que ela fosse capaz de suportar especificações estruturadas em várias máquinas.

- *Utilizar a especificação do comportamento da operação durante o particionamento das classes de equivalência:* na proposta inicial, o particionamento do espaço de entrada de uma operação levava em consideração apenas informações da pré-condição da operação e do invariante da máquina. No entanto, é possível gerar partições mais interessantes se observarmos também o comportamento da operação. Em nosso trabalho, utilizamos informações de comandos condicionais (como *if-then-else* e *case*)

para criar partições mais interessantes, que tendem a exercitar diferentes fluxos de execução dentro da operação.

- *Desenvolver uma ferramenta para automatizar a abordagem:* os estudos de caso iniciais comprovaram que, para que o uso da abordagem fosse viável na indústria, seria necessário desenvolver uma ferramenta que a suportasse. Assim sendo, desenvolvemos uma ferramenta que automatiza o processo de geração de casos de teste a partir de especificação B, gerando especificações de teste que podem ser traduzidas em casos de teste concretos para o software especificado.

1.3 Organização do Trabalho

Este trabalho continua com uma introdução aos conceitos necessários para entendê-lo no Capítulo 2. Nesse capítulo é feita uma breve apresentação do Método B e de alguns conceitos de teste de software. No Capítulo 3 listamos os trabalhos relacionados, cujos objetivos são a geração de testes a partir de modelos formais. No decorrer do capítulo discutimos as estratégias empregadas, os modelos formais utilizados e os resultados alcançados por cada um destes trabalhos. No Capítulo 4 fazemos uma revisão do método desenvolvido em [SOUZA, 2009], enquanto no Capítulo 5 apresentamos as melhorias introduzidas pelo nosso trabalho. No Capítulo 6 apresentamos detalhes sobre o funcionamento e a implementação da ferramenta. Os estudos de caso realizados são apresentados no Capítulo 7. Finalizamos o trabalho no Capítulo 8, discutindo nossos resultados e apontando possíveis melhorias para trabalhos futuros.

2 Fundamentação Teórica

2.1 Considerações Iniciais

Neste capítulo faremos uma pequena introdução aos conceitos necessários para o entendimento do restante do trabalho. Começaremos apresentando o Método B e suas principais características, utilizando exemplos de especificações como referências durante as explicações. Em seguida serão introduzidos alguns conceitos de Teste de Software como: tipos de teste, níveis de teste e critérios de cobertura.

2.2 Método B

O Método B [ABRIAL, 1996] é um método formal utilizado para especificar e construir sistemas de forma segura e robusta. Através dele sistemas podem ser modelados na forma de máquinas de estado abstratas, utilizando conceitos de teoria dos conjuntos, aritmética inteira e lógica de primeira ordem. Por serem modelos matemáticos, estas especificações podem passar por sequências de provas e verificações para garantir que o sistema é seguro e consistente. Após estas verificações, o método permite que os modelos sejam refinados até um nível algorítmico, que possibilita a geração automática de código a partir da especificação.

2.2.1 Tipos de Máquinas B

Especificações no Método B são estruturadas em módulos. A notação possui 3 tipos de módulos que são classificados de acordo com seu nível de abstração. Do mais abstrato para o mais concreto temos: *MACHINE*, *REFINEMENT* e *IMPLEMENTATION*.

O processo de especificação parte da definição de uma máquina abstrata (*MACHINE*). A partir destas máquinas, são feitos refinamentos (*REFINEMENT*) para aproximá-las do nível mais concreto. Refinamentos podem ser feitos a partir de máquinas abstratas ou

outros módulos de refinamento e devem ser verificados para garantir a equivalência com o módulo que está sendo refinado. Após um ou mais refinamentos é obtido o nível mais concreto de especificação (*IMPLEMENTATION*), que pode ser utilizado para a geração de código.

2.2.2 Substituições Generalizadas

Um dos conceitos básicos por trás do Método B são as substituições generalizadas. É a partir destas substituições que variáveis de um predicado podem ser alteradas e analisadas para verificar sua conformidade.

Formalmente uma substituição generalizada funciona da seguinte maneira: dado um predicado P, $[E/x]P$ é o predicado resultante após a substituição da variável livre x pela expressão E no predicado P. Por exemplo, ao aplicarmos a substituição $[2/x]$ ao predicado $x \in NAT$, teríamos como resultado o predicado $2 \in NAT$.

A seguir apresentamos algumas das principais substituições utilizadas no Método B. Outras substituições podem ser encontradas em [CLEARSY, 2011].

Substituição simples:

$$[x := E]P \Rightarrow [E/x]P$$

Nesta substituição x é uma variável da máquina ou parâmetro de saída de uma operação para o qual será atribuída a expressão E. Esta substituição é aplicada a predicado P, que pode ser uma cláusula do invariante da máquina ou da pré-condição da operação.

Substituição múltipla:

$$[x := E, y := F]P \Rightarrow [E, F/x, y]P$$

A substituição múltipla é uma variação da substituição simples. Ela funciona da mesma maneira e é utilizada para fazer substituições em duas ou mais variáveis simultaneamente.

Paralelismo:

$$S_1 \mid\mid S_2$$

Uma outra substituição bastante utilizada no Método B é a substituição paralela. Esta substituição representa duas outras substituições que devem acontecer exatamente no mesmo instante.

Substituição condicional:

$$[\text{IF } E \text{ THEN } S \text{ ELSE } T \text{ END}]P = (E \Rightarrow [S]P) \wedge (\neg E \Rightarrow [T]P)$$

As substituições condicionais provêm um mecanismo para a especificação de comportamentos baseados em uma condição. Ou seja, dada uma expressão lógica, dependendo de seu resultado, uma substituição diferente será realizada. De uma maneira mais formal, se a expressão E resultar em verdadeiro, a substituição S será realizada, caso ela resulte em falso, a substituição T será realizada.

Substituição *ANY* (*Unbounded Choice*):

$$[\text{ANY } X \text{ WHERE } P \text{ THEN } S \text{ END}]R \Leftrightarrow \forall X.(P \Rightarrow [S]R)$$

Esta substituição permite o uso de valores para as variáveis declaradas em X na substituição S, desde que estes valores estejam de acordo com o predicado P. Se há mais de um valor para as variáveis que satisfazem o predicado P, a substituição não especifica qual valor será efetivamente escolhido. Ela definirá um comportamento não-determinístico.

A substituição ANY é utilizada na substituição "se torna elemento de" que utiliza a seguinte notação:

$$X :: SET$$

Esta substituição determina que elementos do conjunto SET serão atribuídos as variáveis de X. Esta substituição é equivalente a seguinte substituição ANY:

$$\text{ANY } Y \text{ WHERE } Y \in SET \text{ THEN } X := Y \text{ END}$$

2.2.3 Símbolos da notação B

Para melhor entender os exemplos de especificação que serão introduzidos no decorrer do texto, é necessário conhecer alguns símbolos utilizados na notação B. As tabelas 2.1,

2.2, 2.3 e 2.4 listam alguns destes símbolos, como os utilizados para representar funções, relações, operações lógicas e operações sobre conjuntos.

Funções e Relações

Símbolo	Descrição
$+->$	função parcial
$-->$	função total
$-->>$	função sobrejetiva
$>+>$	injeção parcial
$>->$	injeção total
$>+>>$	bijeção parcial
$>->>$	bijeção total
$<->$	relação
$\|->$	mapeamento

Tabela 2.1: Notação B: Funções e Relações

Operadores Lógicos

Símbolo	Descrição
&	e
or	ou
#	existe
!	para todo
$=$	igualdade
$/=$	desigualdade
$=>$	implicação
$<=>$	se e somente se
$not(P)$	negação de P

Tabela 2.2: Notação B: Operadores Lógicos

Operações sobre Conjuntos

Símbolo	Descrição
:	pertence
$/:$	não pertence
$<:$	inclusão (subconjunto de)
$/<:$	não inclusão (não é subconjunto de)
$/\backslash$	intersecção
$\backslash/$	união
{}	conjunto vazio
POW	conjunto das partes

Tabela 2.3: Notação B: Operações sobre Conjuntos

Aritmética

Símbolo	Descrição
$+$	adição
$-$	subtração
$*$	multiplicação
$/$	divisão
$<$	menor que
$<=$	menor ou igual a
$>$	maior que
$>=$	maior ou igual a
INT	conjunto dos inteiros
NAT	conjunto dos naturais
$NAT1$	conjunto dos naturais positivos

Tabela 2.4: Notação B: Aritmética

2.2.4 Especificação de Máquinas Abstratas

Na Listagem 2.1, apresentamos um exemplo de máquina abstrata. A máquina *Transport* [GOMES, 2007] representa um cartão eletrônico utilizado em transportes coletivos. Um cartão possui um tipo e armazena o valor de saldo de créditos. O valor da passagem depende do tipo do cartão, que pode ser *inteira*, *estudante* ou *gratuita*. A máquina possui duas operações: *addCredit*, responsável por adicionar créditos no cartão e *queryBalance*, que permite verificar o saldo atual do cartão.

O nome de uma máquina B é definido na cláusula *MACHINE* da especificação. Como pode ser visto na listagem, nossa máquina exemplo recebeu o nome de *Transport* (linha 1).

Uma máquina B pode possuir um conjunto de variáveis que representam seu estado. Estas variáveis são definidas na cláusula *VARIABLES* (linhas 6 e 7). A máquina *Transport* possui duas variáveis: *balance*, responsável por armazenar o saldo de créditos do cartão e *card_type*, que define seu tipo.

Uma máquina B pode possuir ainda definições de conjuntos, que são descritos em uma cláusula *SETS* (linhas 3 e 4). Nosso exemplo possui apenas um conjunto enumerado chamado *CARD_TYPES*, que será utilizado para definir o tipo da variável *card_type*. Os elementos de *CARD_TYPES* são os possíveis tipos de um cartão: *entire_card* (inteira), *student_card* (estudante) e *gratuitous_card* (gratuita).

As variáveis de uma máquina possuem restrições quanto aos valores que podem assu-

Listagem 2.1: Exemplo de máquina abstrata B

```
1  MACHINE Transport
2
3  SETS
4    CARD_TYPES = {entire_card, student_card, gratuitous_card}
5
6  VARIABLES
7    balance, card_type
8
9  INVARIANT
10   balance : NAT &
11   card_type : CARD_TYPES &
12   (not(card_type = gratuitous_card) or balance = 0)
13
14 INITIALISATION
15   balance := 0 ||
16   card_type :: CARD_TYPES
17
18 OPERATIONS
19   addCredit (cr) =
20     PRE
21       cr : NAT1 &
22       card_type /= gratuitous_card &
23       balance + cr <= MAXINT
24     THEN
25       balance := balance + cr
26     END;
27
28   bb <-- queryBalance = BEGIN bb := balance END
29 END
```

mir. Para definirmos estas restrições utilizamos o invariante (cláusula *INVARIANT*) da máquina (linhas 9 a 12). O invariante de uma máquina deve ser sempre preservado em todos os seus estados para que ela se mantenha consistente.

As restrições do invariante são descritas através de cláusulas lógicas que definem os tipos das variáveis de estado e restrições para os estados válidos do sistema. O invariante da máquina *Transport* possui duas cláusulas de tipagem, (*balance : NAT*) e (*card_type : CARD_TYPES*), que definem respectivamente os tipos de *balance* e *card_type* como *"pertence ao conjunto dos números naturais"* e *"pertence ao conjunto CARD_TYPES"*. Além destas, possui a cláusula de restrição (*not(card_type = gratuitous_card) or balance = 0*) que define que, se um cartão possui saldo diferente de 0, ele não pode ser do tipo gratuito.

Para definir os estados iniciais da máquina de estados abstrata, utilizamos a cláusula *INITIALISATION* da notação B (linhas 14 a 16). Ela é utilizada para atribuir valores iniciais às variáveis de estado. Podemos comparar a inicialização de uma máquina B ao

construtor de uma classe no paradigma de orientação a objetos.

Uma máquina de estados abstrata também precisa de algum mecanismo para modificar ou consultar o seu estado. No método B a única forma de alterar o estado de uma máquina é através de operações. As operações de uma máquina são definidas na cláusula *OPERATIONS* (linhas 18 a 28).

Na notação B o cabeçalho de uma operação contém seu nome e pode conter um conjunto de parâmetros e variáveis de retorno (ex: linha 19 apresenta uma operação com parâmetros e a linha 28 uma operação com retorno). Uma operação também pode possuir pré-condições que são utilizadas para definir tipos para seus parâmetros (caso existam) e um conjunto de restrições que devem ser satisfeitas antes de sua execução. A responsabilidade de obedecer a pré-condição de uma operação é do usuário que a invoca. O método não provê nenhuma garantia quanto a execução correta da operação caso a pré-condição seja desrespeitada.

Consideremos inicialmente a operação *addCredit* (linhas 19 a 26). Ela não possui variáveis de retorno e possui apenas um parâmetro de nome *cr*, que representa a quantidade de créditos que serão adicionados no cartão. Sua pré-condição (linhas 20 a 23) possui uma cláusula de tipagem (*cr : NAT1*), que define que *cr* pertence aos naturais positivos, e duas cláusulas de restrição (*card_type /= gratuitous_card*) e (*balance + cr <= MAXINT*), que respectivamente definem que o tipo do cartão deve ser diferente de "gratuito" e que a soma de *cr* ao saldo atual do cartão não deve ultrapassar uma constante *MAXINT*. A constante *MAXINT* representa o valor máximo que um inteiro pode assumir. Esta pré-condição certifica que créditos não sejam adicionados em cartões gratuitos e que após a adição dos créditos o saldo não ultrapasse o limite suportado.

No corpo de uma operação devemos informar seu comportamento e os valores que as variáveis de retorno devem assumir (caso existam). O comportamento da operação *addCredit* é adicionar uma quantidade de créditos *cr* ao saldo atual do cartão (linha 25).

A operação *queryBalance* (linha 28) é utilizada para consultar o saldo atual do cartão. Ela não possui pré-condições e seu comportamento consiste apenas em atribuir o valor da variável *balance* à variável de retorno *bb*.

2.2.5 Obrigações de Prova

Após a especificação de uma máquina B, ela deve ser verificada para garantirmos a sua coerência. No Método B, esta verificação é feita através de *obrigações de prova*. As

obrigações de prova são expressões lógicas geradas a partir da especificação e que devem ser validadas para garantir tal coerência.

Algumas obrigações de prova apresentadas a seguir utilizam a notação de substituição generalizada *[S]P*, apresentada na seção 2.2.2 deste capítulo. Nesta notação, P é um predicado que deve ser mantido após a execução de S. Ou seja, P ainda será uma condição verdadeira após a execução de S.

As principais obrigações de prova para uma máquina B são: consistência do invariante, obrigação de prova da inicialização e obrigações de prova para as operações.

Consistência do invariante

A consistência do invariante é provada pela constatação de que a máquina possui pelo menos um estado válido. Para que isto seja possível, deve existir pelo menos uma combinação de valores para as variáveis da máquina que respeitem o invariante. A fórmula utilizada para tal verificação é a seguinte:

$$\exists v.I$$

Onde v são as variáveis de estado da máquina e I o seu invariante.

Para verificarmos a consistência do invariante da máquina *Transport* utilizamos a seguinte obrigação de prova:

$$\begin{aligned} \exists\ (balance,\ card\_type)\ .\ (&balance\ \in\ \mathbb{N}\ \wedge \\ &card\_type\ \in\ CARD\_TYPES\ \wedge \\ &(not(card\_type\ =\ gratuitous\_card)\ \vee \\ &\quad balance\ =\ 0)) \end{aligned}$$

Ao substituirmos *balance* na fórmula pelo valor 5 e *card_type* por *entire_card* por exemplo, verificamos que a fórmula é valida, ou seja, a máquina possui pelo menos um estado válido.

Obrigação de prova para inicialização da máquina

A próxima obrigação de prova diz respeito à inicialização da máquina. Com ela, provamos que os estados iniciais da máquina satisfazem o invariante. A fórmula para a obrigação de prova da inicialização é a seguinte:

$$[T]I$$

Onde $[T]$ são as substituições realizadas na inicialização e I é o invariante da máquina.

Seguindo com nosso exemplo, temos a seguinte obrigação de prova para a inicialização de *Transport*:

$$\begin{aligned}
&[balance := 0,\ card_type :: CARD_TYPES](\\
&\qquad balance \in \mathbb{N} \wedge\\
&\qquad card_type \in CARD_TYPES \wedge\\
&\qquad (not(card_type = gratuitous_card) \vee balance = 0)\\
&)
\end{aligned}$$

Após a substituição teríamos *balance* assumindo o valor *0* e *card_type* assumindo um elemento qualquer do conjunto $CARD\_TYPES$. Isto resultaria na seguinte obrigação de prova:

$$0 \in \mathbb{N} \wedge$$

$$\forall x.(x \in CARD\_TYPES \Rightarrow card\_type := x) \wedge$$

$$(not(x = gratuitous\_card) \vee 0 = 0)$$

Esta fórmula resulta em verdadeiro e a obrigação de prova para a inicialização da máquina é concluída.

Obrigação de prova para as operações

Finalizamos a verificação de uma máquina realizando as obrigações de provas para suas operações. Esta obrigação de prova tem o objetivo de garantir que, durante a execução de uma operação, a máquina sai de um estado válido para outro estado válido. Obtemos a obrigação de prova para uma operação através da seguinte fórmula:

$$I \wedge P \Rightarrow [S]I$$

Onde I é o invariante da máquina, P é a pré-condição da operação e S é a substituição feita no corpo da operação. Esta fórmula afirma que, se o invariante e a pré-condição da operação são satisfeitos, então após a substituição S o invariante continuará sendo satisfeito. Ou seja, a máquina continua em um estado válido.

Para a operação *addCredit*, esta fórmula resultaria na seguinte obrigação de prova:

$$
\begin{array}{l}
(balance \in \mathbb{N} \wedge \\
\quad card\_type \in CARD\_TYPES \wedge \\
\quad (not(card\_type = gratuitous\_card) \vee balance = 0)) \wedge \\
\\
(cr \in \mathbb{N}* \wedge \\
\quad card\_type \neq gratuitous\_card \wedge \\
\quad balance + cr \leq MAXINT) \Rightarrow \\
\\
[balance := balance + cr]\ (balance \in \mathbb{N} \wedge \\
\qquad\qquad\qquad\qquad card\_type \in CARD\_TYPES \wedge \\
\qquad\qquad\qquad\qquad (not(card\_type = gratuitous\_card) \vee \\
\qquad\qquad\qquad\qquad\quad balance = 0))
\end{array}
$$

Que após a substituição resultaria em:

$$
\begin{array}{l}
(balance \in \mathbb{N} \wedge \\
\quad card\_type \in CARD\_TYPES \wedge \\
\quad (not(card\_type = gratuitous\_card) \vee balance = 0)) \wedge \\
\\
(cr \in \mathbb{N}* \wedge \\
\quad card\_type \neq gratuitous\_card \wedge \\
\quad balance + cr \leq MAXINT) \Rightarrow \\
\\
(balance + cr \in \mathbb{N} \wedge \\
\quad card\_type \in CARD\_TYPES \wedge \\
\quad (not(card\_type = gratuitous\_card) \vee balance + cr = 0))
\end{array}
$$

Como o consequente:

$$(balance + cr \in \mathbb{N} \land \\ card\_type \in CARD\_TYPES \land \\ (not(card\_type = gratuitous\_card) \lor balance + cr = 0))$$

é verdadeiro de acordo com suas premissas, garantimos que o invariante é respeitado e finalizamos a obrigação de prova para a operação *addCredit*, consequentemente provando a coerência da máquina *Transport* (por questão de espaço omitimos as obrigações de prova para *queryBalance*).

2.2.6 Ferramentas para o Método B

O Método B possui algumas ferramentas que auxiliam no processo de especificação de verificação de modelos. A seguir apresentamos duas destas ferramentas: o *AtelierB* e o *ProB*.

AtelierB

O *AtelierB*[1] é uma ferramenta para especificação e verificação de modelos do Método B, desenvolvida pela *Clearsy*[2]. A ferramenta dispõe de uma série de mecanismos que auxiliam o desenvolvedor durante as várias etapas do processo de desenvolvimento formal do Método B. Ela dispõe ainda de funcionalidades que auxiliam práticas comuns da indústria como gerenciamento de projetos e geração de documentação.

Dentre as principais funcionalidades do *AtelierB* podemos citar: um editor para especificação dos módulos da notação B, provadores automáticos, assistentes para o processo de refinamento e um provador iterativo para obrigações de prova que não puderam ser provadas automaticamente.

O *AtelierB* é gratuito, possui versões para Windows, Linux e OS X; e alguns de seus componentes possuem código aberto.

[1] http://www.atelierb.eu/en/
[2] http://www.clearsy.com/

ProB

O *ProB*[3] é um animador e *model-checker* para o Método B, *Event-B*, *Z* e *CSP-M*[ROSCOE, 1997]. Ele permite que especificações destas notações sejam automaticamente animadas, facilitando a detecção de problemas na especificação. A ferramenta possui ainda funcionalidades para geração do grafo de estados de uma máquina, detecção de *deadlocks* e busca por inconsistências no modelo.

O *ProB* é gratuito, possui código-aberto e também possui versões para Windows, Linux e OS X. Em nosso trabalho utilizamos o *parser* do *ProB* para auxiliar na leitura de informações de uma máquina.

2.3 Testes de Software

Quando desenvolvemos um produto precisamos testá-lo para garantir que o mesmo ofereça certo nível de segurança e qualidade para seus usuários. Com software isso não poderia ser diferente. Existem várias estratégias, técnicas e critérios para testá-los.

Testar um software consiste em executá-lo com a finalidade de encontrar erros [MYERS, 2011]. Para testar um sistema costumamos criar *casos de teste* que têm como objetivo exercitar seus possíveis cenários de utilização. Através de casos de teste podemos passar entradas para um sistema e analisar os resultados produzidos como sua saída. Os resultados podem então ser analisados por um *oráculo de testes*, responsável por verificar se programa passou ou não no caso de teste. Um bom caso de teste é aquele que tem alta probabilidade de encontrar defeitos introduzidos no sistema que ainda não foram descobertos.

Podemos classificar testes de software em *tipos* que diferem de acordo com a visão que o engenheiro de testes possui do sistema. Existem também classificações quanto a *níveis*, que variam de acordo o nível do processo de desenvolvimento em que os testes são aplicados. Podemos fazer testes desde os níveis mais abstratos como o de especificação, até níveis mais concretos, como o de implementação do programa. Como não podemos garantir que os testes realizados cobrem todos os possíveis cenários de utilização do sistema – já que em muitos casos isso é computacionalmente impossível – é necessário também definir *critérios* mínimos para que ele seja considerado suficientemente testado. Os conceitos de *tipos*, *níveis de teste* e *critérios de cobertura* são explicados nas próximas

[3] http://www.stups.uni-duesseldorf.de/ProB/

seções.

2.3.1 Terminologia

Antes de continuarmos com este capítulo é necessário apresentar a terminologia que será utilizada no decorrer do texto. A terminologia que empregamos é a apresentada em [AMMANN; OFFUTT, 2008].

Primeiramente precisamos definir os conceitos de *defeito*, *erro* e *falha*:

- *Defeito*: um defeito estático no software. Por exemplo, um defeito no código do programa que poderá resultar em problemas durante a execução;
- *Erro:* é a manifestação do defeito, um estado interno do software que é incorreto;
- *Falha:* é o comportamento externo do software que é resultante do erro.

Para contextualizar estes conceitos imagine um laço dentro de um programa que deve percorrer um vetor de tamanho 5. Como na grande maioria das linguagens de programação os índices de um vetor começam de zero, o laço responsável por percorrer este vetor deveria iterar de 0 a 4. Considere que ao escrever este laço o programador definiu uma iteração de 0 a 5. Tal iteração é o *defeito* do programa. Este defeito irá ocasionar um *erro* quando o programa acessar o índice 5 do vetor. Esse erro pode então ocasionar uma falha, que é a manifestação externa do erro acontecido internamente.

Outros termos que utilizaremos bastante no decorrer do texto dizem respeito a *requisitos de teste* e *critérios de cobertura*. Um *requisito de teste* é um elemento especifico de um artefato de software que um caso de teste deve satisfazer ou cobrir.

Ao projetar os testes de um sistema, o engenheiro costuma definir um conjunto de requisitos de teste que devem ser satisfeitos. Isso pode ser feito através da definição de regras chamadas *critérios de cobertura*. Um *critério de cobertura* é uma regra ou coleção de regras que impõem requisitos de teste a um conjunto de casos de teste. Mais detalhes sobre critérios de cobertura são apresentados nas seção 2.3.4.

Utilizamos ainda em nosso trabalho os conceitos de *testes positivos* e *testes negativos*. Um *teste positivo* é aquele que utiliza dados de teste que respeitam as restrições impostas pela especificação. Por outro lado, um *teste negativo* é aquele que utiliza dados de teste que desrespeitam estas restrições.

2.3.2 Tipos de Teste

Existem várias definições de formas de testar um software e, em geral, podemos classificá-las em dois tipos: *testes estruturais* (ou caixa branca) e *testes funcionais* (ou caixa preta).

Em testes estruturais, o funcionamento interno do programa é conhecido pelo engenheiro de testes, ou seja, todas as funções internas do programa são conhecidas e podem ser testadas tanto individualmente quanto integradas a outros componentes internos.

Uma das vantagens que o conhecimento da estrutura interna do programa traz durante os testes, é a possibilidade de testar diferentes condições lógicas e caminhos de execução que o programa possui. Desta forma podemos mensurar a quantidade de código que está sendo verificado por um conjunto de testes.

Planos de testes estruturais costumam utilizar estruturas de grafos ou fluxogramas para determinar requisitos de teste. Como exemplos de requisitos de teste para este tipo de teste temos: exercitar todos os caminhos de um grafo ou visitar todos os seus nós (ou estados).

Testes funcionais são aqueles projetados sem levar em consideração a estrutura interna do programa. Estes testes são realizados através da interface do software e têm como objetivo validar suas funcionalidades fundamentais.

Este tipo de teste costuma ser aplicado nas etapas finais do ciclo de desenvolvimento, quando o sistema ou parte dele já está pronto para uso. Outra característica deste tipo de teste é que ele pode ser executado por usuários finais do sistema.

Estes testes são geralmente utilizados para encontrar discrepâncias entre o software implementado e os requisitos levantados pelo usuário, detectar problemas de interface, de desempenho do programa, entre outros problemas relacionados à sua utilização.

Estes tipos de teste não são considerados alternativos, ou seja, o engenheiro de testes não deve escolher entre um ou outro. Eles são complementares e na maioria das vezes podem encontrar diferentes tipos de defeitos.

2.3.3 Níveis de Teste

Durante o processo de desenvolvimento de um sistema, defeitos podem ser inseridos em várias de suas etapas. Nesta seção apresentamos as definições de níveis de teste, que

diferem de acordo com as atividades do processo de desenvolvimento a que se relacionam. A relação entre níveis de teste e atividades do ciclo de desenvolvimento são mostradas na Figura 2.1, que apresenta o modelo de testes em V. Na literatura de testes de software, podemos encontrar diversas formas de nomear e classificar níveis de teste. Utilizaremos a classificação feita por [AMMANN; OFFUTT, 2008], que divide os níveis em: *testes de aceitação*, *testes de sistema*, *testes de integração*, *testes de módulo* e *testes de unidade*.

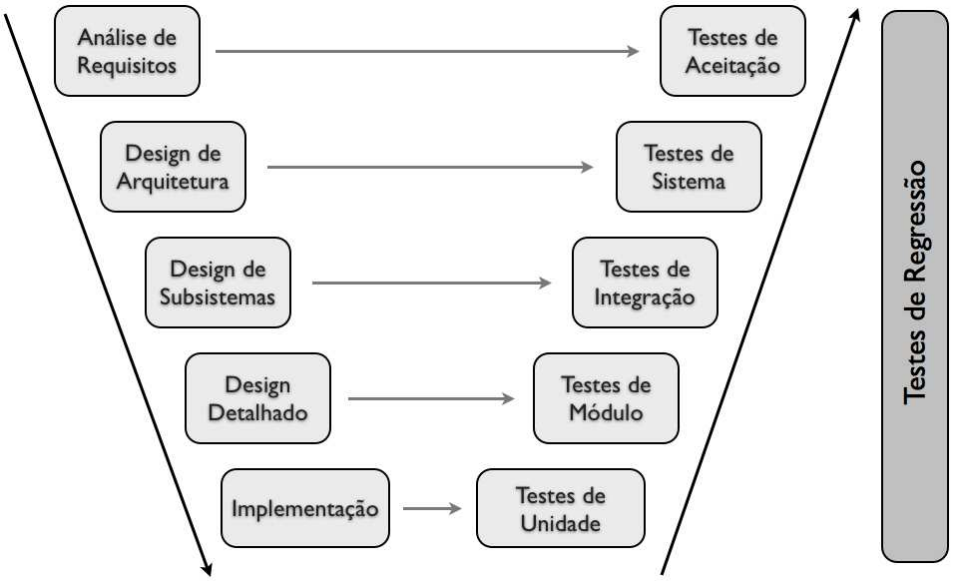

Figura 2.1: Modelo V

Testes de Aceitação

Testes de Aceitação estão relacionados às atividades de análise e elicitação de requisitos, que consistem em identificar e documentar as necessidades do usuário que deverão ser sanadas pelo software desenvolvido. Estes testes têm como objetivo verificar se o software implementado está realmente de acordo com os requisitos levantados. Eles podem ser realizados manualmente, por um usuário final interagindo com o programa, ou automaticamente, por ferramentas que simulam a interação do usuário. A participação do usuário é de extrema importância durante a elaboração de testes de aceitação, já que somente ele pode dar o veredicto final sobre o que foi implementado.

Testes de Sistema

Testes de Sistema estão relacionados à atividade de design do projeto de arquitetura de um sistema. Sistemas computacionais costumam ser construídos a partir da integração de vários componentes de hardware e software, que juntos irão prover uma solução para as

necessidades do usuário. Testes de Sistema tem como objetivos verificar se a composição escolhida atende às especificações e procurar por erros de design e especificação. Neste nível de teste assume-se que cada componente foi testado e funciona corretamente de maneira isolada.

Testes de Integração

Como dito no tópico anterior, sistemas computacionais podem ser construídos a partir de vários componentes. A comunicação entre estes componentes é feita a partir de suas interfaces, que emitem e recebem informações que devem ser processadas por eles. O objetivo dos testes de integração é verificar se estas interfaces se comunicam devidamente. Para isto, são criados casos de teste que exercitem a comunicação entre componentes, detectando possíveis falhas em suas interfaces.

Testes de Módulo

Testes de Módulo estão relacionados à atividade de design detalhado do processo de desenvolvimento. Esta atividade tem o objetivo de determinar a estrutura dos módulos da nossa implementação. Módulos são arquivos que reúnem um conjunto de unidades e variáveis. Unidades são instruções ou procedimentos nomeados, que podem ser chamados em algum ponto do programa através de seu nome. Se consideramos a linguagem Java por exemplo, classes seriam nossos módulos e métodos nossas unidades. Testes de Módulo tem o objetivo de analisar um módulo de forma isolada, verificando como suas unidades e variáveis interagem entre si.

Testes de Unidade

Testes de Unidade estão relacionados ao nível mais concreto do processo de desenvolvimento: a implementação de unidades (métodos, funções, etc). Estes testes tem o objetivo de verificar cada unidade de forma isolada, sem preocupações com o restante do escopo em que ela se encontra. Testes de unidade costumam ser desenvolvidos paralelamente à implementação do programa, para garantir que as funcionalidades implementadas se comportam como o esperado.

Em algumas bibliografias, testes de módulo e de unidade são considerados como um único nível.

Testes de Regressão

Outra atividade presente durante todo o processo de desenvolvimento é a manutenção da implementação. Constantemente fazemos correções e atualizações em nossas implementações, e precisamos garantir que tudo ainda funciona como deveria após estas alterações. Para isso, utilizamos Testes de Regressão. Estes testes devem ser executados sempre que alguma modificação na implementação for realizada e devem assegurar que a versão atualizada ainda possui as mesmas funcionalidades que possuía antes da atualização e funciona ao menos tão bem quanto a versão anterior.

2.3.4 Critérios de Cobertura

Ao testarmos um software, precisamos garantir que ele funciona corretamente em diversos cenários. Para isso, definimos casos de teste que exercitem estes cenários e garantam que o programa se comporta como o esperado. O ideal seria que pudéssemos testar todos os cenários possíveis, mas como isso é inviável na maioria dos casos, devido a limitações econômicas e computacionais, precisamos limitar o número de casos de teste criados para um sistema.

Para reduzir a quantidade de testes necessários em um conjunto de testes, podemos utilizar *critérios de cobertura* que definirão uma regra de parada para a criação de testes. Informalmente, podemos definir critérios de cobertura como um conjunto de regras que impõem requisitos de teste para um conjunto de testes. O objetivo de um critério de cobertura é reduzir a quantidade de testes em um conjunto de testes, sem que o mesmo perca qualidade.

Podemos afirmar que um conjunto de casos de teste obedece um critério de cobertura se, para cada requisito de teste do critério, existe ao menos um teste do conjunto que o satisfaz. Podemos ainda calcular o nível de cobertura de um conjunto de testes T, dividindo a quantidade de requisitos que ele satisfaz, pela quantidade total de requisitos de teste RT. A fórmula é apresentada a seguir:

$$\frac{n^o \text{ de rt's satisfeitos por } T}{\text{tamanho de } RT}$$

Utilizando esta fórmula podemos definir níveis de cobertura para estabelecer metas para nossos casos de teste. Como nem sempre é possível satisfazer todos os requisitos de teste, podemos estabelecer um nível mínimo que deve ser alcançado pela nosso conjunto

de testes.

Existem diversos tipos de critérios de cobertura na literatura como: *cobertura por grafos*, *cobertura lógica*, *particionamento do espaço de entrada* e *testes baseados em sintaxe* [AMMANN; OFFUTT, 2008]. Para nosso trabalho, o critério mais adequado é o de particionamento do espaço de entrada, pois utilizamos especificações para gerar testes. A seguir apresentaremos em detalhes este critério de cobertura.

Particionamento do espaço de entrada

De maneira simplificada, um teste de software consiste em selecionar um valor do espaço de entrada de um programa, passar este valor como entrada para ele e verificar o que acontece após a sua execução. É levando em consideração a importância que as entradas exercem sobre o comportamento de um software, que o particionamento do espaço de entrada define critérios de cobertura. Os conceitos de particionamento do espaço de entrada utilizados nesta seção são apresentados em [AMMANN; OFFUTT, 2008].

Para fazer este particionamento, primeiramente precisamos definir as variáveis que compõem o espaço de entrada da função sob teste. Estas variáveis, dependendo do contexto, podem ser: campos de um formulário, parâmetros de um método, variáveis de uma classe ou de uma máquina de estados.

Após definir as variáveis que fazem parte do espaço de entrada da função sob teste, é necessário determinar os possíveis valores que estas variáveis podem assumir. Ao conjunto universo que contém todos estes valores damos o nome *domínio de entrada*.

O particionamento do espaço de entrada consiste em dividir o domínio de entrada de um programa em subdomínios menores chamados *classes de equivalência* (ou simplesmente *blocos*). Estes blocos são criados com base em *características* do programa ou de suas entradas. Cada característica dá origem a um conjunto de blocos e cada um destes blocos possui um conjunto de dados considerados equivalentes ao testarmos tal característica.

Por exemplo, considere a função de um programa que recebe como parâmetro o número de um jogador de um time de futebol e retorna verdadeiro ou falso, caso o mesmo esteja ou não escalado no time principal. O parâmetro jogador possui como uma de suas características estar escalado ou não no time principal. Sendo assim, podemos fazer o particionamento baseado nesta característica utilizando dois blocos:

- B_1: jogadores escalados no time principal;
- B_2: jogadores não escalados no time principal.

Cada um destes blocos possui um subconjunto do espaço de entrada, cujos elementos são considerados equivalentes ao testarmos o programa levando em consideração esta característica. Ou seja, para qualquer elemento selecionado de uma destas classes, espera-se que o efeito causado durante o teste seja o mesmo que qualquer outro elemento deste bloco causaria.

O particionamento do domínio de entrada em blocos é feito através da criação do *Modelo do Domínio de Entrada*, que pode ser definido seguindo as etapas abaixo:

1. *Identificar funções testáveis:* devemos identificar as funcionalidades que pretendemos testar. Podem ser métodos de uma classe, funcionalidades retiradas de um diagrama de casos de uso, etc;
2. *Identificar os parâmetros que afetam o comportamento da função:* após identificarmos a função a ser testada precisamos encontrar as variáveis que afetam seu comportamento. Estas variáveis podem ser parâmetros de entrada de um método e/ou variáveis de estado das quais o comportamento do método depende. Estas variáveis formam o espaço de entrada da função testada e os valores que elas podem assumir formam o domínio de entrada;
3. *Modelar o domínio de entrada:* são criadas partições para cada característica da função testada. Cada partição é formada por um conjunto de blocos de valores equivalentes.

Toda partição criada deve obedecer a duas propriedades:

1. O particionamento deve cobrir todo o domínio de entrada da função testada, ou seja, a união de todos os blocos deve ser igual ao domínio de entrada;
2. Não deve haver sobreposição entre os blocos criados, ou seja, a intersecção entre os blocos deve ser vazia.

Existem várias estratégias para fazer o particionamento em blocos de maneira mais significativa. Algumas destas estratégias são:

- *Valores válidos*: incluir blocos de valores válidos segundo características do programa;
- *Valores inválidos*: incluir blocos de valores inválidos segundo características do programa;
- *Valores limite*: incluir valores próximos ou no limite de intervalos para as variáveis do espaço de entrada.

Critérios para combinação de dados de teste

Depois de selecionar os blocos de dados de teste, é necessário definir como eles podem ser combinados para a criação de casos de teste. Para isso existem alguns critérios para a combinação de dados [AMMANN; OFFUTT, 2008] que podem nos auxiliar a gerar estas combinações de maneira mais eficaz.

Todas as combinações:

Critério: todas as combinações de blocos de todas as características devem ser testadas.

Imagine o seguinte cenário onde temos 3 partições com os seguintes blocos: [α, β], [1, 2, 3] e [x, y]. Se quiséssemos gerar testes com estes blocos seguindo como critério todas as combinações, teríamos como resultado o seguinte conjunto de testes:

$$\begin{aligned} CTs = \{&(\alpha,\ 1,\ x),\ (\alpha,\ 1,\ y),\ (\alpha,\ 2,\ x),\ (\alpha,\ 2,\ y),\\ &(\alpha,\ 3,\ x),\ (\alpha,\ 3,\ y),\ (\beta,\ 1,\ x),\ (\beta,\ 1,\ y),\\ &(\beta,\ 2,\ x),\ (\beta,\ 2,\ y),\ (\beta,\ 3,\ x),\ (\beta,\ 3,\ y)\} \end{aligned}$$

O número de testes necessários para este critério de cobertura será igual ao produto do número de blocos de cada partição (para o exemplo dado: 2 x 3 x 2 = 12). Criar casos de teste para todas as combinações possíveis costuma ser economicamente inviável e dependendo da quantidade de blocos envolvidos pode resultar em um explosão combinatorial.

Each Choice:

Critério: Um valor de cada bloco para cada característica deve ser utilizado em pelo menos um caso de teste.

Para o exemplo dado anteriormente, poderíamos satisfazer este critérios de diversas maneiras:

$$CTs_1 = \{(\alpha, 1, x), (\beta, 2, y), (\alpha, 3, y)\}$$

$$CTs_2 = \{(\alpha, 1, x), (\beta, 2, y), (\beta, 3, x)\}$$

Tanto o conjunto de testes CTs_1 como CTs_2 obedecem este critério. Este critério é baseado no conceito de particionamento de equivalência clássico, onde basta que algum teste contenha algum dado do bloco para que o bloco seja considerado testado.

Para este critério de cobertura o número de testes necessários para satisfaze-lo será igual ao número de blocos da partição com mais blocos. Para o nosso exemplo apenas três casos de teste são necessários para satisfazer o critério, que é o número de blocos da segunda partição.

Por permitir muita flexibilidade para a seleção dos dados de teste este critério é muitas vezes considerado fraco.

Pairwise:

Critério: Um valor de cada bloco para cada característica deve ser combinado a um valor de todos os blocos para cada outra característica.

Em outras palavras, todas as possíveis combinações dois a dois entre os blocos de duas partições devem estar ser testadas. Para nosso exemplo, devemos garantir que as seguintes duplas de blocos sejam testadas:

$$\begin{array}{llllllll} (\alpha, & 1), & (\alpha, & 2), & (\alpha, & 3), & (\alpha, & x), \\ (\alpha, & y), & (\beta, & 1), & (\beta, & 2), & (\beta, & 3), \\ (\beta, & x), & (\beta, & y), & (1, & x), & (1, & y), \\ (2, & x), & (2, & y), & (3, & x), & (3, & y) \end{array}$$

Os seguintes testes são necessários para cobrir as duplas acima (onde "$-$" é um *wild-card* que pode ser substituído por qualquer bloco da partição):

$$\begin{array}{rcl} CTs & = & \{(\alpha, 1, x), (\alpha, 2, x), (\alpha, 3, x), (\alpha, -, y), \\ & & (\beta, 1, y), (\beta, 2, y), (\beta, 3, y), (\beta, -, y)\} \end{array}$$

2.4 Considerações Finais

Neste capítulo apresentamos os conceitos necessários para o entendimento do restante do trabalho. Fizemos uma introdução as características básicas do Método B, apresentamos conceitos de teste de software como tipos de teste, níveis de teste e critérios de cobertura.

Nosso trabalho utiliza uma abordagem capaz de gerar testes de unidade para um programa a partir de suas especificações em máquinas B. Utilizando restrições descritas nas especificações de uma máquina, definimos características a serem testadas e a partir delas definimos blocos de dados de teste. São criados blocos de valores válidos, inválidos e de valores limite. Por fim, estes blocos podem ser combinados seguindo critérios de combinação de dados de teste como *todas as combinações*, *each choice* e *pairwise*.

3 Trabalhos Relacionados

3.1 Considerações Iniciais

Neste capítulo apresentamos uma lista de trabalhos relacionados que buscam conciliar métodos formais e testes, mais especificamente, artigos que apresentam o uso de especificações formais como fonte para técnicas de Teste Baseado em Modelos.

Apesar do foco do nosso trabalho ser a geração de casos de testes a partir de máquinas B, foram considerados trabalhos sobre geração de casos de teste a partir de diversas linguagens de especificação, tais como: Z [SPIVEY, 1992], Object-Z [SMITH, 1999], Alloy [JACKSON, 2002], JML [LEAVENS *et al.*, 1998], VDM-SL [PLAT; LARSEN, 1992], VDM++ [CSK, 2005] e OCL [WARMER; KLEPPE, 1999].

Tais notações foram escolhidas por compartilharem características com o Método B, como o conceito de máquinas de estado abstratas, transições entre estados através de operações, e preservação da consistência através de pré-condições e invariantes. O gráfico da Figura 3.1 mostra a distribuição dos trabalhos encontrados durante nossa pesquisa de acordo com o método formal utilizado.

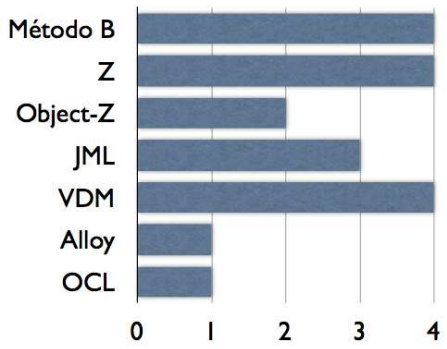

Figura 3.1: Distribuição dos trabalhos de acordo com o método formal utilizado.

Dividimos esta seção em subseções de acordo com a linguagem de especificação utilizada nos trabalhos. No início de cada subseção é apresentada uma tabela com o sumário

das principais características dos artigos que iremos discutir. Cada uma destas tabelas contém as seguintes informações:

- *Tipo de Teste:* que pode ser *Estrutural*, quando o desenvolvimento dos testes é baseado na estrutura do código do software, ou *Funcional*, quando os testes são desenvolvidos baseados apenas nas especificações do software;
- *Nível:* níveis são categorizados em *Aceitação*, *Sistema*, *Integração*, *Módulo*, *Unidade* e *Regressão*. Cada um destes níveis é explicado na seção 2.3.3 do Capitulo 2;
- *Cobertura:* explica o critério de cobertura utilizado no trabalho. Os critérios podem ser divididos em técnicas base como *Cobertura de Grafos*, *Cobertura Lógica*, *Cobertura do Espaço de Entrada* e *Cobertura Baseada na Sintaxe* [AMMANN; OFFUTT, 2008]. No entanto, vale ressaltar que alguns trabalhos desenvolvem seus próprios critérios de cobertura (neste caso tentamos explicar na tabela) ou podem até mesmo não utilizar critério algum;
- *Linguagem de Implementação:* linguagem em que os casos de teste concretos são implementados. Em alguns casos não são gerados testes executáveis mas apenas os dados de teste ou representações abstratas dos testes;

No gráfico da Figura 3.2 temos a distribuição dos trabalhos encontrados em relação ao nível de testes utilizado, enquanto na Figura 3.3 temos a mesma distribuição ao considerarmos apenas trabalhos que utilizam o Método B. Comparações entre nosso trabalho e alguns dos trabalhos encontrados serão feitas nas considerações finais deste capítulo.

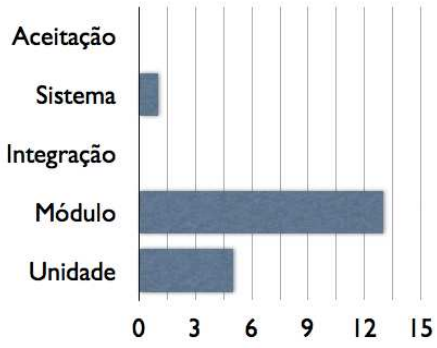

Figura 3.2: Distribuição dos trabalhos de acordo com o nível de testes.

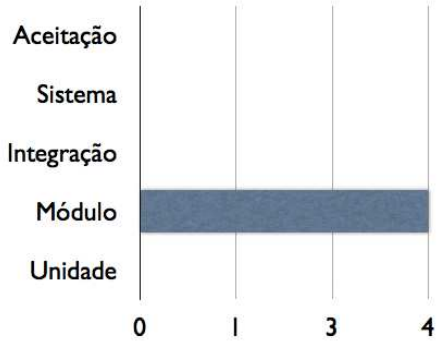

Figura 3.3: Distribuição dos trabalhos que utilizam B de acordo com o nível de testes.

3.2 Método B

Artigo	Tipo	Nível	Cobertura	Ling. de Impl.
[GUPTA; BHATIA, 2010]	Funcional	Módulo	Testar todos os requisitos. Em mais baixo nível, é utilizado cobertura de decisões	Não gera teste executável
[SATPATHY *et al.*, 2007]	Funcional	Módulo	Cada operação deve ser chamada ao menos uma vez	Java
[SATPATHY *et al.*, 2005]	Funcional	Módulo	Cada instância da operação de testes deve ser chamada ao menos uma vez	Java
[AMBERT *et al.*, 2002]	Funcional	Módulo	Análise de Valor Limite	Linguagem qualquer

Tabela 3.1: Trabalhos relacionados que utilizam Método B

Em [AMBERT *et al.*, 2002], os autores desenvolveram a *BZ-TT* (*B and Z Testing Tools*), uma ferramenta para geração de casos de teste, a partir de especificações B e Z, que utiliza a técnica de análise de valor limite. A *BZ-TT* suporta apenas máquinas de teste monolíticas (máquinas cuja especificação deve ser feita em um único arquivo), que possuam somente conjuntos enumerados e pré-condições explícitas (ou seja, todas as operações devem estabelecer pré-condições na cláusula de pré-condição, nenhuma pré-condição deve ficar implícita no corpo da operação).

O método de geração de testes utilizado pela ferramenta é apresentado com detalhes em [LEGEARD *et al.*, 2002]. Ele tem como objetivo exercitar uma máquina quando ela se encontra em um estado limite. Considera-se que uma máquina está em um estado limite quando ao menos uma de suas variáveis de estado possui um valor extremo (máximo ou mínimo).

Nesta abordagem, um caso de teste é uma sequência de chamadas de operações da especificação. Esta sequência é dividida em 4 subsequências:

1. *Preâmbulo:* sequência de operações que leva a máquina a um estado limite.
2. *Corpo:* chamada de uma operação que altera o estado atual da máquina.
3. *Identificação:* chamada de uma operação que observa o estado da máquina.
4. *Finalização:* leva a máquina ao estado inicial novamente para que outros testes possam ser executados.

No *preâmbulo*, para levar uma máquina a um estado limite, o método cria *metas limite*, que são predicados a partir dos quais estados limite podem ser instanciados. Metas limite são criadas a partir de cada operação e é necessário converter a pré e pós-condição de cada operação para a Forma Normal Disjuntiva (FND). Dessa maneira, uma meta limite possuirá a seguinte forma:

$$\exists\ entradas,\ estado',\ saídas\ .\ Pre \wedge Pos_j$$

Onde *entradas* são os parâmetros da operação, *estado'* é o estado da máquina após a execução da operação, *saídas* são as variáveis de retorno da operação, *Pre* é a pré-condição da operação na FND e Pos_j é uma disjunção da FND da pós-condição.

Após definir um conjunto de metas limite, elas são processadas por um solucionador de restrições [BOUQUET *et al.*, 2002] que gera um grafo de estados. Neste grafo, cada caminho que parte do nó inicial é uma sequência de operações que leva a máquina a um estado limite.

Depois de levar a máquina a um estado limite, no *corpo* do caso de teste, uma operação que altere o estado atual da máquina é executada. Esta é instanciada utilizando valores limite para seus parâmetros.

Na etapa de *identificação*, operações que observam o estado da máquina são instanciadas para que um veredicto sobre o teste possa ser dado. Na *finalização* a máquina é levada ao estado inicial para que outros casos de teste possam ser executados.

Depois de definir um conjunto de casos de teste, a *BZ-TT* traduz estes casos de teste em *scripts* de teste executáveis. Além disso, a ferramenta possibilita que um *script* de teste seja executado juntamente com a animação do caso de teste, para que no final os valores retornados por ambos sejam comparados e um resultado seja apresentado.

Finalmente, vale observar que o método utilizado gera tanto casos de teste positivos como negativos. Em um caso de teste positivo, todas as pré-condições da sequência de operações são respeitadas, já em um caso de teste negativo, a pré-condição da última operação da sequência é desrespeitada.

Em [SATPATHY *et al.*, 2005], foi desenvolvida a ferramenta *ProTest*. A *ProTest* utiliza como base o animador *ProB* [LEUSCHEL; BUTLER, 2003] e, através de *model checking*, gera um grafo de estados para uma máquina B. Neste grafo cada nó representa um estado da máquina e cada aresta representa uma operação que a leva de um estado a outro. Cada caminho partindo do estado inicial deste grafo é um caso de teste.

Para gerar este grafo, é feito o particionamento do domínio de entrada de cada operação em subdomínios, pois nesta abordagem, cada subdomínio representa um possível cenário em que a operação pode ser instanciada. O particionamento do espaço de entrada em subdomínios é feito da seguinte forma para cada operação:

1. Transformar a pré-condição da operação para a FND;
2. Caso existam condicionais dentro do corpo da operação, criar fórmulas que representem as possíveis escolhas do condicional e adicioná-las à pré-condição através de conjunção;
3. Filtrar possíveis cláusulas contraditórias que tenham sido adicionadas pelo passo anterior. Após a filtragem temos as disjunções C_1, C_2, ..., C_k que dividem o domínio de entrada da operação em k subdomínios;
4. Criar k instâncias da operação. Assim, cada instância da operação será executada quando a condição C_i for atendida;
5. Explorar a máquina no animador para gerar o grafo.

Os casos de teste são então extraídos do grafo para formar um conjunto de testes. Como critério mínimo de cobertura, o método requer que cada instância de operação apareça ao menos uma vez neste conjunto de testes.

O *ProTest* permite que casos de teste sejam traduzidos em testes executáveis escritos em Java e faz a verificação dos resultados de forma semelhante à [AMBERT *et al.*, 2002], executando os testes juntamente com a animação da especificação para comparar os resultados de ambos.

O trabalho possui algumas restrições quanto às máquinas que podem ser utilizadas: máquinas devem ser monolíticas, operações devem possuir apenas um único retorno, as operações devem ser determinísticas e utilizar apenas tipos básicos (ou seja, tipos que se relacionam a tipagem geralmente utilizada por linguagens de programação como inteiros, booleanos etc.)

Os mesmos autores revisitam o método em [SATPATHY *et al.*, 2007], adaptando-o de forma que possa ser reutilizado para outras linguagens de especificação formal orientadas a modelos como Z e VDM. Também é apresentada uma possível solução para o problema de operações cuja especificação do comportamento possui não-determinismo.

Os autores de [GUPTA; BHATIA, 2010] propõem uma abordagem para geração de testes funcionais a partir de especificações B. Assim como em demais trabalhos, o objetivo da abordagem é testar todos os caminhos alcançáveis pela especificação. Igualmente a outros trabalhos também, um caso de teste consiste em uma sequência de execuções de operações.

O método utiliza anotações de requisitos no código da especificação, que posteriormente são conferidas para garantir que os testes gerados cobrem todos os requisitos elicitados.

Após a especificação e anotação dos requisitos, as máquinas B são validadas no *AtelierB* e depois animadas no *ProB*. Posteriormente as cláusulas de pré e pós condições das máquinas são extraídas e reescritas em uma notação abstrata através de um *parser*.

O resultado do *parser* é usado como entrada para a criação dos casos de teste, juntamente com o critério de cobertura que se deseja utilizar. No artigo é dado um exemplo que utiliza cobertura de decisões, em que é definido como critério que, para cada variável booleana, deve existir um caso de teste em que ela assume o valor verdadeiro e um caso de teste em que ela assume o valor falso. No final temos a descrição de um roteiro de testes onde são listadas sequências de chamadas de operações e os respectivos resultados esperados.

3.3 Z

Artigo	Tipo	Nível	Cobertura	Ling. de Impl.
[BURTON; YORK, 2000]	Funcional	Unidade	Particionamento de Equivalência	Não gera teste executável
[HUAIKOU; LING, 2000]	Funcional	Unidade	Particionamento de Equivalência	Não gera teste executável
[SINGH *et al.*, 1997]	Funcional	Unidade	Cobrir todas as folhas da árvore de classificação	Não gera teste executável
[AMLA; AMMANN, 1992]	Funcional	Módulo	Particionamento de Categorias	Não gera teste executável

Tabela 3.2: Trabalhos relacionados que utilizam Z

Em [AMLA; AMMANN, 1992] é apresentado um método para fazer particionamento de categorias a partir de especificações Z. O particionamento de categorias é uma estratégia de testes baseada em especificações, que utiliza especificações informais para produzir

requisitos de teste [OSTRAND; BALCER, 1988]. Esta técnica utiliza particionamento de equivalência para dividir o espaço de entrada de uma unidade de teste. As partições são definidas a critério de um engenheiro de testes, após observar características importantes de uma unidade de teste presentes na especificação do sistema.

Os autores acreditam que parte do esforço necessário para realizar o particionamento de categorias já é feito durante a especificação formal de um sistema, pois cláusulas do invariante e pré-condições já restringem os valores de variáveis e parâmetros de operações a certas categorias. Desta forma o uso de especificações formais evita o retrabalho, além de prover um método de especificação mais confiável do que requisitos informais.

Para gerar especificações de teste, as seguintes partes de uma especificação Z são utilizadas:

1. esquemas Z que observem ou alterem o estado da máquina e/ou gerem um retorno são consideradas unidades de teste;

2. as entradas de um esquema são os parâmetros de teste;

3. predicados do esquema são utilizados para definir as categorias de teste.

Estas especificações de teste são escritas em uma linguagem chamada TSL [BALCER *et al.*, 1989], que pode ser compilada por uma ferramenta para gerar *scripts* de teste.

Em [SINGH *et al.*, 1997] é apresentado um método para geração de casos de teste, a partir de especificações Z, que combinam as técnicas de particionamento de equivalência (usando FND) e de árvores de classificação.

O método de árvores de classificação particiona o domínio de entrada de um sistema em subcategorias, levando em consideração seus requisitos funcionais. Estas subcategorias podem ser divididas em subcategorias ainda mais específicas, dependendo dos critérios utilizados pelo engenheiro de testes. No final temos uma árvore onde cada nó representa um subdomínio do espaço de entrada que podemos testar.

Os autores integraram as fórmulas geradas pelo particionamento de equivalência aos nós desta árvore, de forma que cada nó possui um conjunto de disjunções que gere dados de teste para seu subdomínio.

A combinação das duas técnicas permite a fácil navegação pelos cenários de teste através da estrutura de árvore, além de tornar mais claro o objetivo de cada conjunto de

fórmulas geradas pelo particionamento de equivalência.

O método foi testado no desenvolvimento de um sistema de navegação para veículos e possui como único suporte ferramental um editor de árvores de classificação: o *CTE* (*Classification-Tree Editor*).

Em [HUAIKOU; LING, 2000] os autores apresentam uma ferramenta que gera casos de teste a partir de especificações Z. Eles apresentam o conceito de *classes de teste*, que são esquemas Z que podem gerar casos de teste para uma operação.

Uma classe de teste define o domínio de entrada de uma operação a partir de cláusulas de tipagem e restrição do esquema para suas variáveis de entrada. Estas cláusulas são utilizadas para gerar os dados de entrada para os casos de teste.

De forma semelhante, os oráculos de um caso de teste são gerados a partir do espaço de saída da operação, que é definido pelas cláusulas de tipagem e cláusulas de restrição da pós-condição para as variáveis de saída da operação.

Ao combinarmos os domínios de entrada e de saída temos um domínio de testes, que define todos os possíveis casos de teste para uma classe de testes. O engenheiro de testes deve então selecionar casos de teste deste espaço utilizando alguma estratégia de testes.

Foi desenvolvida uma ferramenta que recebe como entrada uma classe de teste e gera automaticamente dados de entrada e oráculos. Estes são salvos em um arquivo contendo uma representação abstrata dos dados. A ferramenta não gera casos de teste executáveis.

Em [BURTON; YORK, 2000] é apresentada uma abordagem para geração de casos de teste a partir heurísticas definidas pelo usuário através de especificações Z. Estas heurísticas definem critérios de teste utilizando classes de equivalência ou detecção de faltas.

Classes de equivalência são criadas utilizando a mesma abordagem de trabalhos anteriores, onde predicados de um esquema são utilizados para dividir o domínio de entrada da operação em subdomínios. Já as heurísticas baseadas em detecção de faltas utilizam o conceito de mutantes para gerar dados de testes.

Um mutante é uma réplica da especificação em que foi inserida um defeito que a torna incorreta. Sendo assim, dados de teste gerados a partir de um mutante são úteis para identificar aquele tipo de defeito que foi inserido na especificação.

A abordagem definida pelos autores permite que novas heurísticas sejam criadas e possui como suporte ferramental o *CADiZ*[TOYN, 1996], um provador de teoremas e verificador de tipos.

3.4 Object-Z

Artigo	Tipo	Nível	Cobertura	Ling. de Impl.
[MCDONALD *et al.*, 1997]	Funcional	Módulo	Utiliza Cobertura de Grafos para estados alcançáveis	C++ para framework ClassBench
[FLETCHER; SAJEEV, 1996]	Funcional	Módulo	*Ad hoc*	Eiffel

Tabela 3.3: Trabalhos relacionados que utilizam Object Z

Em [FLETCHER; SAJEEV, 1996] os autores apresentam um framework para geração de casos de teste para sistemas orientados a objetos a partir de especificações escritas em Object-Z. Na abordagem utilizada por este framework, um caso de teste é uma sequência de chamadas de esquemas Z que alteram as variáveis de estado.

É utilizada uma linguagem intermediária para descrever os casos de teste de forma abstrata. Esta linguagem é baseada em expressões regulares e permite que os casos de teste abstratos sejam traduzidos em casos de teste concretos, implementados em uma linguagem de programação orientada a objetos. Esta tradução pode ser feita automaticamente por uma ferramenta.

Ao fazer esta tradução, é definido que a classe de teste criada é uma classe filha da classe que está sendo testada. Desta forma todas as características da classe pai são herdadas pela classe de teste filha. Para que a abordagem funcione, cada classe da especificação deve corresponder a uma classe da implementação.

Após execução dos casos de teste concretos, um arquivo é gerado com os resultados dos testes para análise posterior.

O artigo não cita estratégias de teste ou critérios de cobertura e nada é dito sobre como as sequências são definidas. Além disso, os casos de teste obtidos por esta abordagem não possuem oráculos de teste, não ficando claro como é feita a verificação dos resultados.

Em [MCDONALD *et al.*, 1997] os autores apresentam um método para geração de oráculos de teste a partir de especificações em Object-Z. Os oráculos são gerados em C++ para serem executados no framework *ClassBench*, mas a abordagem pode ser adaptada para outros frameworks de teste.

O método começa com a otimização da especificação para um formato mais apropriado à tradução em C++. Após a otimização é gerado um esqueleto para o código de teste

em C++ e são criadas funções cujo corpo contém validação de expressões definidas nas pré-condições e invariante da especificação.

O artigo não comenta nada sobre automatização do processo ou uso de ferramentas, logo concluímos que o processo é totalmente manual.

3.5 JML

Artigo	Tipo	Nível	Cobertura	Ling. de Impl.
[BOUQUET *et al.*, 2006]	Estrutural	Módulo	Faz cobertura estrutural através de grafos onde os nós são estados alcançáveis	Java
[XU; YANG, 2004]	Estrutural	Unidade	Combinações de entradas, *ad hoc*	Java (JUnit)
[CHEON; LEAVENS, 2002]	Estrutural	Unidade	Dado um conjunto de entradas, são testadas todas as combinações possíveis entre membros desses conjuntos	Java (JUnit)

Tabela 3.4: Trabalhos relacionados que utilizam JML

No aspecto de suporte ferramental, o trabalho que mais se destaca é o *jmlunit* de Cheon e Leavens [CHEON; LEAVENS, 2002]. O *jmlunit* é capaz de gerar esqueletos de teste para o framework *JUnit*[1], a partir de anotações JML feitas nos métodos de uma classe Java. Estes esqueletos de teste contém métodos que verificam se, ao executar o caso de teste, alguma exceção do JML é lançada.

Uma deficiência que a ferramenta possui é a geração dos dados de teste. Estes dados devem ser preenchidos manualmente pelo programador no esqueleto da classe de teste, para que posteriormente ela possa gerar combinações entre estes valores para cada caso de teste.

Em [XU; YANG, 2004], Xu e Yang procuram melhorar o trabalho desenvolvido por Cheon e Leavens criando o *JMLAutoTest*, um framework para geração de casos de teste a partir de JML semelhante ao *jmlunit*, mas com algumas características a mais.

Para resolver o problema da geração dos dados de teste, os autores adicionaram uma classe de suporte: a classe de teste que permite que o programador defina domínios para argumentos de métodos e atributos de classes. Dessa forma, a classe de teste pode gerar

[1] http://junit.org/

valores para seus dados de teste baseados nestes domínios. O programador também pode criar uma estrutura de dados abstrata (que eles chamam de *Operational Profile*) para dividir os valores de teste em partições equivalentes.

Além disso, o *JMLAutoTest* permite que casos de teste que desrespeitam pré-condições sejam removidos através da execução de testes em duas fases. Similaridades entre casos de teste também são tratadas. Sempre que um novo caso de teste é adicionado no conjunto de testes, são feitas comparações com os casos de teste existentes para que não existam casos de teste com os mesmos objetivos.

Finalizando, em [BOUQUET *et al.*, 2006] Bouquet, Dadeau e Legeard aplicam a mesma abordagem utilizada na ferramenta *BZ-TT* [AMBERT *et al.*, 2002] para gerar testes para programas especificados em JML, utilizando análise de valor limite.

3.6 VDM-SL e VDM++

Artigo	Tipo	Nível	Cobertura	Ling. de Impl.
[NADEEM; LYU, 2006]	Funcional	Módulo	Vários critérios de cobertura de grafos	Não são gerados testes executáveis. A saída é um conjunto de sequências de operações
[NADEEM; UR-REHMAN, 2004]	Funcional	Módulo	Particionamento de Equivalência e Análise de Valor de Limite	C
[AICHERNIG, 1999]	Funcional	Módulo	Não geram casos de testes, apenas oráculos	C++
[DICK; FAIVRE, 1993]	Funcional	Módulo	Particionamento de Equivalência	Não gera casos de teste executáveis

Tabela 3.5: Trabalhos relacionados que utilizam VDM-SL ou VDM++

O trabalho de Dick e Faivre [DICK; FAIVRE, 1993] serviu como base para muitos dos artigos citados neste capítulo, principalmente para os que realizam particionamento de equivalência utilizando a forma normal disjuntiva. Neste trabalho, os autores utilizaram a FND de uma especificação para particionar seus estados em classes de equivalência. Desta maneira, cada uma das disjunções presentes na FND representa uma classe de equivalência. Vale ressaltar no entanto que apesar dos autores utilizarem o termo "particionamento em classes de equivalência", a técnica empregada por eles não está de acordo com as regras

de particionamento apresentadas na seção 2.3.4 do Capítulo 2

Outro problema explorado no artigo é a geração de sequências de testes. O método possibilita a criação de um autômato de estados finito, que representa os possíveis estados de uma especificação e as transições entre eles. O autômato é obtido através dos seguintes passos:

1. Fazer a análise de partições para todas as operações individualmente, obtendo assim um conjunto de sub-operações. Uma sub-operação é uma operação de teste criada para cada disjunção de uma FND;

2. Extrair de cada sub-operação as restrições que definem o estado antes de sua execução e as restrições que definem seu estado depois da execução;

3. Fazer a análise de partições nos conjuntos de restrições obtidos no passo 2. As partições resultantes descrevem estados em que ao menos uma sub-operação é capaz de criar (obtido através da pós-condição), ou estados em que uma operação é executável (obtidos através da pré-condição);

4. No passo 1 foram obtidas as transições e no passo 3 os estados da especificação, com estas informações, um solucionador de restrições é utilizado para construir o autômato de estados finito.

Após a criação do autômato, ele é explorado para extrair as sequências de testes. O critério para seleção de sequências de testes utilizado pelo método é que, partindo do estado inicial, uma sequência deve passar por todas as transições possíveis ao menos uma vez. Este processo deve ser feito manualmente já que não foi implementada uma ferramenta para a construção do autômato e seleção das sequências de testes.

Em [AICHERNIG, 1999], Aichernig apresenta um framework que utiliza especificações VDM-SL como oráculos de teste. Na abordagem utilizada, um oráculo é uma função booleana que retorna o valor verdade se o valor retornado pelo caso de teste estiver de acordo com a pós-condição de sua especificação.

Como os dados utilizados no nível de especificação são abstratos, é necessário implementar uma função de mapeamento entre dados abstratos e seus respectivos valores concretos. Desta forma, o processo consiste em: 1) mapear as entradas do caso de teste para sua forma abstrata; 2) passar estes dados abstratos pela especificação e verificar se as pré-condições são obedecidas; 3) mapear as saídas da implementação concreta para

sua forma abstrata; 4) verificar se os valores abstratos das saídas estão de acordo com a pós-condição da especificação.

Nenhuma ferramenta foi desenvolvida durante o trabalho, ao invés disso foi utilizada uma ferramenta comercial – a *IFAD VDM-SL* – para a verificação de modelos e transformações VDM-SL para C++.

Em [NADEEM; UR-REHMAN, 2004], os autores apresentam um framework para geração de testes a partir de especificações VDM-SL. No artigo, os testes são gerados em linguagem C, mas a abordagem pode ser utilizada em outras linguagens de programação.

Nesta abordagem, pré e pós condições são convertidas em funções, de forma semelhante a abordagem utilizada pelo *jmlunit*. É feita uma modificação no código da função para que a mesma chame um método que verifica se a pré-condição é respeitada, garantindo assim que a mesma foi executada em um estado válido. Após a execução da função um método que verifica se a pós-condição foi estabelecida é executado.

Os dados de entrada para os casos de teste são gerados a partir de predicados das pré-condições e do invariante, e através do particionamento do domínio de entrada. Um *driver* de teste executa os casos de teste gerados no código modificado, compara os resultados com a pós-condição e grava um log com resultados dos casos de teste.

Em [NADEEM; LYU, 2006] é apresentada uma abordagem para testes de herança a partir de especificações VDM++. A VDM++ é uma extensão da VDM-SL que suporta abstrações do paradigma de orientação a objetos. Ela permite o uso de classes, subclasses, operações, funções, variáveis de estado e constantes. Especificações VDM++ permitem ainda a definição de *traces* que especificam possíveis ordens para as chamadas de operações de uma classe.

A abordagem para geração de testes consiste das seguintes etapas:

- *Unir uma classe à sua superclasse:* classes e suas respectivas superclasses são unidas em uma única classe de comportamento equivalente.
- *Definir possíveis sequências de operações:* a partir das especificações de *traces*, são definidas as possíveis sequências de operações de uma classe.
- *Definir domínio de entrada para as operações:* usando o invariante, pré-condições e os tipos de cada variável, o domínio de entrada de cada operação é determinado.
- *Estabelecer modelo de testes:* estabelece uma estrutura não linear que representa

uma coleção de sequências de operações. Cada sequência de operações difere das outras pela partição do domínio de entrada que está cobrindo.

- *Gerar caminhos de teste:* escolher sequências de operações do modelo de testes.

3.7 Alloy

Artigo	Tipo	Nível	Cobertura	Ling. de Impl.
[MARINOV; KHURSHID, 2001]	Estrutural	Módulo	O artigo não menciona critérios de cobertura	Java

Tabela 3.6: Trabalhos relacionados que utilizam Alloy

Em [MARINOV; KHURSHID, 2001], Marinov e Khurshid apresentam o *TestEra*, um framework para teste automatizado de programas Java. O framework utiliza a linguagem de especificação Alloy para descrever o domínio de entrada de um programa e para definir critérios de corretude para os casos de teste.

Para gerar dados de teste, o *TestEra* requer que o domínio de entrada seja mapeado através de invariantes descritos em Alloy. A partir destes invariantes, o framework utiliza um solucionador de restrições – o *Alloy Analyzer* [JACKSON *et al.*, 2000] – para gerar um conjunto de instâncias de dados de teste.

Como as instâncias geradas pelo *Alloy Analyzer* são de tipos abstratos, é necessário criar manualmente uma função que traduza estes dados para tipos concretos. Após obter os dados concretos, estes são executados na implementação do programa. As saídas geradas pelo programa são então traduzidas para o nível abstrato, também utilizando uma função de tradução.

Finalmente, utilizando critérios de corretude descritos em Alloy, o *TestEra* verifica se a relação entre as entradas e saídas do programa são válidas. Nesta abordagem, um critério de corretude é uma função que verifica se uma determinada característica é válida. Por exemplo, se o programa que estamos testando tem como objetivo ordenar uma lista e inteiros, deve existir uma função que verifique se uma lista está realmente ordenada. Em casos em que o critério de corretude é desobedecido, o framework apresenta um contra-exemplo que viola o critério.

3.8 OCL

Artigo	Tipo	Nível	Cobertura	Ling. de Impl.
[MENDES *et al.*, 2010]	Estrutural	Sistema	*Ad hoc.* Faz apenas produto cartesiano entre os dados de teste	Não são gerados testes executáveis

Tabela 3.7: Trabalhos relacionados que utilizam OCL

Em [MENDES *et al.*, 2010] é apresentado o *TESTIMONIUM*, um método para geração de casos de teste a partir de regras OCL. O *TESTIMONIUM* é uma extensão de outro método, o *ANIMARE* [SILVEIRA, 2009], que é utilizado para validar os processos de negócio de um sistema. A partir de um diagrama de classes [FOWLER, 2003] representando o modelo de dados, de um diagrama de atividades [FOWLER, 2003] descrevendo os processos do sistema e de regras OCL que definem as regras de negócio, o *ANIMARE* cria um grafo orientado que descreve todos os cenários presentes no sistema. Executando os cenários do grafo, o desenvolvedor pode encontrar inconsistências no modelo e assim corrigi-lo.

O *TESTIMONIUM* é utilizado para auxiliar na seleção de cenários de teste para serem executados. Utilizando critérios de particionamento em classes de equivalência e análise de valor limite o método define os cenários a serem testados. Os valores para os dados de teste são valores fixos, definidos na própria ferramenta, e foram escolhidos de acordo com os tipos para os parâmetros concretos.

Após gerar os dados de teste, a ferramenta faz combinações entre os valores dos parâmetros, resultando em um arquivo contendo um conjunto de cenários de teste. A seleção dos cenários de teste que serão utilizados fica a cargo do engenheiro de testes.

3.9 Considerações Finais

Olhando para as tabelas que classificam os artigos e para os gráficos apresentados inicialmente, fica fácil observar que a grande parte dos trabalhos se concentra no nível de módulo, ou seja, testes focados no comportamento de componentes que agregam várias funções (como classes por exemplo) e na interação entre estas funções. Existem poucos trabalhos cujo foco é o nível de unidade. Se restringirmos as comparações ao método B, não temos nenhum artigo que foque neste nível. No aspecto nível de teste, nosso trabalho apresenta um diferencial pois seria o único a desenvolver testes de unidade a partir de

especificações B.

No quesito critério de cobertura, existem alguns trabalhos que utilizam grafos como técnica de cobertura, mas a grande maioria usa particionamento do espaço de entrada como critério para seus testes. Quando utilizamos especificações formais, podemos facilmente particionar o espaço de entrada de uma operação com base em suas pré-condições e invariantes. As pré-condições e invariantes são escritas na forma de cláusulas lógicas, que podem ser animadas em solucionadores de restrições, para assim obter dados de entradas para os casos testes.

Uma técnica bastante comentada e utilizada por alguns artigos, foi a divisão dos domínios de teste através do uso da forma normal disjuntiva. Nesta esta técnica, cada disjunção da FND representa um domínio que deve ser testado. Após a separação das disjunções, cada uma delas pode ser avaliada por um solucionador de restrições para gerar valores que respeitem suas restrições.

Entre outras técnicas para seleção de dados de teste, as mais comuns são o particionamento em classes de equivalência e a análise de valor limite. Seria interessante avaliarmos outros critérios de cobertura e técnicas de teste.

Trabalhos que geram testes para o nível de unidade a partir de outras linguagens como [SINGH *et al.*, 1997] [HUAIKOU; LING, 2000] e [BURTON; YORK, 2000] não utilizam critérios de teste bem definidos. Existem ainda casos como o trabalho de [CHEON; LEAVENS, 2002], em que dados teste devem ser estabelecidos totalmente a critério do engenheiro de testes. Neste aspecto nosso trabalho se destaca por utilizar conceitos e classificações para seleção e combinação de dados de teste bem estabelecidas pela comunidade de testes.

Outro quesito avaliado foi o suporte ferramental para os métodos propostos. Poucos trabalhos implementaram ferramentas que automatizassem seus métodos. Alguns chegaram a utilizar ferramentas com outros fins – como animadores e solucionadores de restrições – para auxiliar seus métodos como [BURTON; YORK, 2000], outros desenvolveram apenas protótipos que não automatizavam o método por completo. Nos casos em que houve implementação de ferramentas, muitas delas foram descontinuadas ou não estão disponíveis para o público como [AMBERT *et al.*, 2002] e [HUAIKOU; LING, 2000].

Suporte ferramental é uma grande deficiência nos trabalhos relacionados, e como pudemos comprovar no estudo de caso apresentado em [MATOS *et al.*, 2010] é algo essencial para a adoção de uma abordagem de geração de testes na indústria. Nosso trabalho supre

esta necessidade provendo uma ferramenta para geração automática de especificações de teste que podem ser facilmente traduzidas em testes executáveis.

4 Um método para geração de casos de teste de unidade a partir de especificações B

4.1 Considerações Iniciais

Neste capítulo apresentamos o método de geração de casos de teste inicialmente proposto em [SOUZA, 2009]. Iniciamos com uma discussão sobre os critérios para geração de testes utilizados pelo método. Em seguida apresentamos todas as etapas de seu processo enquanto são aplicadas a um exemplo ilustrativo. Durante nosso trabalho aperfeiçoamos este método, adequando-o para que classificações e conceitos de testes de software empregados na abordagem ficassem mais claros. Detalhes sobre as melhorias realizadas no método são apresentados no Capítulo 5.

4.2 O Método para a Geração de Casos de Teste

O trabalhado desenvolvido por Souza apresentou uma técnica de testes baseados em modelos que utiliza como modelo fonte especificações do Método B. A partir de uma máquina B – que deve estar na Forma Normal Conjuntiva (FNC), para facilitar a aplicação do método – são geradas máquinas de teste auxiliares que são posteriormente animadas, utilizando um solucionador de restrições, para gerar dados de teste abstratos. Estes dados são utilizados pelo desenvolvedor para implementar casos de teste concretos em uma linguagem de programação.

O método suporta a geração tanto de casos de teste positivos (que respeitam as restrições da especificação) quanto casos de teste negativos (que desrespeitam as restrições da especificação).

4.2.1 Critérios de Cobertura

O critério de cobertura utilizado pela abordagem é baseado no critério de particionamento do espaço de entrada, apresentado na seção 2.3.4 do Capítulo 2.

A abordagem possui três níveis de cobertura, que diferem de acordo com a forma como os dados de teste são selecionados. De acordo com o nível escolhido, o processo pode selecionar dados de teste através de particionamento em classes de equivalência para casos de especificações que não contenham variáveis do tipo intervalo ou, para casos que contenham intervalos, pode utilizar análise de valor limite para obter dados mais significativos.

Os níveis de cobertura da abordagem são:

- *Nível 1:* utiliza apenas classes de equivalência e gera os testes mínimos para uma operação. Para cada partição é retirado um valor para ser utilizado no caso de teste.
- *Nível 2:* neste nível, dados de teste para variáveis do tipo intervalo são obtidos através de análise de valor limite e para os demais tipos de variáveis são utilizadas classes de equivalência. A combinação dos valores selecionados é realizada através do critério de combinação *Pairwise*. Este nível é recomendado para máquinas que possuam variáveis do tipo intervalo e pode não ser ideal para operações com três ou mais variáveis, pois não gera todas as combinações possíveis para elas.
- *Nível 3:* este nível utiliza os mesmos critérios do nível 2 para geração de dados de teste mas utiliza o critério de "todas as combinações" (*All-combinations*) para a combinação destes dados. Logo, todas as possíveis combinações entre valores para os parâmetros de entrada são cobertos pelo conjunto de testes gerado por este nível. Este nível deve ser utilizado com cautela, pois o critério *All-combinations* pode resultar em uma explosão combinatorial.

4.2.2 Processo de Geração de Casos de Teste

As etapas do processo de geração de casos de teste a partir de máquinas B são apresentadas na Figura 4.1. O processo deve ser executado para cada operação que se pretende testar. Ele inicia com a busca por variáveis e cláusulas do invariante e pré-condições que definam o comportamento da operação. Estas informações são utilizadas para determinar as variáveis de entrada da operação.

Após determinadas as variáveis de entrada, é necessário observar suas restrições, que são definidas pelas cláusulas do invariante e pré-condições. As restrições observadas são utilizadas para particionar os valores das variáveis de entrada em classes de equivalência. Depois do particionamento, são criadas máquinas B auxiliares, que servirão de base para a geração de dados de teste. Para isto, é necessário animar estas máquinas auxiliares em um solucionador de restrições, que nos retornará dados de testes de acordo com as restrições estabelecidas. Com os dados de teste em mãos, podemos implementar os casos de teste concretos na linguagem de programação que bem entendermos.

O método parte do pressuposto que a máquina utilizada para gerar testes foi devidamente verificada e que não possui erros de especificação.

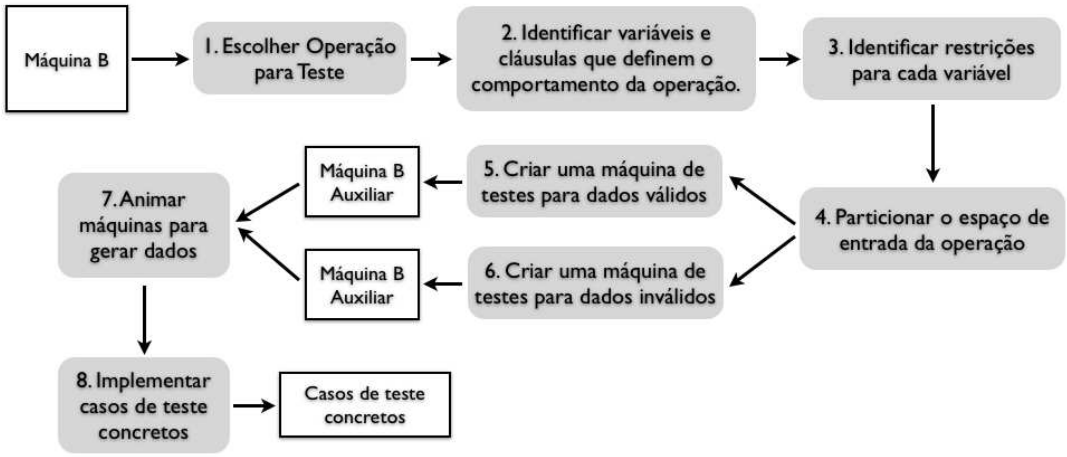

Figura 4.1: Visão geral do processo de geração de casos de teste proposto inicialmente. Os balões em tons de cinza representam as etapas do processo enquanto os quadros brancos representam artefatos produzidos ou utilizados durante sua execução.

Cada uma destas etapas é apresentada em detalhes nas próximas subseções. Durante a apresentação utilizamos a máquina *Transport* (Listagem 4.1), introduzida no Capítulo 2 e replicada aqui por conveniência, para ilustrar cada etapa do processo.

Para explicar cada etapa em detalhes, utilizaremos algoritmos em pseudo-código para descrevê-las. Em alguns pontos, para facilitar a leitura do algoritmo, utilizaremos funções pré-definidas cujas funcionalidades são descritas a seguir:

- *obter_variaveis(predicado)*: recebe um predicado como parâmetro e retorna um conjunto formado pelas variáveis (parâmetros da operação e variáveis de estado) presentes neste predicado;

Listagem 4.1: Especificação da máquina Transport

```
1  MACHINE Transport
2
3  SETS
4    CARD_TYPES = {entire_card, student_card, gratuitous_card}
5
6  VARIABLES
7    balance, card_type
8
9  INVARIANT
10   balance : NAT &
11   card_type : CARD_TYPES &
12   (not(card_type = gratuitous_card) or balance = 0)
13
14 INITIALISATION
15   balance := 0 ||
16   card_type :: CARD_TYPES
17
18 OPERATIONS
19   addCredit (cr) =
20     PRE
21       cr : NAT1 &
22       card_type /= gratuitous_card &
23       balance + cr <= MAXINT
24     THEN
25       balance := balance + cr
26     END;
27
28   bb <-- queryBalance = BEGIN bb := balance END
29 END
```

- *obter_clausulas(predicado)*: recebe um predicado como parâmetro e retorna um conjunto formado pelas cláusulas deste predicado;
- *obter_clausulas_do_inv_com(variaveis)*: recebe como parâmetro um conjunto de variáveis, busca por cláusulas do invariante que contenham estas variáveis e retorna um conjunto formado por estas cláusulas;
- *clausula_contem(clausula, variavel)*: recebe como entrada uma cláusula e uma variável, e verifica se a cláusula contém a variável. Em caso afirmativo a função retorna o valor verdade, caso contrário retorna o valor falsidade;
- *tipo(clausula)*: recebe como entrada uma cláusula e verifica se a mesma é de tipagem. Caso seja, retorna o tipo definido pela cláusula. O tipo definido por uma cláusula é identificado da seguinte maneira: se a cláusula é do tipo *"var : NAT"* ou *"NAT1"* ou *"Z"* ou *"x..y"*, então seu tipo é "intervalo"; se ela é do tipo *"var : EXPRESSAO"* (onde EXPRESSAO é uma *string* em letras maiúsculas), então seu tipo é "pertence

a um conjunto"; se ela é do tipo *"var <: EXPRESSAO"* ou *"var <<: EXPRESSAO"* então seu tipo é "conjunto"; se ela é do tipo *"var = EXPRESSAO"* então seu tipo é "booleano";

- *classificacao(variavel)*: recebe uma variável e retorna seu tipo;
- *clausula_tipagem(variavel)*: recebe uma variável e retorna sua cláusula de tipagem;
- *gerar_todas_combinacoes(formulas)*: recebe como entrada um conjunto, onde cada elemento é um par contendo uma variável e uma lista de cláusulas que geram dados para aquela variável. Retorna um conjunto onde cada elemento é uma combinação entre as cláusulas de cada variável;
- *gerar_combinacoes_part_equiv(formulas)*: recebe como entrada um conjunto de fórmulas (igual à função anterior), gera combinações entre elas utilizando um critério de combinação de dados de teste e retorna um conjunto de combinações de fórmulas, onde cada elemento do conjunto é uma combinação entre as fórmulas recebidas como entrada segundo o critério de combinação utilizado;

1. Definir operação a ser testada

Esta etapa é autoexplicativa. Devemos selecionar a operação para a qual queremos gerar casos de teste. O processo é realizado individualmente para cada operação.

2. Identificar variáveis e cláusulas que determinam o comportamento da operação:

Esta etapa tem como objetivo identificar variáveis e cláusulas relacionadas que ditam o comportamento da operação sob teste. Estas informações serão obtidas a partir das variáveis e do invariante da máquina e da pré-condição da operação que estamos testando.

O processo para extrair estas informações é descrito no algoritmo da Listagem 4.2 e explicado a seguir.

O algoritmo recebe como entradas o invariante da máquina e a pré-condição da operação sob teste. Ele inicia identificando as variáveis e cláusulas da pré-condição como relevantes para o teste da operação (linhas 3-4). Em seguida ele busca no invariante por cláusulas que contenham as variáveis selecionadas na pré-condição. Como elas também restringem estas variáveis, elas também devem ser identificadas como relevantes (linhas 6-7). Existem casos em que variáveis se relacionam através de uma cláusula. Por isso,

Listagem 4.2: Pseudocódigo para extrair variáveis e cláusulas relevantes

```
1  identificar_variaveis_e_clausulas_relevantes(invariante, precondicao) =
2
3  variaveis_relevantes = obter_variaveis(precondicao)
4  clausulas_relevantes = obter_clausulas(precondicao)
5
6  clausulas_invariante_relevantes =
7    obter_clausulas_do_inv_com(variaveis_relevantes)
8
9  variaveis_pendentes =
10   obter_variaveis(clausulas_invariante_relevantes) -
       variaveis_relevantes
11
12 enquanto variaveis_pendentes != ∅:
13   variaveis_relevantes = variaveis_relevantes ∪ variaveis_pendentes
14
15   clausulas_invariante_relevantes =
16     clausulas_invariante_relevantes ∪
17     obter_clausulas_do_inv_com(variaveis_relevantes)
18
19   variaveis_pendentes =
20     obter_variaveis(clausulas_invariante_relevantes) -
         variaveis_relevantes
21
22 clausulas_relevantes = clausulas_relevantes ∪
     clausulas_do_invariante_relevantes
23
24 retorna variaveis_relevantes, clausulas_relevantes
```

precisamos buscar também por cláusulas que contenham estas variáveis relacionadas, pois elas também influenciam no comportamento da operação e são consideradas relevantes (linhas 12-20). O algoritmo encerra retornando os conjuntos de *variaveis_relevantes* e *clausulas_relevantes*.

Se aplicarmos este pseudocódigo na operação *addCredit* da máquina *Transport* teremos as seguintes entradas e saídas:

Entradas:

- $invariante = \{balance : NAT,\ card\_type : CARD\_TYPES,\ (not(card\_type = gratuitous\_card)\ or\ balance = 0)\}$
- $precondicao = \{cr : NAT1,\ card\_type /= gratuitious\_card,\ balance + cr <= MAXINT\}$

Saídas:

- $variaveis\_relevantes = \{cr,\ card\_type,\ balance\}$

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

3. Identificar restrições sobre cada variável

Esta etapa tem o objetivo de identificar as restrições que se aplicam às variáveis do espaço de entrada. Em uma máquina B, estas restrições são encontradas em cláusulas do invariante e das pré-condições de uma operação.

Durante esta etapa os elementos de *clausulas_relevantes* serão classificados como *cláusulas de tipagem*, que são as cláusulas que contém informações sobre o tipo de uma variável, ou como *cláusulas de não tipagem*, que são as cláusulas que definem outro tipo de restrição.

O algoritmo responsável por esta etapa (Listagem 4.3) receberá como entrada as saídas da etapa anterior, ou seja, *variaveis_relevantes* e *clausulas_relevantes* e retornará como saída *clausulas_tipagem*, *clausulas_nao_tipagem* e o conjunto auxilar *classificacoes*. O conjunto *classificacoes* é uma tripla do tipo *(variável que esta sendo tipada, cláusula de tipagem, tipo)* onde *tipo* pode assumir os valores "intervalo", "conjunto", "pertence a um conjunto" e "booleano".

Listagem 4.3: Pseudocódigo para identificar restrições sobre cada variável

```
1  identificar_restricoes(variaveis_relevantes, clausulas_relevantes)
2
3  classificacoes = {}
4  clausulas_tipagem = {}
5  clausulas_nao_tipagem = clausulas_relevantes
6
7  para cada variavel em variaveis_relevantes:
8    para cada clausula em clausulas_relevantes:
9      se clausula_contem_variavel(clausula, variavel) então:
10       classificacoes = classificacoes ∪
11                    {(variavel, clausula, tipo(clausula))}
12       clasulas_nao_tipagem = clausulas_nao_tipagem - clausula
13
14 clausulas_tipagem = clausulas_relevantes - clausulas_nao_tipagem
15
16 retorna classificacoes, clausulas_tipagem, clasulas_nao_tipagem
```

O algoritmo começa inicializando o conjunto *classificacoes* e *clausulas_tipagem* como vazios, e o conjunto *clausulas_nao_tipagem* com todos os elementos de *clausulas_relevantes* (linhas 3-5). Em seguida ele busca nas *clausulas_relevantes* por cláusulas

de tipagem (linhas 7-12), e quando as encontra, adiciona uma tripla no conjunto *classificacoes* (linhas 10-11) e remove a cláusula encontrada do conjunto *clausulas_nao_tipagem* (linha 12). No final de sua execução, o conjunto *clausulas_tipagem* recebe a subtração entre *clausulas_relevantes* e *clausulas_nao_tipagem*.

Em nosso exemplo, após a execução deste algoritmo teríamos as seguintes saídas:

- *classificacoes* = {*(cr, cr : NAT1, "intervalo")*, *(card_type, card_type : CARD_TYPES, "pertence a um conjunto")*, *(balance, balance : NAT, "intervalo")*}
- *clausulas_tipagem* = {*cr : NAT1*, *balance : NAT*, *card_type : CARD_TYPES*}
- *clausulas_nao_tipagem* = {*card_type /= gratuitous_card*, *balance + Cr <= MAXINT*, *(not(card_type = gratuitous_card) or balance = 0)*}

4. Particionar restrições de cada variável em classes de equivalência válidas

Nesta etapa utilizaremos as restrições encontradas na etapa anterior para particionar os valores das variáveis de entrada e definir classes de equivalência para a operação. As cláusulas encontradas anteriormente serão combinadas utilizando um critério de combinação de dados de teste para criar fórmulas que representem um subconjunto do espaço de entrada. Estas fórmulas, quando carregadas em um solucionador de restrições, serão capazes de gerar dados de entrada para nossos casos de teste.

Durante esta etapa será criado o conjunto auxiliar *formulas*, onde cada elemento é um par contendo uma variável e as fórmulas que geram dados válidos para ela de acordo com suas restrições. As fórmulas de cada variável serão combinadas em *combinacoes_de_formulas* e para cada combinação realizada criaremos uma operação de teste (explicadas em detalhes mais adiante), que nesta abordagem, representa um caso de teste.

O algoritmo para esta etapa é apresentado na Listagem 4.4 (note que a geração das fórmulas é diferente para cada nível de cobertura).

O algoritmo começa inicializando a conjunto fórmulas como vazio (linha 3). Em seguida, de acordo com o tipo de cada variável, tuplas contendo a variável e a fórmula responsável por gerar dados para ela são criadas (linhas 5-17). As fórmulas criadas variam de acordo com o nível de teste escolhido. Para o nível 1, apenas cláusulas de tipagem são utilizadas nas fórmulas (ex: linha 8). Para os níveis 2 e 3, que utilizam análise de valor limite, é necessário criar fórmulas que representem o intervalo da variável (ex: linhas

Listagem 4.4: Pseudocódigo para particionar classes de equivalência válidas

```
1  particionar_classes_de_equivalencia_validas(variaveis_relevantes,
       classificacoes)
2
3  formulas = {}
4
5  para cada variavel em variaveis_relevantes:
6    se classificacao(variavel) = "intervalo" então:
7      (nivel 1)
8      formulas = formulas ∪ {(variavel, clausula_tipagem(variavel))}
9      (nivel 2 e 3)
10     formulas =
11       formulas ∪
12         {(variavel, clausula_tipagem(variavel),
13          (!b.((b : intervalo da variavel & b /= classificacoes(n, 1))
                => classificacoes(n, 1) < b))),
14          (!b.((b : intervalo da variavel & b /= classificaoces(n,1)) =>
                classificaoces(n, 1) > b)))}
15   se não então:
16     (nível 1, 2 e 3)
17     formulas = formulas ∪ {(e, clausula_tipagem(variavel))}
18
19 (nível 1 e 3)
20   combinacoes_de_formulas = gerar_todas_combinacoes(formulas)
21
22 (nível 2)
23   combinacoes_de_formulas = gerar_combinacoes_part_equiv(formulas)
```

12-14). Para finalizar, as fórmulas são combinadas e o conjunto combinações de fórmulas é populado (linhas 19-23).

Ao final da execução do algoritmo teremos as seguintes combinações de fórmulas apresentadas nas tabelas 4.1, 4.2 e 4.3. Cada uma destas combinações representa um caso de teste:

Cb	cr	card_type	balance
1	cr : NAT1	card_type : CARD_TYPES	balance : NAT

Tabela 4.1: Combinações para o Nível 1

Os resultados para o nível 2 e 3 foram iguais pois *card_type* possui apenas uma fórmula, logo as combinações foram feitas somente entre *cr* e *balance*.

5. Criar operações de testes para dados válidos

Agora que o particionamento do espaço de entrada da operação foi feito utilizando combinações de fórmulas, devemos utilizar estas combinações para gerar operações de

Cb	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))

Tabela 4.2: Combinações para o Nível 2

Cb	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))

Tabela 4.3: Combinações para o Nível 3

teste. Criaremos uma máquina B auxiliar e cada uma de suas operações representará um caso de teste para a operação sob teste. Desta forma, podemos animar esta máquina auxiliar em um solucionador de restrições para gerar os dados de teste para um respectivo caso de teste.

Para cada combinação de fórmulas em *combinacoes_de_formulas* deverá ser adicionada uma operação em nossa máquina auxiliar de teste. Uma operação será construída através dos seguintes passos:

1. o nome da operação de teste será igual ao nome da operação sob teste, adicionando no final do nome a palavra "Teste" e um número identificador sequencial. (Ex: *addCreditTeste1()*);

2. cada elemento de *variaveis_relevantes* será um parâmetro da operação;

3. cada elemento de *variaveis_relevantes* também será uma variável de retorno da operação;

4. cada elemento de *combinacoes_de_formulas* deve ser posto na pré-condição da operação juntamente com os elementos de *clausulas_nao_tipagem*;

5. no corpo da operação os parâmetros de entrada devem ser passados aos parâmetros de saída.

Se criarmos uma operação de teste para a terceira combinação da Tabela 4.3 teremos a especificação apresentada na Listagem 4.5.

Listagem 4.5: Exemplo de operação de teste para dados válidos

```
1  cr_saida, card_type_saida, balance_saida <-- addCreditTest3(cr,
       card_type, balance) =
2  PRE
3    cr : NAT1 &
4    card_type : CARD_TYPES &
5    balance : NAT &
6    (!b.((b : NAT1 & balance + b <= MAXINT & b/= cr) => cr < b)) &
7    (!b.((b : NAT & b + cr <= MAXINT & b/= balance & (not(card_type =
         gratuitous_card) or b = 0)) => balance > b)) &
8    balance + cr <= MAXINT &
9    card_type /= gratuitous_card &
10   (not(card_type = gratuitous_card) or balance = 0)
11 THEN
12   cr_saida := cr ||
13   card_type_saida := card_type ||
14   balance_saida := balance
15 END
```

6. Particionar retrições de cada variável em classes de equivalência inválidas

De forma semelhante ao passo 3, iremos criar combinações de fórmulas para as variáveis, mas desta vez estas fórmulas deverão gerar valores para casos de teste negativos. O resultado desta etapa é o conjunto *combinacoes_negativas*.

Para popular este conjunto deveremos combinar os elementos de *clausulas_nao_tipagem* entre si, negando um elemento de cada combinação de cláusulas. A negação de um dos elementos da combinação de fórmulas permite que esta combinação seja capaz de produzir dados de entrada em desacordo com as restrições originais. Os dados gerados podem então ser utilizados na criação de casos de teste negativos.

A estrutura da operação de teste também é semelhante a operação de teste para dados válidos, a única mudança será em sua pré-condição que será formada pelos elementos de combinações negativas unidos aos elementos de cláusulas de tipagem.

A Tabela 4.4 mostra exemplos de combinações negativas e na Listagem 4.6 temos uma operação para gerar dados para testes negativos.

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

Tabela 4.4: Combinações Negativas

Listagem 4.6: Exemplo de operação de teste para dados inválidos

```
1  cr_saida, balance_saida, card_types_saida <-- addCreditTeste1 (cr,
       balance, card_types) =
2  PRE
3    cr : NAT1 &
4    balance : NAT &
5    card_type : CARD_TYPES &
6    not(card_type /= gratuitous_card) &
7    balance + Cr <= MAXINT &
8    (not(card_type = gratuitous_card) or balance = 0)
9  THEN
10   cr_saida := cr ||
11   card_type_saida := card_type ||
12   balance_saida := balance
13 END
```

7. Selecionar dados abstratos

Após criadas as máquinas de teste auxiliares para dados válidos e inválidos, elas deverão ser animadas em um solucionador de restrições. Nesta abordagem a ferramenta *ProB* foi utilizada como solucionador de restrições.

Ao abrirmos a máquina de teste no *ProB* ele deverá mostrar possíveis combinações de valores para os parâmetros de cada operação de teste. Este valores estão de acordo com as restrições impostas pelas pré-condições da operação de teste. Se quisermos dados de teste correspondentes à operação *addCreditTest3* (Listagem 4.5), temos os seguintes resultados:

Combinação 1: $cr = 1$, $balance = 126$, $card\_type$ = entire_card

Combinação 2: $cr = 1$, $balance = 126$, $card\_type$ = student_card

Note que para algumas operações mais de uma combinação de valores é aceita pela pré-condição, isso acontece porque podem existir várias combinações de valores que respeitam a pré-condição da operação de teste. Pela definição de classe de equivalência só precisamos selecionar uma dessas combinações pois, já que pertencem a um mesmo subdomínio do espaço de entrada, teriam o mesmo efeito para o caso de teste.

8. Implementação e execução dos testes concretos

No trabalho feito por Souza, os testes concretos foram implementados manualmente em Java utilizando o framework *JUnit*. Na Listagem 4.7 temos um exemplo de caso de teste que foi gerado utilizando a combinação 1 da etapa anterior.

Listagem 4.7: Exemplo de caso de teste concreto

```
1 public void testAddCredit3() {
2   t.setBalance((byte)126);
3   t.setCardType((byte)Constants.ENTIRE_CARD);
4   t.addCredit((byte)1);
5   assertEquals(t.getBalance(), 127);
6 }
```

4.3 Considerações Finais

Neste capítulo apresentamos o método de geração de testes proposto por [SOUZA, 2009], utilizando um exemplo para ilustrar cada uma de suas etapas. No próximo capítulo apresentamos a evolução deste método realizada durante nosso trabalho.

5 Contribuição Teórica: Evolução do Método

5.1 Considerações Iniciais

No capítulo anterior apresentamos o método de geração de casos de teste que foi proposto em [SOUZA, 2009]. Durante nosso trabalho realizamos algumas melhorias e aperfeiçoamos alguns aspectos deste método.

A Figura 5.1 mostra as modificações realizadas no método revisado com relação ao método apresentado na Figura 4.1 do capítulo anterior.

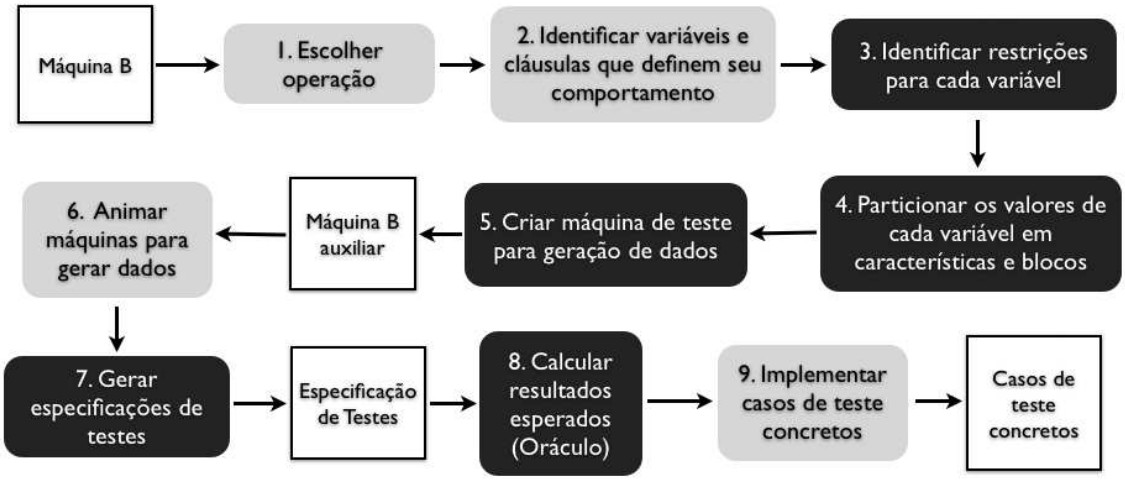

Figura 5.1: Visão geral da abordagem de geração de testes atualizada. Os balões representam etapas da abordagem e os quadros brancos representam artefatos produzidos durante o processo. Os balões escuros são etapas que foram modificadas ou adicionadas durante nosso trabalho.

Os balões escuros representam as etapas do método que sofreram modificações ou foram adicionadas. De forma resumida, as seguintes modificações foram realizadas: a) os passos 3 e 4 foram modificados para trabalhar com os conceitos de características e blocos que foram apresentados na seção 2.3.4 do Capítulo 2, b) os antigos passos 5 e 6 foram unidos em apenas um passo (agora passo 5) e a geração de dados de teste inválidos

deixou de ser uma etapa à parte, c) após a animação da máquina de testes auxiliar é gerado uma especificação de testes que servirá de base para a implementação dos casos de teste concretos, d) adicionamos o passo 8 em que os valores para o oráculo de testes são calculados através de um processo manual. Cada uma destas melhorias será discutida em detalhes nas próximas seções deste capítulo.

5.2 Revisão e formalização dos conceitos de teste utilizados

Os conceitos de teste de software empregados no método original foram apresentados de maneira superficial e necessitavam de explicações mais detalhadas sobre a maneira como eles eram utilizados. Esta deficiência também é encontrada em muitos dos trabalhos relacionados apresentados no Capítulo 3. Nesta seção explicamos as alterações realizadas no método para que suas estratégias e critérios de teste fossem devidamente formalizados.

Nosso método de geração de testes utiliza o critério de *particionamento do espaço de entrada* (apresentado na seção 2.3.4 do Capítulo 2) para elaborar casos de teste. Segundo este critério, o primeiro passo para selecionar valores para serem usados como dados de teste para um programa é identificar os parâmetros que determinam o seu comportamento. Estes parâmetros podem ser parâmetros de um método, variáveis de estado de uma máquina abstrata, atributos de uma classe, ou entradas fornecidas pelo usuário, dependendo do artefato sob teste. Juntos, os parâmetros identificados compõem o conjunto das *variáveis do espaço de entrada* do programa.

Como o método proposto visa gerar casos de teste de unidade para cada operação de uma especificação B, precisamos identificar para cada uma destas operações os parâmetros que determinam o seu comportamento.

Em uma máquina B, o espaço de entrada de uma operação é determinado por seus parâmetros formais e por variáveis de estado da máquina. Por questão de otimização, em nosso método utilizamos os parâmetros formais e apenas as variáveis de estado que influenciam no comportamento da operação sob teste.

Os parâmetros formais são os parâmetros colocados entre parênteses após o nome da operação. Já as variáveis de estado relacionadas são identificadas através das cláusulas da pré-condição da operação. Todas as variáveis de estado citadas na pré-condição são consideradas variáveis relacionadas. Além da pré-condição da operação, deve-se analisar as cláusulas do invariante da máquina. Variáveis que compartilhem cláusulas do

invariante com variáveis relacionadas já detectadas também serão incluídas no conjunto de variáveis relacionadas e consequentemente fazem parte do espaço de entrada da operação. Uma apresentação mais formal da definição de variáveis do espaço de entrada para uma operação em B seria:

Variáveis do espaço de entrada: *o conjunto das variáveis do espaço de entrada de uma operação B é formado pelos parâmetros formais da operação e por variáveis de estado da máquina que influenciam seu comportamento.*

Considere como exemplo a especificação da máquina *Player* [SCHNEIDER, 2001] apresentada na Listagem 5.1. Esta máquina representa um time de futebol. Ela possui uma variável de estado *team* que representa um conjunto de jogadores (*PLAYER*) escalados no time principal que possui exatamente onze jogadores. Ela possui duas operações: *substitute* e *query*. A operação *substitute* é responsável por substituir um jogador *pp* do time principal por um jogador *rr* do time reserva. Já a operação *query* verifica se um jogador *pp* está escalado no time principal.

Para ilustrar o processo de identificação das variáveis do espaço de entrada, vamos utilizar como exemplo a operação *substitute*. Começamos selecionando os parâmetros formais, que neste caso são *pp* e *rr*. Em seguida verificamos se nas cláusulas da pré-condição da operação existe alguma referência a variáveis de estado. Neste exemplo, a variável *team* é citada na pré-condição, logo ela também faz parte do espaço de entrada. Finalmente verificamos se alguma das variáveis já identificadas se relacionam com outras variáveis através do invariante. Como não existem outras variáveis relacionadas, concluímos que o espaço de entrada da operação *substitute* é formado por: *pp*, *rr* e *team*.

Após identificar as variáveis do espaço de entrada de uma operação, precisamos identificar o seu *domínio de entrada*, ou seja, os possíveis valores que estas variáveis podem assumir. Em uma máquina B, as restrições quanto aos valores permitidos pelo domínio de entrada de uma operação são estabelecidas pela pré-condição da operação e pelo invariante da máquina. Logo, o domínio de entrada de uma operação é estabelecido da seguinte maneira:

Domínio de entrada de uma operação: *é a conjunção de todas as cláusulas da pré-condição e cláusulas do invariante que referenciem alguma variável do espaço de entrada.*

Listagem 5.1: Especificação da máquina Player

```
1  MACHINE Player
2
3  SETS ANSWER = {in, out};
4       PLAYER
5
6  PROPERTIES
7       card(PLAYER) > 11
8
9  VARIABLES team
10
11 INVARIANT
12      team <: PLAYER &
13      card(team) = 11
14
15 INITIALISATION
16      ANY tt
17      WHERE tt <: PLAYER & card(tt) = 11
18      THEN team := tt
19      END
20
21 OPERATIONS
22      substitute(pp,rr) =
23          PRE pp : team & rr : PLAYER & rr /: team
24          THEN team := (team \/ {rr}) - {pp}
25          END;
26
27      aa <-- query(pp) =
28          PRE pp : PLAYER
29          THEN IF pp : team
30               THEN aa := in
31               ELSE aa := out
32          END
33 END
34 END
```

A fórmula resultante desta conjunção estabelece os valores válidos para as variáveis do espaço de entrada, de forma que a operação se comporte da maneira esperada.

Se considerarmos a operação *substitute* do exemplo anterior, seu domínio de entrada é estabelecido pela seguinte fórmula:

$$pp \in team \wedge rr \in PLAYER \wedge rr \notin team \wedge team \subset PLAYER \wedge card(team) = 11$$

Ou seja, para pertencer ao domínio de entrada da operação, os valores para as variáveis do espaço de entrada da operação *substitute* devem obedecer às seguinte restrições:

- Para ser substituído, o jogador *pp* do time principal deve estar de fato escalado ($pp \in team$);

- O jogador reserva *rr* que entrará em seu lugar deve pertencer ao conjunto geral de jogadores ($rr \in PLAYER$) e não deve estar escalado no time principal no momento ($rr \notin team$);

- O time principal *team* deve ser um subconjunto do conjunto geral de jogadores ($team \subset PLAYER$) e deve possuir exatamente onze jogadores ($card(team) = 11$).

Após estabelecer o domínio de entrada precisamos particioná-lo em blocos para que então possamos selecionar valores de cada bloco para nossos casos de teste. O particionamento é feito considerando *características* individuais de cada variável do espaço de entrada. Para cada característica são criados blocos de valores para testá-la.

Na abordagem original, a maneira como as cláusulas eram utilizadas para realizar o particionamento não era bem definida. As cláusulas eram combinadas para estabelecer um conjunto de fórmulas em que cada fórmula representava um caso de teste. No entanto, o processo para gerar estas combinações não utilizava uma estratégia bem definida.

Em nossa abordagem cada cláusula do domínio de entrada representa uma característica a ser testada. A partir de cada cláusula são gerados blocos de dados de teste. Para cada cláusula são gerados dois blocos, um bloco de valores que respeitam a cláusula (ou seja, um bloco de valores válidos) e um bloco de valores que desrespeitam a cláusula (neste caso, um bloco de valores inválidos).

Em casos em que a cláusula representa um intervalo e deseja-se selecionar os dados utilizando apenas o critério de classes de equivalência, são gerados três blocos: um bloco com valores dentro do intervalo, um bloco com valores que antecedem o intervalo e um bloco com valores posteriores ao intervalo. Quando deseja-se utilizar análise de valor limite, são gerados quatro blocos: um bloco que representa a margem esquerda do limite inferior, um bloco que representa a margem direita do limite inferior, um bloco para a margem esquerda do limite superior e um bloco para a margem direita do limite superior. Em nossa abordagem geramos testes apenas para os limites (ou bordas) de um intervalo. Outra estratégia comum para testes baseados em valores limite é a seleção de valores aleatórios próximos às bordas do intervalo.

Para o exemplo da operação *substitute* teremos as características e blocos apresentadas na Tabela 5.1.

Após estabelecermos os blocos para cada característica da operação, é feita a combinação entre blocos segundo algum critério de combinação de dados de teste. Nossa

Características	Blocos	
$pp \in team$	b_1: $pp \in team$	b_2: $not(pp \in team)$
$rr \in PLAYER$	b_1: $rr \in PLAYER$	b_2: $not(rr \in PLAYER)$
$rr \notin team$	b_1: $rr \notin team$	b_2: $not(rr \notin team)$
$team \subset PLAYER$	b_1: $team \subset PLAYER$	b_2: $not(team \subset PLAYER)$
$card(team) = 11$	b_1: $card(team) = 11$	b_2: $not(card(team) = 11)$

Tabela 5.1: Características e respectivos blocos para a operação *substitute*

abordagem utiliza três critérios de combinação: *All-Combinations*, *Each-choice* e *Pairwise*. Cada um destes critérios é apresentado na seção 2.3.4 do Capítulo 2.

Após a combinação dos blocos, teremos um conjunto de combinações em que cada combinação é uma fórmula que representa um sub-domínio do domínio de entrada da operação. A seguir mostramos as tabelas com combinações seguindo os critérios *Each-choice* (Tabela 5.2) e *Pairwise* (Tabela 5.3). Para o algoritmo *Pairwise* utilizamos o *In-parameter-order Pairwise* apresentado em [TAI; LEI, 2002].

N.	Combinação
1	$not(card(team) = 11) \wedge not(pp \in team) \wedge not(rr \notin team) \wedge not(rr \in PLAYER) \wedge not(team \subset PLAYER)$
2	$card(team) = 11 \wedge pp \in team \wedge rr \notin team \wedge rr \in PLAYER \wedge team \subset PLAYER$

Tabela 5.2: Combinação de blocos utilizando o critério *Each-choice*

N.	Combinação
1	$pp \in team \wedge not(rr \in PLAYER) \wedge rr \notin team \wedge team \subset PLAYER \wedge not(card(team) = 11)$
2	$pp \in team \wedge rr \in PLAYER \wedge rr \notin team \wedge team \subset PLAYER \wedge not(card(team) = 11)$
3	$not(pp \in team) \wedge rr \in PLAYER \wedge not(rr \notin team) \wedge team \subset PLAYER \wedge not(card(team) = 11)$
4	$pp \in team \wedge not(rr \in PLAYER) \wedge not(rr \notin team) \wedge team \subset PLAYER \wedge card(team) = 11$
5	$pp \in team \wedge rr \in PLAYER \wedge rr \notin team \wedge not(team \subset PLAYER) \wedge card(team) = 11$
6	$not(pp \in team) \wedge not(rr \in PLAYER) \wedge rr \notin team \wedge not(team \subset PLAYER) \wedge card(team) = 11$
7	$pp \in team \wedge rr \in PLAYER \wedge not(rr \notin team) \wedge not(team \subset PLAYER) \wedge not(card(team) = 11)$

Tabela 5.3: Combinação de blocos utilizando o critério *Pairwise*

Estas fórmulas são animadas em um solucionador de restrições para obter dados de

teste para o sub-domínio que cada uma delas representa. O processo de animação é semelhante a abordagem inicial em que criamos uma máquina de testes auxiliar para gerar os dados de teste.

Na abordagem original eram criadas duas máquinas de teste auxiliares, uma para testes positivos e outra para testes negativos. Em nossa abordagem, durante o particionamento de uma característica, são criados blocos de valores válidos e inválidos para ela. Estes blocos são tratados igualmente durante o processo de combinação para criação de casos de testes, sendo assim necessário apenas uma máquina de testes auxiliar.

5.3 Mudanças no uso de critérios de teste

Na abordagem proposta inicialmente, os critérios de teste utilizados eram divididos de acordo com níveis de teste pré-estabelecidos. O trabalho propunha 3 níveis de teste, cada um destes níveis utilizava critérios de particionamento e combinação diferentes para a geração de casos de teste. O engenheiro de testes deveria escolher um nível de testes de acordo com sua necessidade. Um resumo sobre estes níveis é mostrado na Figura 5.2 e mais detalhes são apresentados na seção 4.2.1 do Capítulo 4.

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

Figura 5.2: Níveis de cobertura propostos por [SOUZA, 2009]

Em nosso trabalho simplificamos a utilização dos critérios de teste. O engenheiro de teste poderá escolher um critério de particionamento do espaço de entrada (Particionamento de Equivalência ou Análise de Valor Limite) e um critério combinatório para combinação dos blocos de dados de teste (*All-Combinations*, *Each-choice* ou *Pairwise*). Desta forma ficam explícitos para o engenheiro de testes quais critérios estão sendo utilizados no momento.

No final, devido a estratégia que utilizamos para fazer o particionamento dos blocos e

em seguida combiná-los, os casos de teste gerados pela nossa abordagem serão diferentes. Na abordagem original, as combinações eram feitas utilizando as próprias características, enquanto que em nossa abordagem estas características são particionadas em blocos e estes blocos são então combinados. Também há diferença na maneira que geramos os casos de testes negativos. Enquanto, este tipo de teste recebia um tratamento especial na abordagem original, em nossa abordagem os blocos de dados negativos recebem o mesmo tratamento que os blocos de dados positivos. Mais detalhes sobre os testes negativos em nossa abordagem são apresentados na próxima seção.

5.4 Integração de testes negativos à geração de blocos

No método proposto por Souza, a geração de casos de teste para dados inválidos era realizada em uma etapa separada dentro do processo. Em nosso método consideramos a geração deste tipo de caso de teste como parte do processo padrão e o integramos ao processo para que funcione de tal forma.

Ao invés de criarmos uma máquina de testes separada para geração de dados de teste inválidos, nós geramos dados inválidos através do uso de blocos que contenham valores que desrespeitam uma determinada característica. Para cada característica a ser testada são gerados blocos de dados válidos e inválidos.

Ainda levando em consideração a operação *substitute* dada como exemplo, uma característica que precisa ser testada a seu respeito é a de que o jogador *rr* que entrará como substituto não deve pertencer o time principal, ou seja, $rr \notin team$. Neste caso além de testar a operação com um bloco de valores válidos, devemos testá-la com um bloco de valores inválidos representado por $rr \in team$.

5.5 Mudanças na utilização das cláusulas de tipagem

De acordo com o método proposto, cláusulas de tipagem são consideradas características especiais. Elas não podem ser negadas como as outras características, pois ao implementarmos casos de teste concretos obteríamos erros de compilação. Isso acontece porque em nosso trabalho consideramos que a linguagem de programação utilizada para a implementação dos casos de teste concretos é fortemente tipada. Logo, para este tipo de linguagem, um caso de teste que requer que uma variável "booleana" passe a ser "não booleana" seria impossível de implementar.

No método proposto por Souza, todas as cláusulas do tipo "pertence à" ($\in$), "subconjunto de" ($\subset$), "subconjunto estrito de" ($\subseteq$) e "igual a" ($=$) eram consideradas cláusulas de tipagem e por isso não poderiam ser negadas ou manipuladas durante a geração de dados de teste. Não eram feitas análises mais detalhadas sobre a semântica das cláusulas deste tipo. Em nosso trabalho analisamos as cláusulas de tipagem de maneira mais profunda.

Além de verificar se a cláusula é de tipagem, verificamos o que a cláusula está especificando. Por exemplo, considere uma cláusula do tipo $xx \in 1..10$. De acordo com o método proposto anteriormente esta cláusula seria considerada de tipagem e não poderia ser negada ou manipulada. Em nosso método analisamos os detalhes desta cláusula. Além de definir o tipo da variável *xx*, que neste caso poderia ser implementada como um inteiro, esta cláusula contém informações importantes sobre uma característica a ser testada. Ela define que *xx* pertence à um intervalo entre os inteiros 1 e 10. É interessante gerar dados que desobedeçam essa regra para que possamos testar o comportamento da operação com dados inválidos.

Após a revisão do método chegamos as seguintes regras para as cláusulas de tipagem:

- xx ($\in$ / $\subset$ / $\subseteq$) INT;
- xx ($\in$ / $\subset$ / $\subseteq$) $BOOL$;
- xx ($\in$ / $\subset$ / $\subseteq$) $STRING$;
- xx ($\in$ / $\subset$ / $\subseteq$) SET, onde SET é um conjunto predefinido;
- xx ($\in$ / $\subset$ / $\subseteq$) $\mathcal{P}(T)$, onde $\mathcal{P}(T)$ é o conjunto das partes de T;
- xx, yy ($\in$ / $\subset$ / $\subseteq$) $T_1 \times T_2$, onde T_1 e T_2 são tipos;
- xx ($\in$ / $\subset$ / $\subseteq$) id, onde id representa um tipo.

5.6 Uso do comportamento da operação no particionamento

Em nosso trabalho passamos a utilizar informações do corpo da operação para gerar particionamentos mais interessantes. É feita uma busca no corpo da operação por comandos condicionais e a partir destes comandos elaboramos partições avaliando as várias possibilidades de interação dentro da operação.

Por exemplo, se uma operação possui em seu corpo um comando do tipo *IF predicado THEN* $inst_1$ *ELSE* $inst_2$, podemos criar partições de modo que tanto o caminho que executa $inst_1$ quanto o caminho que executa $inst_2$ sejam cobertos por nosso conjunto de testes.

A técnica é baseada no trabalho realizado em [SATPATHY *et al.*, 2005]. Neste trabalho os autores comentam a importância de considerar estruturas condicionais durante o particionamento, pois costumam resultar em partições mais interessantes. Além de comandos *IF-ELSE*, a abordagem suporta comandos do tipo: *IF-ELSEIF*, *SELECT-WHEN* e *CASE*. Esta técnica corresponde ao conceito por trás do particionamento de equivalência clássico, que requer que dados de teste exercitem os diversos caminhos de execução de um programa.

Para incluir estruturas condicionais no particionamento, utilizamos o predicado da condição como uma característica a mais a ser testada. Estas características são trabalhadas da mesma maneira que as características extraídas da pré-condição e do invariante. A partir dela, geramos blocos para testar cada caminho de execução do condicional.

Por exemplo, se considerarmos um comando *IF-ELSE* do tipo:

$$IF\ P\ THEN\ inst_1\ ELSE\ inst_2$$

A partir deste comando é extraída a característica P – onde P é uma cláusula lógica – e a partir dela geramos dois blocos: $b_1 = P$ e $b_2 = \neg P$.

De forma semelhante podemos construir blocos a partir de um comando *SELECT-WHEN* como:

$$SELECT\ P_1\ THEN\ inst_1\ WHEN\ P_2\ THEN\ inst_2$$

Neste caso temos duas características: P_1 a partir da qual podemos gerar os blocos $b_1 = P_1$ e $b_2 = \neg P_1$, e P_2 da qual podemos gerar os blocos $b_1 = P_2$ e $b_2 = \neg P_2$.

Finalmente para casos de comandos *CASE* como:

$$CASE\ E\ OF\ EITHER\ X\ THEN\ inst_1\ EITHER\ Y\ THEN\ inst_2$$

Teremos uma característica para cada instrução *EITHER*. Para o exemplo acima temos as características: $E \in X$, que gera os blocos $b_1 = E \in X$ e $b_2 = not(E \in X)$, e $E \in Y$ que gera os blocos $b_1 = E \in Y$ e $b_2 = not(E \in Y)$.

Continuando com o exemplo da especificação da máquina *Player* (Listagem 5.1), considere a operação *query*. Esta operação possui um condicional que verifica se o jogador *pp* passado como parâmetro está escalado no time. Se o jogador estiver escalado a operação retorna *in*, caso contrário, retorna *out*.

Como operação *query* possui um condicional em seu corpo, ao gerarmos o particionamento para ela, devemos considerar o predicado presente em seu condicional como uma característica a ser testada. Neste caso esta característica é $pp \in team$. Utilizando esta característica adicionamos uma partição com dois novos blocos em nosso particionamento $b_1 = pp \in team$ e $b_2 = not(pp \in team)$, que nos ajudarão a testar dois caminhos de execução presentes na operação.

5.7 Suporte a especificações estruturadas

Outro problema da abordagem original e que também pode ser encontrado em muitos dos trabalhos relacionados, diz respeito ao suporte a especificações estruturadas em múltiplos componentes. As abordagens propostas por estes trabalhos exigem que as máquinas utilizadas para a geração de testes sejam especificadas em um único arquivo.

Esta restrição pode dificultar a adoção de tais abordagens em ambientes encontrados no mercado, visto que a estruturação de especificações B em vários componentes é uma prática bastante comum. Tal prática pode trazer muitos benefícios, como por exemplo: encorajar o reuso de componentes, facilitar a verificação de máquinas (componentes menores são mais fáceis de verificar) e reduzir a quantidade de obrigações de prova geradas.

O Método B provê vários mecanismos para estruturar especificações. Estes mecanismos são invocados através das cláusulas *SEES*, *USES*, *INCLUDES* e *EXTENDS* [SCHNEIDER, 2001]. Cada uma destas cláusulas provê uma visão diferente da estrutura da máquina agregada.

Em nossa abordagem levamos em consideração todas estas cláusulas quando geramos casos de teste para uma operação de uma máquina. Durante o processo de geração de testes, buscamos por variáveis de estado e cláusulas do invariante de máquinas agregadas que influenciam no comportamento da operação sob teste. As variáveis de estado encontradas integram o conjunto de variáveis do espaço de entrada enquanto as cláusulas encontradas compõe o domínio de entrada da operação sob teste e definem novas características que precisam ser testadas.

Durante as buscas, criamos um conjunto global de variáveis do espaço de entrada que contém variáveis de todas as máquinas agregadas que influenciam no comportamento da operação sob teste. Após obtermos este conjunto, iniciamos a busca em todas as máquinas agregadas por cláusulas lógicas da especificação que contenham restrições aplicadas a estas variáveis. Todas estas cláusulas são adicionadas a um conjunto global de restrições. Após a criação destes conjuntos o processo segue normalmente com o particionamento das características em blocos e com a combinação dos blocos criados segundo critérios de teste selecionados.

5.8 Criação de oráculos de teste

Apesar de citar a possibilidade de criação de oráculos de teste para o método, no trabalho de Souza não foi apresentado um processo para a geração dos mesmos. Em nosso trabalho introduzimos um processo para geração de oráculos de teste que, apesar de atualmente ser manual, pode ser automatizado futuramente.

O oráculo gerado por este processo tem o objetivo de verificar se, para um determinado conjunto de dados de teste, os resultados obtidos após a execução da operação sob teste estão de acordo com sua especificação.

O calculo do oráculo para casos de teste positivos é simples. Dado um conjunto de dados de teste válidos para uma operação, um solucionador de restrições pode obter os resultados esperados através do calculo da pós-condição utilizando substituições generalizadas.

No entanto, no caso de oráculos para casos de teste negativos, a violação das pré-condições impede que a operação sob teste seja animada com dados inválidos. Isto ocorre porque o comportamento esperado da operação para dados inválidos (que desrespeitam a pré-condição) não é especificado. Neste caso, a metodologia utilizada pelo oráculo para avaliar os resultados obtidos deve ser definida pelo engenheiro de testes, segundo critérios próprios. Existem várias possibilidades para a avaliação de resultados, uma delas seria verificar se a operação executa corretamente, de acordo com comportamento especificado. Caso isto ocorra temos um forte indício de que a implementação não está de acordo com a especificação, pois os dados de teste utilizados violam as restrições impostas e portanto deveriam revelar um comportamento inesperado, que está em desacordo com a especificação.

A seguir o processo de criação dos oráculos é explicado utilizando como exemplo a

máquina *Paperround* [SCHNEIDER, 2001]. A especificação da máquina representa um pequeno programa para controlar entrega de jornais e revistas em uma vizinhança. A máquina possui duas variáveis (linha 3), *papers* e *magazines*, que armazenaram os números das casas na vizinhança que recebem jornais e revistas. O invariante da máquina define que *papers* será um subconjunto de números presentes no intervalo de 1 a 163 (linha 6), também define que *magazines* é um subconjunto de *papers* (linha 7), ou seja, apenas casas que recebem jornais podem receber revistas. A última cláusula do invariante (linha 8) estabelece que o número de casas que podem receber jornais (e consequentemente revistas) deve ser menor ou igual a 60. A máquina possui como principal operação *addPaper*, responsável por adicionar casas que passarão a receber jornais (linhas 15 a 20). Esta operação recebe como parâmetro o número *hh* de uma casa que passará a receber jornais. Como pré-condições ela estabelece que *hh* deve pertencer ao intervalo de 1 a 163 e que a quantidade de casas já registradas deve ser menor do que 60 (linha 17). As demais operações (*set_papers* e *set_magazines*) são apenas *setters* para as variáveis de estado da máquina.

Durante o processo de geração de casos de teste a partir de uma máquina, nossa ferramenta gera uma máquina de testes auxiliar que deverá ser animada em um solucionador de restrições. Após essa animação obtemos dados de entrada para serem utilizados em nossos casos de teste. Considere a máquina auxiliar de testes apresentada na Listagem 5.3, criada durante o processo de geração de casos de teste para a operação *addPaper*.

Ao animarmos estas operações em um solucionador de restrições, obteremos várias combinações de valores para o espaço de entrada que obedecem as restrições das pré-condições. Por exemplo teríamos as seguintes possíveis combinações de valores para o primeiro teste:

$$
\begin{aligned}
&In_1 : hh = 1, papers = \emptyset, magazines = \emptyset \\
&In_2 : hh = 1, papers = \{1\}, magazines = \emptyset \\
&In_3 : hh = 1, papers = \{1\}, magazines = \{1\} \\
&In_4 : hh = 1, papers = \{1,2\}, magazines = \emptyset \\
&In_5 : hh = 1, papers = \{1,2\}, magazines = \{2\}
\end{aligned}
$$

E para o segundo teste os seguintes valores:

$$In_1 : hh = 1, papers = \{1,2,3,\ldots,60,61\}, magazines = \{0\}$$

Listagem 5.2: Especificação da máquina Paperround

```
MACHINE Paperround

VARIABLES papers, magazines

INVARIANT
  papers <: 1..163 &
  magazines <: papers &
  card(papers) <= 60

INITIALISATION
  papers := {} ||
  magazines := {}

OPERATIONS
  addPaper(hh) =
    PRE
      hh : 1..163 & card(papers) < 60
    THEN
      papers := papers \/ {hh}
    END;

  set_papers(pp) =
    PRE
      pp : NAT1
    THEN
       papers := pp
    END;

  set_magazines(mm) =
    PRE
      mm : NAT1
    THEN
      magazines := mm
    END
END
```

$$In_2 : hh = 1, papers = \{1,2,3,\ldots,60,61\}, magazines = \{0,1\}$$
$$In_3 : hh = 1, papers = \{1,2,3,\ldots,60,61\}, magazines = \{0,1,2\}$$
$$In_4 : hh = 1, papers = \{1,2,3,\ldots,60,61\}, magazines = \{0,1,2,3\}$$
$$In_5 : hh = 1, papers = \{1,2,3,\ldots,60,61\}, magazines = \{0,1,3\}$$

No entanto, apenas os dados de entrada não são suficientes para criar um caso de teste. É necessário saber qual resposta é esperada do programa ao executarmos uma operação com estas entradas. É a partir destas respostas que podemos analisar os resultados do teste e estabelecer se ele foi bem sucedido ou não. Este é o trabalho de um oráculo de testes.

Listagem 5.3: Especificação da máquina de testes para Paperround

```
1  MACHINE TestsForOp_addPaper_From_Paperround
2
3  OPERATIONS
4
5  addPaper_test1(data__hh, data__papers, data__magazines) =
6    PRE
7      data__hh : 1..163 &
8      card(data__papers) < 60 &
9      card(data__papers) <= 60 &
10     data__magazines <: data__papers &
11     data__papers <: 1..163
12   THEN
13     skip
14   END;
15
16 addPaper_test2(data__hh, data__papers, data__magazines) =
17   PRE
18     data__hh : 1..163 &
19     not(card(data__papers) < 60) &
20     not(card(data__papers) <= 60) &
21     not(data__magazines <: data__papers) &
22     data__papers <: 1..163
23   THEN
24     skip
25   END
26
27 END
```

Nossos oráculos serão criados a partir das máquinas de teste auxiliares geradas pela ferramenta. Para montarmos o oráculo precisamos fazer algumas alterações nesta máquina de testes auxiliar. Primeiro, precisamos que ela inclua a máquina original (através da cláusula *INCLUDES*) para a qual estamos gerando casos de teste. Desta forma podemos acessar suas operações e suas variáveis de estado a partir das operações de testes.

Além disso, devemos modificar o corpo da operação de testes, alterando o estado da máquina com os valores que queremos testar através das operações *setters*. Para isto a abordagem exige que operações de *set* para as variáveis de estado da máquina sejam criadas durante a especificação.

Logo em seguida fazemos a chamada da operação sob teste, para que seja feita a simulação de sua execução no estado recém estabelecido. Por fim, verificamos o novo estado da máquina acessando suas variáveis de estado. Precisamos também adicionar variáveis de retorno para capturarmos a resposta esperada. No final, para o exemplo da máquina *Paperround*, teremos a máquina apresentada na Listagem 5.4.

Ao animarmos a primeira operação de testes, obteremos o seguinte conjunto de

Listagem 5.4: Especificação da máquina de testes para Paperround

```
1  MACHINE TestsForOp_addPaper_From_Paperround
2
3  INCLUDES Paperround
4
5  OPERATIONS
6
7  r_papers, r_magazines <-- addPaper_test1(data__hh, data__papers,
       data__magazines) =
8    PRE
9      data__hh : 1..163 &
10     card(data__papers) < 60 &
11     card(data__papers) <= 60 &
12     data__magazines <: data__papers &
13     data__papers <: 1..163
14   THEN
15     set_papers(data__papers);
16     set_magazines(data__magazines);
17     addPaper(data__hh);
18     r_papers := papers;
19     r_magazines := magazines
20   END;
21
22 r_papers, r_magazines <-- addPaper_test2(data__hh, data__papers,
       data__magazines) =
23   PRE
24     data__hh : 1..163 &
25     not(card(data__papers) < 60) &
26     not(card(data__papers) <= 60) &
27     not(data__magazines <: data__papers) &
28     data__papers <: 1..163
29   THEN
30     set_papers(data__papers);
31     set_magazines(data__magazines);
32     addPaper(data__hh);
33     r_papers := papers;
34     r_magazines := magazines
35   END
36
37 END
```

relações entre dados de entrada e respostas esperadas do programa:

$In_1 : hh = 1, papers = \emptyset, magazines = \emptyset \qquad Out_1 : papers = \{1\}, magazines = \emptyset$

$In_2 : hh = 1, papers = \{1\}, magazines = \emptyset \qquad Out_2 : papers = \{1\}, magazines = \emptyset$

$In_3 : hh = 1, papers = \{1\}, magazines = \{1\} \qquad Out_3 : papers = \{1\}, magazines = \{1\}$

$In_4 : hh = 1, papers = \{1,2\}, magazines = \emptyset \qquad Out_4 : papers = \{1,2\} magazines = \emptyset$

$In_5 : hh = 1, papers = \{1,2\}, magazines = \{2\} \qquad Out_5 : papers = \{1,2\} magazines = \{2\}$

Já para a segunda operação de testes temos um resultado diferente. Como para este caso de teste definimos que o conjunto *papers* receberia o conjunto $\{1,2,3,\ldots,60,61\}$ e como este é um valor inválido para a variável segundo o invariante da máquina (que estabelece que $(card(papers) < 60)$), não existe valor que satisfaça as restrições correspondentes ao segundo caso de teste, pois há inconsistência entre as cláusulas $card(parpers) < 60$ que veio do invariante e $papers = \{1,2,3,\ldots,60,61\}$ que define uma situação de teste negativo.

Neste caso, como não existe um comportamento esperado para um conjunto de dados de teste negativos, é necessário definir o oráculo de testes manualmente.

Este processo pode ser automatizado se utilizarmos o arquivo gerado durante animação da máquina auxiliar que especifica o oráculo. Já implementamos na ferramenta um processo similar que lê os dados de entrada gerados pela animação de uma máquina de testes auxiliar como a da Listagem 5.3, no entanto, ainda seria necessário fazer algumas adaptações neste processo para recuperar os valores do oráculo.

5.9 Considerações Finais

Neste capítulo apresentamos as melhorias que realizamos na abordagem original. Algumas destas melhorias incluíram a revisão dos conceitos de testes de software utilizados pela abordagem, tornando mais clara sua atuação na abordagem. Restruturamos também a maneira como critérios de cobertura eram empregados, deixando a classificação em níveis para trás e utilizando técnicas de classes de equivalência e análise de valores limite para seleção de blocos de teste e critérios combinatórios para combinação destes blocos. Passamos também a integrar a geração de casos de teste negativos ao processo de geração de testes padrão. Fizemos algumas alterações no que diz respeito ao tratamento de cláusulas de tipagem. Introduzimos na abordagem a utilização do corpo da operação para definir classes de equivalência e passamos a suportar especificações estruturadas em várias máquinas. Por fim, definimos um processo para a criação de oráculos de teste.

6 Contribuição Prática: a ferramenta BETA

6.1 Considerações Iniciais

Neste capítulo apresentamos a ferramenta desenvolvida durante nosso trabalho. Começamos discutindo alguns detalhes técnicos sobre a implementação, como linguagem de programação, frameworks e componentes utilizados. Em seguida, apresentamos a arquitetura da ferramenta e seu funcionamento interno. Por fim, mostramos sua interface gráfica e como ela deve ser utilizada para gerar especificações de teste.

6.2 Detalhes técnicos

Desenvolvemos a *BETA* (*B Based Testing Approach*) utilizando a linguagem de programação Java[1]. A linguagem foi escolhida devido à familiaridade do autor com a mesma e por facilitar a integração com outros componentes utilizados pela ferramenta.

Foram utilizados alguns componentes do *ProB*, como o seu *parser* (o *BParser*) e o *probcli*, sua interface através de linhas de comando. Como solução para o *parser* da ferramenta consideramos também utilizar o *BCompiler*[2], o *parser* para a notação B utilizado pelo *AtelierB*. No final optamos por utilizar o *BParser* pois seria mais fácil integrá-lo a ferramenta, visto que o mesmo também foi desenvolvido em Java, e por ele gerar uma árvore sintática mais fácil de ser manipulada.

Para desenvolver a interface gráfica da ferramenta utilizamos o framework *WindowBuilder*[3], que é gratuito e de código aberto, mantido pela Google. Ele possui um *plugin* para a plataforma *Eclipse*[4] com funcionalidades WYSIWYG (*What You See Is What You*

[1] http://www.java.com
[2] http://www.tools.clearsy.com/index.php5?title=B-Compiler_Project
[3] http://code.google.com/javadevtools/wbpro/index.html
[4] http://www.eclipse.org/

Get), além de permitir que o código gerado seja facilmente manipulado caso necessário.

Utilizamos desenvolvimento dirigido por testes (*Test Driven Development*, TDD) [BECK, 2002] desde o início da implementação da ferramenta. Até o momento da escrita deste trabalho, a *BETA* possuía aproximadamente 12 mil linhas de código (sendo destas, 4 mil linhas de teste), 120 casos de teste automatizados responsáveis por mais de 90% de cobertura do código.

6.3 Implementação

Nossa ferramenta é capaz de gerar especificações com testes de unidade para um programa a partir de sua especificação em B. Uma visão geral de sua arquitetura é apresentada na Figura 6.1 e o funcionamento de cada um de seus componentes será explicado em detalhes no decorrer desta seção.

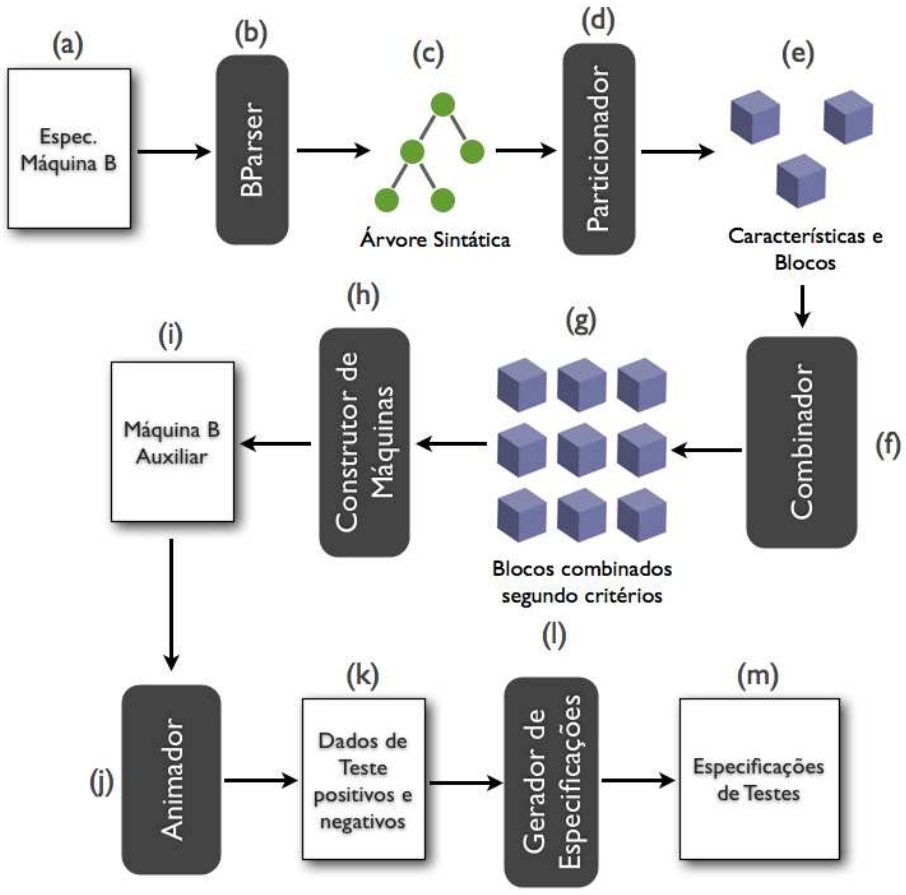

Figura 6.1: Visão geral do funcionamento da ferramenta

A ferramenta recebe como entrada a especificação de uma máquina abstrata B (a). Internamente, esta máquina passa pelo *BParser* (b), que verificará se a máquina está

sintaticamente correta e, caso esteja, devolverá como saída uma árvore sintática com a estrutura da máquina (c).

O particionador (d) utiliza esta árvore para retirar as informações necessárias para conduzir nossa abordagem. Estas informações são: operações da máquina, variáveis de estado, parâmetros das operações, cláusulas do invariante da máquina, pré-condições e corpo das operações, entre outras.

Após o usuário selecionar uma operação para gerar testes, o particionador calcula o espaço de entrada desta operação, e com base em suas características, devolve um conjunto de blocos de dados de teste (e). Cada um destes blocos é representado por uma cláusula lógica.

Estes blocos passam então por um combinador (f) que utiliza critérios de cobertura como *All-combinations*, *Each-choice* e *Pairwise* para combiná-los em fórmulas lógicas que são utilizadas para gerar os dados de teste (g). Cada uma destas fórmulas é uma conjunção dos blocos definidos previamente.

Para gerar os dados de entrada, estas fórmulas precisam ser avaliadas por um solucionador de restrições. Em nosso trabalho, utilizamos o *ProB* como solucionador de restrições e devido a isso precisamos fazer as animações utilizando máquinas B. Desta forma, utilizamos um construtor de máquinas (h) que recebe um conjunto de combinações e cria uma máquina de testes auxiliar (i). Esta máquina contém uma operação para cada combinação de fórmulas. Cada fórmula é colocada na pré-condição de uma operação, gerando uma máquina como a apresentada na Listagem 5.3.

Internamente, a ferramenta faz a chamada do animador do *ProB* (j) através de linhas de comando, passando como parâmetro a máquina de testes auxiliar. Este comando gera como saída um arquivo Prolog [CLOCKSIN; MELLISH, 2003], contendo os valores para os parâmetros das operações de teste gerados durante a animação (k).

Finalmente estes valores são passados como entrada para o gerador de especificações da ferramenta (l), que organiza estes valores em casos de teste que podem ser reescritos em uma linguagem de programação para a implementação dos testes concretos.

6.4 Utilizando a ferramenta

Nesta seção apresentamos a interface da ferramenta e descrevemos como ela deve ser utilizada para gerar uma especificação de testes para uma operação. Ao abrir a ferramenta,

o usuário será apresentado a janela ilustrada na Figura 6.2.

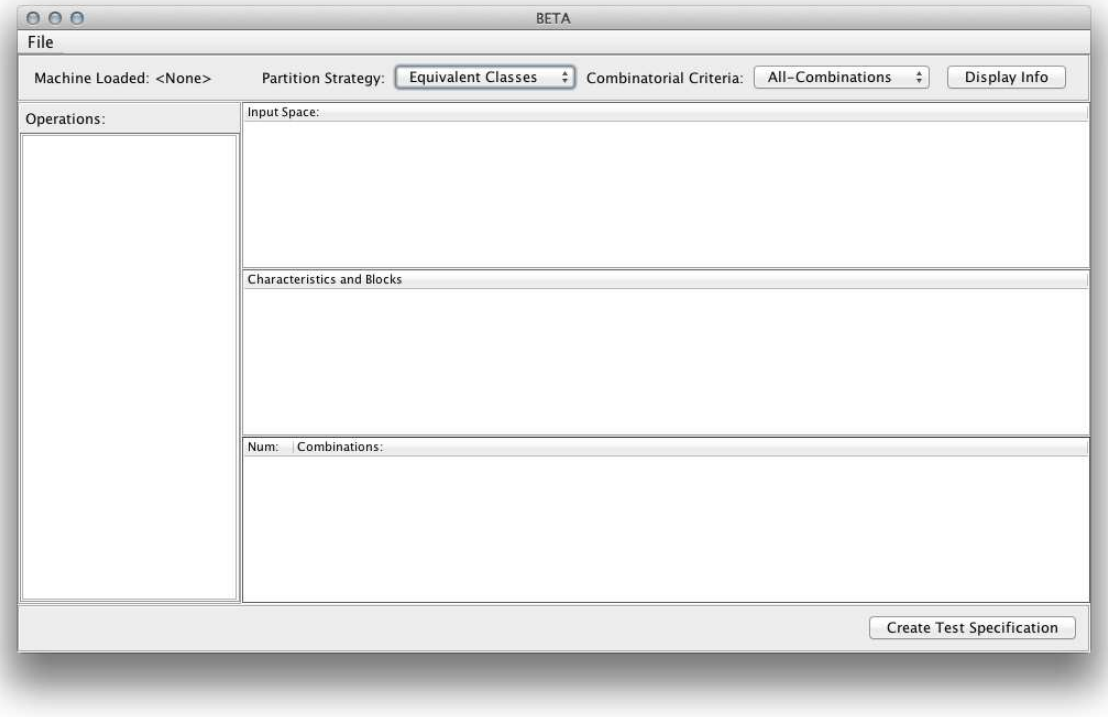

Figura 6.2: Janela principal da ferramenta

Para gerar os casos de teste o usuário deve primeiramente carregar uma máquina na ferramenta. Isso é feito selecionando o menu *"File"* e em seguida *"Load Machine"*. Ao carregar uma máquina, a mesma será verificada sintaticamente e, caso algum erro seja encontrado, a ferramenta solicitará que o problema seja corrigido antes de continuar com o processo. O foco da ferramenta não é a especificação de máquinas, logo qualquer mudança necessária na especificação deverá ser feita através de outro editor.

Após carregar uma máquina na ferramenta, será apresentada uma tela como a da Figura 6.3, com as seguintes informações:

1. O nome da máquina que foi carregada;
2. As estratégias de particionamento que o usuário pode selecionar. Como opções temos: *"Equivalent Classes"* (Classes de Equivalência) e *"Boundary Analisys"* (Análise de Valor Limite);
3. Os critérios para combinação dos blocos gerados. As opções são: *"All-combinations"*, *"Each-choice"* e *"Pairwise"*;

4. O botão *"Display Info"* é responsável por carregar as informações dos itens 6, 7 e 8;

5. Uma lista com as operações da máquina. Esta lista é populada após o carregamento da máquina e uma de suas operações deve ser selecionada antes de pressionar o botão *"Display Info"*;

6. A lista de variáveis do espaço de entrada da operação sob teste;

7. As características das variáveis do espaço de entrada da operação sob teste e seus respectivos blocos, criados de acordo com a estratégia de particionamento selecionada;

8. A lista de combinações de blocos criada de acordo com o critério de combinações selecionado;

9. O botão *"Create Test Specification"* é responsável por gerar a especificação de testes para a operação sob teste.

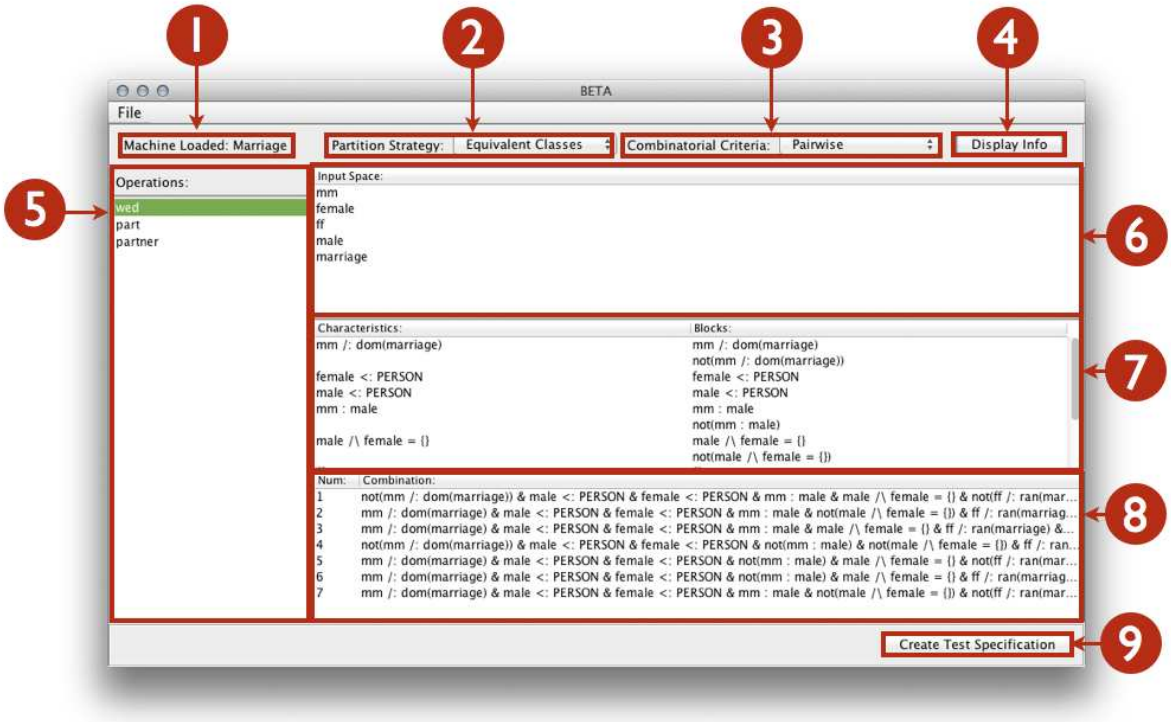

Figura 6.3: Visão da ferramenta após carregar uma máquina

Após selecionar o botão *"Create Test Specification"*, um arquivo com a especificação de testes para a operação será criado no mesmo diretório onde o fonte da máquina B carregada se encontra. Para a operação *addPaper* da máquina *Paperround* (Listagem 5.2) teríamos

como resultado a especificação apresentada na Listagem 6.1. Esta especificação foi gerada utilizando classes de equivalência como estratégia de particionamento e *Pairwise* como critério de combinação. Originalmente ela possuía cinco casos de teste mas foi reduzida aqui a somente dois testes por questões de espaço.

Listagem 6.1: Especificação de testes para a operação *addPaper*

```
1  TEST REPORT
2  MACHINE: Paperround
3
4  addPaperTest1() {
5  set_magazines({0});
6  set_papers({});
7  addPaper(164);
8  // Oracle Value must be calculated
9  expectedValueForMagazines = nil;
10 actualValueForMagazines = getMagazines();
11 assertEquals(expectedValueForMagazines, actualValueForMagazines);
12 expectedValueForPapers = nil;
13 actualValueForPapers = getPapers();
14 assertEquals(expectedValueForPapers, actualValueForPapers);
15 }
16
17 addPaperTest2() {
18 set_magazines({});
19 set_papers({1,2,3,4,5,6,7,8,9,10, ..., 60});
20 addPaper(-4);
21 // Oracle Value must be calculated
22 expectedValueForMagazines = nil;
23 actualValueForMagazines = getMagazines();
24 assertEquals(expectedValueForMagazines, actualValueForMagazines);
25 expectedValueForPapers = nil;
26 actualValueForPapers = getPapers();
27 assertEquals(expectedValueForPapers, actualValueForPapers);
28 }
```

Cada teste na especificação tenta simular a maneira como um caso de teste concreto seria implementado em uma linguagem de programação convencional. Consideremos como exemplo o primeiro teste (linhas 4 a 15). Inicialmente precisamos redefinir o estado da máquina para o estado que pretendemos testar. Isto é feito atribuindo valores necessários a suas variáveis de estado (linhas 5 e 6). Logo em seguida, a operação que estamos testando é executada com um valor de teste (linha 7). Finalmente verificamos se os valores que obtemos após a execução estão de acordo com o que o oráculo esperava (linhas 9 a 14). Os valores do oráculo podem ser calculados através do método apresentado na seção 5.8 do Capítulo 5. Futuramente pretendemos fazer este calculo do oráculo de maneira automática.

6.5 Considerações Finais

Neste capítulo apresentamos a ferramenta que desenvolvemos durante nosso trabalho: a *BETA*. Foi feita uma explanação sobre sua implementação, arquitetura e interfaces, além de apresentarmos um exemplo de especificação de testes gerada por ela.

A ferramenta ainda possui algumas limitações. Ela ainda não suporta todas as possibilidades de especificação que a gramática da notação B oferece. Também precisamos estudar a fundo características que envolvem cláusulas com quantificadores universais e existenciais, e definir como suas variáveis devem ser trabalhadas. Outro ponto importante é que a ferramenta até o presente momento só aceita como entrada máquinas abstratas, não sendo possível utilizar refinamentos e implementações.

Além disso, como dito anteriormente, ainda precisamos automatizar a criação dos oráculos de teste para complementar os casos de teste gerados por ela. Ao implementarmos os oráculos, seria possível adicionar a funcionalidade de geração de casos de teste concretos, complementando as especificações de teste.

Enquanto desenvolvemos a ferramenta procuramos facilitar possíveis extensões. Algumas extensões interessantes seriam o suporte a *Event-B* e a adição de novos critérios para combinação dos blocos de teste. Outras possibilidades de extensão deste projeto serão discutidas na seção de trabalhos relacionados do Capítulo 8.

A ferramenta continua em constante desenvolvimento e o projeto pode ser acompanhado em seu repositório no *Google Code*: http://code.google.com/p/btestgen/.

7 Estudos de Caso

7.1 Considerações Iniciais

Neste capítulo apresentamos os dois estudos de caso realizados durante nosso trabalho. O primeiro estudo de caso utilizou especificações de um módulo controlador de portas de metrô, para validar a abordagem de geração de testes proposta inicialmente. Já no segundo estudo de caso utilizamos especificações do *FreeRTOS*, um *kernel* para sistemas de tempo real, para validar nossa abordagem revisada e a ferramenta que desenvolvemos.

7.2 Primeiro estudo de caso: Controlador Geral de Portas

Em nosso primeiro estudo de caso utilizamos especificações de um módulo controlador de portas de metrôs, o *CGP* (Controlador Geral de Portas). Este módulo é desenvolvido pelo *Grupo AeS*[1], uma empresa situada em São Paulo que possui com uma de suas especialidades o desenvolvimento de software crítico para o setor ferroviário.

O *CGP* controla o estado das portas de um metrô e possui operações para abri-las e fechá-las. Estas operações respeitam um conjunto de restrições de segurança que levam em consideração, por exemplo, a velocidade do metrô no momento, sua posição na plataforma, seu modo de operação e possíveis sinais de emergência.

A especificação da máquina do *CGP* contém 19 operações, sendo 4 delas operações que definem o real intuito da máquina – ou seja, abrir e fechar portas da direita e esquerda do metrô – e 15 operações "setters" para variáveis de estado da máquina. Ela possui ainda 29 variáveis de estado, 46 cláusulas no invariante e pelo menos uma pré-condição para cada operação. A especificação desta e de outras máquinas que compõem o módulo controlador de portas foram realizadas por Barbosa em [BARBOSA, 2010]. Na Listagem

[1] http://www.grupo-aes.com.br

7.1 apresentamos um trecho da especificação do *CGP*.

Listagem 7.1: Trecho da especificação do CGP

```
1  MACHINE CGP
2
3  SEES Sets
4
5  INCLUDES Opening, Closing, Emergency
6
7  VARIABLES
8  doors_right_open_simultaneously,
9  doors_right_close_simultaneously,
10 internal_speed_over_6km_h,
11 conditions_to_open_satisfied
12 ...
13
14 INVARIANT
15 doors_right_open_simultaneously : BOOL &
16 doors_right_close_simultaneously : BOOL &
17 internal_speed_over_6km_h : BOOL &
18 conditions_to_open_satisfied : BOOL &
19 ((internal_speed_over_6km_h = TRUE) =>
20    (doors_right_open_simultaneously = FALSE &
21     doors_left_open_simultaneously = FALSE)) &
22 ((doors_right_open_simultaneously = TRUE or
23    doors_left_open_simultaneously = TRUE) =>
24    (conditions_to_open_satisfied = TRUE))
25 ...
26
27 INITIALISATION
28 ...
29
30 OPERATIONS
31 simultaneous_opening_all_doors_right =
32 PRE
33 internal_speed_over_6km_h = FALSE &
34 conditions_to_open_satisfied = TRUE
35 THEN ...
36 END;
37 ...
38 END
```

7.2.1 Objetivos

O objetivo deste primeiro estudo de caso era validar a abordagem proposta em [SOUZA, 2009]. Para isso, utilizamos um usuário sem conhecimento prévio da abordagem e sem formação em métodos formais para aplicar a abordagem em uma máquina da especificação do módulo controlador de portas.

7.2.2 Estudo de caso

Da mesma maneira que foi descrita na abordagem original, todo o processo de geração de casos de teste para uma operação da máquina *CGP* foi realizado manualmente. Escolhemos uma operação da máquina para gerarmos casos de teste: a *simultaneous_opening_all_doors_right*.

O processo de geração de testes ocorreu sem problemas até a etapa de criar as máquinas de teste auxiliares. Para a máquina de testes auxiliar responsável por gerar dados de teste válidos tivemos um problema de explosão combinatorial durante a animação. As restrições utilizadas nesta máquina permitiam uma grande quantidade de possíveis dados de teste. Este problema não foi considerado pela abordagem original, logo, nenhum procedimento para evitar este tipo de problema foi definido. Solucionamos este problema alterando parâmetros do solucionador de restrições, definindo uma quantidade máxima de estados computados durante a animação da máquina auxiliar. Na ferramenta que desenvolvemos, limitamos a quantidade de animações feitas para uma operação através de parâmetros de execução do *ProB*. Para cada operação, apenas uma possibilidade de animação é calculada. Já que cada fórmula animada representa um sub-domínio da operação sob teste, não existe a necessidade de gerar mais de uma combinação de dados para um mesmo sub-domínio, visto que todas as combinações geradas seriam equivalentes para testá-lo.

Encontramos um problema também durante a geração de dados de teste inválidos. A máquina de testes auxiliar criada possuía uma inconsistência na pré-condição que tornava impossível para o solucionador de restrições encontrar valores que a satisfizessem. Assim como o problema anterior, este também não foi considerado pela abordagem inicial e nada era dito sobre como proceder nesta situação. Concluímos que a pré-condição inconsistente presente na máquina tinha como objetivo testar uma situação que de acordo com as especificações nunca poderiam acontecer. Assim, concluímos que este caso de teste em particular poderia ser desconsiderado. Durante nossos estudos percebemos que, em grande parte dos casos, estas inconsistências ocorriam devido ao uso de várias cláusulas negadas (múltiplos blocos negativos) nas fórmulas de teste. Ainda precisamos trabalhar neste problema em nossa abordagem revisada, possivelmente alterando a estratégia de geração de casos de teste negativos para que apenas um bloco negativo esteja presente em uma fórmula de testes, reduzindo assim o número de inconsistências.

7.2.3 Resultados e conclusões

Após este estudo de caso concluímos que a abordagem era utilizável em projetos a nível de indústria, e poderia até mesmo ser aplicado por usuários que não fossem especialistas em métodos formais. Por outro lado, confirmamos que o processo de geração de testes da abordagem era demorado e bastante suscetível a erros quando realizado manualmente. Com isso em mente, iniciamos o desenvolvimento de uma ferramenta para automatizar este processo.

Os resultados obtidos por este estudo de caso foram avaliados pelo gerente do *Grupo AeS* e foram considerados como satisfatórios, principalmente no que diz respeito a possibilidade de geração de casos de teste negativos.

Vale ressaltarmos uma característica importante deste estudo de caso no que diz respeito as propriedades das máquinas utilizadas. Como a *CGP* é basicamente um sistema de sinalização e controle de operações, seu estado é formado somente por variáveis booleanas. Por isso, somente este estudo de caso não foi suficiente para validar todo o potencial da abordagem. Para avaliar características como, por exemplo, a geração de testes utilizando análise de valor limite, seria necessário a realização de outros estudos de caso com máquinas que tirassem proveito desta característica.

Por fim, consideramos como muito importante a realização deste estudo de caso, pois a partir dele obtivemos subsídios para propôr melhorias para a abordagem original. Mais detalhes sobre este estudo de caso podem ser encontrados em [MATOS *et al.*, 2010].

7.3 Segundo estudo de caso: FreeRTOS

O *FreeRTOS* é um *kernel* para sistemas de tempo real simples, de código aberto, escrito em C e *assembly*. Ele provê uma camada de abstração entre a aplicação e o hardware, facilitando o acesso aos recursos do hardware através de uma biblioteca de tipos e funções que pode ser utilizada pela aplicação.

Ele se destaca por ser simples, possuindo cerca de duas mil linhas de código distribuídas entre quatro arquivos (*task.c*, *queue.c*, *croutine.c* e *list.c*), e por sua portabilidade, suportando 17 arquiteturas diferentes.

Dentre as funcionalidades providas pelo *FreeRTOS* estão: gerenciamento de tarefas, comunicação e sincronização entre tarefas, gerenciamento de memória e controle de dispositivos de entrada e saída. Mais informações sobre *FreeRTOS* podem ser encontradas

na página oficial: http://www.freertos.org.

A especificação do *FreeRTOS* utilizando o Método B foi realizada durante o mestrado de Stepheson Galvão. Em sua dissertação [GALVÃO, 2010], foi feita a especificação dos seguintes componentes do *FreeRTOS*: *tarefas*, *escalonador*, *fila de mensagens* e *mutex*.

A especificação foi organizada nas seguintes máquinas:

- *Types*: especifica as abstrações de tipos utilizadas na modelagem do *FreeRTOS*;
- *FreeRTOSConfig*: especifica as variáveis de configuração do *FreeRTOS*, usadas antes da compilação;
- *Task*: especifica as abstrações de hardware *tarefa* e *escalonador*, assim como suas propriedades;
- *Queue*: especifica as abstrações de hardware *fila de mensagens*, *semáforo*, *mutex* e suas respectivas propriedades;
- *FreeRTOSBasic*: especifica algumas funcionalidades do sistema, agrupa operações dos módulos de abstrações em operações mais complexas e especifica propriedades compostas pela união das propriedades *Task* e *Queue*;
- *FreeRTOS*: especifica funcionalidades mais complexas utilizando as operações do módulo anterior.

7.3.1 Objetivos

Nosso objetivo com este estudo de caso era validar a abordagem revisada e a ferramenta desenvolvida durante nosso trabalho. Utilizamos a especificação da máquina *Queue* do *FreeRTOS* para isto. As especificações de *Queue* possuem características que procuramos considerar durante nosso trabalho como estruturação em vários componentes e condicionais no corpo da operação. Além disso as especificações do *FreeRTOS* refletem exemplos de especificações que podemos encontrar na indústria.

7.3.2 Estudo de Caso

Nosso estudo de caso focou na especificação da máquina *Queue* (apresentada na Listagem 8.1 no Anexo A), que representa a estrutura de fila de mensagens do sistema. Ela controla a comunicação entre tarefas, através de mensagens que são enviadas e retiradas

de uma fila. A especificação das propriedades da fila foi feita utilizando conjuntos como variáveis de estado. Dentre estas variáveis de estado temos:

- *queues*: união de todas as filas de mensagens, semáforos e mutex;
- *queue_msg*: conjunto de filas gerenciado pelo sistema;
- *queue_msg_full*: conjunto contendo as filas de mensagens que estão cheias;
- *queue_msg_empty*: conjunto contendo as filas de mensagens que estão vazias;
- *queue_items*: conjunto de itens aos quais a fila de mensagens é associada;
- *queue_receiving*: conjunto de tarefas bloqueadas por leitura;
- *queue_sending*: conjunto de tarefas bloqueadas por escrita;
- *first_sending*: primeira tarefa que deve ser retirada do conjunto de tarefas bloqueadas por escrita;
- *first_receiving*: primeira tarefa que deve ser retirada do conjunto de tarefas bloqueadas por leitura;

A máquina possui ainda operações que manipulam estes conjuntos, enviam e recebem mensagens de uma fila e adicionam e excluem tarefas de uma das listas de tarefas bloqueadas pela fila. A seguir listamos algumas destas operações que foram avaliadas em nosso estudo de caso:

- *q_queueCreate*: cria uma nova fila de mensagens no sistema;
- *q_queueDelete*: exclui uma fila de mensagens do sistema;
- *q_sendItem*: envia uma mensagem para a fila de mensagens;
- *q_receivedItem*: recebe uma mensagem de uma fila de mensagens;
- *q_insertTaskWaitingToSend*: insere uma tarefa no conjunto de tarefas bloqueadas por escrita de uma fila de mensagens;
- *q_insertTaskWaitingToReceive*: insere uma tarefa no conjunto de tarefas bloqueadas por leitura de uma fila de mensagens;

- *q_removeFromEventListQueue*: remove uma tarefa dos conjuntos de tarefas bloqueadas de uma fila de mensagens.

Como o estado de *Queue* é composto somente por conjuntos, utilizamos apenas o particionamento em classes de equivalência na geração dos casos de teste. Algumas informações como quantidade de variáveis que compõem o espaço de entrada, quantidade de características e total de blocos que compõem o particionamento de cada operação são apresentadas na Tabela 7.1.

Operação	Variáveis E. E.	Características	Total Blocos
q_queueCreate	2	2	3
q_queueDelete	16	42	76
q_sendItem	14	42	74
q_receivedItem	17	38	67
q_insertTaskWaitingToSend	17	39	69
q_insertTaskWaitingToReceive	17	39	69
q_removeFromEventListQeue	1	1	1

Tabela 7.1: Informações sobre a máquina *Queue*.

As variáveis do espaço de entrada de uma operação são formadas por todas as variáveis da especificação que influenciam no comportamento da operação. As características são cláusulas lógicas que impõem restrições a estas variáveis. A partir destas características são gerados blocos de dados que são utilizados na geração de teste.

Considere como exemplo a operação *q_queueDelete*. Ao gerar testes para ela a ferramenta calcula que 16 variáveis da especificação (entre parâmetros da operação e variáveis de estado) influenciam em seu comportamento. Ela calcula ainda que 42 restrições (características) são aplicadas a estas variáveis. A seguir listamos três destas características:

1. $pxQueue \in queue$
2. $queue\_receiving(pxQueue) = \emptyset$
3. $queue\_sending(pxQueue) = \emptyset$

Respectivamente, estas características definem que: 1) *pxQueue* (a fila que será apagada) deve pertencer ao conjunto geral de filas ativas, 2) *pxQueue* não deve estar bloqueada por leitura e 3) *pxQueue* não deve estar bloqueada por escrita.

Após detectar estas características, a ferramenta utiliza a estratégia de particionamento selecionada pelo usuário (Classes de Equivalência ou Análise de Valor Limite) para gerar blocos de teste para elas. Neste exemplo utilizamos particionamento em classes de equivalência e obtivemos os seguintes blocos:

1. *pxQueue* deve pertencer ao conjunto geral de filas ativas:

   B1: $pxQueue \in queue$ (*pxQueue* pertence ao conjunto)

   B2: $pxQueue \notin queue$ (*pxQueue* não pertence ao conjunto)

2. *pxQueue* não deve estar bloqueada por leitura:

   B1: $queue\_receiving(pxQueue) = \emptyset$ (*pxQueue* está bloqueada por leitura)

   B2: $queue\_receiving(pxQueue) \neq \emptyset$ (*pxQueue* não está bloqueada por leitura)

3. *pxQueue* não deve estar bloqueada por escrita:

   B1: $queue\_sending(pxQueue) = \emptyset$ (*pxQueue* está bloqueada por escrita)

   B2: $queue\_sending(pxQueue) \neq \emptyset$ (*pxQueue* não está bloqueada por escrita)

Estes blocos são então combinados pela ferramenta seguindo o critério de combinação selecionado, gerando assim casos de teste para diferentes cenários. Para os blocos acima teríamos as seguintes possibilidades de casos de teste para cada critério de combinação:

Utilizando *All-Combinations*:

1. $pxQueue \in queue \wedge queue\_receiving(pxQueue) = \emptyset \wedge queue\_sending(pxQueue) = \emptyset$
2. $pxQueue \in queue \wedge queue\_receiving(pxQueue) \neq \emptyset \wedge queue\_sending(pxQueue) = \emptyset$
3. $pxQueue \in queue \wedge queue\_receiving(pxQueue) = \emptyset \wedge queue\_sending(pxQueue) \neq \emptyset$
4. $pxQueue \in queue \wedge queue\_receiving(pxQueue) \neq \emptyset \wedge queue\_sending(pxQueue) \neq \emptyset$
5. $pxQueue \notin queue \wedge queue\_receiving(pxQueue) = \emptyset \wedge queue\_sending(pxQueue) = \emptyset$
6. $pxQueue \notin queue \wedge queue\_receiving(pxQueue) \neq \emptyset \wedge queue\_sending(pxQueue) = \emptyset$
7. $pxQueue \notin queue \wedge queue\_receiving(pxQueue) = \emptyset \wedge queue\_sending(pxQueue) \neq \emptyset$
8. $pxQueue \notin queue \wedge queue\_receiving(pxQueue) \neq \emptyset \wedge queue\_sending(pxQueue) \neq \emptyset$

Utilizando *Each-Choice*:

1. $pxQueue \in queue \wedge queue\_receiving(pxQueue) = \emptyset \wedge queue\_sending(pxQueue) = \emptyset$
2. $pxQueue \notin queue \wedge queue\_receiving(pxQueue) \neq \emptyset \wedge queue\_sending(pxQueue) \neq \emptyset$

Utilizando *Pairwise*:

1. $pxQueue \in queue \wedge queue\_receiving(pxQueue) = \emptyset \wedge queue\_sending(pxQueue) = \emptyset$
2. $pxQueue \in queue \wedge queue\_receiving(pxQueue) \neq \emptyset \wedge queue\_sending(pxQueue) \neq \emptyset$
3. $pxQueue \notin queue \wedge queue\_receiving(pxQueue) = \emptyset \wedge queue\_sending(pxQueue) \neq \emptyset$
4. $pxQueue \notin queue \wedge queue\_receiving(pxQueue) \neq \emptyset \wedge queue\_sending(pxQueue) = \emptyset$

Considerando os testes gerados utilizando *All-Combinations*, no primeiro caso de teste temos o cenário em que *pxQueue* pertence ao conjunto geral de filas ativas e não está bloqueado por leitura e por escrita. No segundo, *pxQueue* pertence ao conjunto geral de filas ativas, está bloqueado por leitura, mas não está bloqueado por escrita. No terceiro, *pxQueue* pertence ao conjunto geral de filas ativas, não está bloqueado por leitura, mas está bloqueado por escrita, e assim por diante. Vale ressaltar que em alguns casos os critérios de combinação podem ser satisfeitos por outras combinações de blocos. Ou seja, existe mais de um conjunto de casos de teste que satisfazem o critério.

Por fim, após a combinação dos dados de teste, os casos de teste para a operação sob teste gerados pela ferramenta são compilados em uma especificação de testes. A Tabela 7.2 apresenta um resumo sobre as especificações de teste geradas para as operações de *Queue*. Todos os casos de teste foram gerados utilizando particionamento em classes de equivalência e utilizaram como critérios para combinação dos blocos os critérios *All combinations* (*AC*), *Each choice* (*EC*) e *Pairwise* (PW). Após a combinação dos blocos, ao animarmos as combinações para a geração de dados de teste, algumas destas combinações podem se mostrar inviáveis (explicamos o porquê disto na próxima seção), desta forma contabilizamos os casos de teste viáveis nas colunas *AC**, *EC** e *PW** da tabela.

Operação	AC	AC*	EC	EC*	PW	PW*
q_queueCreate	2	2	2	2	2	2
q_queueDelete	100+	0	2	0	18	1
q_sendItem	100+	100+	2	1	15	3
q_receivedItem	100+	100+	2	1	16	3
q_insertTaskWaitingToSend	100+	100+	2	1	16	2
q_insertTaskWaitingToReceive	100+	100+	2	1	14	2
q_removeFromEventListQeue	1	1	1	1	0	0

Tabela 7.2: Relatório de casos de teste gerados.

7.3.3 Resultados e conclusões

Primeiramente devemos apontar que, para alguns casos de teste como os gerados por *Each-choice* para a operação *q_queueDelete*, não foi possível gerar cenários de teste viáveis (os exemplos de combinações que mostramos anteriormente estão simplificados e não contém todos os blocos, a primeira combinação utilizando *Each-Choice* pode parecer viável mas, se considerarmos os blocos omitidos, ela não é). Este é um exemplo de caso de teste que requer um estado impossível de ser calculado pelo solucionador de restrições. Ou seja, a combinação de alguns blocos em particular gerou uma insatisfabilidade.

O principal motivo para a geração de fórmulas de teste insatisfatíveis diz respeito a restrições contraditórias entre blocos de um mesmo caso de teste. Por exemplo, em um mesmo caso de teste podemos ter um bloco que requer que um elemento xx pertença a um conjunto yy ($xx \in yy$) e que o conjunto yy seja vazio ($yy = \emptyset$). É impossível para o solucionador de restrições encontrar um conjunto de dados que satisfaça essas duas regras, logo este cenário de teste é insatisfatível.

Outro motivo para não conseguirmos obter dados de teste para algumas fórmulas diz respeito ao funcionamento do *ProB*. Em alguns casos o *ProB* pode inferir o tipo de uma variável através de alguma cláusula da fórmula de testes. Por exemplo, se temos que uma variável xx pertence a um conjunto yy ($xx \in yy$) e que yy é subconjunto dos naturais ($yy \subset NAT$), o *ProB* consegue inferir que o tipo de xx é NAT. Em alguns de nossos casos de teste nós podemos negar uma cláusula que seria utilizada pelo *ProB* para realizar esta inferência, quando negamos tal clásula, o *ProB* não consegue mais fazer inferências de tipo através dela. Quando esta situação ocorre, o *ProB* não consegue animar a fórmula de testes. Uma possível solução para este problema seria forçar o uso de tipagem explícita nas especificações, como por exemplo: $xx \in NAT$. Isto resultaria em um novo bloco a ser considerado durante as combinações, mas já que este bloco é um bloco simples de tipagem

que não gera blocos negativos, não haveria mudanças no resultado final obtido.

Também obtivemos casos em que, ao utilizarmos *All-Combinations* como critério de combinação, ocorreram explosões combinatoriais gerando, em alguns casos, milhões de casos de teste. Isto ocorre devido ao grande número de características a serem testadas de certas operações. Consideremos como exemplo a operação *q_queueDelete*, ela possui 42 características a serem testadas, 34 destas características geram dois blocos e 8 geram um bloco. Se quiséssemos testar todas as combinações possíveis de blocos para esta operação teríamos $2^{34} \times 1^8$ que resultaria em 17.179.869.184 combinações. Para evitar este tipo de explosão combinatorial na ferramenta, limitamos o número de casos de teste gerados por este critério para no máximo 100.

No caso dos testes combinados através de *Each Choice*, o número reduzido de testes se deve a simplicidade do critério. Esse critério exige apenas que cada bloco apareça ao menos uma vez no conjunto de testes. O número de testes necessários para satisfazer este critério é igual a quantidade de blocos da característica com mais blocos. Como em nossos exemplos são gerados no máximo dois blocos por característica, dois casos de teste são suficientes para satisfazê-lo.

Para os casos de teste gerados utilizando *Pairwise*, nota-se a quantidade de casos de teste perdidos por serem considerados insatisfatíveis pelo solucionador de restrições. Isso ocorre devido a combinação de múltiplos blocos negativos em um único caso de teste, tornando impossível a geração de um conjunto de dados de teste que satisfaça o cenário em questão. Ainda precisamos avaliar a possibilidade de utilizar apenas um bloco negativo por caso de teste, reduzindo assim a ocorrência deste tipo de situação.

Outra possibilidade de mudança na abordagem para futuro diz respeito ao tratamento das cláusulas do invariante para a geração de blocos teste. Atualmente cláusulas do invariante e das pré-condições são tratadas igualmente durante a geração de blocos de teste. Trabalhando desta forma, temos a possibilidade de gerar casos de teste que desrespeitam o invariante da máquina. Nestes casos de teste estamos estabelecendo que a operação sob teste poderia ser executada em um cenário em que o estado do sistema estava inconsistente (o caso de teste poderia ser executado em um cenário em que o invariante está "quebrado"). Como nosso foco são testes de unidade, podemos considerar que a operação sob teste só será executada em cenários onde o estado do sistema está consistente, não sendo mais necessário gerar blocos negativos para cláusulas do invariante. Esta mudança reduziria bastante a quantidade de casos de teste insatisfatíveis, já que teríamos menos conflitos com blocos negativos. Futuramente podemos voltar a trabalhar com altera-

ções no invariante da máquina, quando começarmos a focar em testes de segurança, por exemplo.

Por fim, outro fator que devemos observar é a diferença entre os resultados obtidos pela abordagem original e pela abordagem revisada. Ainda considerando a operação *q_queueDelete*, enquanto nossa abordagem gerou mais de 100 casos de teste para esta operação, a abordagem de Souza gerou apenas 19. Pode parecer que a abordagem de Souza é mais eficiente que a nossa por reduzir a quantidade de casos de teste gerados, mas na verdade esta quantidade é reflexo da estratégia de particionamento empregada.

Nossa estratégia de particionamento é capaz de computar mais cenários de teste com base em características de uma especificação, pois levamos em consideração fatores como: a presença de condicionais dentro de uma operação, características provenientes de outras máquinas que compunham a estrutura da especificação e a semântica de algumas cláusulas de tipagem.

Este segundo estudo de caso nos mostrou outros pontos da abordagem que ainda precisam ser melhor trabalhados. O primeiro deles diz respeito a geração de casos de teste insatisfatíveis. Como mencionado anteriormente, a abordagem ainda gera uma quantidade considerável de cenários de teste insatisfatíveis, principalmente devido a combinação de múltiplos blocos de teste negativos. Podemos evitar este problema se desenvolvermos uma estratégia especial para a geração de casos de teste negativos, que utilize apenas um bloco negativo por caso de teste.

Outro ponto que pode ser melhor trabalhado é a maneira como tratamos as cláusulas do invariante durante o particionamento em blocos. Ao gerarmos blocos negativos a partir de cláusulas do invariate, estamos criando cenários de teste em que uma operação poderia ser executada a partir de um estado inconsistente do sistema. Já que o foco deste trabalho é a geração de testes de unidade, talvez seja mais interessante considerar que uma operação sempre será executada a partir de um estado consistente do sistema e não gerar blocos negativos para cláusulas do invariante. Esta alteração na abordagem também ajudaria a reduzir a quantidade de casos de teste insatisfatíveis gerados pela abordagem.

Este estudo de caso também nos mostrou problemas a respeito da usabilidade da ferramenta, principalmente sobre os relatórios de teste gerados. Atualmente os resultados gerados são difíceis de ser interpretados pelo usuário. Precisamos melhorar a legibilidade dos relatórios gerados para que o usuário possa fazer melhores interpretações sobre os resultados obtidos.

7.4 Considerações Finais

Neste capítulo apresentamos os dois estudos de caso realizados em nosso trabalho: o *Controlador Geral de Portas* e o *FreeRTOS*. O primeiro estudo de caso com o *CGP* foi importante para validar a abordagem de geração de testes original, prover subsídios para a revisão da abordagem e confirmar a necessidade de uma ferramenta para automatizá-la. Com o segundo estudo de caso, pretendíamos validar a abordagem revisada e a ferramenta desenvolvida, gerando especificações de teste para especificações B baseadas em sistemas presentes na indústria.

8 Considerações Finais

Como dito anteriormente, atualmente existe o interesse das comunidades de métodos formais e testes de software em integrar ambas disciplinas. Este trabalho contribui para esta integração, apresentando uma abordagem para geração de testes de unidade a partir de uma notação formal.

Em nosso trabalho aperfeiçoamos uma abordagem para geração de casos de teste a partir de especificações B, inicialmente proposta em [SOUZA, 2009]. Além disso, implementamos uma ferramenta para automatizar o processo de geração testes e realizamos alguns estudos de caso para validar a abordagem e a ferramenta desenvolvida.

Iniciamos o trabalho com um estudo de caso em que um usuário, sem conhecimento prévio sobre a abordagem e sem formação em métodos formais, deveria aplicar a abordagem em especificações B baseadas em um sistema proveniente da indústria, neste caso, o *Controlador Geral de Portas* desenvolvido pelo *Grupo AeS*. Este estudo de caso nos proveu subsídios para propor melhorias à abordagem original.

A evolução da abordagem original começou com a formalização dos conceitos de teste empregados por ela. Da maneira que a abordagem foi descrita inicialmente, era difícil visualizar como estes conceitos eram utilizados durante o processo de geração de teste. Em nosso trabalho, a abordagem foi reescrita, e alterada em alguns pontos, para que ficasse de acordo com conceitos de teste apresentados em [AMMANN; OFFUTT, 2008], além de tornar mais clara a forma como eles eram empregados.

Outras deficiências confirmadas durante o primeiro estudo de caso foram a demora na geração de casos de teste e a suscetibilidade a erros do processo quando o mesmo era executado manualmente. Devido a isso, tomamos como prioridade o desenvolvimento de uma ferramenta que automatizasse a abordagem, fazendo com que todo o processo de geração de casos de teste pudesse ser realizado com poucos cliques.

A ferramenta que desenvolvemos é capaz de, a partir de uma máquina B previamente verificada, fazer o particionamento do espaço de entrada de suas operações utilizando

técnicas de particionamento em classes de equivalência e análise de valor limite. Após o particionamento, é possível combinar os blocos de dados gerados utilizando critérios para combinação de dados de teste como o *Each-choice* e *Pairwise*. Por fim, a ferramenta é capaz de animar as combinações geradas em um solucionador de restrições e gerar especificações contendo casos de teste para a operação.

Após o desenvolvimento da ferramenta realizamos um outro estudo de caso para validá-la. Com este estudo de caso pudemos analisar as especificações de teste geradas pela ferramenta e comparar os resultados da abordagem revisada com os resultados da abordagem original.

8.1 Trabalhos Futuros

Em nosso trabalho estabelecemos uma boa base para que outros trabalhos pudessem ser derivados a partir dele. A seguir listamos alguns destes trabalhos e possíveis melhorias que ainda podem ser realizadas na abordagem e na ferramenta:

- *Verificação formal da abordagem:* a nossa abordagem foi definida utilizando descrições informais e pseudocódigo. Seria interessante especificar os algoritmos utilizados através do Método B. Desta forma poderíamos provar a abordagem formalmente e futuramente utilizar estas especificações para gerar testes para nossa ferramenta. Este seria um bom exemplo de estudo de caso para situações em que temos a especificação formal de um software que não foi desenvolvido formalmente, e pretendemos gerar testes para tal software.

- *Automatização da geração de oráculos de teste:* criar oráculos de teste é uma tarefa difícil e que exige habilidades de um engenheiro de testes. Utilizando animações de uma máquina B podemos verificar os resultados esperados após a execução de um determinado caso de teste. Atualmente, a ferramenta é capaz de gerar especificações com casos de teste e seus respectivos dados de entrada, mas o cálculo dos valores do oráculo é feito através de um processo manual, que utiliza como apoio o *ProB*. Poderíamos automatizar este processo de geração de oráculos através do uso de uma *API* que está sendo desenvolvida pelo grupo que desenvolve o *ProB*. Esta *API* permitirá a melhor manipulação do processo de animação de uma especificação, facilitando a criação de nossos oráculos. Outro problema que ainda precisa ser trabalhado em relação aos oráculo diz respeito a operações cujo comportamento é

não-determinístico. Ainda é necessário estabelecer uma abordagem para trabalhar com este tipo de operação.

- *Expandir a abordagem para a geração de testes que precisam executar uma sequência de operações antes de sua execução:* existem casos em que precisamos de uma maneira para levar o sistema ao estado necessário para que alguns casos de teste sejam executados. Atualmente simulamos este comportamento utilizando operações de *set* para levar a máquina ao estado para o qual pretendemos executar um caso de teste. Isto poderia ser melhorado se, ao invés de operações de *set*, utilizássemos uma ferramenta de *model-checking* para detectar as sequências de chamadas de operações necessárias para levar a máquina ao estado desejado, através dos grafos gerados por ela.

- *Adicionar novos critérios de cobertura e técnicas de teste na abordagem:* é possível expandir a ferramenta para que ela utilize novos critérios de cobertura e técnicas para a seleção de dados de teste. Seria necessário estudar quais técnicas e critérios se destacam atualmente na comunidade de testes e verificar sua compatibilidade com nossa abordagem.

- *Elaborar uma abordagem para a geração de testes de sistema a partir de Event-B*: o *Event-B* é um método formal para especificação a nível de sistemas derivado do Método B. Existem casos de projetos em que partes do sistema, relacionadas a um nível mais abstrato, são especificadas utilizando *Event-B* enquanto detalhes mais concretos sobre as operações são especificados utilizando o Método B. Seria interessante estudar e elaborar uma estratégia para a geração de testes de nível de sistema a partir de especificações em *Event-B*, complementando os testes já gerados para os níveis mais concretos de implementação.

- *Estudar áreas de aplicação específicas:* em nosso grupo de pesquisa temos a oportunidade de realizar estudos de caso com sistemas de diferentes gêneros. Podemos trabalhar com sistemas de tempo real como o *FreeRTOS* [GALVÃO, 2010], com sistemas de cartões inteligentes como o *KitSmart* [GOMES *et al.*, 2010] e sistemas de controle como o *CGP* [DEHARBE *et al.*, 2007]. Desta forma podemos avaliar a eficácia da ferramente para diversas áreas de aplicação específicas.

- *Integrar nossa ferramenta a alguma ferramenta já estabelecida no mercado:* outro projeto interessante seria a integração da nossa ferramenta a alguma IDE para desenvolvimento formal já estabelecida no mercado. Desta forma, o processo de es-

pecificação, desenvolvimento e testes seria completamente integrado. Alguma possibilidades de integração seriam as ferramenta *ProB*, *AtelierB* e *Rodin*[1].

[1]http://www.event-b.org/

Referências Bibliográficas

ABRIAL, J. R. *The B Book: Assigning Programs to Meanings*. Cambridge: Cambridge University Press, 1996.

ABRIAL, J. R. *et al.* Rodin: an open toolset for modelling and reasoning in Event-B. *STTT*, v. 12, n. 6, p. 447–466, 2010.

AICHERNIG, B. Automated black-box testing with abstract VDM oracle. In: *Computer Safety, Reliability and Security*. Berlin: Springer Berlin / Heidelberg, 1999, (Lecture Notes in Computer Science, v. 1698). p. 688–688.

AMBERT, F. *et al.* BZ-TT: A tool-set for test generation from Z and B using constraint logic programming. *Proc. of Formal Approaches to Testing of Software, FATES 2002 (workshop of CONCUR'02)*, p. 105–120, 2002.

AMLA, N.; AMMANN, P. Using Z specifications in category partition testing. *Proceedings of the Seventh Annual Conference on Computer Assurance, COMPASS '92*, p. 3–10, 1992.

AMMANN, P.; OFFUTT, J. *Introduction to Software Testing*. 1. ed. Cambridge: Cambridge University Press, 2008.

BALCER, M.; HASLING, W.; OSTRAND, T. Automatic generation of test scripts from formal test specifications. *Proceedings of the ACM SIGSOFT '89 third symposium on Software testing, analysis, and verification*, ACM, p. 210–218, 1989.

BARBOSA, H. *Desenvolvendo um sistema crítico através de formalização de requisitos utilizando o método B*. 2010. Monografia de Graduação, DIMAp/UFRN.

BECK, K. *Test Driven Development: By Example*. 1. ed. Toronto: Addison-Wesley Professional, 2002.

BOUQUET, F.; DADEAU, F.; LEGEARD, B. Automated test generation from JML specifications. In: *FM 2006: Formal Methods*. Berlin: Springer Berlin / Heidelberg, 2006, (Lecture Notes in Computer Science, v. 4085). p. 428–443.

BOUQUET, F.; LEGEARD, B.; PEUREUX, F. CLPS-B: A constraint solver for B. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin: Springer Berlin / Heidelberg, 2002, (Lecture Notes in Computer Science, v. 2280). p. 235–256.

BURTON, S.; YORK, H. *Automated Testing from Z Specifications*. York, 2000. Report: University of York.

CHEON, Y.; LEAVENS, G. A simple and practical approach to unit testing: The JML and JUnit way. In: *ECOOP 2002 - Object-Oriented Programming*. Berlin: Springer Berlin / Heidelberg, 2002, (Lecture Notes in Computer Science, v. 2374). p. 1789–1901.

CLEARSY. *B Language Reference Manual*. 2011. `http://www.tools.clearsy.com/images/0/07/Manrefb_en.pdf`. Version 1.8.7.

CLOCKSIN, W. F.; MELLISH, C. S. *Programming in Prolog: Using the ISO Standard*. 1. ed. Berlin: Springer, 2003.

CSK, C. *VDMTools: The VDM++ Language*. 2005. `http://www.csk.co.jp/`. Version 6.8.1.

DEHARBE, D. *et al.* Modelling control systems in B: an industrial case study. *Proceedings of Brazilian Symposium on Formal Methods*, 2007.

DICK, J.; FAIVRE, A. Automating the generation and sequencing of test cases from model-based specifications. In: *FME '93: Industrial-Strength Formal Methods*. Berlin: Springer Berlin / Heidelberg, 1993, (Lecture Notes in Computer Science, v. 670). p. 268–284.

DIJKSTRA, E. W. The humble programmer. *Commun. ACM*, ACM, v. 15, p. 859–866, 1972.

FLETCHER, R.; SAJEEV, A. A framework for testing object oriented software using formal specifications. In: *Reliable Software Technologies - Ada-Europe '96*. Berlin: Springer Berlin / Heidelberg, 1996, (Lecture Notes in Computer Science, v. 1088). p. 159–170.

FMWIKI. *Formal Methods*. nov. 2011. `http://formalmethods.wikia.com`.

FOWLER, M. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. 3. ed. Toronto: Addison-Wesley Professional, 2003.

GALVÃO, S. d. S. L. *Especificação do micronúcleo FreeRTOS utilizando método B*. 2010. Tese de Mestrado, DIMAp/UFRN.

GOMES, B. *et al.* Applying the B method for the rigorous development of smart card applications. In: *Abstract State Machines, Alloy, B and Z*. Berlin: Springer Berlin / Heidelberg, 2010, (Lecture Notes in Computer Science, v. 5977). p. 203–216.

GOMES, B. E. G. *BSmart: Desenvolvimento Rigoroso de Aplicações Java Card com base no Método Formal B*. Dissertação (Mestrado) — Departamento de Informática e Matemática Aplicada, Universidade Federal do Rio Grande do Norte, Natal, 2007.

GUPTA, A.; BHATIA, R. Testing functional requirements using B model specifications. *SIGSOFT Softw. Eng. Notes*, ACM, v. 35, n. 2, p. 1–7, 2010.

HUAIKOU, M.; LING, L. A test class framework for generating test cases from Z specifications. *Engineering of Complex Computer Systems, IEEE International Conference on*, IEEE Computer Society, p. 0164, 2000.

JACKSON, D. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, ACM, v. 11, p. 256–290, April 2002.

JACKSON, D.; SCHECHTER, I.; SHLYAKHTER, I. Alcoa: the Alloy constraint analyzer. *Proceedings of the 22nd. International Conference on Software Engineering*, Society, ACM Press, 2000.

LEAVENS, G.; BAKER, A.; RUBY, C. JML: a java modeling language. *Proceedings of Formal Underpinnings of Java Workshop (at OOPSLA'98)*, 1998.

LEGEARD, B.; PEUREUX, F.; UTTING, M. Automated boundary testing from Z and B. In: *FME 2002: Formal Methods, Getting IT Right*. Berlin: Springer Berlin / Heidelberg, 2002, (Lecture Notes in Computer Science, v. 2391). p. 221–236.

LEUSCHEL, M.; BUTLER, M. ProB: A model checker for B. In: *FME 2003: Formal Methods*. Berlin: Springer Berlin / Heidelberg, 2003, (Lecture Notes in Computer Science, v. 2805). p. 855–874.

MARINOV, D.; KHURSHID, S. TestEra: A novel framework for automated testing of Java programs. *International Conference on Automated Software Engineering*, IEEE Computer Society, v. 0, p. 22, 2001.

MATOS, E. C. B. *et al.* Generating test cases from B specifications: An industrial case study. *Proceedings of 22nd IFIP International Conference on Testing Software and Systems*, 2010.

MCDONALD, J.; MURRAY, L.; STROOPER, P. Translating Object-Z specifications to object-oriented test oracles. *Asia-Pacific Software Engineering Conference*, IEEE Computer Society, 1997.

MENDES, E.; SILVEIRA, D. S.; LENCASTRE, M. Testimonium: Um método para geração de casos de teste a partir de regras de negócio expressas em OCL. *IV Brazilian Workshop on Systematic and Automated Software Testing, SAST*, 2010.

MYERS, G. J. *The Art of Software Testing*. 3. ed. Hoboken, New Jersey: John Wiley & Sons, Inc., 2011.

NADEEM, A.; LYU, M. R. A framework for inheritance testing from VDM++ specifications. *Pacific Rim International Symposium on Dependable Computing, IEEE*, IEEE Computer Society, p. 81–88, 2006.

NADEEM, A.; UR-REHMAN, M. J. A framework for automated testing from VDM-SL specifications. *Proceedings of INMIC 2004, 8th IEEE International Multitopic Conference*, p. 428–433, 2004.

NIST. *The Economic Impacts of Inadequate Infrastructure for Software Testing*. 2002. `http://www.nist.gov/director/planning/upload/report02-3.pdf`. (National Institute of Standards and Technology).

OSTRAND, T. J.; BALCER, M. J. The category-partition method for specifying and generating fuctional tests. *Commun. ACM*, ACM, v. 31, p. 676–686, June 1988.

PLAT, N.; LARSEN, P. G. An overview of the ISO/VDM-SL standard. *SIGPLAN Not.*, ACM, v. 27, p. 76–82, August 1992.

ROSCOE, A. *Theory and Practice of Concurrency*. 1. ed. New Jersey: Prentice Hall, 1997.

SATPATHY, M. *et al.* Automatic testing from formal specifications. *TAP'07: Proceedings of the 1st international conference on tests and proofs*, Springer-Verlag, p. 95–113, 2007.

SATPATHY, M.; LEUSCHEL, M.; BUTLER, M. Protest: An automatic test environment for B specifications. *Electronic Notes in Theoretical Computer Science*, EMC Corporation, p. 111–113, 2005.

SCHNEIDER, S. *B Method, An Introduction*. 1. ed. Basingstoke: Palgrave, 2001.

SILVEIRA, D. S. *Animare: Um método de Validação de Processos de Negócio Através de Animação*. 2009. Ph.D Dissertation, COPPE, UFRJ.

SINGH, H. *et al.* Test case design based on Z and the classification-tree method. *First IEEE International Conference on Formal Engineering Methods*, IEEE Computer Society, p. 81–90, 1997.

SMITH, G. *The Object-Z Specification Language*. 1. ed. Berlin: Springer, 1999.

SOUZA, F. M. *Geração de Casos de Teste a partir de Especificações B*. 2009. Tese de Mestrado, DIMAp/UFRN.

SPIVEY, J. M. *The Z Notation: A Reference Manual*. 2. ed. New Jersey: Prentice-Hall, 1992.

TAI, K.; LEI, Y. A test generation strategy for pairwise testing. *Software Engineering, IEEE Transactions on*, v. 28, n. 1, p. 109 –111, 2002.

TOYN, I. Formal reasoning in the Z notation using CADiZ. *2nd International Workshop on User Interface Design for Theorem Proving Systems*, 1996.

UTTING, M.; PRETSCHNER, A.; LEGEARD, B. A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability*, John Wiley & Sons, Ltd., 2011.

WARMER, J.; KLEPPE, A. *The Object Constraint Language: Precise Modeling with UML*. 1st. ed. Toronto: Addison-Wesley, 1999.

XU, G.; YANG, Z. JMLAutoTest: A novel automated testing framework based on JML and JUnit. In: *Formal Approaches to Software Testing*. Berlin: Springer Berlin / Heidelberg, 2004, (Lecture Notes in Computer Science, v. 2931). p. 1103–1104.

Anexo A

Especificação resumida da máquina *Queue*, utilizada no estudo de caso apresentado no Capítulo 7. A especificação foi retirada de [GALVÃO, 2010].

Listagem 8.1: Especificação da máquina *Queue*

```
/**********************************
Basic layer of definitions for queue support in FreeRTOS.
This machine defines the concept of a queue in FreeRTOS with no
concern about the length of the queue and the size of the queue items.
Queues are modelled as sets.

Authors: Stephenson Galvao, David Deharbe
Universidade Federal do Rio Grande do Norte
Departamento de Informatica e Matematica Aplicada
Programa de Pos-Graduacao em Sistemas e Computacao
Formal Methods and Languages Research Laboratory
**********************************/

MACHINE Queue

SEES Types

VARIABLES
  /** Different queues in FreeRTOS */
  queues, /** Set of all active queues. Union of all message queues */
  queues_msg, /** Set of message queues managed by the system */
  queues_msg_full, /** Set of message queues that are full */
  queues_msg_empty, /** Set of message queues that are empty */

  queue_items, /** Set of items associated to the message queues */
  queue_receiving, /** Set of tasks blocked by reading */
  queue_sending, /** Set of tasks blocked by writing */

  /** First task that should be removed from the set of
  tasks blocked by writing */
```

```
31    first_sending,
32
33    /** First task that should be removed from the set of
34    tasks blocked by reading */
35    first_receiving
36
37  INVARIANT
38    /** Typing Clauses */
39
40    /** queue is a set of QUEUE elements */
41    queues : POW(QUEUE) &
42    /** queues_msg is a subset of queues */
43    queues_msg <: queues &
44    /** queues_msg_full is a subset of queues_msg */
45    queues_msg_full <: queues_msg &
46    /** queues_msg_empty is a subset of queues_msg */
47    queues_msg_empty <: queues_msg &
48    /** queue_items is a function mapping
49    a QUEUE to a set of ITEMs */
50    queue_items : QUEUE +-> POW(ITEM) &
51    /** queue_receiving is a function mapping
52    a QUEUE to a set of TASKs */
53    queue_receiving : QUEUE +-> POW(TASK) &
54    /** queue_sending is a function mapping
55    a QUEUE to a set of TASKs */
56    queue_sending : QUEUE +-> POW(TASK) &
57    /** first_receiving maps a QUEUE to the first TASK that should be
58    removed from that queue */
59    first_receiving : QUEUE +-> TASK &
60    /** first_sending maps a QUEUE to the first TASK that should be
61    removed from that queue */
62    first_sending : QUEUE +-> TASK &
63
64    /** Restriction Clauses */
65    queues = dom(queue_receiving) &
66    queues = dom(queue_sending) &
67    dom(first_receiving) = dom(queue_receiving) &
68    dom(first_sending) = dom(queue_sending)&
69
70    /** assert that firt_receiving belongs to queue_receiving and
71    first_sending belongs to queue_sending */
72    !(q1).(q1 : queues & q1 : dom(first_receiving) &
73           queue_receiving(q1) /= {}
```

```
74            => first_receiving(q1) : queue_receiving(q1)) &
75    !(q1).(q1 : queues & q1 : dom(first_sending) &
76           queue_sending(q1) /= {}
77            => first_sending(q1) : queue_sending(q1)) &
78
79    queues_msg_full /\ queues_msg_empty = {} &
80    queues_msg = dom(queue_items) &
81
82    /** Assert that no task is in more than one set of
83    blocked tasks */
84    !(q1,q2,tk).(q1 : queues & q2 : queues & tk : TASK &
85                 tk : queue_receiving(q1)
86                   => tk /: queue_sending(q2)) &
87    !(q1,q2,tk).(q1 : queues & q2 : queues & tk : TASK &
88                 tk : queue_sending(q2)
89                   => tk /: queue_receiving(q1)) &
90    !(q1,q2,tk).(q1 : queues & q2 : queues & q1 /= q2 &
91                 tk : TASK & tk : queue_receiving(q1)
92                   => tk /: queue_receiving(q2)) &
93    !(q1,q2,tk).(q1 : queues & q2 : queues & q1 /= q2 &
94                 tk : TASK & tk : queue_sending(q1)
95                   => tk /: queue_sending(q2))
96
97  CONSTANTS
98    remove_task
99
100 PROPERTIES
101   remove_task : ((QUEUE +-> POW(TASK)) * POW(TASK))
102                       --> (QUEUE +-> POW(TASK)) &
103   remove_task = %(q_task,unblocked).(q_task : QUEUE +-> POW(TASK) &
104                                      unblocked : POW(TASK) |
105                  %(q1).(q1 : QUEUE & q1 : dom(q_task) |
106                  q_task(q1)-unblocked))
107
108 INITIALISATION
109   queues := {} ||
110   queues_msg := {} ||
111   queue_items := {} ||
112   queue_receiving := {} ||
113   queue_sending := {} ||
114   queues_msg_full := {} ||
115   queues_msg_empty := {} ||
116   first_sending := {} ||
```

```
117    first_receiving := {}
118
119
120  OPERATIONS
121    /*******************************************************
122    Create Queue:
123    Create a new queue that will be managed by FreeRTOS.
124
125    Parameters:
126    uxQueueLength - Queue length - Not used in this level of abstraction
127    uxItemSize - the size of one item of queue - No used in this level of
128    abstraction
129    *******************************************************/
130
131    xQueueHandle <-- q_queueCreate(uxQueueLength, uxItemSize) =
132    PRE
133      uxQueueLength : QUEUE_LENGTH &
134      uxItemSize : NAT
135    THEN
136      ANY
137        pxQueue
138      WHERE
139        pxQueue : QUEUE &
140        pxQueue /: queues
141      THEN
142        queues := queues \/ {pxQueue} ||
143        queues_msg := queues_msg \/ {pxQueue} ||
144        queue_items := queue_items \/ {pxQueue |-> {}} ||
145        queue_receiving := queue_receiving \/ {pxQueue |-> {}} ||
146        queue_sending := queue_sending \/ {pxQueue |-> {}}||
147        queues_msg_empty := queues_msg_empty \/ {pxQueue}||
148        xQueueHandle := pxQueue ||
149        first_sending := first_sending \/ {pxQueue |-> TASK_NULL} ||
150        first_receiving := first_receiving \/ {pxQueue |-> TASK_NULL}
151      END
152    END;
153
154    /*******************************************************
155    Delete Queue:
156    Delete a Queue. This operation will delete
157    only Queues that are not mutex or semaphore (not shown here)
158
159    Parameters:
```

```
160   queue - Queue that will be deleted
161   ******************************************************/
162
163   q_queueDelete(pxQueue) =
164   PRE
165     pxQueue : queues &
166     pxQueue /: semaphores &
167     pxQueue /: mutexes &
168     queue_receiving(pxQueue) = {} &
169     queue_sending(pxQueue) = {}
170   THEN
171     queues := queues - {pxQueue} ||
172     queues_msg := queues_msg - {pxQueue} ||
173     queue_items := {pxQueue} <<| queue_items ||
174     queue_receiving := {pxQueue} <<| queue_receiving ||
175     queue_sending := {pxQueue} <<| queue_sending ||
176     first_sending := {pxQueue} <<| first_sending ||
177     first_receiving := {pxQueue} <<| first_receiving ||
178     IF pxQueue : queues_msg_full
179     THEN  queues_msg_full := queues_msg_full - {pxQueue}
180     END ||
181     IF pxQueue : queues_msg_empty
182     THEN queues_msg_empty := queues_msg_empty - {pxQueue}
183     END
184   END;
185
186   /******************************************************
187   Send Item:
188   Inserts one item in the queue and remove the task from the
189   set of tasks that are waiting to receive an item. The given
190   task shall be waiting on the given queue.
191
192   Parameters:
193   pxQueue - Queue that will send the item
194   pxItem - Item that will be sent to the queue
195   copy_position - Position that the item will be inserted
196   in the queue
197   ******************************************************/
198
199   q_sendItem(pxQueue, pxItem, copy_position) =
200   PRE
201     pxQueue : queues_msg &
202     pxItem : ITEM &
```

```
203       pxItem /: queue_items(pxQueue) &
204       copy_position : COPY_POSITION &
205       pxQueue /: queues_msg_full
206     THEN
207       queue_items(pxQueue) := queue_items(pxQueue) \/ {pxItem} ||
208       IF queue_receiving(pxQueue) /= {} THEN
209         ANY n_receiving, n_first
210         WHERE
211           n_receiving : POW(TASK) &
212           n_first : TASK &
213           n_receiving = (queue_receiving(pxQueue) -
214                          {first_receiving(pxQueue)}) &
215           n_first : n_receiving
216         THEN
217           queue_receiving(pxQueue) := n_receiving ||
218           first_receiving(pxQueue):= n_first
219         END
220       END ||
221       IF pxQueue:queues_msg_empty
222       THEN queues_msg_empty := queues_msg_empty-{pxQueue}
223       END ||
224       CHOICE
225         queues_msg_full:= queues_msg_full \/ {pxQueue}
226       OR
227         skip
228       END
229     END;
230
231     /*******************************************************
232     Received Item:
233     Remove (or not) one item from the queue passed as a parameter
234     and remove the task also passed as a parameter from the set of
235     tasks waiting to send.
236
237     Parameters:
238     pxQueue - Queue in wich the task will be inserted
239     justPeeking - Flag that indicates if the item will read and removed
240     or only read
241     pxTask - Task that have the item that will be read
242     *******************************************************/
243
244     xItem <-- q_receivedItem(pxQueue,justPeeking) =
245     PRE
```

```
246       pxQueue : queues_msg &
247       justPeeking : BOOL &
248       pxQueue /: queues_msg_empty
249     THEN
250       ANY
251         item
252       WHERE
253         item : ITEM &
254         item : queue_items(pxQueue)
255       THEN
256         IF justPeeking = FALSE THEN
257           IF queue_sending(pxQueue) /= {} THEN
258             ANY n_sending, n_first
259             WHERE
260               n_sending : POW(TASK) &
261               n_first : TASK &
262               n_sending = (queue_sending(pxQueue) -
263                            {first_sending(pxQueue)}) &
264               n_first : n_sending
265             THEN
266               queue_items(pxQueue) := queue_items(pxQueue) - {item} ||
267               queue_sending(pxQueue) := n_sending ||
268               first_sending(pxQueue):=n_first ||
269               IF pxQueue : queues_msg_full
270               THEN queues_msg_full := queues_msg_full - {pxQueue}
271               END ||
272               CHOICE
273                 queues_msg_empty := queues_msg_empty \/ {pxQueue}
274               OR
275                 skip
276               END
277             END
278           END
279         END ||
280         xItem:=item
281       END
282     END;
283
284     /*************************************************************
285     Insert Task in Wainting To Send:
286     Insert a task in the waiting to send queue
287
288     Parameters:
```

```
289   pxQueue: Queue in wich the task will be inserted
290   pxTask: Task to be inserted in the queue
291   ************************************************************/
292
293   q_insertTaskWaitingToSend(pxQueue, pxTask) =
294   PRE
295     pxQueue : queues &
296     pxTask : TASK &
297     !q1.(q1 : queues => pxTask /: queue_sending(q1) &
298                         pxTask /: queue_receiving(q1))
299   THEN
300     queue_sending(pxQueue) := queue_sending(pxQueue) \/ {pxTask} ||
301     IF queue_sending(pxQueue) = {}
302     THEN first_sending(pxQueue) := pxTask
303     END
304   END;
305
306   /************************************************************
307   Insert Task in Wainting To Receive:
308   Insert a task in the waiting to receive queue
309
310   Parameters:
311   pxQueue: Queue in wich the task will be inserted
312   pxTask: Task to be inserted in the queue
313   ************************************************************/
314
315   q_insertTaskWaitingToReceive(pxQueue, pxTask) =
316   PRE
317     pxQueue : queues &
318     pxTask : TASK &
319     !q1.(q1 : queues => pxTask /: queue_sending(q1) &
320                         pxTask /: queue_receiving(q1))
321   THEN
322     queue_receiving(pxQueue) := queue_receiving(pxQueue) \/ {pxTask} ||
323     IF queue_receiving(pxQueue) = {}
324     THEN first_receiving(pxQueue) := pxTask
325     END
326   END;
327
328   /************************************************************
329   Remove From Event List Queue:
330   Remove a task from all event sets (receiving, sending) in all
331   queues of FreeRTOS.
```

```
332
333   Parameters:
334   task - Task that will be removed
335   ************************************************************/
336
337   q_removeFromEventListQueue(task) =
338   PRE
339     task : TASK
340   THEN
341     ANY
342       pxQueue, n_receiving , n_first
343     WHERE
344       pxQueue : queues &
345       task : queue_receiving(pxQueue) &
346       n_receiving : POW(TASK) &
347       n_receiving = queue_receiving(pxQueue) -
348                     {first_receiving(pxQueue)} &
349       n_first : TASK &
350       n_first : queue_receiving(pxQueue)
351     THEN
352       queue_receiving(pxQueue) := n_receiving ||
353       first_receiving(pxQueue) := n_first
354     END ||
355     ANY
356       pxQueue, n_sending ,n_first
357     WHERE
358       pxQueue : queues &
359       n_sending : POW(TASK) &
360       n_first : TASK &
361       task : queue_sending(pxQueue) &
362       n_sending = queue_sending(pxQueue) - {first_sending(pxQueue)} &
363       n_first = first_sending(pxQueue)
364     THEN
365       queue_sending(pxQueue) := n_sending ||
366       first_sending(pxQueue) := n_first
367     END
368   END
369 END
```