João Batista de Souza Neto

Um estudo empírico sobre geração de testes com BETA: avaliação e aperfeiçoamento

Natal-RN

2015

João Batista de Souza Neto

Um estudo empírico sobre geração de testes com BETA: avaliação e aperfeiçoamento

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 Sistemas e Computação.

PPgSC – Programa de Pós-Graduação em Sistemas e Computação

DIMAp – Departamento de Informática e Matemática Aplicada

CCET – Centro de Ciências Exatas e da Terra

UFRN – Universidade Federal do Rio Grande do Norte

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

Natal-RN

2015

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

Souza Neto, João Batista de.
Um estudo empírico sobre geração de testes com BETA: avaliação e aperfeiçoamento / João Batista de Souza Neto. - Natal, 2015.
144 f.: il.

Orientadora: Profª. Drª. Anamaria Martins Moreira.

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

1. Teste de software – Dissertação. 2. Qualidade – Dissertação. 3. Métodos formais – Dissertação. 4. Método B – Dissertação. 5. Teste de unidade – Dissertação. 6. Avaliação – Dissertação. I. Moreira, Anamaria Martins. II. Título.

RN/UF/BSE-CCET CDU: 004.415.53

João Batista de Souza Neto

Um estudo empírico sobre geração de testes com BETA: avaliação e aperfeiçoamento

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 Sistemas e Computação.

Esta dissertação foi avaliada e considerada aprovada pelo Programa de Pós-Graduação em Sistemas e Computação da Universidade Federal do Rio Grande do Norte.

Profª. Drª. Anamaria Martins Moreira – Orientadora
Universidade Federal do Rio de Janeiro

Prof. Dr. David Boris Paul Déharbe
Universidade Federal do Rio Grande do Norte

Prof. Dr. Rohit Gheyi
Universidade Federal de Campina Grande

Natal-RN
2015

Aos meus pais, Junior e Maria, e a minha irmã, Carol.

Agradecimentos

Primeiramente agradeço a minha família, em especial meus pais e minha irmã, que sempre me apoiaram e incentivaram nos momentos mais importantes de minha vida.

Agradeço a minha orientadora, Anamaria Martins Moreira, por todos os ensinamentos e pela ajuda, paciência e dedicação na orientação deste trabalho. Sem todo seu apoio e confiança este trabalho não seria possível.

Agradeço ao colega de pesquisa Ernesto Cid pela ajuda, contribuição e trabalho em conjunto. Seu apoio também foi fundamental para que este trabalho pudesse ser realizado.

Agradeço aos professores e colegas do grupo de pesquisa *Forall*, David Déharbe, Martin Musicante, Valério Medeiros, Cleverton Hentz, Paulo Ewerton e Márcio Alves, pelas contribuições e trabalhos em grupo.

Agradeço ao pessoal do laboratório *LabLua* da *PUC-Rio*, Roberto Ierusalimschy, Ana Lúcia, Hisham Muhammad e demais integrantes, pelas contribuições e por terem me acolhido de forma amigável no Rio de Janeiro.

Agradeço a todos os colegas de pós-graduação pela ajuda e momentos de descontração.

Agradeço a todos os professores e funcionários do *DIMAp*.

Agradeço a todos os amigos que me apoiaram e proporcionaram momentos alegres, essenciais em qualquer jornada.

"Não é preciso ter olhos abertos para ver o sol, nem é preciso ter ouvidos afiados para ouvir o trovão. Para ser vitorioso você precisa ver o que não está visível"

Sun Tzu

Resumo

A demanda de sistemas seguros e robustos fez com que crescesse a preocupação em desenvolver software de qualidade. Teste de Software e Métodos Formais são duas abordagens que possuem essa finalidade. Neste contexto, vários esforços vem sendo feitos para unir essas duas abordagens, que podem se complementar e trazer mais qualidade para o software. Em um desses esforços, foi desenvolvida a abordagem e ferramenta BETA (*B Based Testing Approach*). BETA gera testes de unidade a partir de especificações formais escritas na notação do Método B. O presente trabalho tem o objetivo de contribuir com a evolução e o aperfeiçoamento da abordagem e ferramenta BETA. Com essa finalidade, duas linhas de ação foram tomadas. Na primeira linha de ação, este trabalho trouxe contribuições para as duas últimas etapas da abordagem BETA. Para a penúltima etapa, a de definição dos oráculos de teste, este trabalho propôs *estratégias de oráculos de teste* que trouxeram mais flexibilidade e uma melhor definição para a etapa. Para a última etapa, a de implementação dos testes concretos, este trabalho desenvolveu um *gerador de scripts de teste* que automatizou parte da implementação e, assim, contribuiu para reduzir o esforço gasto na etapa. Na segunda linha de ação, este trabalho realizou um estudo empírico para avaliar a abordagem e ferramenta. Dessa forma, BETA foi aplicada em dois estudos de caso que possuíam diferentes objetivos e complexidades. No primeiro estudo de caso, BETA foi utilizada para gerar testes para a API da linguagem de programação Lua. No segundo estudo de caso, BETA foi utilizada para contribuir com a validação de dois geradores de código para o Método B, o *b2llvm* e o *C4B*. Em ambos os estudos de caso, os resultados de BETA foram avaliados quantitativamente e qualitativamente. Como resultado, este trabalho conseguiu identificar qualidades e limitações de BETA e, com isso, estabelecer uma base para propor e implementar melhorias para a abordagem e ferramenta.

Palavras-chaves: Qualidade; Teste de Software; Métodos Formais; Método B; Teste de Unidade; Avaliação.

Abstract

The demand for secure and robust systems has made to grow the concern for developing quality software. *Software Testing* and *Formal Methods* are two approaches that have this purpose. In this context, several efforts are being made to unite these two approaches, which can complement each other and bring more quality to the software. In one of these efforts, the approach and tool *BETA* (*B Based Testing Approach*) was developed. BETA generates unit tests from formal specifications written in *B Method* notation. The present work aims to contribute with the evolution and improvement of the approach and tool BETA. For this purpose, two lines of action were taken. In the first line of action, this work brought contributions for the last two stages of the BETA approach. For the penultimate stage, the test oracles definition, this work proposed *test oracles strategies* that brought more flexibility and a better definition for the stage. For the last stage, the concrete tests implementation, this work developed a *test scripts generator* that automates part of the implementation and, thus, contribute to reduce the effort spent in the stage. In the second line of action, this work performed an empirical study to evaluate the approach and tool. Thus, BETA was applied in two case studies that had different objectives and complexities. In the first case study, BETA was used to generate tests for the API of the Lua programming language. In the second case study, BETA was used to contribute to the validation of two code generators for the B Method, the *b2llvm* and *C4B*. In both case studies, the results of BETA were evaluated quantitatively and qualitatively. With this study, this work was able to identify qualities and limitations of BETA and, with that, establish a basis for proposing improvements to the approach and tool.

Keywords: Quality; Software Testing; Formal Methods; B Method; Unit Test; Evaluation.

Lista de ilustrações

Lista de tabelas

Sumário

1 Introdução

O software se tornou indispensável para a vida moderna, estando presente em todos os segmentos da sociedade. Setores da indústria e serviços possuem demandas crescentes por software, uma vez que é impraticável não utilizar algum software em um cenário em que a competição entre as empresas só aumenta. Esses setores exigem sistemas robustos e seguros, uma vez que falhas em um software podem trazer grandes prejuízos sociais e econômicos, que por muitas vezes são irremediáveis. Por esse motivo, desenvolver software de qualidade tem sido uma das maiores preocupações na indústria do software. Dentro da *Engenharia de Software* existem várias abordagens que visam a qualidade. Este trabalho aborda duas das principais técnicas utilizadas pelo mercado para se desenvolver software de qualidade, *Métodos Formais* e *Teste de Software*.

Métodos formais são utilizados para especificar, desenvolver e verificar software através da aplicação de formalismos matemáticos (ABRAN et al., 2004). O rigor matemático das notações e linguagens formais permite que os requisitos de um software possam ser descritos de uma forma completa, consistente e não-ambígua, garantindo uma maior segurança com relação ao desenvolvimento. Entre alguns dos principais métodos e notações formais, se destacam: a *Notação Z* (SPIVEY, 1989), *Alloy* (JACKSON, 2012), *Event-B* (ABRIAL, 2010) e o *Método B* (ABRIAL, 2005). Este último, é o método formal abordado neste trabalho.

Em geral, a aplicação de métodos formais em um projeto possui um alto custo, fazendo com que esta abordagem seja utilizada apenas em projetos de sistemas de alta integridade ou no desenvolvimento de funcionalidades críticas. Em contrapartida, teste de software é empregado em, basicamente, todos os projetos de desenvolvimento de software, sendo a abordagem de verificação e validação mais utilizada no mercado. Teste de software consiste na verificação dinâmica de que um programa se comporta da maneira esperada (ABRAN et al., 2004). Apesar de sua efetividade, testes podem apenas identificar a presença de problemas no software, mas nunca a ausência (DIJKSTRA, 1979).

Apesar de prover provas matemáticas de que um software está correto, métodos formais também não conseguem eliminar todos os problemas, pois o software é dependente de agentes externos que, em geral, não têm sua correção provada. Problemas relacionados a compilação, sistema operacional ou hardware, por exemplo, não podem ser evitados apenas com o uso de métodos formais, necessitando de testes que, apesar de não garantir a ausência, são mais efetivos em encontrar problemas dessa magnitude. Ambas as abordagens são efetivas nos seus respectivos propósitos, abrangendo diferentes aspectos no desenvolvimento de software. As duas abordagens não competem entre si e podem,

inclusive, se complementar (TANENBAUM, 1976).

Utilizar métodos formais e teste de software em conjunto não só traz benefícios para a qualidade do software, mas também pode reduzir os custos do desenvolvimento, através da aplicação de técnicas de teste nas fases iniciais do desenvolvimento, enquanto os defeitos ainda são relativamente baratos de serem corrigidos, e de uma maior automação no processo de teste, com a geração de testes a partir de especificações formais (HIERONS et al., 2009). Derivar testes a partir de especificações formais também é uma maneira conveniente de verificar de forma independente a corretude de uma implementação (CARRINGTON; STOCKS, 1994), pois, dado o alto custo de uma verificação formal, o processo de desenvolvimento formal nem sempre é totalmente seguido e, muitas vezes, fica limitado a fase inicial do desenvolvimento, envolvendo apenas a especificação do sistema.

Nesse contexto, existem várias iniciativas que buscam unir métodos formais e teste de software. Entre os vários trabalhos existentes na área, se destacam: (LEGEARD; PEUREUX; UTTING, 2002), que apresenta a ferramenta *BZ-TT* (*BZ-Testing-Tools*), que gera testes a partir de especificações escritas na notação Z ou B; (SATPATHY; LEUSCHEL; BUTLER, 2005), que apresenta o *Protest*, um ambiente de teste automático para especificações em B; (AYDAL et al., 2009), que apresenta uma abordagem de teste a partir de especificações em Alloy; e (MALIK; LILIUS; LAIBINIS, 2009a), que apresenta uma abordagem para a criação de testes a partir de modelos formais feitos em Event-B. Estas são apenas algumas das várias iniciativas que unem métodos formais e teste de software, mostrando que esta é uma área de estudo bastante ativa.

Em um trabalho do grupo de pesquisa *ForAll* (*Formal Methods and Languages Research Laboratory*)[1], iniciado em (SOUZA, 2009) e aprimorado em (MATOS; MOREIRA, 2012) e (MATOS; MOREIRA, 2013), foi desenvolvida a abordagem e ferramenta *BETA* (*B Based Testing Approach*). Fazendo uso de particionamento do espaço de entrada (AMMANN; OFFUTT, 2010), BETA gera testes de unidade a partir de especificações formais escritas na notação do Método B. Com a abordagem, é possível gerar testes positivos, que exercitam situações esperadas de acordo com a especificação, e negativos, que exercitam situações não esperadas de acordo com a especificação, permitindo avaliar a robustez e segurança de um software e ajudando a detectar os pontos fracos que podem ser explorados por agentes mal intencionados.

De forma sucinta, a abordagem consiste em particionar (ou dividir) o domínio de entrada de uma operação (unidade) de uma especificação B em blocos (ou subdomínios); combinar os diferentes blocos para gerar casos de teste (situações a serem testadas); obter

[1] *ForAll* é o grupo de pesquisa de Métodos Formais e Linguagens do Departamento de Informática e Matemática Aplicada (DIMAp) da UFRN, informações sobre o grupo podem ser encontradas em <http://forall.ufrn.br>.

dados de entrada que satisfazem as restrições dos casos de teste; definir os oráculos de teste (mecanismos que determinam a falha ou sucesso de um teste); e, por último, implementar os testes concretos. As etapas de particionamento do espaço de entrada, geração de casos de teste e obtenção dos dados de teste são automatizadas pela ferramenta BETA, que gera como saída uma especificação de casos de teste. Para a etapa de definição dos oráculos de teste, a abordagem propõe o uso de uma ferramenta de animação que simula a execução da operação sob teste para se obter os resultados esperados no teste. O método proposto para obtenção dos oráculos não é aplicável em casos de teste negativos, uma vez que não é possível determinar um comportamento específico nessas situações a partir da especificação, ficando a cargo do engenheiro de testes determinar o comportamento esperado segundo os seus critérios. A última etapa da abordagem é a concretização de todo o processo com a implementação de testes executáveis.

Nos primeiros trabalhos, a abordagem e ferramenta BETA foi testada em dois estudos de caso. No primeiro estudo de caso, realizado em (MATOS et al., 2010), a abordagem, proposta em (SOUZA, 2009), foi utilizada para projetar casos de teste para um sistema de controle de portas de metrô a partir da especificação criada em (BARBOSA, 2010). Esse primeiro estudo mostrou a viabilidade da abordagem, assim como levantou a necessidade de melhorias e automação, uma vez que, até então, era feita de forma manual. Com esse resultado, os trabalhos seguintes, (MATOS; MOREIRA, 2012) e (MATOS; MOREIRA, 2013), foram focados no aperfeiçoamento da abordagem e no desenvolvimento de uma ferramenta que automatizasse parte do processo. Para validar a abordagem revisada e a ferramenta desenvolvida, foi realizado um segundo estudo de caso. Nesse estudo, BETA foi utilizada para projetar casos de teste para o *FreeRTOS* (FREERTOS, 2010), um *kernel* para sistemas de tempo real simples, a partir da especificação apresentada em (GALVÃO, 2010).

Os dois estudos de caso realizados foram importantes para uma validação inicial da abordagem e ferramenta. Entretanto, ficaram limitados apenas ao projeto de casos de teste, não tendo sido realizadas as etapas finais do processo de teste, como a implementação, execução e análise dos resultados. Com isso, os estudos foram focados em mostrar a aplicabilidade de BETA, mas forneceram poucas informações com relação a qualidade dos testes gerados.

Nesse cenário, o presente trabalho teve o objetivo de contribuir para a evolução e aperfeiçoamento da abordagem e ferramenta BETA. Com essa finalidade, duas linhas de ação foram tomadas. Na primeira linha de ação, este trabalho trouxe contribuições para as duas últimas etapas de BETA. Para a etapa de definição dos oráculos de teste a abordagem propunha um método para a obtenção dos resultados esperados. Entretanto, o método proposto não é aplicável em casos de teste negativos, não oferece flexibilidade e não define bem os elementos que devem ser verificados pelos oráculos. Então, para

reduzir essas limitação, este trabalho propôs estratégias de oráculos de teste baseadas nas estratégias apresentadas em (LI; OFFUTT, 2014). As estratégias propostas oferecem mais flexibilidade para a etapa de definição dos oráculos de teste de BETA e podem ser aplicadas em casos de teste negativos. Na última etapa de BETA os testes concretos são implementados de forma manual, algo que pode demandar muito tempo e esforço. Para reduzir esse esforço, este trabalho desenvolveu um gerador de *scripts* de teste que automatiza parte do processo de implementação. O gerador traduz a especificação de casos de teste gerada por BETA em um *script* de teste parcial. O *script* deve ser adaptado para adequar o refinamento dos dados de teste gerados por BETA e inserir os valores esperados nos oráculos de teste para, então, poder ser executado.

Na segunda linha de ação, o presente trabalho realizou um estudo empírico de BETA com o intuito de avaliar a abordagem e ferramenta de uma forma mais completa, realizando todo o processo de teste, algo que ainda não havia sido feito. Dessa forma, BETA foi aplicada em dois estudos de caso. No primeiro estudo de caso, BETA foi utilizada para gerar casos de teste para a API da linguagem de programação Lua (IERUSALIMSCHY, 2013) a partir do modelo em B desenvolvido em (MOREIRA; IERUSALIMSCHY, 2013). No segundo estudo de caso, BETA foi utilizada para contribuir com a validação de dois geradores de código para o Método B, o *b2llvm* (DÉHARBE; MEDEIROS JR., 2013) e o *C4B*[2]. Em ambos os estudos de caso, todo o processo de teste foi realizado, ou seja, do projeto dos casos de teste até a execução dos testes, e os testes foram avaliados através de análise de cobertura de código (MILLER; MALONEY, 1963) e teste de mutação (YOUNG; PEZZE, 2005). Os testes gerados por BETA conseguiram identificar erros nos dois estudos de caso, mostrando que a abordagem e ferramenta BETA contribui para o processo de verificação e validação. Além disso, foram identificadas limitações e pontos a serem melhorados em BETA, que serão explorados futuramente.

1.1 Objetivos

De forma geral, o objetivo deste trabalho é contribuir com a evolução e aperfeiçoamento da abordagem e ferramenta BETA. Em uma primeira linha de ação, o objetivo deste trabalho é propor e implementar melhorias para as duas últimas etapas da abordagem BETA. Com o intuito de diminuir as limitações e trazer mais flexibilidade para a etapa de definição dos oráculos de teste, este trabalho propôs estratégias de projeto de oráculos de teste inspiradas nas estratégias apresentadas em (LI; OFFUTT, 2014). Para reduzir o esforço gasto na última etapa da abordagem BETA, este trabalho desenvolveu um gerador de *scripts* de teste que automatiza parte do processo de implementação dos testes concretos.

[2] Gerador de código C a partir de implementações B distribuído e integrado com a IDE *Atelier B* disponível em <http://www.atelierb.eu>.

Em uma segunda linha de ação, o objetivo deste trabalho é realizar um estudo empírico de BETA, possuindo um caráter exploratório, visando exercitar BETA em diferentes situações, além de situações não exploradas em trabalhos anteriores, e avaliativo, com o objetivo de identificar qualidades e limitações na abordagem e ferramenta. Desta forma, este trabalho aplicou a abordagem e ferramenta BETA em dois estudos de caso. No primeiro estudo de caso, BETA foi utilizada para gerar testes para a API da linguagem de programação Lua a partir do modelo parcial em B apresentado em (MOREIRA; IERUSALIMSCHY, 2013). O modelo apresentava uma estruturação complexa, definia dados e estados compostos e possuía um número elevado de operações, apresentando situações que não foram exploradas em trabalhos anteriores.

No segundo estudo de caso, a abordagem e ferramenta BETA foi utilizada para contribuir com a validação de dois geradores de código para o Método B, o *b2llvm* e o *C4B*. Neste estudo, um conjunto de módulos B foi aplicado em ambos os geradores, e os códigos gerados pelo *b2llvm* e *C4B* foram verificados através de testes gerados por BETA. Em ambos os estudos de caso, os resultados de BETA foram avaliados de forma quantitativa e qualitativa, fazendo uso de métricas como a análise de cobertura de comandos e ramificações, e teste de mutação.

1.2 Organização do trabalho

Este trabalho está organizado da seguinte forma: no Capítulo 2 são apresentados conceitos relacionados a Teste de Software e é feita uma breve introdução ao Método B, fornecendo uma base para o entendimento deste trabalho. Os trabalhos relacionados são discutidos ao longo dos Capítulos 3, 4 e 5. No Capítulo 3, a abordagem e ferramenta BETA é apresentada. Neste Capítulo, cada etapa da abordagem é apresentada em detalhes e o funcionamento da ferramenta é explicado. No Capítulo 4, são apresentadas as contribuições para a abordagem e ferramenta que foram desenvolvidas neste trabalho. Primeiro, são apresentadas as estratégias de oráculos de teste que foram propostas para diminuir limitações e flexibilizar a etapa de definição dos oráculos de teste da abordagem. Em seguida, é apresentado o gerador de *scripts* de teste desenvolvido com a finalidade de automatizar parte do processo de implementação dos testes concretos. Os dois estudos de caso realizados são apresentados no Capítulo 5. Para ambos, inicialmente é apresentado o seu contexto e motivação. Em seguida, são relatados os processos de geração de testes com BETA e implementação dos testes concretos. Por último, os resultados dos testes realizados são apresentados e discutidos. Finalizando o trabalho, as considerações finais sobre os resultados obtidos, assim como, diretivas para trabalhos futuros são apresentadas no Capítulo 6.

2 Fundamentação Teórica

Neste capítulo é feita uma introdução aos conceitos necessários para o entendimento deste trabalho. Inicialmente, são apresentados alguns conceitos importantes relacionados a *Teste de Software*, como as principais terminologias, atividades do processo de teste, critério de cobertura e oráculo de teste. Em seguida, é feita uma breve introdução ao *Método B*, de forma exemplificada são apresentadas as suas principais características e conceitos, como máquina abstrata, substituições generalizadas, refinamento e implementação.

2.1 Teste de Software

Teste é o processo de executar um software com o intuito de verificar se o software se comporta da maneira esperada. Em (MYERS et al., 2004), teste é definido como o processo de executar um software com a finalidade de encontrar erros. Com estas definições, é possível ver que o teste de software é uma atividade importante do processo de desenvolvimento e que está diretamente ligada à qualidade do software. O teste é o principal mecanismo de validação e verificação utilizado pela indústria e uma importante área de estudo dentro da Engenharia de Software. O teste de software é uma grande área e envolve uma série de conceitos. Nas seções a seguir, serão apresentados os principais conceitos e definições relacionadas a teste de software que são relevantes para este trabalho.

2.1.1 Terminologia

A seguir será apresentada uma série de termos que são importantes no contexto de teste de software, em geral, e neste trabalho. Existem diferentes terminologias relacionadas a teste de software na literatura, neste trabalho são adotados os termos apresentados em (AMMANN; OFFUTT, 2010). Dois termos importantes e que costumam ser confundidos são validação e verificação.

Definição 2.1.1 (Validação) *É o processo de avaliar o software no final do seu desenvolvimento para garantir que ele atende ao que foi requisitado.*

Definição 2.1.2 (Verificação) *É o processo de determinar se os resultados de uma determinada fase do processo de desenvolvimento de software atendem ao que foi planejado na fase anterior.*

A verificação é, geralmente, um processo mais técnico que requer conhecimentos sobre os artefatos do software, requisitos e especificações. Já a validação, requer conhecimentos sobre o domínio da aplicação para o qual o software foi desenvolvido. Outra distinção que precisa ser feita é entre os termos defeito, erro e falha, muito utilizados em teste de software e que são constantemente confundidos.

Definição 2.1.3 (Defeito) *Um defeito estático no software que pode resultar em um problema durante a sua execução.*

Definição 2.1.4 (Erro) *Um estado interno incorreto que é manifestado por um defeito no software.*

Definição 2.1.5 (Falha) *É a externalização do erro. Um comportamento incorreto do software com relação ao que foi requisitado ou o que é esperado.*

As definições acima são importantes porque ajudam a distinguir teste de depuração (*debugging*).

Definição 2.1.6 (Teste) *É o processo de avaliar um software observando a sua execução.*

Definição 2.1.7 (Falha de Teste) *É a execução que resulta em uma falha, ou seja, um comportamento inesperado.*

Definição 2.1.8 (Depuração (*Debugging*)) *É o processo de encontrar um defeito no software dada uma falha na execução.*

Mesmo com a existência de defeitos, nem toda a execução do software resultará em uma falha. Para que uma falha possa ser observada são necessárias três condições: primeiro, o código (local) que possui o defeito deve ser executado; segundo, é necessário que a execução do defeito ocasione um erro, infectando o estado do software; terceiro, é necessário que o estado infectado resulte em alguma falha do software. O conjuntos das três condições necessária para que um defeito ocasione uma falha é conhecido como Modelo "RIP" (AMMANN; OFFUTT, 2010). Este modelo é muito importante para análises estáticas do software e é muito utilizado em *Teste de Mutação*, que será apresentado mais a frente.

Definição 2.1.9 (Alcançabilidade (*Reachability*)) *O local ou locais que contém defeitos no programa devem ser alcançados.*

Definição 2.1.10 (Infecção (*Infection*)) *Após a execução do local que possui um defeito, o estado do programa deve ser incorreto.*

Definição 2.1.11 (Propagação (*Propagation*)) *O estado infectado deve causar alguma saída incorreta ou o estado final do programa deve ser incorreto.*

Um importante conceito é o de *caso de teste*, que, de forma geral, representa uma situação a ser testada no software. Um caso de teste inclui as condições necessárias para a execução do teste, os seus valores de entrada e os seus resultados esperados.

Definição 2.1.12 (Valores de Caso de Teste) *São os valores de entrada necessários para completar alguma execução do software sob teste.*

Definição 2.1.13 (Resultados esperados) *É o resultado que será produzido durante a execução do teste se, e somente se, o programa satisfaz seu comportamento pretendido.*

Definição 2.1.14 (Caso de Teste) *É um conjunto de artefatos que são necessários para uma completa execução e avaliação do software sob teste em uma determinada situação. Esses artefatos incluem os valores de caso de teste e resultados esperados, além das condições necessárias para que o software possa ser executado na situação a ser testada.*

O projeto de casos de teste não é uma tarefa trivial e, em várias situações, esbarra em problemas relacionados a como fornecer os valores de entrada para testar o software e como observar detalhadamente o seu comportamento.

Definição 2.1.15 (Observabilidade) *O quão fácil é observar o comportamento de um programa em termos de suas saídas, efeitos nos ambiente e em outros componentes de hardware ou software.*

Definição 2.1.16 (Controlabilidade) *O quão fácil é fornecer ao programa as entradas necessárias, em termos de valores, operações e comportamentos.*

A observabilidade e a controlabilidade são importantes para determinar o quão testável é um software. Outros termos que são bastante utilizados neste trabalho são *teste positivo* e *teste negativo*. Os testes positivos são aqueles que utilizam dados que respeitam as restrições impostas pela especificação. Já os testes negativos, são aqueles que utilizam dados que desrespeitam as restrições impostas pela especificação. Testes negativos ajudam a garantir que o software consegue tratar situações inesperadas, como um usuário fornecer um dado inválido por exemplo, ajudando a melhorar a qualidade do software e a encontrar seus pontos fracos.

Outras duas terminologias importantes no testes de software, que dizem respeito ao tipo do teste, são *teste funcional*, classicamente chamado de teste de caixa preta, e *teste estrutural*, classicamente chamado de teste de caixa branca.

Definição 2.1.17 (Teste Funcional) *Neste tipo de teste, os testes são derivados a partir de uma descrição externa do software, como a especificação ou requisitos, por exemplo.*

Definição 2.1.18 (Teste Estrutural) *Neste tipo de teste, os testes são derivados a partir do código fonte interno do software.*

Testes podem ser obtidos a partir de diversos artefatos, como requisitos e especificações, artefatos de projeto e código fonte. Cada artefato é relacionada a alguma atividade de desenvolvimento de software. Cada atividade do desenvolvimento é relacionada a um nível de teste, descritos a seguir:

Teste de Aceitação: tem o objetivo de verificar se o software atende ao que é esperado pelo usuário;

Teste de Sistema: tem o objetivo de verificar se o sistema desenvolvido atende ao que foi especificado;

Teste de Integração: tem o objetivo de verificar se as interfaces dos diferentes módulos do software se comunicam corretamente;

Teste de Módulo: tem o objetivo de analisar cada módulo do software de forma isolada, verificando a interação entre seus componentes (unidades);

Teste de Unidade: tem o objetivo de verificar cada unidade do software produzida na fase de implementação, ou seja, a menor parte testável do software. Uma unidade pode ser considerada uma função ou um método, por exemplo, e, geralmente, é de responsabilidade do desenvolvedor.

Em algumas literaturas não existe a distinção entre teste de módulo e teste de unidade. Este último, é o nível de teste que é abordado neste trabalho. Além desses níveis, também existe o *Teste de Regressão*, que é o teste que é realizado após uma manutenção ou atualização do software, que tem o objetivo de verificar se o software atualizado ainda possui as funcionalidades que tinha antes de ser modificado.

2.1.2 Atividades de Teste

O teste de software envolve várias atividades que, geralmente, são realizadas pelo engenheiro de teste, um profissional da *Tecnologia da Informação* (TI) que é encarregado

por uma ou mais atividades de teste. De maneira geral, o processo de teste de software pode ser dividido em quatro grandes tipos de atividades: *Projeto de Teste*, *Automação de Teste*, *Execução de Teste* e *Avaliação de Teste*. Cada tipo de atividade requer diferentes níveis de habilidades e conhecimentos. Cada um dos tipos de atividade são descritos a seguir:

1. *Projeto de Teste*: é o processo de projetar casos de teste que vão testar o software de maneira efetiva. Os testes podem ser projetados a partir de critérios de cobertura ou a partir de conhecimentos específicos sobre o domínio do sistema. Critérios de cobertura, que serão apresentados mais a frente, fornecem uma maneira sistemática de projetar testes com um nível satisfatório de qualidade;

2. *Automação de Teste*: é o processo de embutir os casos de teste (valores de entrada, resultados esperados, etc.) em *scripts* executáveis;

3. *Execução de Teste*: é o processo de executar os scripts de teste no software e registrar os seus resultados;

4. *Avaliação de Teste*: é o processo de avaliar os resultados dos testes e reportá-los para a equipe de desenvolvimento.

2.1.3 Critérios de Cobertura

Em um cenário ideal, o software deveria ser testado com todas as suas possíveis entradas, algo que é conhecido como *teste exaustivo*. Entretanto, uma vez que o número de possíveis entradas pode ser muito grande, possivelmente infinito, se torna inviável realizar um teste exaustivo. Por causa disso, é necessário projetar um conjunto de testes, não exaustivo, que consiga testar o software de uma maneira efetiva e viável. *Critérios de cobertura* são alguns dos mecanismos que ajudam a projetar conjuntos de testes mais efetivos e viáveis. Critérios de cobertura são definidos em termos de *requisitos de teste*:

Definição 2.1.19 (Requisito de teste) *Um requisito de teste é um elemento específico de um artefato de software que deve ser satisfeito ou coberto por algum caso de teste.*

Os requisitos de teste podem ser obtidos a partir de uma variedade de artefatos do software, como o código fonte, componentes do projeto, modelos da especificação e descrições do espaço de entrada, entre outros. Um critério de cobertura é apenas uma forma de se obter requisitos de teste de uma maneira sistemática:

Definição 2.1.20 (Critério de cobertura) *Um critério de cobertura é uma regra ou conjunto de regras que estabelecem requisitos de teste para um conjunto de testes.*

Para o engenheiro de testes, é importante saber a qualidade do seu conjunto de testes. É possível medir a qualidade de um conjunto de teste através de sua cobertura com relação a algum critério de cobertura.

Definição 2.1.21 (Cobertura) *Dado um conjunto de requisitos de teste TR para um critério de cobertura C, um conjunto de testes T satisfaz C se e somente se para cada requisito de teste tr pertencente a TR, existe pelo menos um teste t pertencente a T que satisfaz tr.*

Definição 2.1.22 (Nível de cobertura) *Dado um conjunto de requisitos de teste TR e um conjunto de testes T, o nível de cobertura é simplesmente a relação entre o número de requisitos satisfeitos por T com o número total de requisitos em TR.*

Uma vez que é possível obter requisitos de teste a partir de diferentes artefatos do software, existem vários tipos de critérios de cobertura que estabelecem requisitos de diferentes maneiras e a partir de diferentes artefatos. Em (AMMANN; OFFUTT, 2010), são apresentados os principais tipos de critério de cobertura: *cobertura de grafos*, *cobertura lógica*, *particionamento do espaço de entrada* e *testes baseados em sintaxe*. O tipo de critério utilizado por BETA para gerar os casos de teste é o de particionamento do espaço de entrada. Neste trabalho, também foram utilizados os critérios de *cobertura de código* e *teste de mutação* para avaliar o conjunto de testes gerado por BETA nos estudos de caso. Os critérios são apresentados a seguir.

2.1.3.1 Particionamento do Espaço de Entrada

Neste tipo de critério de cobertura, requisitos de teste são criados a partir da divisão do *domínio de entrada* do programa sob teste em subdomínios. O domínio de entrada de um programa contém todos os valores que os parâmetros de entrada podem assumir. Os parâmetros de entrada podem ser os parâmetros de um método ou função, variáveis globais, entradas de um programa a nível de usuário, como um campo em um formulário por exemplo, entre outros, que dependem do tipo de artefato do software que está sendo analisado. O particionamento do espaço de entrada consiste em dividir o domínio de entrada em regiões, chamadas de *classes de equivalência* ou *blocos*, e selecionar ao menos um valor de cada região para testes.

O domínio é dividido com base em *características* do programa. Cada característica permite ao testador definir uma partição, que deve seguir duas propriedades:

1. A partição deve cobrir todo o domínio (completude);
2. Os blocos da partição não podem se sobrepor (disjuntos).

Um exemplo de característica é a que um determinado campo X de um formulário deve ter ao menos 6 letras. Com essa característica, uma das possíveis formas de dividir o domínio de entrada seria criar um bloco contendo os valores com 6 letras ou menos, e um bloco contendo os valores com mais de 6 letras, dividindo o domínio de entrada em dois blocos completos e disjuntos. Após a divisão do domínio de entrada, o passo seguinte é combinar valores de cada bloco das diferentes características para criar os casos de teste.

O processo de particionar o domínio de entrada em blocos e selecionar valores de cada bloco é chamado de *modelagem do domínio de entrada* (AMMANN; OFFUTT, 2010). Estre processo é dividido em cinco passos:

Passo 1. **Identificar funções testáveis:** Consiste em identificar as funcionalidades a serem testadas, como um método individual, uma classe com vários métodos ou um programa;

Passo 2. **Identificar os parâmetros de entrada:** Consiste em identificar todos os parâmetros do domínio de entrada que influenciam o comportamento da funcionalidade a ser testada. No caso de um método, por exemplo, seus parâmetros são os parâmetros de entrada do método e todas as variáveis que são utilizadas;

Passo 3. **Modelar o domínio de entrada:** Consiste em identificar as características dos parâmetros de entrada, e particionar o domínio em blocos com base nessas características. Existem várias estratégias para dividir o domínio em blocos, algumas podem ser vistas abaixo:

- **Valores válidos**: incluir ao menos um grupo com valores válidos.
- **Valores inválidos**: incluir ao menos um grupo com valores inválidos.
- **Valores limites**: incluir grupos com valores próximos e dentro de limites.
- **Valores especiais**: incluir grupos com valores especiais ou menos comuns, como valores nulos, por exemplo.

Passo 4. **Selecionar combinações de valores:** Uma vez que as características e seus blocos foram definidos, o passo seguinte é selecionar valores de teste. Como é possível ter várias características e blocos, os valores dos blocos, das diferentes características, devem ser combinados para gerar os valores de entrada para cada teste. Existem critérios que podem auxiliar na combinação de valores dos diferentes blocos, em (AMMANN; OFFUTT, 2010) são apresentados alguns:

All Combinations: Todas as combinações dos blocos de todas as características devem ser utilizadas.

O exemplo do campo X de um formulário apresentado anteriormente será estendido para exemplificar os critérios de combinação. A característica de que o campo tem

que ter ao menos 6 letras, já apresentada anteriormente, será chamada de C_1, o bloco contendo os valores com 6 letras ou menos será chamados de b_{11}, e o bloco contendo os valores com mais de 6 letras será chamado de b_{12}. Outras duas características serão criadas para que possa ser feita a combinação com C_1, a característica C_2 que determina que o campo X pode ter no máximo 12 letras, que dividirá o domínio em dois blocos, o bloco b_{21} com os valores com 12 letras ou menos e o bloco b_{22} com os valores com mais de 12 letras; e a característica C_3 que determina que o campo X não pode ser vazio, que dividirá o domínio em dois blocos, o bloco b_{31} contendo os valores vazios e o bloco b_{32} contendo os valores não vazios.

Para criar os testes seguindo o critério *All Combinations*, todos os blocos das três características deverão ser combinados, gerando, ao todo, oito requisitos de teste ($2 \times 2 \times 2 = 8$). No caso do critério *All Combinations*, os oito requisitos de teste são, também, os casos de teste, uma vez que cada combinação abrange todas as características. O conjunto contendo os casos de teste pode ser visto abaixo:

$$CTs = \{(b_{11}, b_{21}, b_{31}), (b_{11}, b_{21}, b_{32}), (b_{11}, b_{22}, b_{31}), (b_{11}, b_{22}, b_{32}),$$
$$(b_{12}, b_{21}, b_{31}), (b_{12}, b_{21}, b_{32}), (b_{12}, b_{22}, b_{31}), (b_{12}, b_{22}, b_{32})\}$$

Com este exemplo, é possível ver que algumas combinações de blocos podem ser inviáveis ou insatisfatíves, não existindo valores que possam satisfazer o que é requisitado. Um exemplo de combinação insatisfatível é a (b_{11}, b_{22}, b_{32}), que exige que o campo possua 6 letras ou menos (b_{11}) e que possua mais de 12 letras (b_{22}).

Each Choice: Um valor de cada bloco para cada característica deve ser utilizado em ao menos um caso de teste.

Este critério de combinação é mais simples e exige um número menor de testes para satisfazê-lo. Para o exemplo do campo X, existem seis requisitos, que são os seis blocos, e são necessários apenas dois casos de teste para satisfazer o critério:

$$CTs = \{(b_{11}, b_{21}, b_{31}), (b_{12}, b_{22}, b_{32})\}$$

Pairwise: Um valor de cada bloco para cada característica deve ser combinado com um valor de todos os blocos das outras características.

O critério *Pairwise* exige que todas as combinações dois a dois de blocos sejam testadas. O critério costuma exigir menos testes que o critério *All Combinations* e gerar melhores testes que o critério *Each Choice*. Para o exemplo apresentado aqui, as seguintes combinações (requisitos) devem ser garantidas:

$$(b_{11}, b_{21}), (b_{11}, b_{22}), (b_{11}, b_{31}), (b_{11}, b_{32}),$$

$$(b_{12}, b_{21}), (b_{12}, b_{22}), (b_{12}, b_{31}), (b_{12}, b_{32}),$$
$$(b_{21}, b_{31}), (b_{21}, b_{32}), (b_{22}, b_{31}), (b_{22}, b_{32})$$

É possível garantir esse critério com cinco casos de teste:

$$CTs = \{(b_{11}, b_{21}, b_{31}), (b_{11}, b_{22}, b_{31}), (b_{12}, b_{21}, b_{32}), (b_{12}, b_{22}, b_{31}), (b_{12}, b_{22}, b_{32})\}$$

Passo 5. **Refinar as combinações de blocos em entradas de teste:** Este último passo consiste em selecionar valores apropriados que satisfaçam as combinações para criar os testes.

2.1.3.2 Cobertura de Código

Neste tipo de cobertura, os requisitos de teste são obtidos diretamente do código fonte do programa sob teste. A cobertura é calculada medindo a porcentagem de elementos do código que são cobertos pelo conjunto de testes. Entre os elementos que são verificados, os mais comuns são: *funções*, em que é avaliada a quantidade de funções do programa sob teste que são chamadas pelo conjunto de testes; *comandos*, em que é avaliada a quantidade de comandos do programa sob teste que são executados pelo conjunto de testes; e *ramificações*, em que é avaliada a quantidade de ramificações, ou possíveis decisões, do programa sob teste que são executadas pelo conjunto de testes. A cobertura de código foi um dos primeiros métodos criados para se testar um software de maneira sistemática (MILLER; MALONEY, 1963). Neste trabalho, os testes gerados por BETA nos estudos de caso foram avaliados através da cobertura de comandos e ramificações. Para essa avaliação, foi utilizado, como ferramenta auxiliar, o *Gcov*[1], uma ferramenta para análise de cobertura de código em *C*.

2.1.3.3 Teste de Mutação

Teste de mutação (ou análise de mutação) é uma técnica de testes baseados em defeitos que consiste em criar variantes de um programa original, chamados de mutantes, a partir da simulação de defeitos comuns feitas por mudanças sintáticas, e desenvolver testes que consigam distinguir os mutantes do programa original (YOUNG; PEZZE, 2005). Nesta abordagem, um teste deve identificar que o resultado obtido com um mutante é diferente do resultado obtido com o programa original. Quando isso ocorre, é dito que o teste matou o mutante. Em certas situações, não é possível matar um mutante porque este, mesmo com alguma mudança sintática, possui o mesmo comportamento do programa original, o que faz com que ambos não possam ser distinguidos, neste caso, o mutante é dito equivalente. O processo de matar um mutante ou determinar que ele é equivalente

[1] <https://gcc.gnu.org/onlinedocs/gcc/Gcov.html>

nem sempre é trivial. Uma das formas de se analisar um mutante para determinar que ele é equivalente ou ver quais são as condições necessárias para matá-lo é através do Modelo *RIP*, apresentado na seção 2.1.1.

Definição 2.1.23 (Mutante) *Um programa que difere do original devido a alguma mudança sintática no código.*

Definição 2.1.24 (Mutante morto) *Dado um mutante m de um programa P e um teste t, é dito que m foi morto por t se, e somente se, a saída que P provê para t é diferente da saída que m provê para t.*

Definição 2.1.25 (Mutante equivalente) *Um mutante que não pode ser distinguido do programa original, ou seja, possui o mesmo comportamento do programa original.*

Teste de mutação pode ser utilizado como um critério de cobertura, em que matar cada um dos mutantes criados são requisitos de teste. Os mutantes são criados a partir da inserção de defeitos no código, que são pequenas modificações sintáticas feitas a partir de regras com padrões de mudança. Essas regras são chamadas de operadores de mutação e dependem da sintaxe de linguagem de programação utilizada, o que faz com que teste de mutação possa ser classificado como um critério de cobertura baseado em sintaxe (AMMANN; OFFUTT, 2010). Como exemplo, um operador de mutação pode modificar o comando "x = a + b" para "x = a - b", "x = a * b" ou "x = a / b". Teste de mutação também pode ser utilizado para medir a qualidade de um conjunto de testes, em que é avaliada a quantidade de mutantes, não considerando os mutantes equivalentes, que o conjunto de testes consegue matar, o que é chamado de escore de mutação (*mutation score*).

Definição 2.1.26 (Cobertura de mutação) *Dado um conjunto de mutantes M de um programa P, para cada mutante m pertencente a M, matar m é um requisito de teste.*

Definição 2.1.27 (Operador de mutação) *Uma regra para produzir um programa mutante a partir de uma modificação sintática no programa original.*

Definição 2.1.28 (Escore de mutação (*mutation score*)) *É a razão entre a quantidade de mutantes mortos e a quantidade de mutantes não equivalentes.*

Os operadores de mutação são variados, sendo construídos com base na sintaxe de uma linguagem de programação específica. Em (AGRAWAL et al., 1989) são apresentadas uma série de operadores de mutação que podem ser aplicados em programas escritos em *C* ou em linguagens que possuem uma sintaxe semelhante, como *Java* ou *C++*. A Tabela

Sigla	Operador de Mutação	Descrição	Exemplo
OAAN	Substituição de operador aritmético	Substitui a ocorrência de um operador aritmético (+, -, *, /, %) por cada um dos outros operadores aritméticos	a + b → a * b
OBBN	Substituição de operador bit a bit	Substitui a ocorrência de um operador bit a bit (&, ^, |, >>, <<) por cada um dos outros operadores bit a bit	a & b → a | b
OAAA	Substituição de operador de atribuição aritmética	Substitui a ocorrência de um operador de atribuição aritmética (+=, -=, *=, /=, %=) por cada um dos outros operadores de atribuição aritmética	a += b → a -= b
OBBA	Substituição de operador de atribuição bit a bit	Substitui a ocorrência de um operador de atribuição bit a bit (&=, ^=, |=, >>=, <<=) por cada um dos outros operadores de atribuição bit a bit	a &= b → a |= b
ORRN	Substituição de operador relacional	Substitui a ocorrência de um operador relacional (<, <=, >, >=, ==, !=) por cada um dos outros operadores relacionais	a < b → a <= b
OLLN	Substituição de operador lógico	Substitui a ocorrência de um operador lógico (&&, ||) por cada um dos outros operadores lógicos	a && b → a || b
OLNG	Negação lógica	Faz a negação lógica de cada um dos operandos em uma expressão lógica, assim como a negação da expressão lógica como um todo	a && b → a && !b
OCNG	Negação de contexto lógico	Faz a negação lógica de uma expressão que se encontra em um comando de repetição ou um comando condicional, com exceção do comando *switch*	if (exp) comando → if (!exp) comando
CRCR	Substituição de constante	Faz a substituição de uma constante numérica por outras constantes numéricas	a = b + 1 → a = b + 0
OIDO	Substituição de incremento ou decremento	Substitui uma operação de incremento ou decremento pela substituição contrária (decremento ou incremento) e a substituição por suas variantes, antes ou após a variável	a++ → a-- ou ++a
SBRC	Substituição de *break* por *continue*	Substitui o comando *break*, que para uma iteração, pelo comando *continue*, que continua uma iteração	break; → continue;
SSDL	Eliminação de comando	Elimina um comando	c = a + b; → ;
ABS	Inserção de valor absoluto	Substitui o valor de uma variável em uma expressão aritmética por seu valor absoluto positivo e negativo	a = 3 * b → a = 3 * abs(b)
UOI	Inserção de operador unário	Insere um operador unário (+, -) antes de uma variável em uma expressão aritmética	a = 3 * b → a = 3 * -b

Tabela 2.1 – Operadores de mutação

2.1 apresenta alguns dos operadores de mutação apresentados em (AGRAWAL et al., 1989).

Teste de mutação é um dos critérios mais difíceis de satisfazer, e estudos empíricos mostram que é um dos critérios mais eficientes em detectar defeitos (AMMANN; OFFUTT, 2010). Também é considerado um dos critérios mais difíceis de serem aplicados manualmente, o que faz com que automação seja necessária. Dentre algumas ferramentas que automatizam o teste de mutação, se destacam a *μJava* (MA; OFFUTT; KWON, 2005) e a *PIT*[2], ambas para a linguagem *Java*, a *Nester*[3], para a linguagem *C#*, e a *Milu* (JIA; HARMAN, 2008), para a linguagem *C*. Esta última, a ferramenta *Milu*, foi utilizada neste trabalho para avaliar os testes em um dos estudos de caso. A ferramenta

[2] A ferramenta *PIT* pode ser obtida no site <http://www.pitest.org>.

[3] A ferramenta *Nester* pode ser obtida no site <http://www.nester.sourceforge.net>.

Milu implementa os operadores de mutação que foram apresentados aqui.

2.1.4 Oráculos de Teste

Além dos casos de teste, um outro elemento fundamental no teste de software é o mecanismo que avalia os resultados do software para verificar se um teste foi bem sucedido ou não, ou seja, se houve a ocorrência de falhas. Esses mecanismos são conhecidos como *oráculos de teste* e têm o objetivo de verificar a correta execução do software dada uma certa entrada.

Definição 2.1.29 (Oráculo de teste) *É o mecanismo que determina se um programa se comportou corretamente em um dado teste, resultando no veredicto "passou" ou "falhou".*

Projetar oráculos é uma das maiores dificuldades dentro do teste de software, pois os oráculos têm relação direta com a qualidade dos testes desenvolvidos, uma vez que oráculos mal projetados podem não detectar falhas. Abordagens comuns para oráculos de teste consistem em verificação direta das saídas, computação de redundâncias, verificação de consistência e redundância de dados (AMMANN; OFFUTT, 2010).

Em (LI; OFFUTT, 2014) são apresentadas algumas estratégias de oráculo de teste (abreviada como OS), que são regras que especificam quais elementos do programa devem ser verificados em um teste. As estratégias vão desde a simples verificação de alguma ocorrência não esperada até verificações mais profundas, como análise do estado interno do software e de suas saídas. As estratégias apresentadas em (LI; OFFUTT, 2014) estão inseridas no contexto de teste de sistemas feitos em *Java*, por isso essas estratégias são relacionadas à orientação a objetos. Os elementos que são verificados pelas estratégias são: invariante de estado, membros dos objetos (atributos dos objetos), valores de retorno e membros dos parâmetros (atributos dos objetos passados como argumento). De forma simplificada (adaptada para teste de unidade), as estratégias apresentadas são:

NOS Consiste, simplesmente, em verificar a ocorrência de exceções ou finalizações inesperadas do software;

SIOS Consiste em verificar se o invariante de estado foi mantido após a execução do método sob teste. As estratégias apresentadas a seguir também seguem esta estratégia;

OS1 Consiste em verificar os valores dos membros de todos os objetos utilizados pelo método sob teste;

OS2 Consiste em verificar os valores de retorno do método sob teste;

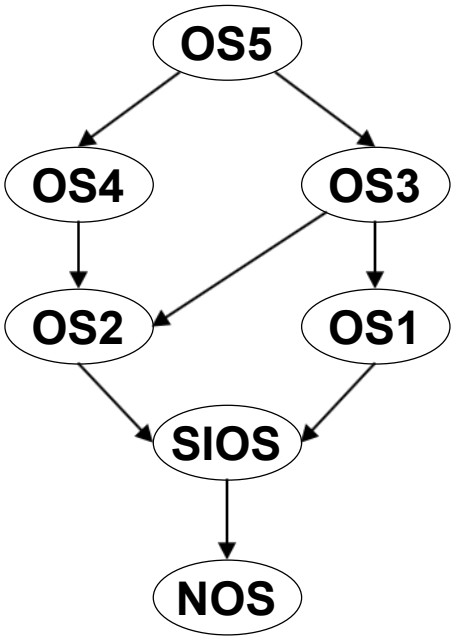


Figura 2.1 – Relação de precisão entre as estratégias de oráculo de teste

Fonte: adaptado de (LI; OFFUTT, 2014)

OS3 Esta estratégia consiste na união das estratégias *OS1* e *OS2*, em que os valores dos membros de todos os objetos utilizados e os valores retornados pelo método sob teste são verificados;

OS4 Consiste em verificar os valores dos membros de todos os parâmetros do método sob teste, além dos seus valores de retorno;

OS5 Esta estratégia une as estratégias anteriores em apenas uma, em que os valores dos membros de todos os objetos utilizados, os valores dos membros de todos os parâmetros e os valores de retorno do método sob teste são verificados.

Uma propriedade importante em uma estratégia de oráculo de teste é a sua precisão (BRIAND; PENTA; LABICHE, 2004), que se refere ao grau em que os elementos são verificados. Quanto mais precisa uma estratégia é, mais defeitos ela consegue detectar (BRIAND; PENTA; LABICHE, 2004), (SHRESTHA; RUTHERFORD, 2011), (STAATS; WHALEN; HEIMDAHL, 2011) e (XIE; MEMON, 2007). A precisão do oráculo de teste também é relacionada a observabilidade do software sob teste, pois quanto mais "observável" um software é, mais estados podem ser verificados e, consequentemente, mais defeitos podem ser detectados. A Figura 2.1 apresenta a relação de precisão entre as estratégias de oráculo de teste apresentadas. Uma seta a partir de uma estratégia para outra indica que a mais alta é a estratégia mais precisa.

2.2 Método B

A especificação de um sistema descreve como ele deve funcionar, descrevendo as funcionalidades que serão oferecidas para o usuário, e o que será alcançado quando uma funcionalidade for escolhida. Para isso, a especificação deve conter uma quantidade significativa de informações sobre o sistema. Uma das dificuldades encontradas ao especificar um sistema é gerir o grande volume de informações detalhadas sobre o sistema de forma que se possa ter uma especificação precisa e confiável. Para isso, é essencial que se utilize uma abordagem estruturada de especificação para produzir uma descrição do sistema com maior qualidade e sem inconsistências. O *Método B* oferece tal abordagem.

O Método B é uma metodologia formal para especificação, projeto e codificação de software (ABRIAL, 2005). Utilizando conceitos da teoria dos conjuntos, lógica de primeira ordem, aritmética de inteiros e *substituições generalizadas* (transformadores de predicados), o processo do Método B se inicia com a criação de um módulo abstrato, chamado *máquina abstrata*, que especifica o comportamento do sistema. Grandes especificações podem ser construídas a partir de pequenas, utilizando um mecanismo de estruturação que permite a separação da especificação em várias partes, dividindo o problema em partes menores e mais entendíveis. Esta abordagem é composicional, em que a combinação de máquinas abstratas é por si só uma máquina abstrata, permitindo uma especificação hierárquica (SCHNEIDER, 2001).

Máquinas abstratas descrevem *o que* o sistema deve fazer, entretanto, elas não, necessariamente, descrevem *como* deve ser feito. Isso permite que construções abstratas ou não determinísticas possam ser utilizadas para especificar o comportamento do sistema. Porém, essas construções não são entendidas por um computador, que depende de instruções bem definidas de como fazer para poder ter o comportamento esperado. O Método B fornece um processo completo que permite refinar uma máquina abstrata até que ela se torne entendível para um computador, ou seja, até que um código correspondente a especificação abstrata seja produzido. Os módulos de refinamento se encontram no centro deste processo. De forma gradual, uma máquina abstrata pode ser refinada através de vários módulos de refinamento que visam transformar as construções da especificação em detalhes de implementação, reduzindo as abstrações e não determinismos da especificação. O último passo do processo de refinamento é a implementação, que é o nível mais concreto da especificação, de modo que o código do sistema pode ser produzido.

A Figura 2.2 apresenta o processo completo do Método B. Nela é possível ver que cada uma das etapas é formalmente provada para que a consistência seja garantida, inclusive os módulos de refinamento e implementação que devem ser consistentes com a máquina abstrata de origem. Desse modo, partindo de uma especificação não formal dos requisitos, é possível gerar um código consistente do sistema. A seguir será feita uma breve introdução aos módulos e principais conceitos, além de uma breve apresentação de

algumas ferramentas do Método B.

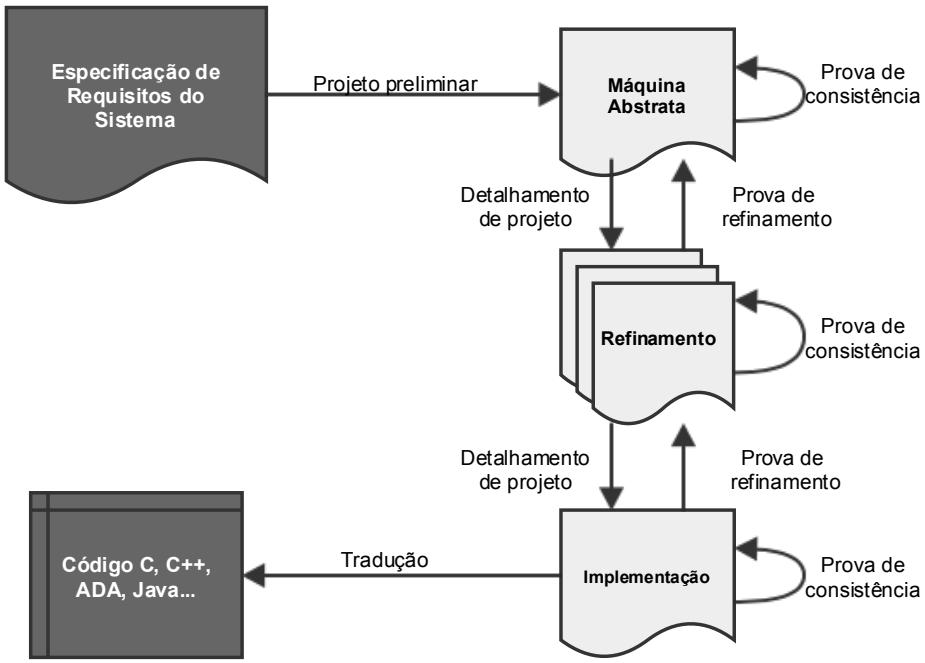


Figura 2.2 – Processo do Método B

2.2.1 Máquina Abstrata

Uma máquina abstrata é a especificação de (ou parte de) um sistema. Ela contém informações que descrevem vários aspectos do sistema, como seu estado interno, suas propriedades e funcionalidades. Primeiramente, as máquinas abstratas devem descrever o que o sistema pode fazer. Isso é feito através das *operações*, que descrevem as ações que podem ser realizadas pelo sistema. As operações podem receber entradas do usuário, fornecer saídas para o usuário e realizar mudanças dentro do sistema. As operações da máquina abstrata são listadas na cláusula *OPERATIONS*.

Em uma máquina abstrata pode ser necessário guardar e processar informações. Essas informações ficam contidas em um estado local, que é representado pelas *variáveis de estado*. Essas variáveis são listadas na cláusula *VARIABLES*. Todas as informações relacionadas a uma variável, como seu tipo e restrições, ficam contidas na cláusula *INVARIANT*. As informações contidas nessa cláusula sempre devem ser verdadeiras, mesmo após a execução de operações. Também deve ser especificado o *estado inicial* para o sistema. Isso é feito na cláusula *INITIALISATION*, que atribui valores iniciais para todas as variáveis de estado.

Por último, uma máquina abstrata deve possuir um nome para que outros, possíveis, módulos ou componentes possam referenciá-la. A cláusula *MACHINE* define o nome da máquina abstrata. Além dessas cláusulas, máquinas abstratas também podem conter

conjuntos, que são definidos na cláusula *SETS*, e constantes, que são definidas na cláusula *CONSTANTS*. As condições que devem ser satisfeitas pelos conjuntos e constantes, como seu tipo, valores e restrições, por exemplo, ficam contidas na cláusula *PROPERTIES*.

Para ilustrar uma máquina abstrata em mais detalhes, será apresentado como exemplo uma especificação de um jogo popular conhecido como "Jogo da Velha". O jogo possui regras simples e é jogado por dois jogadores em turnos. Em cada turno, um jogador deve marcar uma posição no tabuleiro, que inicialmente não tem nenhuma marcação. O tabuleiro possui nove posições dispostas em uma matriz de três linhas e três colunas. A Figura 2.3 apresenta um exemplo de tabuleiro do "Jogo da Velha" com as marcações dos jogadores.

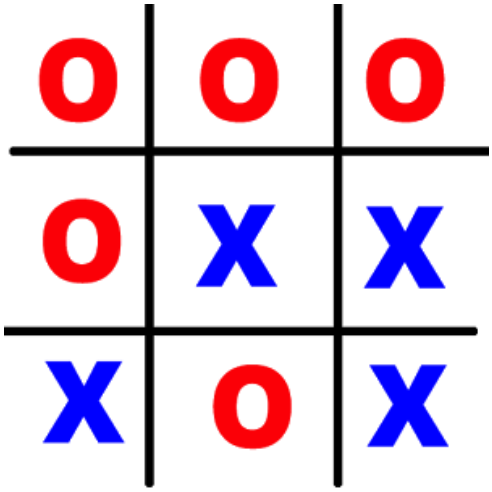


Figura 2.3 – Tabuleiro do "Jogo da Velha"

O objetivo do jogo é marcar três posições em linha no tabuleiro, dispostas na horizontal, vertical ou diagonal. O jogo pode terminar com a vitória de um jogador, aquele que conseguiu marcar três posições em linha primeiro, ou em empate, quando todas as posições são marcadas mas nenhuma em linha por um só jogador. Na Figura 2.3 também é possível ver um exemplo de vitória, em que o jogador com a marcação "**O**", em vermelho, conseguiu marcar três posições em linha. Para especificar o jogo, foi necessário fazer uma abstração do tabuleiro, enumerando cada uma de suas posições de 0 a 8. A Figura 2.4 apresenta os números correspondentes a cada uma das posições. Com essa abstração, uma posição válida do tabuleiro pertence ao intervalo de 0 a 8.

A Figura 2.5 apresenta a máquina abstrata *TicTacToe* que especifica o "Jogo da Velha". Essa máquina foi inspirada na especificação apresentada em (ROSE; DUKE, 2000). Na linha 1, a cláusula *MACHINE* provê o nome da máquina. A cláusula *SETS* (linhas 2 a 4) define os conjuntos da máquina, que podem ser conjuntos abstratos ou enumerados. Foram definidos dois conjuntos enumerados nesta máquina, o conjunto *Player* (linha 3), que possui os valores *blue* e *red*, que são as abstrações dos dois jogadores, e o conjunto *Result* (linha 4), que possui três valores que representam os possíveis resultados

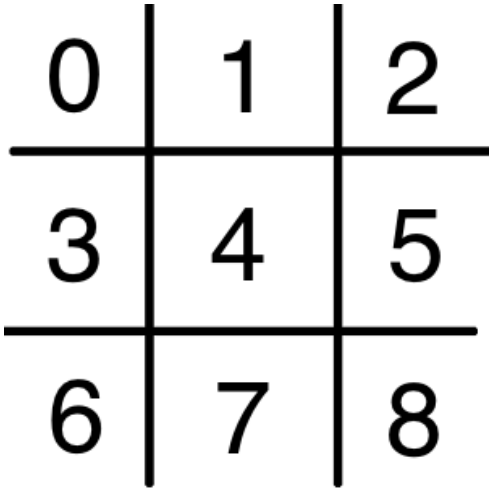


Figura 2.4 – Abstração do tabuleiro do "Jogo da Velha"

do jogo, que são: *blue_win*, que significa que o jogador representado por *blue* ganhou; *red_win*, que significa que o jogador representado por *red* ganhou; e *draw*, que significa que o jogo empatou.

A cláusula *CONSTANTS* (linhas 5 e 6) define os nomes das constantes que serão utilizadas na máquina. São definidas duas constantes: *WinnerRows* e *ThreeInRow*. A cláusula *PROPERTIES* (linhas 7 a 16) descreve as condições dos conjuntos e constantes definidos nas cláusulas anteriores. Nas linhas 8 a 10, a constante *WinnerRows* é definida como um conjunto contendo todos os subconjuntos que representam as linhas na vertical, horizontal e diagonal do tabuleiro, ou seja, é um conjunto contém todas as linhas vencedoras. Nas linhas 11 a 16, a constante *ThreeInRow* é definida como uma função que recebe um subconjunto do intervalo de 0 a 8 e retorna verdadeiro, caso o subconjunto contenha três números que representam uma linha, ou falso, caso o subconjunto não contenha três números que representam uma linha. A função *ThreeInRow* será utilizada nas operações para verificar se um jogador ganhou o jogo.

A cláusula *VARIABLES* (linhas 17 e 18) define o nome das variáveis locais que irão guardar as informações do estado da máquina. Na máquina *TicTacToe* são definidas três variáveis: a variável *bposn*, que irá guardar as posições marcadas no tabuleiro pelo jogador representado por *blue*; a variável *rposn*, que irá guardar as posições marcadas no tabuleiro pelo jogador representado por *red*; e a variável *turn*, que guarda o jogador que vai jogar no turno.

A cláusula *INVARIANT* (linhas 19 a 28) define todas as informações sobre as variáveis de estado, como os seus tipos e restrições nos valores que podem assumir. Os valores das variáveis podem mudar durante a execução da máquina, mas o invariante define as propriedades das variáveis de estado que serão sempre verdadeiras. As variáveis *bposn* e *rposn* são definidas como um subconjunto do intervalo de de 0 a 8 (linha 20); a interseção entre *bposn* e *rposn* deve ser vazia (linha 21), pois uma mesma posição não

```
1  MACHINE TicTacToe
2  SETS
3      Player = {blue, red};
4      Result = {blue_win, red_win, draw}
5  CONSTANTS
6      WinnerRows, ThreeInRow
7  PROPERTIES
8      WinnerRows ⊆ ℙ(0..8) ∧
9      WinnerRows = {{0,1,2}, {3,4,5}, {6,7,8}, {0,3,6},
10                   {1,4,7}, {2,5,8}, {0,4,8}, {6,4,2}} ∧
11     ThreeInRow ∈ ℙ(0..8) → BOOL ∧
12     ThreeInRow =
13       λps.(ps ∈ ℙ(0..8) ∧
14             ∃wn.(wn ∈ WinnerRows ∧ wn ⊆ ps)|TRUE) ∪
15       λps.(ps ∈ ℙ(0..8) ∧
16             ¬(∃wn.(wn ∈ WinnerRows ∧ wn ⊆ ps))|FALSE)
17 VARIABLES
18     bposn, rposn, turn
19 INVARIANT
20     bposn ⊆ 0..8 ∧ rposn ⊆ 0..8 ∧
21     bposn ∩ rposn = ∅ ∧
22     (ThreeInRow(bposn) = TRUE ⇒ ThreeInRow(rposn) = FALSE) ∧
23     (ThreeInRow(rposn) = TRUE ⇒ ThreeInRow(bposn) = FALSE) ∧
24     turn ∈ Player ∧
25     (turn = blue ⇒ card(bposn) ≤ card(rposn) ∧
26           card(rposn) ≤ (card(bposn) + 1)) ∧
27     (turn = red ⇒ card(rposn) ≤ card(bposn) ∧
28           card(bposn) ≤ (card(rposn) + 1))
29 INITIALISATION
30     bposn := ∅ || rposn := ∅ || turn := blue
31 OPERATIONS
32     BlueMove(pp) =
33     PRE
34         pp ∈ 0..8 ∧
35         pp ∉ (bposn ∪ rposn) ∧
36         turn = blue ∧
37         ThreeInRow(rposn) = FALSE
38     THEN
39         bposn := bposn ∪ {pp} ||
40         turn := red
41     END;
42     RedMove(pp) =
43     PRE
44         pp ∈ 0..8 ∧
45         pp ∉ (bposn ∪ rposn) ∧
46         turn = red ∧
47         ThreeInRow(bposn) = FALSE
48     THEN
49         rposn := rposn ∪ {pp} ||
50         turn := blue
51     END;
52     result ← GameResult =
53     PRE
54         ThreeInRow(bposn) = TRUE ∨
55         ThreeInRow(rposn) = TRUE ∨
56         (bposn ∪ rposn) = 0..8
57     THEN
58         SELECT ThreeInRow(bposn) = TRUE THEN
59             result := blue_win
60         WHEN ThreeInRow(rposn) = TRUE THEN
61             result := red_win
62         ELSE
63             result := draw
64         END
65     END
66 END
```

Figura 2.5 – Máquina abstrata *TicTacToe* que especifica o "Jogo da Velha"

pode ser marcada por dois jogadores; e se um jogador tiver marcado três posições em linha, então o outro jogador não pode ter marcado outras três posições em linha (linhas 22 e 23), que significa que apenas um jogador pode ganhar. A variável *turn* pertence ao conjunto *Player* (linha 24), que armazena o jogador que irá jogar no turno; e quando o turno é de um jogador, o número de posições marcadas pelo outro jogador deve ser igual ou superior, não excedendo uma posição a mais que o jogador da vez (linhas 25 a 28), essa restrição obriga que o jogo tem que ser em turnos. A cláusula *INITIALISATION* (linhas 29 e 30) define o estado inicial da máquina. Todas as variáveis descritas na cláusula *VARIABLES* devem ter algum valor atribuído. As variáveis *bposn* e *rposn* são atribuídas com o conjunto vazio, pois inicialmente nenhuma posição é marcada por algum jogador, e a variável *turn* é atribuída com *blue*, o primeiro jogador a jogar.

A cláusula *OPERATIONS* define as operações da máquina (linhas 31 a 65). As operações definem as ações que podem ser feitas na máquina, como a consulta e a modificação do estado. A assinatura de uma operação contém o nome da operação e seus parâmetros de entrada e saída, estes últimos não são obrigatórios. A especificação de uma operação pode ser dividida em duas partes, uma parte contendo as *precondições* e uma parte contendo o *corpo* da operação. As precondições contém a tipagem dos parâmetros de entrada e restrições para os parâmetros de entrada e variáveis de estado. Quando as precondições são cumpridas, é garantido que a operação vai ser executada sem inconsistências.

O corpo da operação descreve o que a operação quer alcançar. No corpo são atribuídos valores para os parâmetros de saída e o estado da máquina pode ser atualizado. O efeito de uma operação é, portanto, definido como uma relação entre o estado antes de sua chamada com os valores de entrada, e o estado após a sua chamada junto com os valores de saída. Isso é feito através de declarações abstratas que determinam como o estado deve ser atualizado e como as saídas devem ser em termos do estado inicial e dos valores de entrada da operação. No Método B, essas declarações são definidas como substituições generalizadas (transformadores de predicados), e podem ser determinísticas ou não determinísticas.

Entre as principais substituições determinísticas estão a atribuição simples, a condicional *IF*, que possui o mesmo comportamento do comando de decisão *if then else* de linguagens de programação imperativas, e o *CASE*, que se assemelha ao comando de decisão *switch* de linguagens imperativas. Entre as principais substituições não determinísticas estão a *ANY*, que possibilita a utilização de qualquer valor que satisfaça uma determinada condição, a *CHOICE*, que possibilita listar várias possíveis declarações sem determinar qual será selecionada, e a *SELECT*, que funciona de forma semelhante a *CHOICE*, mas com a possibilidade de impor condições para que uma declaração possa ser selecionada. Declarações não determinísticas são importantes porque fornecem flexibili-

dade e possibilitam que decisões sejam tomadas mais a frente, nas etapas de refinamento e implementação, por exemplo.

A operação *BlueMove* (linhas 32 a 41) é responsável por fazer a jogada do jogador representado por *blue*. A operação não possui parâmetros de saída e possui apenas um parâmetro de entrada, que representa a posição que o jogador quer marcar no tabuleiro. Suas precondições determinam que a posição selecionada deve estar dentro do intervalo de 0 a 8 (linha 34), que a posição não tenha sido marcada por nenhum dos dois jogadores (linha 35), que o turno seja do jogador representado por *blue* (linha 36) e que o jogador representado por *red* não tenha marcado três posições em linha, ou seja, que não tenha ganhado (linha 37). O comportamento da operação *BlueMove* é atualizar as variáveis *bposn* (linha 39), inserindo a nova posição no conjunto de posições marcadas pelo jogador representado por *blue*, e *turn* (linha 40), passando o turno para o outro jogador representado por *red*. A operação *RedMove* (linhas 42 a 51) é análoga a operação *BlueMove*, sendo responsável pela jogada do jogador representado por *red*.

A operação *GameResult* (linhas 52 a 65) possui apenas um parâmetro de saída e é responsável por indicar o resultado final do jogo. Sua precondição exige que o jogador representado por *blue* tenha marcado três posições em linha (linha 54), ou que o jogador representado por *red* tenha marcado três posições em linha (linha 55), ou que todas as posições já tenham sido marcadas (linha 56), sendo essas as condições que determinam o fim de uma partida. No corpo da operação (linhas 58 a 64) foi utilizado a substituição não determinística *SELECT* para indicar os possíveis resultados do jogo e suas condições para ocorrer. Se o jogador representado por *blue* tiver marcado três posições em linha, então a saída da operação é *blue_win*, indicando que ele foi o vencedor (linhas 58 e 59). Caso o jogador representado por *red* tenha marcado três posições em linha, então a saída da operação é *red_win*, indicando que ele foi o vencedor (linhas 60 e 61). Senão, a saída da operação é *draw*, indicando que o jogo terminou empatado.

2.2.2 Refinamento e Implementação

As construções matemáticas utilizadas em uma máquina abstrata são adequadas para uma especificação, pois conseguem descrever de forma eficiente qual é o comportamento esperado do sistema, em termos do relacionamento entre o estado inicial e o estado final. Entretanto, essas construções não são adequadas para um computador, uma vez que as linguagens de programação convencionais não permitem muitas delas, como alguns conjuntos, funções e não determinismos, por exemplo, e dependem de instruções bem definidas de como o computador deve se comportar para que o comportamento esperado possa ser atingido.

Para que o quê está descrito na máquina abstrata possa ser transformado em um programa executável, a equipe de desenvolvimento do sistema deve tomar decisões

de projeto sobre qual é a melhor forma de representar as abstrações da especificação em uma linguagem de programação. No Método B, essas decisões e detalhes de projeto são inseridos através dos módulos de refinamento, que inserem informações em uma máquina abstrata com o intuito de aproximá-la de um código executável.

As novas informações inseridas em um módulo de refinamento devem ser consistentes com a máquina abstrata, pois estas devem, de uma forma mais concreta, descrever o mesmo comportamento. Isso é feito através de um *invariante de ligação* que relaciona o estado da máquina abstrata ao estado do módulo de refinamento, mantendo a consistência entre os dois módulos. Um módulo de refinamento também deve possuir a mesma interface que a máquina abstrata a qual ele está refinando, ou seja, possuir as mesmas operações da máquina abstrata e as mesmas assinaturas (mesmo nome da operação e de seus parâmetros de entrada e saída).

O refinamento de uma máquina abstrata pode ser feito de forma gradual, através da criação de vários módulos de refinamento que, pouco a pouco, vão inserindo detalhes de projeto. Nesse processo vão sendo encapsulados detalhes de projeto como estruturas de dados e algoritmos, além da resolução de não determinismos. O objetivo final desse processo é produzir uma descrição detalhada o suficiente para ser entendível como instruções para um computador, possibilitando a sua tradução direta para um código em alguma linguagem de programação. Dessa forma, o módulo de refinamento deve conter apenas construções que podem ser transformadas em código. Quando um módulo de refinamento atinge esse ponto, é chamado de módulo de implementação.

Uma implementação é, por si só, um módulo de refinamento, o que faz com que tenha que ser consistente com a máquina abstrata a qual está refinando. Um módulo de implementação deve conter apenas construções que podem ser entendidas por um computador. O subconjunto de construções B que podem ser utilizadas é chamado de *B0*. Este subconjunto contém apenas construções que são passíveis de ser implementadas em linguagens de programação imperativas e sequenciais.

Para exemplificar o processo de refinamento e implementação, a Figura 2.6 apresenta um trecho de um possível módulo de implementação para a máquina abstrata *TicTacToe* apresentada na Figura 2.5. A cláusula *IMPLEMENTATION* (linha 1) provê o nome *TicTacToeI* para o módulo de implementação. Em um módulo de refinamento normal, é a cláusula *REFINEMENT* que provê o nome para o módulo. A cláusula *REFINES* (linha 2) indica qual é o módulo que está sendo refinado, no caso é a máquina abstrata *TicTacToe*. A cláusula *CONCRETE_VARIABLES* (linhas 3 e 4) define o nome das variáveis *board* e *blue_turn* que farão parte da implementação, em contrapartida à cláusula *VARIABLES* do módulo abstrato que define o nome de variáveis que, não necessariamente, precisam estar na implementação.

A cláusula *INVARIANT* (linhas 5 a 11) define as informações sobre o estado do

```
1  IMPLEMENTATION TicTacToeI
2  REFINES TicTacToe
3  CONCRETE_VARIABLES
4      board, blue_turn
5  INVARIANT
6      board ∈ (0..8) → (0..2) ∧
7      dom(board ▷ {1}) = bposn ∧
8      dom(board ▷ {2}) = rposn ∧
9      blue_turn ∈ BOOL ∧
10     (blue_turn = TRUE ⇒ turn = blue) ∧
11     (blue_turn = FALSE ⇒ turn = red)
12 INITIALISATION
13     board := {0|->0, 1|->0, 2|->0, 3|->0, 4|->0,
14                 5|->0, 6|->0, 7|->0, 8|->0};
15     blue_turn := TRUE
16 OPERATIONS
17     BlueMove(pp) =
18     BEGIN
19         board(pp) := 1;
20         blue_turn := FALSE
21     END;
22     /* ... */
23 END
```

Figura 2.6 – Possível implementação B para a máquina abstrata *TicTacToe*

módulo, assim como ocorre em uma máquina abstrata, além de definir a relação entre o estado da implementação e o estado da máquina abstrata, através do invariante de ligação. Na máquina *TicTacToe* foram utilizados conjuntos para representar as marcações no tabuleiro. Entretanto, esses conjuntos não podem ser utilizados em uma implementação, o que faz com que o tabuleiro e suas marcações tenham que ser representados de uma nova maneira.

Na programação é comum utilizar *arrays* para guardar os dados de uma coleção, como um conjunto, por exemplo. Um *array* é uma coleção de valores indexados. Seus valores podem ser acessados através de um índice, que podem ser atualizados e lidos como uma variável normal. No Método B, um *array* é definido como o mapeamento entre um conjunto de índices e um conjunto de dados. Como as posições no tabuleiro foram abstraídas como números no intervalo de 0 a 8, como é mostrado na Figura 2.4, foi decidido que o tabuleiro seria representado por um *array* de nove posições com os seus índices indo de 0 a 8. Cada índice representa a sua respectiva posição no tabuleiro e armazena o valor da posição no momento, que pode ser: 0, indicando que esta posição não foi marcada por nenhum jogador; 1, indicado que esta posição foi marcada pelo jogador representado por *blue*; e 2, indicando que esta posição foi marcada pelo jogador representado por *red*.

A variável *board* representa o tabuleiro e é definida como sendo uma relação entre o conjunto do intervalo de 0 a 8, que representa os índices do *array*, e o conjunto do intervalo de 0 a 2, que representa as três possíveis situações de uma posição (linha 6). Uma vez que os dados estão sendo descritos de uma nova maneira, é preciso relacioná-los com os dados da máquina abstrata. Dessa forma, os índices dos valores que foram atribuídos com 1 no *array* foram relacionados a variável *bposn* da máquina abstrata (linha 7), uma vez

que estes representam as posições marcadas pelo jogador representado por *blue*. De forma análoga, os índices dos valores que foram atribuídos com 2 no *array* foram relacionados a variável *rposn* da máquina abstrata (linha 8), uma vez que estes representam as posições marcadas pelo jogador representado por *red*.

A variável *blue_turn* guarda a informação de que é ou não o turno do jogador representado por *blue*, por isso foi definida como sendo do tipo booleano (linha 9). Para fazer a relação com a variável *turn* da máquina abstrata, foi definido que se *blue_turn* for verdadeiro, então o turno é do jogador representado por *blue* (linha 10), e se for falso, então o turno é do jogador representado por *red* (linha 11). A cláusula *INITIALISATION* (linha 12 a 15) atribui valores iniciais para as variáveis de estado. Todos os índices do *array board* foram atribuídos com o valor 0 (linhas 13 e 14), indicando que nenhuma posição foi marcada até o momento. A variável *blue_turn* foi atribuída com o valor verdadeiro (linha 15), indicando que o turno é do jogador representado por *blue*.

A cláusula *OPERATIONS* define as operações do módulo de implementação, devendo ter as mesmas operações e assinaturas definidas na máquina abstrata. Na Figura 2.6 é possível ver um trecho do módulo de implementação em que a operação *BlueMove* (linhas 17 a 21) é mostrada. Assim como na máquina abstrata, essa operação possui apenas um parâmetro de entrada, a posição que o jogador deseja marcar. Nos módulos de refinamento e implementação as precondições das operações, descritas na máquina abstrata, são implícitas, por isso não é necessário declará-las novamente. No corpo da operação *BlueMove*, o valor no índice do *array board* passado como parâmetro é atribuído com o valor 1 (linha 19), indicando que a posição representada pelo índice foi marcada pelo jogador representado por *blue*, e a variável *blue_turn* foi atribuída com falso (linha 20), indicando que o turno passou para o jogador representado por *red*.

A implementação B apresentada na Figura 2.6 é apenas uma das possíveis formas de implementar o módulo *TicTacToe*. A partir desse módulo de implementação pode ser produzido código executável em alguma linguagem de programação. Como exemplo, a Figura 2.7 apresenta um trecho de um código em *C* que implementa o módulo *TicTacToe*. Este código foi gerado de forma automática a partir da implementação B *TicTacToeI*.

2.2.3 Ferramentas

Para auxiliar o desenvolvimento de um sistema com o Método B existem várias ferramentas no mercado. Entre as principais ferramentas estão o *ProB* e o *Atelier B*, ambas foram utilizadas neste trabalho. O *ProB*[4] (LEUSCHEL; BUTLER, 2003) é um animador e verificador de modelos (*model checker*) para o Método B. Ele permite que uma especificação B seja animada, simulando a sua execução. O *ProB* também pode ser

[4] A ferramenta *ProB* pode ser obtida no site <http://stups.hhu.de/ProB/>.

```
1  static int32_t TicTacToe__board[9];
2  static bool TicTacToe__blue_turn;
3
4  void TicTacToe__INITIALISATION(void)
5  {
6      TicTacToe__board[0] = 0;
7      TicTacToe__board[1] = 0;
8      TicTacToe__board[2] = 0;
9      TicTacToe__board[3] = 0;
10     TicTacToe__board[4] = 0;
11     TicTacToe__board[5] = 0;
12     TicTacToe__board[6] = 0;
13     TicTacToe__board[7] = 0;
14     TicTacToe__board[8] = 0;
15     TicTacToe__blue_turn = true;
16 }
17
18 void TicTacToe__BlueMove(int32_t pp)
19 {
20     TicTacToe__board[pp] = 1;
21     TicTacToe__blue_turn = false;
22 }
23 // ...
```

Figura 2.7 – Código *C* que implementa a máquina abstrata *TicTacToe*

utilizado como solucionador de restrições (*constraint solving*) para auxiliar na detecção de erros e inconsistências na especificação, como um *deadlock* por exemplo.

O *Atelier B*[5] é uma *IDE* (*Integrated Development Environment*) que permite a utilização operacional do Método B. O *Atelier B* auxilia no desenvolvimento com o Método B ao fornecer um ambiente confortável para edição e gerenciamento de módulos B. A ferramenta também fornece um provador de teoremas automático que auxilia a provar que um módulo B é consistente. Além disso, o *Atelier B* também gera código em alguma linguagem de programação a partir de uma implementação B. A versão gratuita da ferramenta traz, de forma integrada, o gerador de código *C4B*, que gera código *C* a partir de uma implementação B. O código apresentado na Figura 2.7 é um exemplo de código gerado pelo *C4B*. Atualmente, o *Atelier B* se encontra na versão 4.2, mas neste trabalho foi utilizada a versão 4.1, pois essa era a versão vigente quando este trabalho foi iniciado.

[5] A ferramenta *Atelier B* pode ser obtida no site <http://www.atelierb.eu>.

3 BETA

Neste Capítulo a abordagem e ferramenta BETA (*B Based Testing Approach*) (MATOS; MOREIRA, 2012) são apresentadas. Inicialmente, a abordagem é apresentada em detalhes. Para ilustrar a abordagem, foi utilizado como exemplo a máquina abstrata *TicTacToe* apresentada no Capítulo 2. Em seguida, é apresentada a ferramenta que automatiza parte da abordagem, com a explicação de seu funcionamento e os seus resultados. Por último, é feita uma discussão sobre trabalhos relacionados a teste de software e métodos formais que podem trazer contribuições para BETA.

3.1 Abordagem

BETA é uma abordagem para geração de casos de teste de unidade a partir de especificações formais escritas na notação do Método B. Foi proposta em (SOUZA, 2009), que desenvolveu uma versão inicial da abordagem, e aprimorada em (MATOS; MOREIRA, 2012) e (MATOS; MOREIRA, 2013), que melhorou a abordagem inicial e desenvolveu uma ferramenta que automatiza parte de suas etapas. Neste trabalho, foi considerada a primeira versão de BETA, que foi apresentada em (MATOS; MOREIRA, 2013), entretanto, a abordagem e ferramenta BETA, atualmente, já evoluiu desde a sua versão inicial.

Utilizando como tipo de critério de cobertura o particionamento do espaço de entrada, que foi apresentado no Capítulo 2, BETA gera casos de teste positivos e negativos para uma unidade (operação) de uma máquina abstrata B. As informações para o particionamento são obtidas a partir do invariante de estado, das precondições e outros predicados da operação sob teste. Feito o particionamento, os blocos são combinados (fazendo uso dos critérios *Each Choice*, *Pairwise* ou *All Combinations* apresentados no Capítulo 2) e o resultado é um conjunto de fórmulas que representam as condições que devem ser testadas, onde cada fórmula representa um caso de teste.

De posse das fórmulas que representam os casos de teste, a etapa seguinte é obter valores para as variáveis do espaço de entrada que satisfazem as restrições impostas pelas fórmulas, de modo a obter os dados de teste. A abordagem propõe o uso de um solucionador de restrições para esse fim. Obtidos os dados de teste, uma especificação de casos de teste, que contém os valores das variáveis de estado e dos parâmetros de entrada para cada caso de teste, é gerada. As duas últimas etapas da abordagem consistem em obter os resultados esperados no teste, que serão utilizados nos oráculos para verificar se um teste passou ou falhou, e implementar cada caso de teste, concretizando toda a abordagem. A Figura 3.1 apresenta uma visão geral da abordagem BETA. Os artefatos utilizados e

produzidos durante a abordagem estão em cinza claro, e as etapas da abordagem estão em cinza escuro.

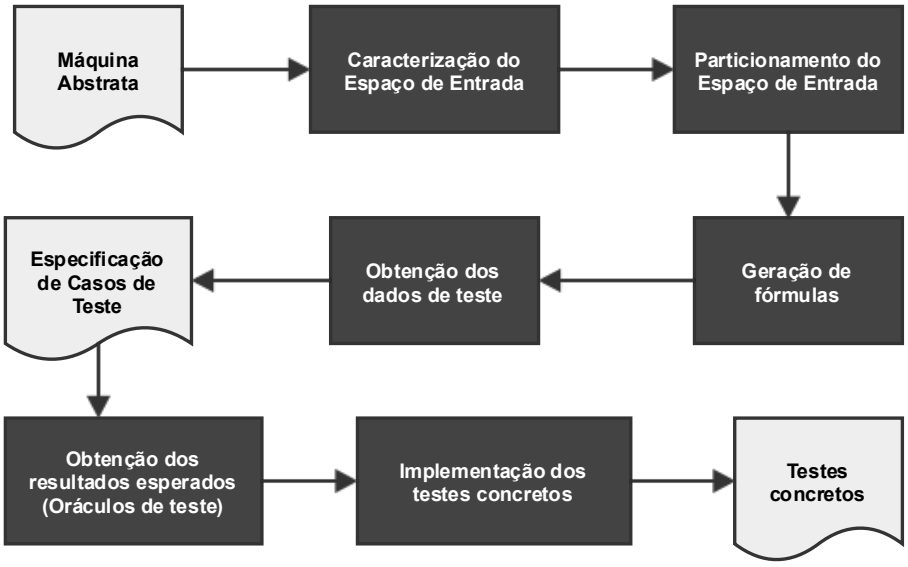


Figura 3.1 – Visão geral da abordagem BETA

Para ilustrar a abordagem, será utilizado como exemplo a máquina abstrata *TicTacToe* apresentada na Figura 2.5 do Capítulo 2. A seguir, será apresentado como são realizadas cada uma das etapas da abordagem para gerar testes de unidade para a operação *BlueMove* da máquina *TicTacToe*.

***Etapa 1.* Caracterização do espaço de entrada**

Esta etapa consiste em identificar as variáveis do espaço de entrada da operação e caracterizá-las. Em BETA, são consideradas como variáveis do espaço de entrada de uma operação os seus parâmetros de entrada e as variáveis de estado que estão presentes nos seus predicados, além das variáveis relacionadas na cláusula do invariante. Para a operação *BlueMove*, as variáveis do espaço de entrada são o parâmetro de entrada *pp*, e as variáveis de estado *bposn*, *rposn* e *turn*.

Após a identificação das variáveis do espaço de entrada da operação sob teste, deve-se identificar as restrições nos valores que essas variáveis podem assumir. Essas restrições são as características da operação sob teste. As características são extraídas das cláusulas da precondição da operação, predicados contidos no corpo da operação e o invariante da máquina. Para a operação *BlueMove* foram identificadas 12 características, que foram extraídas do invariante de estado e das precondições da operação. As características estão listadas a seguir:

(1) $bposn \subseteq 0..8$

(2) $rposn \subseteq 0..8$

(3) $bposn \cap rposn = \emptyset$

(4) $ThreeInRow(bposn) = TRUE \Rightarrow ThreeInRow(rposn) = FALSE$

(5) $ThreeInRow(rposn) = TRUE \Rightarrow ThreeInRow(bposn) = FALSE$

(6) $turn \in Player$

(7) $turn = blue \Rightarrow card(bposn) \leq card(rposn) \wedge card(rposn) \leq (card(bposn) + 1)$

(8) $turn = red \Rightarrow card(rposn) \leq card(bposn) \wedge card(bposn) \leq (card(rposn) + 1)$

(9) $pp \in 0..8$

(10) $pp \notin bposn \cup rposn$

(11) $turn = blue$

(12) $ThreeInRow(rposn) = FALSE$

Etapa 2. **Particionamento do espaço de entrada**

Esta etapa consiste em criar partições no espaço de entrada a partir das características obtidas na etapa anterior. Cada característica divide o espaço de entrada em um ou mais blocos, que depende da estratégia de partição escolhida. Atualmente, BETA suporta duas estratégias de particionamento: *Classes de Equivalência* e *Análise do Valor Limite*. Para a maioria dos casos, ambas as estratégias de particionamento dividem o espaço de entrada em dois blocos, um positivo, que contém valores que respeitam a restrição da característica, e um negativo, que contém valores que não respeitam a restrição da característica.

As duas estratégias de particionamento se diferenciam quando são utilizados intervalos numéricos nas restrições da operação sob teste. Neste caso, a estratégia de particionamento em Classes de Equivalência divide o domínio de entrada em três blocos, um bloco com os valores que antecedem o intervalo, um bloco com os valores que pertencem ao intervalo e um bloco com os valores posteriores ao intervalo; e a estratégia de Análise de Valor Limite divide o domínio em quatro blocos, um bloco que representa a margem esquerda do limite inferior, um bloco que representa a margem direita do limite inferior, um bloco que representa a margem esquerda do limite superior e um bloco que representa a margem direita do limite superior.

Em duas situações, ambas as estratégias de particionamento dividem o domínio de entrada em apenas um bloco. Essas situações são: a tipagem de uma variável, uma vez que valores que não respeitam a tipagem da variável, geralmente, acarretariam em falha de compilação; e restrições do invariante de estado, uma vez que valores que desrespeitam

Tabela 3.1 – Blocos gerados com a estratégia de particionamento em Classes de Equivalência

C	Bloco 1	Bloco 2	Bloco 3
(1)	$bposn \subseteq 0..8$	–	–
(2)	$rposn \subseteq 0..8$	–	–
(3)	$bposn \cap rposn = \emptyset$	–	–
(4)	$ThreeInRow(bposn) = TRUE \Rightarrow$ $ThreeInRow(rposn) = FALSE$	–	–
(5)	$ThreeInRow(rposn) = TRUE \Rightarrow$ $ThreeInRow(bposn) = FALSE$	–	–
(6)	$turn \in Player$	–	–
(7)	$turn = blue \Rightarrow card(bposn) \leq card(rposn) \wedge$ $card(rposn) \leq (card(bposn) + 1)$	–	–
(8)	$turn = red \Rightarrow card(rposn) \leq card(bposn) \wedge$ $card(bposn) \leq (card(rposn) + 1)$	–	–
(9)	$pp \in MININT.. - 1$	$pp \in 0..8$	$pp \in 9..MAXINT$
(10)	$pp \notin bposn \cup rposn$	$\neg(pp \notin bposn \cup rposn)$	–
(11)	$turn = blue$	$\neg(turn = blue)$	–
(12)	$ThreeInRow(rposn) = FALSE$	$\neg(ThreeInRow(rposn) = FALSE)$	–

Legenda: C - Característica.

o invariante não seriam interessantes (a não ser para um sistema de alta integridade) porque a operação sob teste seria executada com um estado interno inválido.

A Tabela 3.1 apresenta o particionamento do espaço de entrada para o exemplo da operação *BlueMove* utilizando a estratégia de particionamento em Classes de Equivalência. Como pode ser visto na tabela, as oito primeiras características dividiram o domínio de entrada em apenas um bloco, uma vez que estas foram extraídas a partir do invariante de estado da máquina *TicTacToe*. A característica (9) dividiu o domínio em três blocos, uma vez que utiliza um intervalo numérico. E as características (10), (11) e (12) dividiram o domínio em dois blocos, um contendo os valores positivos e outro contendo os valores negativos.

Etapa 3. **Geração de fórmulas**

Nesta etapa é preciso combinar os blocos gerados na etapa anterior para gerar fórmulas que representam os casos de teste. Para a combinação dos blocos, a abordagem suporta três critérios de combinação, que são: *All Combinations*, *Each Choice* e *Pairwise*. Todos os três critérios foram apresentados no Capítulo 2. O resultado da combinação dos blocos é um conjunto de fórmulas que representam as situações a serem testadas. Cada fórmula é uma conjunção de predicados do conjunto de blocos e representa um subconjunto do domínio de entrada da operação, onde cada fórmula gerada representa um caso de teste para a operação.

A seguir, serão apresentadas as oito fórmulas geradas ao utilizar o critério de combinação *Pairwise*. Uma vez que as restrições do invariante não são negadas, estas serão representadas pelo nome *INV* com o intuito de simplificar o entendimento das

fórmulas geradas. As fórmulas são:

(1) $INV \wedge pp \in 0..8 \wedge ThreeInRow(rposn) = FALSE \wedge pp \notin bposn \cup rposn \wedge turn = blue$

(2) $INV \wedge pp \in 0..8 \wedge \neg(ThreeInRow(rposn) = FALSE) \wedge \neg(pp \notin bposn \cup rposn) \wedge \neg(turn = blue)$

(3) $INV \wedge pp \in MININT..-1 \wedge ThreeInRow(rposn) = FALSE \wedge pp \notin bposn \cup rposn \wedge \neg(turn = blue)$

(4) $INV \wedge pp \in MININT..-1 \wedge \neg(ThreeInRow(rposn) = FALSE) \wedge pp \notin bposn \cup rposn \wedge turn = blue$

(5) $INV \wedge pp \in MININT..-1 \wedge ThreeInRow(rposn) = FALSE \wedge \neg(pp \notin bposn \cup rposn) \wedge turn = blue$

(6) $INV \wedge pp \in 9..MAXINT \wedge ThreeInRow(rposn) = FALSE \wedge pp \notin bposn \cup rposn \wedge turn = blue$

(7) $INV \wedge pp \in 9..MAXINT \wedge \neg(ThreeInRow(rposn) = FALSE) \wedge pp \notin bposn \cup rposn \wedge \neg(turn = blue)$

(8) $INV \wedge pp \in 9..MAXINT \wedge ThreeInRow(rposn) = FALSE \wedge \neg(pp \notin bposn \cup rposn) \wedge \neg(turn = blue)$

Etapa 4. **Obtenção dos dados de teste**

Obtidas as fórmulas que representam os casos de teste, a etapa seguinte é obter valores para as variáveis do domínio de entrada da operação que satisfazem as restrições dos casos de teste. Em alguns casos, a restrição do caso de teste pode ser inviável, não sendo possível obter um valor que a satisfaça. Quando isso ocorre, o caso de teste é considerado insatisfatível. Um exemplo de caso insatisfatível pode ser visto na fórmula (5), que requer que o parâmetro *pp* pertença ao intervalo entre o menor número inteiro e -1 ($pp \in MININT..-1$) e que pertença a união entre *bposn* e *rposn* ($\neg(pp \notin bposn \cup rposn)$), algo que não é possível uma vez que a interseção entre o intervalo do menor número inteiro a zero e a união entre *bposn* e *rposn* é vazia. A fórmula (8) também é insatisfatível. Quando diferentes combinações de valores satisfazem uma fórmula, qualquer uma pode ser escolhida para o teste.

Uma maneira tradicional de se obter os valores para os casos de teste, e a maneira sugerida pela a abordagem, é aplicar as fórmulas em um solucionador de restrições, que gera valores que satisfazem as condições das fórmulas. A ferramenta BETA faz uso do

ProB como solucionador de restrições. A forma que a ferramenta BETA utiliza o *ProB* para gerar os dados de teste será abordada na próxima seção.

Ao utilizar um solucionador de restrições é possível obter os seguintes dados: para a fórmula (1) $bposn = \emptyset$, $rposn = \emptyset$, $turn = blue$ e $pp = 0$; para a fórmula (2) $bposn = \{3, 4, 5\}$, $rposn = \{0, 1, 2\}$, $turn = red$ e $pp = 0$; para a fórmula (3) $bposn = \emptyset$, $rposn = \emptyset$, $turn = red$ e $pp = -1$; para a fórmula (4) $bposn = \{3, 4\}$, $rposn = \{0, 1, 2\}$, $turn = blue$ e $pp = -1$; para a fórmula (6) $bposn = \emptyset$, $rposn = \emptyset$, $turn = blue$ e $pp = 9$; e para a fórmula (7) $bposn = \{3, 4, 6\}$, $rposn = \{0, 1, 2\}$, $turn = red$ e $pp = 9$. Como já foi mencionado, as fórmulas (5) e (8) são insatisfatíveis, por isso não é possível obter dados que as satisfaçam. Dessa forma, elas não são consideradas nas etapas seguintes.

De posse dos dados de teste é criada, então, uma especificação de casos de teste, que traz as informações e os valores para as variáveis do domínio de entrada da operação em cada caso de teste. A especificação servirá como referência para o desenvolvedor poder implementar e executar os testes.

Etapa 5. **Obtenção dos resultados esperados (Oráculos de teste)**

Outros artefatos importantes para o desenvolvimento de testes, além dos dados de entrada, são os oráculos, que determinam a falha ou sucesso de um teste. Esta etapa consiste em obter os resultados esperados em um teste para verificar se, após a execução da operação sob teste, estes foram obtidos. A abordagem BETA propõe um método para obter os resultados esperados. Este método consiste em animar a máquina abstrata ao qual a operação sob teste faz parte, passando os dados de teste que foram gerados para o caso de teste. Ao animar a máquina, a operação sob teste é executada e os valores das variáveis do domínio de entrada após a execução da operação são calculados. A abordagem sugere o uso do *ProB* para esse fim. O método proposto não pode ser aplicado em casos de teste negativos, uma vez que não está especificado na máquina qual deve ser o comportamento do sistema sob essas condições. O método também não pode ser aplicado nos casos em que são utilizados não determinismos na especificação, pois, nesses casos, o *ProB* pode retornar qualquer valor que satisfaça as restrições, logo, não se pode esperar que o programa sob teste retorne o mesmo valor. Para esses casos, a abordagem deixa a cargo do engenheiro de testes projetar os oráculos de teste, seguindo os critérios estabelecidos para essas situações.

Para o exemplo da operação *BlueMove*, apenas o caso de teste representado pela fórmula (1) é positivo, portanto é o único caso em que o método proposto pela abordagem pode ser utilizado. Ao aplicar os dados de teste obtidos na etapa anterior no *ProB*, os seguintes valores são calculados após a execução da operação *BlueMove*: $bposn = \{0\}$, $rposn = \emptyset$ e $turn = red$.

Etapa 6. **Implementação dos testes concretos**

De posse dos dados e oráculos de teste, esta última etapa consiste em implementar os casos de teste, que é a concretização de todo o processo apresentado até aqui. BETA gera casos de teste a partir de uma máquina abstrata, por causa disso, podem ser gerados dados abstratos para os casos de teste. Dessa forma, é nesta etapa que os dados gerados por BETA devem ser refinados, de forma manual, seguindo as decisões de projeto que foram tomadas, para que possam ser utilizados na implementação dos testes.

Em geral, o teste concreto pode ser divido em três partes: a primeira consiste em levar o estado do programa sob teste ao estado requisitado pelo caso de teste, ou seja, fazer com que as variáveis de estado do programa fiquem com os valores apresentados na especificação de casos de teste; a segunda parte consiste na chamada da operação sob teste passando os valores dos parâmetros de entrada; e a terceira parte consiste nos oráculos de teste, com assertivas que verificam se os valores esperados foram obtidos. A Figura 3.2 apresenta a implementação do caso de teste definido pela fórmula (1). Para a implementação deste teste, foram utilizadas como base as decisões de projeto inseridas na implementação B *TicTacToeI*, apresentada na Figura 2.6, e o código em *C* apresentado na Figura 2.7.

Entre as linhas 8 e 19 da Figura 3.2 é feita a inicialização do estado e a atribuição das variáveis de estado, considerando os valores refinados dos dados gerados para o caso de teste. Na abordagem, é considerado que as variáveis de estado possuem livre acesso ou que existem métodos acessadores (como métodos *setters* e *getters* utilizados em programação orientada a objetos) para fazer a atribuição. Entre as linhas 21 e 22 é feita a chamada da operação sob teste com os seus parâmetros de entrada. Entre as linhas 24 e 33 estão os oráculos de teste que verificam se as variáveis de estado ficaram com os valores esperados.

3.2 Ferramenta

A ferramenta BETA foi desenvolvida para automatizar parte do processo da abordagem. A ferramenta automatiza as etapas *1* a *4*, além de parte da etapa *5*, da abordagem. A ferramenta BETA recebe como entrada uma máquina abstrata B e gera como saída uma especificação de casos de teste. A ferramenta foi desenvolvida em *Java* e faz uso do *ProB* para gerar os dados de teste. Mais informações sobre a ferramenta e detalhes técnicos do desenvolvimento podem ser encontrados em (MATOS; MOREIRA, 2012), (MATOS; MOREIRA, 2013) e no site <http://www.beta-tool.info/>, onde a ferramenta pode ser obtida.

As etapas *1*, *2* e *3* são feitas de forma simples pela ferramenta, necessitando apenas que o usuário escolha a operação a ser testada e selecione uma estratégia de particionamento (Classes de Equivalência ou Análise de Valor Limite) e um critério de combinação (*All Combinations*, *Pairwise* ou *Each Choice*). Feito isso, a ferramenta gera

```
1  /**
2  * Test Case 1
3  * Formula: ThreeInRow(rposn) = FALSE & pp /: bposn \/ rposn & pp :
       0..8 & turn = blue
4  * Positive Test
5  */
6  void TicTacToe_BlueMove_test_case_1(CuTest* tc)
7  {
8      TicTacToe__INITIALISATION();
9
10     TicTacToe__blue_turn = true;
11     TicTacToe__board[0] = 0;
12     TicTacToe__board[1] = 0;
13     TicTacToe__board[2] = 0;
14     TicTacToe__board[3] = 0;
15     TicTacToe__board[4] = 0;
16     TicTacToe__board[5] = 0;
17     TicTacToe__board[6] = 0;
18     TicTacToe__board[7] = 0;
19     TicTacToe__board[8] = 0;
20
21     int32_t pp = 0;
22     TicTacToe__BlueMove(pp);
23
24     CuAssertTrue(tc, TicTacToe__blue_turn == false);
25     CuAssertTrue(tc, TicTacToe__board[0] == 1);
26     CuAssertTrue(tc, TicTacToe__board[1] == 0);
27     CuAssertTrue(tc, TicTacToe__board[2] == 0);
28     CuAssertTrue(tc, TicTacToe__board[3] == 0);
29     CuAssertTrue(tc, TicTacToe__board[4] == 0);
30     CuAssertTrue(tc, TicTacToe__board[5] == 0);
31     CuAssertTrue(tc, TicTacToe__board[6] == 0);
32     CuAssertTrue(tc, TicTacToe__board[7] == 0);
33     CuAssertTrue(tc, TicTacToe__board[8] == 0);
34 }
```

Figura 3.2 – Exemplo de teste concreto em *C* para a operação *BlueMove*

as fórmulas que representam os casos de teste. Para gerar os valores que satisfazem as fórmulas, a ferramenta utiliza o *ProB*. Fazendo uso de um pequeno artifício, a ferramenta cria uma máquina B auxiliar para ser animada no *ProB* e gerar os valores que satisfazem as fórmulas.

A máquina B auxiliar contém uma operação para cada caso de teste. Cada operação da máquina B auxiliar tem como parâmetros de entrada as variáveis do espaço de entrada da operação sob teste, que são os parâmetros de entrada da operação e as variáveis de estado, e a fórmula do caso de teste como precondição. Além disso, a máquina auxiliar também possui os mesmos conjuntos, constantes e propriedades da máquina da operação sob teste. A Figura 3.3 apresenta um trecho da máquina B auxiliar criada para o exemplo da operação *BlueMove*. A primeira operação da máquina auxiliar corresponde ao caso de teste representado pela fórmula (1).

```
1  MACHINE
2     TestsForOp_BlueMove_From_TicTacToe
3  SETS
4     Player = {blue, red};
5     Result = {blue_win, red_win, draw}
6  CONSTANTS
7     WinnerRows, ThreeInRow
8  PROPERTIES
9      WinnerRows ⊆ ℙ(0..8) ∧
10     WinnerRows = {{0,1,2}, {3,4,5}, {6,7,8}, {0,3,6},
11                   {1,4,7}, {2,5,8}, {0,4,8}, {6,4,2}} ∧
12     ThreeInRow ∈ ℙ(0..8) → BOOL ∧
13     ThreeInRow =
14       λps.(ps ∈ ℙ(0..8) ∧
15             ∃wn.(wn ∈ WinnerRows ∧ wn ⊆ ps)|TRUE) ∪
16       λps.(ps ∈ ℙ(0..8) ∧
17             ¬(∃wn.(wn ∈ WinnerRows ∧ wn ⊆ ps))|FALSE)
18 OPERATIONS
19    BlueMove_test1(rposn, bposn, turn, pp) =
20    PRE
21       pp ∈ 0..8 ∧ ThreeInRow(rposn) = FALSE ∧ bposn ⊆ 0..8 ∧ pp ∉ bposn ∪
           rposn ∧ turn = blue ∧ rposn ⊆ 0..8 ∧ turn ∈ Player ∧ bposn ∩ rposn
            = ∅
22    THEN
23       skip
24    END
25    /* ... */
26 END
```

Figura 3.3 – Máquina B auxiliar para a operação *BlueMove*

Ao animar a máquina B auxiliar, o *ProB* gera valores, quando possível, que satisfazem as restrições das fórmulas. De posse desses valores, a ferramenta gera uma especificação de casos de teste. Além disso, a ferramenta BETA consegue, atualmente, automatizar parte do passo (5) da abordagem, calculando os valores esperados para as variáveis de estado nos casos de teste positivos, fazendo uso do método apresentado pela abordagem e do *ProB* para esse fim. Esses valores também se encontram na especificação de casos de teste gerada pela ferramenta. Atualmente, a ferramenta gera a especificação em dois formatos, *HTML*, que é de fácil leitura, e *XML*, que pode ser utilizado por uma outra ferramenta. A Figura 3.4 apresenta uma parte da especificação de casos de teste no formato *HTML* gerada pela ferramenta para a operação *BlueMove*.

3.3 Trabalhos Relacionados

Com o intuito de encontrar diretrizes para possíveis melhorias e evoluções na abordagem e ferramenta BETA, este trabalho fez uma revisão bibliográfica de trabalhos relacionados a teste de software e métodos formais. Em (MATOS; MOREIRA, 2012), trabalho que reformulou a abordagem e desenvolveu a ferramenta BETA, também foi feita uma revisão bibliográfica que, de forma mais profunda, caracterizou e classificou trabalhos que apresentavam abordagens de geração de testes a partir de modelos formais. De modo a fornecer uma nova visão sobre trabalhos relacionados a teste de software e

Test Case 1 (Positive)

Formula: *ThreeInRow(rposn) = FALSE & pp /: bposn \/ rposn & pp : 0..8 & turn = blue*

Preamble:

No preamble generated

Input data:

State	
Variable	Value
bposn	{}
rposn	{}
turn	blue
Input Values	
Parameter	Value
pp	0

Expected results:

State	
Variable	Value
bposn	{0}
rposn	{}
turn	red
Return Variables	
Return Variable	Value
This operation has no return variables	

Figura 3.4 – Especificação de casos de teste em *HTML* gerada por BETA

métodos formais, não foram incluídos, neste trabalho, os trabalhos que foram revisados em (MATOS; MOREIRA, 2012).

Em (STOCKS; CARRINGTON, 1996) e (MACCOLL; CARRINGTON, 1998) é apresentado o *Test Template Framework* (*TTF*), uma abordagem para geração de testes a partir de especificações formais escritas na *Notação Z* (SPIVEY, 1989). O *TTF* gera testes de unidade fazendo uso de particionamento do espaço de entrada. Inicialmente, é feita, para cada operação da especificação Z, a definição do *Espaço de Entrada* (EE), que é o conjunto de todos os possíveis valores das entradas e variáveis de estado da operação, e do *Espaço de Entrada Válido* (EEV), que é o subconjunto do EE pra o qual a operação foi definida, que, em outras palavras, é o conjunto dos valores que respeitam as precondições da operação.

Em seguida, o EEV é dividido em classes de equivalência chamadas de *classes de teste* (ou *templates* de teste, originalmente), que são subdomínios do EEV. A divisão é feita através da aplicação de estratégias de teste (também chamadas de táticas de teste ou estratégias de particionamento). O *TTF* deixa a cargo do engenheiro de testes decidir quais estratégias de teste aplicar, entretanto, são sugeridas algumas: *Forma Normal Disjuntiva* (FND), que consiste em escrever os predicados da especificação na FND e transformar cada predicado em uma classe de teste; e o *Particionamento Padrão* (PP), que consiste em fazer o particionamento com base em operadores matemáticos específicos, como operadores de conjuntos, por exemplo. O *TTF* proporciona uma flexibilidade

ao permitir que o engenheiro de testes decida quais estratégias de teste (estratégias de particionamento) utilizar, podendo incluir suas próprias estratégias. As classes de teste podem ser subdivididas em outras classes de teste através da aplicação de outras estratégias de teste. Este processo pode continuar até o engenheiro de testes decidir que existe um número razoável de classes.

Feita a divisão, as classes de teste são dispostas em uma *árvore de testes*, que é uma árvore hierárquica em que a raiz é o EEV, e seus nós filhos são as classes de teste geradas a partir de sua divisão. Cada classe de teste que foi subdividida também possui nós filhos. No *TTF*, os casos de teste são obtidos a partir dos nós folhas da árvore de testes, que são aquelas classes de teste que não foram subdivididas. Em uma etapa intermediária, a árvore de testes é podada, que consiste em eliminar da árvore todas as classes de teste que são vazias devido a contradições, que, em outras palavras, consiste em remover da árvore todos os casos que são insatisfatíves. Removidas as classes de teste insatisfatíveis, a última etapa consiste em gerar dados de entrada para os casos de teste, que são os nós folhas que sobraram na árvore de testes. O *TTF* também abrange os oráculos de teste, consistindo em fazer uso das pós-condições da especificação Z pra determinar os resultados esperados.

O *TTF* foi automatizado em (CRISTIÁ; MONETTI, 2009) com a criação da ferramenta *Fastest*. A ferramenta implementa as estratégias de teste FND e PP, além de propor uma nova estratégia, a de Tipos Livres (TL). Na estratégia TL, o particionamento é feito com base em conjuntos enumerados, em que são criadas classes de teste para cada valor do conjunto. Assim como o *TTF*, o *Fastest* também flexibiliza ao deixar o usuário escolher quais estratégias de teste utilizar ou adicionar as suas próprias. Todos os artefatos gerados pelo *Fastest*, como as classes de teste e os casos de teste, também são escritos na Notação Z. Em (CRISTIÁ et al., 2011) é apresentada uma linguagem de refinamento de dados para o *TTF*, chamada de *Fastest Test Case Refinement Language* (*FTCRL*). Nesta linguagem, são escritas regras de refinamento que transformam os casos de teste abstratos gerados pelo *Fastest* em casos de teste concretos escritos na linguagem de programação do sistema sob teste.

Em (LIU; YANG, 2009) é proposto o uso do critério de teste *MC/DC* (*Modified Condition/Decision Coverage*) para geração de testes a partir de gráficos de estado e especificações escritas na notação do Método B. Este critério requer que cada ponto de entrada e saída do programa sob teste seja invocado pelo menos uma vez; que cada decisão (predicado) do programa sob teste assuma todos os seus possíveis resultados; que cada condição (cláusula) em uma decisão do programa sob teste assuma todos os seus possíveis resultados; e que cada condição em uma decisão seja mostrada para afetar de forma independente o resultado da decisão. De forma sucinta, o objetivo do *MC/DC* é testar o efeito de cada condição em um decisão de forma isolada, isso garante que a condição seja

testada sem que seu resultado seja mascarado por outra condição. O *MC/DC* é um dos critérios de teste recomendados para sistemas de alta integridade, como sistemas para a área da aviação, por exemplo.

Em (MALIK; LILIUS; LAIBINIS, 2009a) é apresentada uma abordagem que gera casos de teste a partir de modelos em *Event-B* (ABRIAL, 2010). Nesta abordagem cenários de teste devem ser fornecidos pelo engenheiro de teste. Um cenário de teste é uma possível sequencia de execução de eventos do sistema sob teste. Os cenários de teste são, então, traduzidos em expressões *CSP* (HOARE, 1978). Na medida que o modelo em *Event-B* vai sendo refinado (o refinamento no *Event-B* é mais abrangente que no Método B, permitindo, também, inserir novas funcionalidades, por exemplo), os cenários de teste em *CSP* também vão sendo através de transformações sintáticas. Terminado os refinamentos, no modelo e cenários, os cenários de teste são utilizados para guiar a animação do modelo em *Event-B*. Após a execução de cada evento do modelo, informações sobre o estado do sistema sob teste são armazenadas. Com esse processo, cada cenário de teste é transformado em um caso de teste, que consiste em uma sequencia de execução de eventos e o estado após a execução de cada evento (pós-condições). Esta abordagem é estendida em (MALIK; LILIUS; LAIBINIS, 2009b) com a proposta de uma abordagem para geração de código em *Java*, a partir do modelo do sistema em *Event-B*, e testes executáveis em *JUnit*, a partir dos casos de teste gerados. No geral, esta abordagem é feita de forma manual, mas possui diretrizes que a permitem ser automatizada.

Em (AYDAL et al., 2009) é proposta uma metodologia que faz uso de testes baseados em modelos formais para a validação de modelos de sistema. Nessa metodologia, o sistema é modelado de duas formas, um modelo principal, que vai guiar a implementação do sistema, e um modelo de teste, que modela uma parte específica do sistema para ser testada. Ambos os modelos são feitos a partir dos mesmos requisitos do sistema. O modelo de teste é desenvolvido em *Alloy* (JACKSON, 2012). A partir do modelo de teste, são geradas especificações de casos de teste positivo (que respeitam as precondições) e negativo (que desrespeitam as precondições) em termos de precondições e pós-condições do modelo *Alloy*. Essas especificações de casos de teste são, então, negadas em assertivas *Alloy*, e essas são verificadas em um *Analisador Alloy*, que tenta gerar contra exemplos para elas. Os contra exemplos gerados são os casos de teste abstratos utilizados para verificar o modelo principal do sistema. Este modelo principal pode ser desenvolvido em outra linguagem de especificação formal, por isso os contra exemplos devem ser adaptados para a linguagem deste modelo. Para facilitar esse processo, o *Analisador Alloy* gera os casos de teste em formato *XML*, que pode ser passado para um adaptador ou gerador que traduz os casos de teste para a linguagem do modelo principal. Os casos de teste gerados podem ajudar a encontrar inconsistências no modelo principal que vai guiar a implementação do sistema, sendo uma alternativa para a validação formal.

3.3.1 Discussões e Conclusões

A abordagem *TTF* (STOCKS; CARRINGTON, 1996; MACCOLL; CARRINGTON, 1998), possui várias semelhanças com a abordagem BETA. Além de também fazer uso de particionamento do espaço de entrada na geração dos testes, várias das etapas da *TTF* são análogas as etapas de BETA. Uma das diferenças entre as duas abordagens é a que a *TTF* considera apenas o espaço de entrada válido, que é o espaço de entrada que respeita as precondições da operação sob teste, enquanto BETA considera todo o espaço de entrada. Isso faz com que a abordagem *TTF* gere apenas casos de teste positivos, enquanto BETA oferece a vantagem de também gerar casos de teste negativos. Outra diferença entre as duas abordagens é a que na *TTF* o particionamento é feito no EEV como um todo, enquanto em BETA é feito com base em características de forma separada para, só após, fazer a junção com os critérios de combinação. Um dos diferenciais da abordagem *TTF* é a maior flexibilização na etapa de particionamento, permitindo que o engenheiro de testes utilize seus próprios critérios de particionamento e que vários particionamentos possam ser feitos. Flexibilizar a etapa de particionamento de BETA permitiria variar mais os dados de teste e, consequentemente, gerar casos de teste melhores.

Em BETA os casos de teste insatisfatíveis são detectados na etapa de obtenção dos dados de teste, já no *TTF* os casos insatisfatíveis são detectados antes da obtenção dos dados, através da busca de contradições nas classes de teste. Em (CRISTIA; ALBERTENGO; MONETTI, 2010) é apresentado como pode ser feito essa busca, que de forma sucinta consiste em escrever teoremas que definem certos tipos de contradições e, depois, verificar se as classes de teste geradas correspondem com esses teoremas. Quando isso ocorre, significa que existe uma contradição na classe de teste e, por isso, ela é insatisfatível. A vantagem dessa abordagem é que a busca por contradições acaba sendo mais rápida do que a feita com um solucionador de restrições, como foi mostrado no artigo. Em contrapartida, essa abordagem só busca as contradições que foram definidas nos teoremas, logo, as contradições que não foram definidas nos teoremas não serão encontradas com essa abordagem. Incluir uma etapa para busca de contradições em BETA poderia ser interessante porque evitaria gastar tempo desnecessário com o solucionador de restrições em alguns casos insatisfatíveis.

A ferramenta *Fastest* (CRISTIÁ; MONETTI, 2009) automatiza a abordagem *TTF*. A ferramenta BETA oferece uma interface gráfica simples e fácil de utilizar, sendo necessário apenas que o usuário selecione a estratégia de particionamento, o critério de combinação e o formato da especificação para a ferramenta gerar a especificação de casos de teste. Já a ferramenta *Fastest* não oferece a mesma usabilidade, fornecendo uma interface via linhas de comando, fazendo com que o usuário precise saber vários comandos para conseguir gerar os casos de teste, uma vez que cada etapa da abordagem é feita através de um comando diferente. A linguagem de refinamento *FTCRL* (CRISTIÁ et al.,

2011) permite que regras de refinamento para transformar os dados abstratos dos casos de teste gerados pelo *Fastest* em dados concretos, traduzindo os dados diretamente para a linguagem de implementação do sistema sob teste. Dessa forma, a linguagem *FTCRL* provê um mecanismo para o processo de refinamento de dados na *Notação Z*. Em BETA isso poderia ser feito com a criação de uma linguagem semelhante a *FTCRL* ou fazendo uso do próprio módulo de implementação, ou refinamento, do módulo B do sistema sob teste.

Em (LIU; YANG, 2009) foi proposto o uso do critério lógico *MC/DC* para gerar testes a partir de gráficos de estado e especificações em B. O critério *MC/DC* se destaca por ser recomendado para teste de sistemas de alta integridade, algo que mostra que o critério ajuda a criar testes de qualidade que são essenciais para sistemas desse nível. Atualmente, se encontra em desenvolvimento a inclusão do critério *MC/DC* em BETA, algo que contribuirá para a geração de testes com mais qualidade, além de tornar BETA mais atrativa para o desenvolvimento de sistemas de alta integridade.

A abordagem apresentada em (MALIK; LILIUS; LAIBINIS, 2009a) gera testes a partir de especificações em *Event-B*, uma extensão do Método B. O *Event-B* é voltado para modelagem de sistemas baseados em eventos, como sistemas reativos, sistemas embarcados e sistemas *web*, por exemplo, dessa forma, abordagens de teste que se baseiam em *Event-B* vem ganhando muito interesse. Por ser uma extensão do Método B, o *Event-B* apresenta uma série de características em comum, o que permite que a abordagem BETA possa ser adaptada para o método futuramente.

A metodologia proposta em (AYDAL et al., 2009) faz uso de testes gerados a partir de especificações em *Alloy* para validar a especificação que vai ser utilizada para modelar um sistema, ao invés de validar o sistema em si. Este é um dos trabalhos que mostram que testes também podem contribuir na validação de modelos de software, dessa forma, este é um campo de atuação que pode ser explorado em uma abordagem como BETA futuramente. Os trabalhos discutidos nessa seção apresentaram diferentes visões e métodos que podem ser explorados em BETA, podendo contribuir com a evolução da abordagem e ferramenta.

4 Contribuições

Neste capítulo são apresentadas as contribuições propostas e desenvolvidas neste trabalho para a abordagem e ferramenta BETA. Inicialmente, são apresentadas as *Estratégias de Oráculos de Teste* propostas para a abordagem BETA, que têm o objetivo de trazer mais flexibilidade e uma melhor definição para a etapa de obtenção dos resultados esperados (oráculos de teste) da abordagem. Em seguida, é apresentado o *Gerador de Scripts de Teste* desenvolvido para a ferramenta BETA, que tem o objetivo de automatizar parte da etapa de implementação dos testes concretos.

4.1 Estratégias de Oráculos de Teste

A penúltima etapa da abordagem BETA (MATOS; MOREIRA, 2012; MATOS; MOREIRA, 2013) consiste na obtenção dos resultados esperados nos casos de teste. Estes resultados são verificados em oráculos de teste, que são mecanismos que determinam o sucesso ou a falha de um teste. Em BETA, é proposto um método para a obtenção dos resultados esperados. Este método consiste em aplicar os dados de teste em um animador de especificações, como o *ProB* por exemplo, e executar a operação sob teste. Como resultado, o animador calcula os resultados esperados após a execução da operação sob teste. Esses resultados devem, então, ser verificados através de assertivas nos testes concretos. O método proposto só pode ser aplicado em casos de teste positivos, pois não é possível aplicar os dados de testes negativos no animador, uma vez que estes não respeitam as precondições da operação sob teste. Dessa forma, os resultados esperados em um caso de teste negativo não podem ser determinados a partir da própria especificação (máquina abstrata). Nesse caso, a abordagem BETA deixa a cargo do engenheiro de testes determinar o método que vai aplicar em oráculos para testes negativos, seguindo seus próprios critérios.

Apesar de BETA propor um método para os oráculos de testes positivos, este método não oferece flexibilidade e, também, não especifica, de forma precisa, quais são os elementos que devem ser verificados nos oráculos. Além disso, não são apontadas diretrizes para os oráculos em testes negativos. Dessa forma, este trabalho está propondo estratégias de oráculos de teste para a abordagem BETA. As estratégias propostas trazem uma maior flexibilidade para a etapa de obtenção dos resultados esperados, definem de forma precisa quais são os elementos que devem ser verificados nos oráculos de teste e podem ser aplicadas em testes negativos (com a limitação de que é necessário ter informações externas à especificação para serem utilizadas nessas situações).

As estratégias de oráculos de teste propostas neste trabalho foram baseadas nas

estratégias apresentadas em (LI; OFFUTT, 2014) (discutidas na Seção 2.1.4 do Capítulo 2). As estratégias apresentadas em (LI; OFFUTT, 2014) são inseridas no contexto de teste de sistemas orientados a objetos. Por esse motivo, este trabalho adaptou as estratégias de oráculos para o contexto de teste de unidade aplicado ao Método B. As estratégias verificam execuções anormais do programa sob teste, a preservação do invariante de estado e os valores resultantes nas variáveis de estado e retornos da operação sob teste. As estratégias propostas estão listadas abaixo:

1. *Exceções*: consiste em verificar a ocorrência de finalizações inesperadas no programa após a execução da operação sob teste, como o levantamento de exceções ou erros;
2. *Invariante de estado*: consiste em verificar se o invariante de estado (invariante da máquina abstrata) foi preservado após a execução da operação sob teste;
3. *Variáveis de estado*: consiste em verificar se as variáveis de estado ficaram com os valores esperados após a execução da operação sob teste;
4. *Valores de retorno*: consiste em verificar se os valores retornados pela operação sob teste foram os esperados.

As estratégias de oráculos de teste propostas podem ser aplicadas individualmente ou em conjunto para uma melhor verificação. Além disso, as estratégias são adaptáveis para diferentes contextos. A seguir, são feitas discussões e conclusões acerca da estratégias de oráculos de teste propostas.

4.1.1 Discussões e Conclusões

As estratégias de oráculos de teste propostas verificam diferentes elementos, como o invariante de estado, variáveis de estado, valores de retorno e interrupções na execução. Algumas das estratégias propostas em (LI; OFFUTT, 2014) consistiam na união de duas ou mais estratégias simples (que verificavam apenas um tipo de elemento). Quanto mais elementos são verificados, mais preciso é o oráculo, e, consequentemente, mais defeitos ele consegue detectar. A relação entre a precisão e o número de defeitos detectados já foi mostrada em vários trabalhos: (BRIAND; PENTA; LABICHE, 2004), (SHRESTHA; RUTHERFORD, 2011), (STAATS; WHALEN; HEIMDAHL, 2011) e (XIE; MEMON, 2007). Para manter a flexibilidade, este trabalho optou por propor estratégias que verificam apenas um tipo de elemento, ficando a cargo do engenheiro de testes fazer uso de mais de uma estratégia para aumentar a precisão dos oráculos de teste no seu projeto.

As estratégias de verificação de variáveis de estado e valores de retorno (estratégias 3 e 4, respectivamente) são as que apresentam melhores resultados na detecção de erros. Isso ocorre porque elas trabalham com a verificação direta dos resultados esperados

da operação sob teste. Dessa forma, as duas estratégias dão mais garantias na verificação da execução correta da operação sob teste. Por causa disso, a união dessas duas estratégias de oráculos de teste também é conhecida como *oráculo máximo* (STAATS; WHALEN; HEIMDAHL, 2011) ou *oráculo preciso* (BRIAND; PENTA; LABICHE, 2004). Devido a sua precisão, o oráculo máximo faz um maior número de verificações em comparação a cada uma das estratégias de oráculos de teste propostas de forma individual. Esse fato faz com que o custo do oráculo máximo seja maior que os outros, como foi mostrado em (LI; OFFUTT, 2014).

Uma alternativa de menor custo que o oráculo máximo é a verificação do invariante de estado (estratégia 2). O menor custo decorre do fato de que o invariante de estado deve ser sempre mantido, portanto, em cada caso de teste sempre vai ser feita a mesma verificação, o que permite a reutilização de código. Apesar do seu resultado não ser tão efetivo quanto o oráculo máximo, como foi mostrado em (BRIAND; PENTA; LABICHE, 2004), a verificação do invariante de estado ainda apresenta resultados consideráveis. Em (LI; OFFUTT, 2014) foi mostrado que a estratégia de verificação do invariante conseguiu detectar em torno de 80% dos defeitos detectados com o oráculo máximo, o que faz com que a estratégia seja uma boa alternativa quando o engenheiro de testes está disposto a sacrificar um pouco da qualidade para diminuir os custos.

A estratégia de exceções (estratégia 1) é a que possui o menor custo entre todas as estratégias. Entretanto, é a estratégia que apresenta o pior resultado na detecção de defeitos, como foi mostrado em (SHRESTHA; RUTHERFORD, 2011) e (LI; OFFUTT, 2014), o que sugere que a estratégia de exceções não é suficiente para a detecção de defeitos. Entretanto, ao se analisar apenas os casos de teste negativos, a estratégia de exceções pode ser considerada uma boa alternativa, dependendo dos critérios do engenheiro de testes.

Na programação de computadores é comum fazer uso de exceções para detectar e processar erros ou situações excepcionais (ABRAN et al., 2004). O tratamento de exceções mantem o sistema robusto, garantindo que ele continua executando corretamente mesmo com a ocorrência de situações não esperadas. As precondições de uma operação em uma especificação B determinam as condições que devem ser satisfeitas para que a operação possa ser executada de forma correta. Ao se quebrar as precondições de uma operação, algo que é feito em testes negativos, não se pode esperar uma correta execução desta operação, logo, seus resultados não podem ser determinados. Portanto, a quebra das precondições é uma situação excepcional na operação sob teste. Mesmo não estando na especificação B, o tratamento de situações excepcionais é um requisito comum no desenvolvimento de sistemas robustos. Por isso, pode ser esperado que uma operação lavante uma exceção quando suas precondições não forem respeitadas. Dessa forma, a estratégia de exceções pode ser utilizada para verificar que uma operação levantou uma exceção quando assim for esperado, ou seja, quando suas precondições forem quebradas.

A estratégia de verificação do invariante de estado também se apresenta como uma alternativa para os casos de teste negativos. O invariante de estado determina as condições que sempre devem ser válidas no sistema sob teste. Isso faz com que garantir o invariante seja um requisito para o sistema como um todo. Com isso, pode ser desejado que, mesmo em situações não esperadas, uma operação sempre garanta o invariante. Logo, em casos de teste negativos, o invariante de estado pode ser verificado para garantir que a operação manteve a consistência do sistema. Outra técnica utilizada para tratar situações não esperadas em um sistema é retornar valores neutros ou constantes para simbolizar a ocorrência de um erro (ABRAN et al., 2004). Quando esta técnica for utilizada, as estratégias de verificação de variáveis de estado e valores de retorno podem ser aplicadas para verificar se esses valores neutros ou constantes foram retornadas nos casos de teste negativos.

As estratégias de oráculos de teste apresentadas neste capítulo contribuíram para diminuir as limitações existentes na etapa de definição dos oráculos de teste de BETA, pois definem bem quais elementos devem ser verificados e oferecem flexibilidade, permitindo que o engenheiro de testes escolha quais elementos verificar, podendo definir a precisão que deseja aplicar no projeto, algo que a abordagem não oferecia anteriormente. As estratégias de oráculos de teste propostas também oferecem diretivas para os oráculos em casos de teste negativos, entretanto, dependem de informações e decisões externas à especificação B para que possam ser utilizadas.

Uma possível solução para os oráculos de teste em casos de teste negativos seria especificar em B o comportamento do sistema em situações não esperadas. Isso poderia ser feito na especificação principal, utilizada como base no desenvolvimento do sistema e nos testes, ou em uma especificação complementar criada apenas para essa finalidade. Dessa forma, seria possível determinar de forma precisa o comportamento esperado do sistema nessas situações. Além disso, ao especificar em B o comportamento excepcional do sistema, é possível aplicar o método proposto em (MATOS; MOREIRA, 2012) para a obtenção dos resultados esperados. Portanto, especificar em B o comportamento do sistema em situações negativas (que quebram as precondições) pode contribuir para a qualidade dos testes.

Como foi discutido, a estratégia de exceções (1) é a estratégia mais fraca, mas que pode ser uma boa alternativa quando os requisitos (não especificados em B, necessariamente) determinarem que exceções devem ser levantadas em certas ocasiões. A estratégia de verificação do invariante de estado (2) é uma estratégia intermediária que apresenta resultados razoáveis, podendo ser uma boa alternativa quando se quer uma economia no projeto dos oráculos de teste, principalmente nos casos em que o invariante de estado é simples. E as estratégias de verificação das variáveis de estado e valores de retorno (3 e 4, respectivamente) são estratégias que apresentam bons resultados, e a união das

duas estratégias, que é conhecido como oráculo máximo ou preciso, é a que apresenta os melhores resultados na detecção de defeitos, sendo a melhor opção para ser aplicada na abordagem.

4.2 Gerador de Scripts de Teste

A última etapa da abordagem BETA consiste na implementação dos testes concretos. Nesta etapa, o desenvolvedor deve pegar os dados de teste contidos na especificação de casos de teste gerada por BETA, e traduzi-los em um código de teste ou *script* de teste que automatiza a execução e verificação dos testes. Como foi apresentado no Capítulo 3, cada caso de teste implementado pode, basicamente, ser dividido em três partes. Na primeira parte, são realizadas as atribuições nas variáveis de estado para que o módulo fique no estado requisitado pelo teste. Na segunda parte, é feita a chamada da operação sob teste, onde são passados os valores de entrada gerados para o caso de teste. Por último, são colocados os oráculos que determinam o sucesso ou a falha do teste. Esta estrutura comumente se repete nos testes.

Atualmente, a etapa de implementação dos testes concretos é feita de forma manual na abordagem BETA. Esta etapa pode demandar muito tempo e esforço, principalmente nos casos em que for gerado um grande número de testes e o sistema sob teste for complexo. Pelo fato da ferramenta BETA também gerar a especificação de casos de teste em formato *XML*, um formato simples que possui fácil leitura para programas de computador, e os testes possuírem a mesma estrutura, é possível automatizar o processo (ou parte) de implementação dos testes concretos. Dessa forma, este trabalho contribuiu com a ferramenta BETA ao desenvolver um gerados de *scripts* de teste que automatiza parte da etapa de implementação.

O gerador de traduz os dados contidos na especificação de casos de teste gerada por BETA em um *script* de teste parcial. Este *script* deve, então, ser adaptado para adequar os dados de teste abstratos em dados concretos, seguindo as decisões de projeto tomadas, além de outras adequações, como informações adicionais sobre os oráculos de teste, por exemplo. Depois de adaptado, o *script* de teste pode ser executado para realizar os testes da operação sob teste. O gerador também automatiza, de forma parcial, a implementação dos oráculos de teste com base nas estratégias propostas neste trabalho. O gerador foi desenvolvido de forma modular para ser adaptável para diferentes contextos e projetos. Detalhes técnicos e de implementação do gerador de *scripts* de teste são dados a seguir.

4.2.1 Detalhes técnicos e implementação

Para o desenvolvimento do gerador de *scripts* de teste foi utilizada a linguagem de metaprogramação *Rascal* (KLINT; STORM; VINJU, 2009). A linguagem *Rascal* foi escolhida porque traz recursos que integram análise, transformação e geração de código em apenas uma linguagem, sendo ideal para o desenvolvimento de compiladores e geradores de código, como o que foi desenvolvido neste trabalho. Outro fator que contribuiu para a escolha do *Rascal* foi o fato dele ser desenvolvido em *Java*, a mesma linguagem em que a ferramenta BETA foi desenvolvida. Dessa forma, a integração do gerador de *scripts* de teste desenvolvido neste trabalho com a ferramenta BETA é facilitada. A Figura 4.1 apresenta uma visão geral, de forma simplificada, da arquitetura do gerador de *scripts* de teste implementado em *Rascal*. O código completo do gerador de *scripts* de teste em *Rascal* é apresentado no Apêndice A.

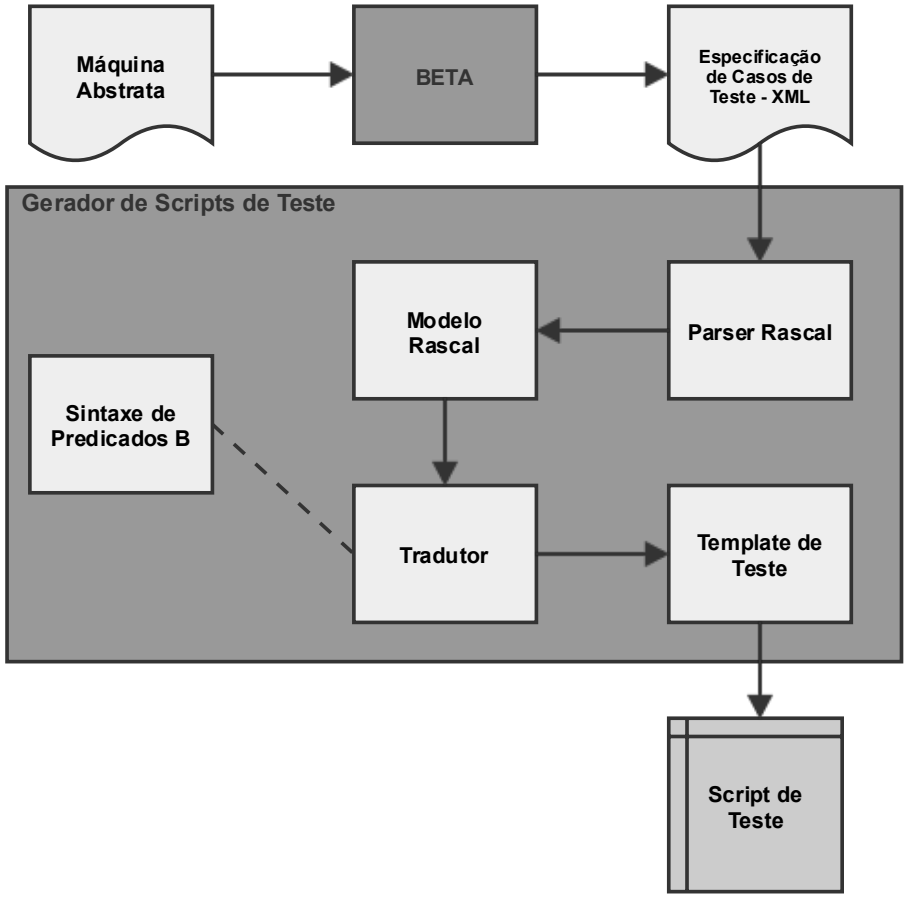


Figura 4.1 – Visão geral da arquitetura do gerador de *scripts* de teste

Como é mostrado na Figura 4.1, o gerador de *scripts* de teste recebe como entrada uma especificação de casos de teste em formato *XML* gerada pela ferramenta BETA a partir de uma máquina abstrata B. A especificação é aplicada em um *Parser*, que separa as informações da especificação e as convertem para um modelo de dados que representa a especificação em *Rascal*.

Após gerado o modelo com as informações da especificação de casos de teste em *Rascal*, alguns dados do modelo podem ser traduzidos para uma linguagem de programação. Os dados que podem ser traduzidos são os predicados do invariante de estado e expressões para serem utilizadas nos oráculos de teste. Esses dados são traduzidos em um módulo de tradução. Para poder traduzir os dados, foi necessário definir a sintaxe de predicados e expressões do Método B. Esta sintaxe foi definida em *Rascal* e foi baseada na sintaxe do Método B apresentada em (CLEARSY, 2011), considerando apenas a parte dos predicados e expressões.

A sintaxe dos predicados e expressões é utilizada por um módulo tradutor, que define funções de tradução para cada um dos predicados. Podem ser desenvolvidos módulos de tradução para linguagens de programação específicas ou, ainda, para projetos específicos, dependendo da necessidade e do contexto do sistema sob teste. Neste trabalho foram desenvolvidos dois módulos de tradução, um para a linguagem *C* e outro para a linguagem *Java*. Estes módulos não traduzem todos os predicados e expressões definidas na sintaxe em *Rascal*, mas podem ser complementados, adaptados ou utilizados como referência para o desenvolvimento de outros módulos de tradução. As regras definidas para a tradução de predicados e expressões em *C* e *Java* são apresentadas na Tabela 4.1. Na tabela os predicados são representados pela letra P e as expressões pela letra E, as regras de tradução são definidas em termos das funções ρ, que traduz os predicados, e ϵ, que traduz as expressões. Os predicados e expressões que não estão na Tabela 4.1 não são traduzidos.

Os predicados traduzidos são utilizados em um módulo que implementa um *template* de teste. É o módulo de *template* que transforma os dados da especificação de casos de teste em um código de teste. Assim como os módulos de tradução de predicados e expressões, podem ser desenvolvidos vários módulos de *template* para linguagens de programação diferentes e projetos diferentes. Neste trabalho foram desenvolvidos dois módulos de *template* para a linguagem *C*, um utilizando a notação de testes do *framework CuTest*[1] e outro utilizando a notação do *framework Check*[2], ambos utilizados neste trabalho, e um módulo de *template* para a linguagem *Java*, utilizando a notação de testes do *framework JUnit*[3].

Os módulos de *template* desenvolvidos implementam, de forma parcial, as estratégias de oráculos de teste propostas neste trabalho. Para as estratégias de verificação das variáveis de estado e valores de retorno, estratégias (3) e (4), respectivamente, são inseridas assertivas após a chamada da operação sob teste em cada caso de teste. As assertivas

[1] *CuTest* é uma biblioteca de teste de unidade para a linguagem *C* que pode ser obtida no site <http://cutest.sourceforge.net>.

[2] *Check* é um *framework* de teste de unidade para a linguagem *C* que pode ser obtido no site <http://check.sourceforge.net/>.

[3] *JUnit* é um *framework* de teste de unidade para a linguagem *Java* que pode ser obtido no site <http://www.junit.org/>.

Tabela 4.1 – Regras de tradução para alguns predicados e expressões B

Predicado/Expressão	**Tradução**
$\rho[P_1 \wedge P_2]$	$\rho[P_1]$ && $\rho[P_2]$
$\rho[P_1 \vee P_2]$	$\rho[P_1]$ || $\rho[P_2]$
$\rho[\neg P]$	!($\rho[P]$)
$\rho[P_1 \Rightarrow P_2]$	!($\rho[P_1]$) || $\rho[P_2]$
$\rho[P_1 \Leftrightarrow P_2]$	($\rho[P_1]$ && $\rho[P_2]$) || !($\rho[P_1]$ || $\rho[P_2]$)
$\rho[E_1 = E_2]$	$\epsilon[E_1]$ == $\epsilon[E_2]$
$\rho[E_1 \neq E_2]$	$\epsilon[E_1]$!= $\epsilon[E_2]$
$\rho[E_1 \leq E_2]$	$\epsilon[E_1]$ <= $\epsilon[E_2]$
$\rho[E_1 < E_2]$	$\epsilon[E_1]$ < $\epsilon[E_2]$
$\rho[E_1 \geq E_2]$	$\epsilon[E_1]$ >= $\epsilon[E_2]$
$\rho[E_1 > E_2]$	$\epsilon[E_1]$ > $\epsilon[E_2]$
$\rho[E_1 \in E_2..E_3]$	$\epsilon[E_1]$ >= $\epsilon[E_2]$ && $\epsilon[E_1]$ <= $\epsilon[E_3]$
$\rho[E_1 \in E_2]$	$\epsilon[E_1]$!= *null*
$\epsilon[id]$	*id*
$\epsilon[TRUE]$	*true*
$\epsilon[FALSE]$	*false*
$\epsilon[num]$, onde $num \in \mathbb{Z}$	*num*
$\epsilon[E_1 + E_2]$	$\epsilon[E_1]$ + $\epsilon[E_2]$
$\epsilon[E_1 - E_2]$	$\epsilon[E_1]$ - $\epsilon[E_2]$
$\epsilon[E_1 \times E_2]$	$\epsilon[E_1]$ * $\epsilon[E_2]$
$\epsilon[E_1 \ / \ E_2]$	$\epsilon[E_1]$ / $\epsilon[E_2]$
$\epsilon[E_1 \ mod \ E_2]$	$\epsilon[E_1]$ % $\epsilon[E_2]$
$\epsilon[E_1^{E_2}]$	pow($\epsilon[E_1]$, $\epsilon[E_2]$)
$\epsilon[-E]$	-$\epsilon[E]$
$\epsilon[succ(E)]$	$\epsilon[E]$ + 1
$\epsilon[pred(E)]$	$\epsilon[E]$ - 1

comparam o valor da variável de estado ou resultado com o valor que é esperado, que, atualmente, deve ser inserido manualmente. Para a estratégia de verificação do invariante de estado (2), é definida uma função que verifica se o invariante de estado do sistema sob teste foi preservado. Nesta função, cada predicado do invariante é verificado individualmente em um comando condicional, em que é verificado se a negação do predicado é verdadeira. Quando isso ocorre, significa que o predicado não foi respeitado, dessa forma, é levantado um erro no teste para indicar que o invariante de estado foi quebrado. Esta função de verificação do invariante foi baseada no método de verificação do invariante proposto em (COSTA et al., 2012). O módulo de tradução dos predicados e expressões é utilizado para gerar a função de verificação do invariante. Atualmente, os predicados do invariante que o módulo de tradução não traduz aparecem em forma de comentário no código. Após o código de teste ser montado, o gerador gera como saída um arquivo com o *script* de teste.

4.2.2 Discussões e Conclusões

Para ilustrar o resultado do gerador de *scripts* de teste desenvolvido neste trabalho, a Figura 4.2 apresenta um trecho do *script* de teste em *C*, utilizando a notação do *framework CuTest*, gerado a partir da especificação de casos de teste da operação *BlueMove* da máquina *TicTacToe*, apresentada no Capítulo 2. Neste *script* foram aplicadas as estratégias de oráculo de verificação do invariante e verificação das variáveis de estado.

Na Figura 4.2 é possível ver um trecho da função que verifica o invariante (linhas 9 a 20) e a implementação de um caso de teste (linhas 22 a 44). Na função de verificação do invariante, os predicados que foram traduzidos, de forma completa ou parcial, são negados em comandos condicionais *if*. Quando a sua negação é verdadeira, significa que o invariante não foi preservado, então o teste falha. Os predicados que não conseguiram ser traduzidos, de forma completa ou parcial, aparecem como comentários (linhas 15 a 17), dessa forma, fica indicado para o desenvolvedor dos testes que esses predicados não estão sendo verificados, permitindo que ele possa inserir a sua própria verificação. Na implementação do caso de teste é possível ver as três partes do teste de forma divida, a inicialização do estado (linhas 22 a 29), a chamada da operação sob teste com os seus parâmetros (linhas 31 e 32) e os oráculos de teste (linhas 34 a 41), com as assertivas que verificam as variáveis de estado e a chamada da função de verificação do invariante de estado.

Como pode ser observado na Figura 4.2, o código gerado pelo gerador de *scripts* de teste é, de maneira geral, baseado na especificação B (máquina abstrata) que originou os casos de teste. Logo, as variáveis e dados utilizados nos testes são abstratos. Por causa disso, o *script* de teste gerado precisa ser adaptado para poder ser executado. Essa adaptação envolve fazer as adequações de refinamento, seguindo o detalhamento feito no projeto, como transformar variáveis abstratas em variáveis concretas e o refinamento dos dados abstratos, além da inserção dos valores esperados nos oráculos de teste. No *script* de teste apresentado na Figura 4.2 é possível ver que a função de verificação do invariante (linhas 9 a 20) não verifica todos os predicados do invariante, que são os que aparecem em forma de comentário (linhas 15 a 17), e, além disso, os dois predicados que são verificados utilizam variáveis e constantes abstratas (linhas 8 a 14). Dessa forma, a verificação dos predicados que aparecem nos comentários deve ser feita de forma manual e os predicados que são verificados precisam ser refinados seguindo as decisões de projeto que foram tomadas. As outras adaptações envolvem o refinamento das variáveis e dados de teste (linhas 24 a 31) e a inserção dos valores esperados nos oráculos de teste (linhas 34 a 39).

O gerador de *scripts* de teste desenvolvido neste trabalho trouxe uma automação parcial para a etapa de implementação dos testes concretos da abordagem BETA. Dessa forma, o esforço gasto na última etapa de BETA foi reduzido, fazendo com que o

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <stdint.h>
4  #include "CuTest.h"
5  #include "TicTacToe.h"
6
7  void check_invariant(CuTest* tc) {
8     if(!(!(!((ThreeInRow(TicTacToe__bposn) == true)) || (ThreeInRow(
          TicTacToe__rposn) == false)))){
9        CuFail(tc, "The invariant '((ThreeInRow(TicTacToe__bposn) =
             TRUE) => (ThreeInRow(TicTacToe__rposn) = FALSE))' was
             unsatisfied");
10    }
11    /* ... */
12    if(!(TicTacToe__bposn /\ TicTacToe__bposn == {})){
13       CuFail(tc, "The invariant 'bposn /\ rposn = {}' was
             unsatisfied");
14    }
15    // Predicate 'bposn <: 0..8' can't be automatically translated
16    // Predicate 'rposn <: 0..8' can't be automatically translated
17    // Predicate 'turn : Player' can't be automatically translated
18 }
19
20 void TicTacToe_BlueMove_test_case_1(CuTest* tc)
21 {
22    TicTacToe__INITIALISATION();
23
24    int32_t[] bposn = {};
25    TicTacToe__bposn = bposn;
26    int32_t[] rposn = {};
27    TicTacToe__rposn = rposn;
28    Player turn = blue;
29    TicTacToe__turn = turn;
30
31    int32_t pp = 0;
32    TicTacToe__BlueMove(pp);
33
34    int32_t[] bposnExpected; // Add expected value here.
35    CuAssertTrue(tc, TicTacToe__bposn == bposnExpected);
36    int32_t[] rposnExpected; // Add expected value here.
37    CuAssertTrue(tc, TicTacToe__rposn == rposnExpected);
38    Player turnExpected; // Add expected value here.
39    CuAssertTrue(tc, TicTacToe__turn == turnExpected);
40
41    check_invariant(tc);
42 }
```

Figura 4.2 – *Script* de teste em *C* gerado para a operação *BlueMove* da máquina *TicTacToe*

desenvolvedor dos testes precise apenas fazer as adequações relacionadas ao refinamento de dados e oráculos de teste. O gerador foi desenvolvido de forma modular para ser adaptável, dessa forma, os módulos de tradução e *templates* desenvolvidos podem ser customizados ou utilizados como referência para o desenvolvimento de módulos específicos, oferecendo uma maior flexibilidade para que o gerador possa ser utilizado em diferentes

projetos. O gerador também automatiza, de forma parcial, as estratégias de oráculos de teste propostas neste trabalho.

A adaptação automática dos *scripts* de teste gerados poderá ser explorada futuramente, o que diminuirá, ainda mais, os custos da etapa de implementação dos testes concretos. Uma possibilidade seria fazer uso de uma linguagem de refinamento semelhante a *FTCRL* apresentada em (CRISTIÁ et al., 2011), que foi discutida na Seção 3.3 do Capítulo 3. Poderia ser desenvolvida uma linguagem de refinamento própria pra BETA ou, ainda, fazer uso da linguagem *Rascal* pra essa finalidade, uma vez que a linguagem oferece recursos para transformações de dados, semelhante ao que foi feito nos módulos de tradução de predicados e expressões. Outra possibilidade seria fazer uso do próprio módulo de implementação B para fazer o refinamento de dados.

5 Estudos de Caso

Neste capítulo são apresentados os dois estudos de caso realizados neste trabalho. No primeiro estudo de caso, a abordagem e ferramenta BETA foi utilizada para gerar testes a partir de um modelo parcial em B da API *C* da linguagem de programação Lua. No segundo estudo de caso, BETA foi utilizada para contribuir com a validação de dois geradores de código para o Método B, o *b2llvm* e o *C4B*. Ambos os estudos de caso serviram para avaliar a abordagem e ferramenta, assim como avaliar as melhorias propostas e desenvolvidas neste trabalho.

5.1 Primeiro estudo de caso: API de Lua

Neste estudo de caso BETA foi utilizada para gerar testes a partir uma especificação parcial em B da API *C* da linguagem de programação Lua. Inicialmente, é feita uma breve introdução à linguagem de programação Lua e sua API. Em seguida, é apresentada a especificação da API de Lua em B. Por último, são apresentados os resultados do processo de geração e execução dos testes.

5.1.1 Linguagem Lua

Lua é uma linguagem de programação de extensão projetada para dar suporte a programação procedural em geral com facilidades para descrição de dados (IERUSALIMSCHY; FIGUEIREDO; CELES, 2014). Foi projetada para ser poderosa, rápida e leve, combinando uma sintaxe simples para programação procedural com poderosas construções para descrição de dados baseadas em *arrays* associativos e semântica extensível. A linguagem oferece um bom suporte para vários paradigmas, como programação orientada a objetos, programação funcional e programação dirigida a dados. Além de permitir estender aplicações, é possível estender a linguagem Lua ao registrar funções escritas em *C*, ou em outra linguagem, para adicionar funcionalidades que não podem ser escritas diretamente em Lua. Essa característica faz com que Lua seja uma linguagem de extensão extensível (IERUSALIMSCHY; FIGUEIREDO; CELES, 1996).

A linguagem Lua foi criada em 1993 a partir da evolução de outras duas linguagens que eram utilizadas no *Tecgraf* (Grupo de Tecnologia em Computação Gráfica da PUC-Rio)[1] (IERUSALIMSCHY; FIGUEIREDO; CELES, 2007). Desde a criação, Lua vem se apresentando como uma linguagem robusta e já é estabelecida no mercado, sendo utilizada

[1] <http://www.tecgraf.puc-rio.br/>

em grandes projetos, como a aplicação *Adobe Photoshop Lightroom*[2], o *middleware* Ginga[3] para TV digital, e os jogos *World of Warcraft*[4] e *Angry Birds*[5]. Atualmente a linguagem Lua se encontra na versão *5.3*.

Lua é uma linguagem dinamicamente tipada, que significa que variáveis não possuem tipos, apenas valores possuem tipo, além de que todos os valores carregam o seu próprio tipo. Todos os valores em Lua são *valores de primeira classe*, que significa que todos os valores podem ser armazenados em variáveis, passados como argumentos para funções e retornados como resultado. Existem oito tipos básicos em Lua: *nil*, *boolean*, *number*, *string*, *function*, *userdata*, *thread* e *table*.

O mecanismo utilizado para estender uma aplicação e, ao mesmo tempo, estender a linguagem Lua é a API (*Application Program Interface*) *C*. A API possui em torno de 113 funções (ou macros) que permitem que um programa hospedeiro se comunique com Lua e vice versa. Todas as funções, tipos e constantes da API são declaradas no arquivo de cabeçalho *lua.h*. A Figura 5.1 apresenta um pequeno exemplo de código em *C* que faz uso da API de Lua.

```
1  #include "lua.h"
2  #include "lauxlib.h"
3
4  int main(int argc, char **argv) {
5     lua_State *L = luaL_newstate();
6     luaL_openlibs(L);
7     luaL_loadfile(L, argv[1]);
8     lua_call(L, 0, 0);
9     lua_close(L);
10    return 0;
11 }
```

Figura 5.1 – Exemplo de uso da API de Lua

A Figura 5.1 apresenta um pequeno interpretador de Lua, que recebe como entrada um arquivo de programa em Lua e o executa. Na linha 1, é incluído o arquivo de cabeçalho *lua.h*, que declara as funções e tipos da API. Na linha 2, é incluído o arquivo de cabeçalho *lauxlib.h*, que declara a biblioteca auxiliar que fornece várias funções convenientes que auxiliam na interface entre *C* e Lua. As funções da API padrão possuem o prefixo *lua_* e as funções da biblioteca auxiliar possuem o prefixo *luaL_*. Na linha 5 é declarado um ponteiro para uma estrutura *lua_State*, que é uma estrutura dinâmica definida em *lua.h* que armazena o estado da API. Todas as funções da API e da biblioteca auxiliar recebem um estado (lua_State) como primeiro argumento, com exceção das funções que

[2] <http://www.adobe.com/products/photoshop-lightroom.html>
[3] <http://www.ginga.org.br/>
[4] <http://us.battle.net/wow/>
[5] <https://www.angrybirds.com/>

criam um novo estado. Todo estado é auto-contido e deve ser explicitamente criado (e destruído) pela aplicação *C*.

Uma das dificuldades de interação entre uma linguagem como Lua e *C* é administrar as diferenças entre uma linguagem dinamicamente tipada e uma estaticamente tipada, além do gerenciamento de memória, em que uma faz de forma automática e a outra manual. Uma solução para esse problema seria definir algum tipo em *C* para representar os valores Lua, solução que é utilizada por outras linguagens como *Python*[6] e *Ruby*[7]. Entretanto, o problema desta abordagem é que é fácil de criar referências pendentes e vazamentos de memória, o que dificulta a coleta de lixo. Para evitar esse tipo de problema, a API não define nenhum tipo em *C* para representar os valores Lua.

A troca de valores entre Lua e *C* é feita através de uma *pilha virtual* que fica armazenada em *lua_State*. Cada espaço na pilha pode armazenar um valor Lua e esses valores podem ser acessados e manipulados através de funções da API. Toda a troca de informações entre as duas linguagens é feita exclusivamente através da pilha, assim sendo, para passar algum valor para Lua é necessário primeiro inserir este valor na pilha e, após, acessar a pilha através de Lua. Esta mesma forma é utilizada para passar valores Lua para uma aplicação *C*, em que os valores devem ser inseridos na pilha e, após, serem acessados pela aplicação *C*, através de funções da API que fazem as conversões dos valores Lua para os tipos equivalentes em *C* (IERUSALIMSCHY, 2013). O uso de uma pilha para manipular e trocar informações com *C* faz com que a API de Lua seja uma das mais simples e ortogonais em comparação com as APIs de outras linguagens, como *Python* e *Ruby* (MUHAMMAD; IERUSALIMSCHY, 2007).

O acesso ao conteúdo da pilha virtual não segue uma disciplina estrita da estrutura de dados pilha uma vez que seus elementos também podem ser acessados através de índices. Uma região da pilha é reservada para o registro, que é uma tabela Lua que pode ser utilizada por qualquer aplicação *C* para armazenar valores no estado, os índices dessa área reservada são chamados de *pseudo-índices*. As funções que recebem índices da pilha trabalham com *índices válidos* ou *índices aceitáveis*. Os índices válidos são aqueles que se referem ao conteúdo que está na pilha, sem considerar os pseudo-índices, ou seja, o que se encontra entre o primeiro elemento, que tem o índice 1, e o elemento que está no topo. Os índices válidos também podem ser negativos e possuem o topo da pilha como referência, como o índice -1 que se refere ao elemento que está no topo, por exemplo. Os índices aceitáveis são qualquer índice válido, pseudo-índices ou qualquer número inteiro positivo após o topo da pilha dentro do espaço alocado para a pilha (IERUSALIMSCHY; FIGUEIREDO; CELES, 2014).

Retomando o exemplo da Figura 5.1, na linha 5 o estado é criado pela função

[6] <https://www.python.org/>
[7] <https://www.ruby-lang.org>

luaL_newstate da biblioteca auxiliar. Na linha 6, é feita a chamada a função *luaL_openlibs* da biblioteca auxiliar, que abre todas as bibliotecas Lua padrão no estado fornecido. Na linha 7, é feita a chamada à função *luaL_loadfile* da biblioteca auxiliar, que recebe o estado e um arquivo de programa como argumentos. Esta função carrega o conteúdo de um arquivo, coloca-o dentro de uma função Lua e insere esta função no topo da pilha. Na linha 8 a função *lua_call* da API é chamada. Esta função recebe como argumento o número de argumentos que foram passados para a função Lua e o número de resultados esperados. Para se executar uma função Lua deve-se, primeiro, empilhar uma função na pilha (que foi feito ao chamar *luaL_loadfile*), e em seguida empilhar os argumentos que serão passados para a função em ordem, ou seja, o primeiro argumento é o primeiro a ser empilhado, por fim chamar *lua_call*, que se baseia pelo número de argumentos passado como parâmetro para encontrar a função na pilha, executá-la, passando os argumentos empilhados, e depois empilhar os seus resultados, de acordo com a quantidade de resultados esperados. No exemplo foi passado zero para os dois parâmetros, pois não foi passado nenhum valor como argumento para a função Lua e não era esperado nenhum resultado.

Além da função *lua_call* para chamar funções Lua, a API possui outras funções básicas que manipulam a pilha. As funções que servem para inserir um elemento na pilha possuem, em sua maioria, o nome *lua_push* seguido do tipo do elemento a ser inserido, como exemplo *lua_pushnumber*, que recebe como argumentos o estado e o número a ser inserido na pilha. As funções que servem para verificar se um elemento da pilha é de um tipo específico possuem o nome *lua_is* seguido do tipo tipo a ser verificado, como exemplo *lua_isnumber*, que recebe como argumento o estado e o índice do elemento a ser verificado. Todos os valores Lua que podem ser convertidos para valores em *C* são obtidos da pilha através de funções que possuem o nome *lua_to* seguido do tipo do elemento a ser obtido, como exemplo *lua_tonumber*, que recebe como argumentos o estado e o índice do elemento a ser obtido. Essas são as principais funções da API que permitem a interação entre *C* e Lua.

5.1.2 Especificação da API de Lua

Em (MOREIRA; IERUSALIMSCHY, 2013) é apresentado um modelo parcial da API de Lua em B. O modelo é focado na consistência da pilha, tipos e *heap*. O modelo é baseado na descrição da API do manual de referência da linguagem Lua na versão *5.2* (IERUSALIMSCHY; FIGUEIREDO; CELES, 2014). O modelo é estruturado a partir de máquinas que especificam os tipos e valores da linguagem, seguido de máquinas que especificam o estado da API e máquinas que especificam as operações da API. Ao todo, a especificação da API de Lua possui 23 máquinas abstratas. As principais máquinas da especificação são agrupadas e descritas a seguir:

5.1.2.1 Tipos de Lua em B

Os tipos e valores de Lua são definidos na máquina *BasicTypes* e existem máquinas que especificam funcionalidades específicas para cada tipo. As principais máquinas são descritas abaixo:

- *BasicTypes*: define os tipos básicos de Lua em B. Cada tipo de Lua é modelado através de um conjunto abstrato ou enumerado, com exceção do conjunto LUA_NUMBER que representa o tipo *number* que é apenas um outro nome para o conjunto dos inteiros. Uma vez que Lua é uma linguagem bastante flexível e que não é possível trabalhar com esse mesmo tipo de flexibilidade em B, um valor Lua foi modelado através de uma tupla (LUA_VALUES) que contém um campo para cada tipo de Lua e um campo para indicar o tipo do valor, para apenas o campo do tipo específico ser considerado;

- *BasicTypeProjections*: especifica mecanismos de seleção para acessar os diferentes valores da tupla LUA_VALUES;

- Máquinas ADT (*Abstract Data Type*): para cada tipo de Lua, existe uma máquina que define constantes e funções que auxiliam no tratamento e utilização do tipo na especificação. Essas máquinas possuem o nome do tipo acrescido do sufixo ADT, como $BooleanADT$ por exemplo;

5.1.2.2 Estado da API em B

O estado da API de Lua foi dividido em quatro máquinas. As máquinas *Pseudo_Stack*, *TableHeap* e *UserdataHeap* especificam partes específicas do estado, como a pilha e os *heaps* de memória, e a máquina *Memory* unifica todo o estado. As máquinas são descritas abaixo:

- *Pseudo_Stack*: modela a pilha virtual da API de Lua em B. A pilha é definida através das variáveis *stack*, que é uma função total do conjunto dos números inteiros, que representam os índices, para valores do conjunto LUA_VALUES, que representa valores Lua; *stack_top*, que é o índice para o elemento que se encontra no topo da pilha; *max_stack_top*, que limita o valor que *stack_top* pode assumir; e *pseudo_bottom*, que define o início dos chamados pseudo-índices na pilha, que são espaços reservados para informações adicionais. A máquina também especifica uma série de operações que manipulam a pilha;

- *TableHeap*: modela o *heap* que armazena as tabelas de Lua, que são associações entre valores LUA_TABLE, que funcionam como identificadores, e o conteúdo da tabela, que são relações do tipo chave e valor entre dois valores Lua;

- *UserdataHeap*: de forma análoga a *TableHeap*, modela o *heap* que armazena valores do tipo *userdata*;
- *Memory*: unifica as três máquinas que representam o estado da API (*Pseudo_Stack*, *TableHeap* e *UserdataHeap*) em uma única máquina, especificando as restrições que fazem as três máquinas serem consistentes entre si e unificando as suas operações.

5.1.2.3 Operações da API em B

No modelo são especificadas 71 operações finais da API, que são as operações que contém o prefixo *lua_*. As operações foram divididas em seis máquinas abstratas que foram organizadas de acordo com seus objetivos e similaridades. As máquinas são descritas abaixo:

- *PseudoStackOperations*: especifica as operações que fazem manipulações na pilha, como a inserção, modificação ou remoção de um valor;
- *PseudoStackReadingOperations*: especifica algumas operações que apenas fazem leituras na pilha;
- *TypeCastOperations*: especifica as operações que convertem valores Lua da pilha em valores equivalentes em *C*;
- *TypeCheckOperations*: especifica as operações que fazem verificações de tipos dos valores da pilha;
- *TableOperations*: especifica as operações da API que são relacionadas ao tipo *table*;
- *Function*: especifica a operação que executa uma função Lua e modifica o estado, caso seja necessário.

A Figura 5.2 apresenta uma visão geral da estrutura de máquinas abstratas da especificação da API de Lua e apresenta a ligação entre elas.

5.1.3 Objetivos

O objetivo deste estudo de caso era fazer uma avaliação de BETA, analisando o seu comportamento e resultados ao se utilizar uma especificação B complexa que continha situações ainda não exploradas na ferramenta, como uma estruturação em várias máquinas, tipos complexos, como o tipo LUA_VALUES, e um número elevado de operações, 71 operações finais ao todo. Também era objetivo deste estudo de caso realizar todas as etapas da abordagem e avaliar os resultados finais, relatando as dificuldades encontradas nas etapas de definição dos oráculos de teste e implementação, e avaliando a qualidade dos

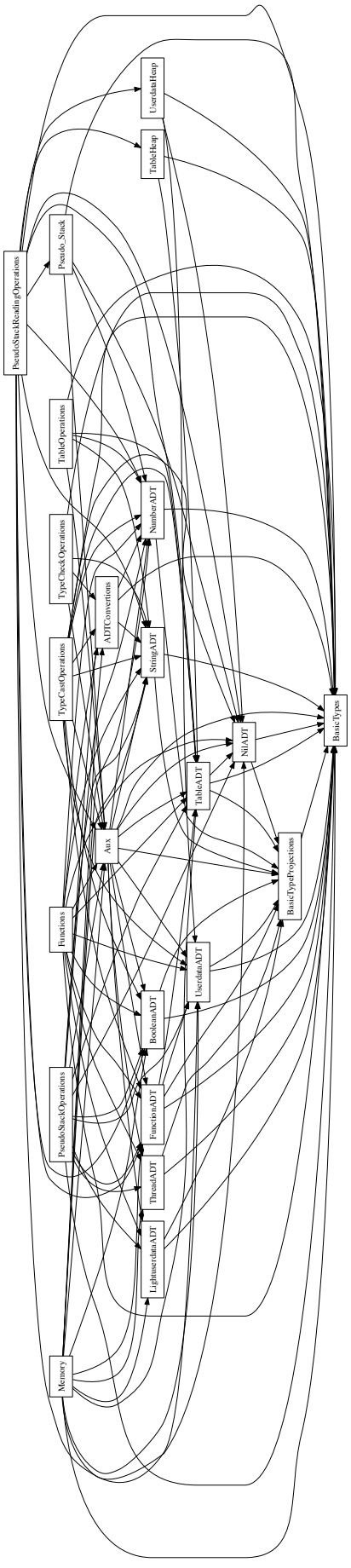


Figura 5.2 – Visão geral da estrutura de máquinas da especificação da API de Lua

testes desenvolvidos através de análise de cobertura de comandos e ramificações. Além disso, também era objetivo deste estudo de caso contribuir com a validação da especificação da API de Lua, verificando se o comportamento especificado é compatível com o comportamento real da API.

5.1.4 Execução do Estudo de Caso

Para gerar os testes para as operações da API de Lua, foram aplicadas na ferramenta BETA todas as máquinas que continham operações finais da API (operações com o prefixo *lua_*), que são as máquinas *PseudoStackOperations*, *PseudoStackReadingOperations*, *TypeCastOperations*, *TypeCheckOperations*, *TableOperations* e *Function*.

Cada máquina foi aplicada individualmente na ferramenta, utilizando, inicialmente, o particionamento em Classes de Equivalência e o critério de combinação *All Combinations*. Em um primeiro momento, a ferramenta não conseguiu gerar a especificação de casos de teste para nenhuma operação da API. A mensagem levantada pela ferramenta apenas informava que nenhuma combinação (fórmula) era válida, não sendo suficiente para indicar a causa do problema.

Após uma investigação, foi identificado que o problema estava relacionado ao solucionador de restrições utilizado pela ferramenta, o *ProB*. A ferramenta BETA utiliza máquinas auxiliares B para gerar os dados de teste, como foi explicado no capítulo 3. Essas máquinas são aplicadas no *ProB* que gera dados que satisfazem as restrições das fórmulas de teste. Ao aplicar as máquinas auxiliares no *ProB*, foi levantado um erro que informava que a ferramenta não havia conseguido gerar valores que satisfaziam as propriedades da especificação, indicando que o possível problema era a faixa de inteiros que estava sendo considerada (por *default* o *ProB* utiliza a faixa de inteiros de -1 a 3, uma faixa reduzida que busca evitar explosão combinatória). Uma vez que o possível problema indicado pela ferramenta podia ser corrigido ao se alterar as configurações, a faixa de números inteiros foi aumentada para de -5 a 25.

Após a mudança na configuração, o *ProB* não conseguiu animar a especificação, levantando um novo erro relacionado a explosão combinatória e extrapolação do tempo de computação da ferramenta. Este novo problema estava relacionado ao tipo LUA_VALUES que representa os valores Lua e é definido na máquina *BasicTypes*. Uma vez que LUA_VALUES é definido através de uma tupla com um produto cartesiano entre todos os tipos de Lua, o *ProB* calcula um número elevado de elementos para o conjunto, 39.680 elementos no total considerando a faixa de inteiros escolhida. Por causa desse número elevado, o *ProB* excedia o seu tempo de computação quando era necessário computar valores e verificar propriedades em cima de todos os valores de LUA_VALUES, como as funções definidas em *BasicTypesProjections* por exemplo, em que era necessário mapear todos os valores do conjunto para valores de outro conjunto. Assim como no problema

anterior, a configuração do *ProB* foi alterada, aumentando-se o tempo de computação da ferramenta. Mesmo com o aumento no tempo de computação da ferramenta o problema persistiu, fazendo com que não fosse possível animar as máquinas.

Ao verificar que não era possível gerar os dados de teste com a ferramenta BETA devido as limitações do *ProB*, e dado que a especificação era muito extensa e complexa para se realizar todo o processo manualmente, este trabalho optou por realizar modificações para reduzir a complexidade da especificação e, assim, dar continuidade ao estudo de caso. Como os problemas levantados eram relacionados a LUA_VALUES, que é um produto cartesiano entre todos os conjuntos de tipos de Lua, este trabalho optou por reduzir a quantidade de tipos de lua considerados na especificação e, consequentemente, presentes no produto cartesiano em LUA_VALUES para, assim, reduzir o número de elementos a serem calculados e não sobrecarregar o *ProB*. Para fazer essa redução foi necessário realizar experimentos com alterações na especificação da API de Lua no *ProB*, com o intuito de verificar quais tipos poderiam ser considerados na especificação de modo que o solucionador de restrições suportasse sem levantar erros. Como resultado, foi verificado que o *ProB* suportava apenas três tipos na especificação, os tipos de menor complexidade *nil*, *boolean* e *number*. Ao considerar apenas três tipos de Lua, a quantidade de elementos no conjunto LUA_VALUES diminuiu consideravelmente, passando a ter apenas 248 elementos com a mesma faixa de inteiros (-5 a 25).

Para manter a consistência na especificação, foi considerado apenas o subconjunto de máquinas abstratas e operações que eram relacionadas aos três tipos considerados (*nil*, *boolean* e *number*). Todas as máquinas e operações que eram relacionadas aos outros tipos não considerados foram retiradas dessa versão da especificação, referida neste trabalho como especificação reduzida. Como consequência, o número de máquinas abstratas e operações na especificação também diminuiu de forma considerável. A especificação passou a ter apenas 11 máquinas abstratas, das 23 que existem na especificação original, e a especificar 25 operações finais da API, das 71 especificadas na versão original. Apesar da versão reduzida ser consideravelmente mais simples que a original, ainda assim possui um nível de complexidade que não havia sido explorada anteriormente em BETA. A Figura 5.3 apresenta uma visão geral da estrutura de máquinas abstratas da especificação reduzida.

Na especificação reduzida da API de Lua foi adicionado o sufixo R em todas as máquinas abstratas para simbolizar que estas são versões reduzidas das originais. Na máquina *BasicTypesR*, o tipo LUA_VALUES também foi definido através de uma tupla, assim como na versão original, mas contendo um produto cartesiano apenas dos tipos considerados. A máquina *BasicTypesProjectionsR* ficou apenas com os mecanismos de seleção dos tipos considerados, além de que apenas as máquinas ADT dos tipo considerados continuaram na especificação ($NilADT$, $BooleanADT$ e $NumberADT$).

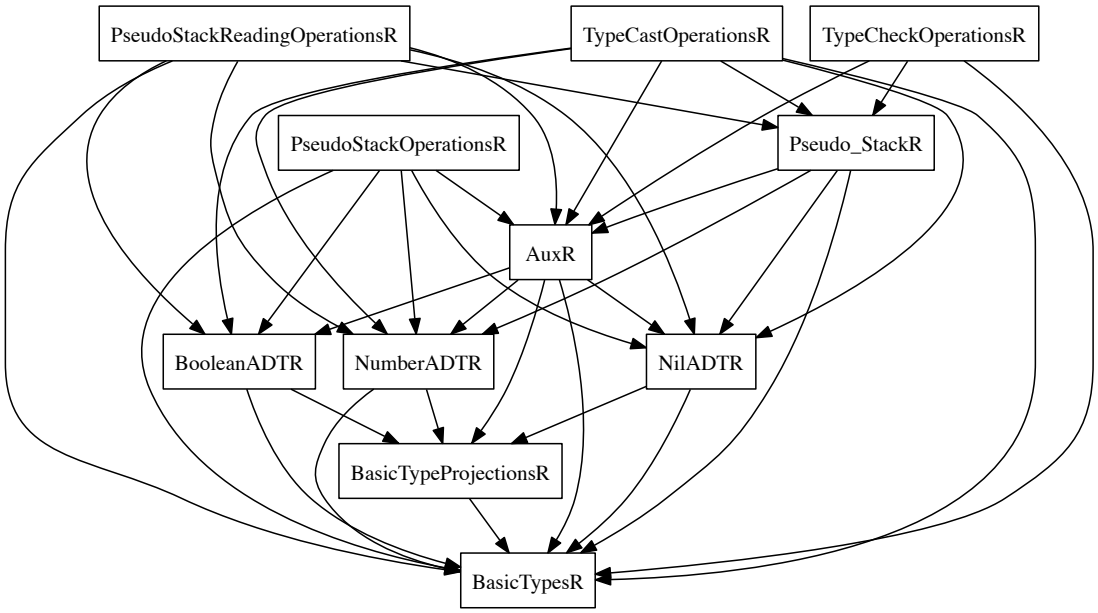


Figura 5.3 – Visão geral da estrutura de máquinas da especificação reduzida da API de Lua

Na versão original quatro máquinas compunham a especificação do estado da API, a máquina *Pseudo_Stack*, que define a pilha, as máquinas *TableHeap* e *UserdataHeap*, que definem os *heaps* de tabela e *userdata*, respectivamente, e a máquina *Memory* que unifica todo o estado. Uma vez que os tipos *table* e *userdata* não foram considerados na especificação reduzida, as máquinas *TableHeap* e *UserdataHeap* não eram necessárias, e, consequentemente, a máquina *Memory* também não era necessária, uma vez que apenas uma máquina foi utilizada para definir o estado da API (*Pseudo_StackR*).

Para manter a consistência com a especificação original, todas as operações da API que continham qualquer relação com algum dos tipos não considerados foram removidas da especificação, dessa forma, apenas 25 operações se encaixaram na especificação reduzida e todas foram consideradas no estudo de caso. A seguir, é feita uma descrição parcial de todas as operações da API que se encontram na especificação reduzida:

- *lua_checkstack*: recebe um número como parâmetro e garante que a pilha possui esse número de espaços livres, aumentando seu espaço caso seja possível e necessário;
- *lua_copy*: recebe dois índices como parâmetro e copia o elemento de um índice para o outro sem deslocar nenhum outro elemento;
- *lua_insert*: recebe um índice como parâmetro e move o elemento que se encontra no topo da pilha para ele, deslocando os elementos que se encontram acima desse índice para abrir espaço;

- *lua_pop*: recebe um número como parâmetro e remove da pilha esse número de elementos;
- *lua_pushboolean*: recebe um valor booleano como parâmetro e o insere no topo da pilha;
- *lua_pushinteger*: recebe um número inteiro como parâmetro e o insere no topo da pilha;
- *lua_pushnil*: insere o valor *nil* no topo da pilha;
- *lua_pushnumber*: recebe um número como parâmetro e o insere no topo da pilha;
- *lua_pushvalue*: recebe um índice como parâmetro e insere no topo da pilha uma cópia do elemento que se encontra nesse índice;
- *lua_remove*: recebe um índice como parâmetro e remove o elemento que se encontra nele, deslocando para baixo os elementos que se encontram acima desse índice;
- *lua_replace*: recebe um índice como parâmetro e move o elemento que se encontra no topo da pilha para esse índice sem deslocar nenhum outro elemento, removendo o elemento no topo da pilha;
- *lua_setglobal*: recebe um nome como parâmetro e estabelece como um valor global o elemento que se encontra no topo da pilha com o nome passado, removendo o elemento no topo da pilha;
- *lua_settop*: recebe um índice como parâmetro e o estabelece como o topo da pilha. Quando o índice passado é maior que o anterior, os elementos nos índices que estão acima do anterior passam a ter o valor *nil*, e quando o índice passado é 0, todos os elementos da pilha são removidos;
- *lua_status*: retorna o status do estado atual da API, que pode ser LUA_OK, LUA_YIELD ou algum código de erro;
- *lua_arith*: realiza uma operação aritmética sob os dois elementos no topo da pilha (ou um em caso de negação unária), remove os elementos e insere o resultado no topo da pilha. A operação aritmética a ser realizada é passada como parâmetro para a operação;
- *lua_absindex*: recebe um índice aceitável como parâmetro e o converte em um índice absoluto, que é um índice que não depende do topo da pilha, como um índice negativo por exemplo;
- *lua_compare*: recebe dois índices e um tipo de comparação como parâmetros e compara os elementos que se encontram nos índices passados;

- *lua_gettop*: retorna o índice do topo da pilha;
- *lua_rawequal*: recebe dois índices como parâmetro e compara se seus elementos são primitivamente iguais (comparação sem a necessidade de chamar algum metamétodo);
- *lua_toboolean*: recebe um índice como parâmetro e retorna o seu valor convertido em um valor booleano em *C* (0 ou 1). A operação retorna 1 para todo valor Lua diferente de *false* e *nil* e retorna 0 em caso contrário;
- *lua_isboolean*: recebe um índice como parâmetro e retorna 1 caso o valor do elemento no índice seja do tipo *boolean* e 0 em caso contrário;
- *lua_isnil*: recebe um índice como parâmetro e retorna 1 caso o valor do elemento no índice seja *nil* e 0 em caso contrário;
- *lua_isnone*: recebe um índice como parâmetro e retorna 1 caso o índice não seja válido e 0 em caso contrário;
- *lua_isnoneornil*: recebe um índice como parâmetro e retorna 1 caso o valor do elemento no índice seja *nil* ou o índice não seja válido e retorna 0 em caso contrário;
- *lua_type*: recebe um índice como parâmetro e retorna o tipo do elemento caso o índice seja válido ou retorna *LUA_TNONE* em caso contrário;

Após a redução da especificação, o *ProB* conseguiu animar todas as máquinas e, consequentemente, a ferramenta BETA conseguiu gerar as especificações de casos de teste. Inicialmente a ferramenta BETA só conseguiu gerar as especificações para nove operações das 25 existentes na especificação. Após uma investigação das operações que não tiveram suas especificações de casos de teste geradas e uma investigação na ferramenta, foram identificados problemas nas máquinas auxiliares geradas por BETA para a obtenção dos dados de teste, o que impossibilitou a animação destas no *ProB*. As máquinas auxiliares não estavam incluindo todo o contexto em que as operações sob teste estavam inseridas, fazendo com que o *ProB* não identificasse algumas definições e propriedades que eram necessárias em alguns casos. Após este problema ser reportado e corrigido, a ferramenta BETA gerou as especificações de casos de teste. Detalhes sobre a geração dos casos de teste e uma análise quantitativa são feitas a seguir.

5.1.4.1 Casos de teste gerados por BETA

No estudo de caso foi utilizada apenas a estratégia de particionamento em Classes de Equivalência na geração dos casos de teste, pois as restrições das variáveis do espaço de entrada não utilizavam intervalos numéricos, dessa forma, a estratégia de Análise de

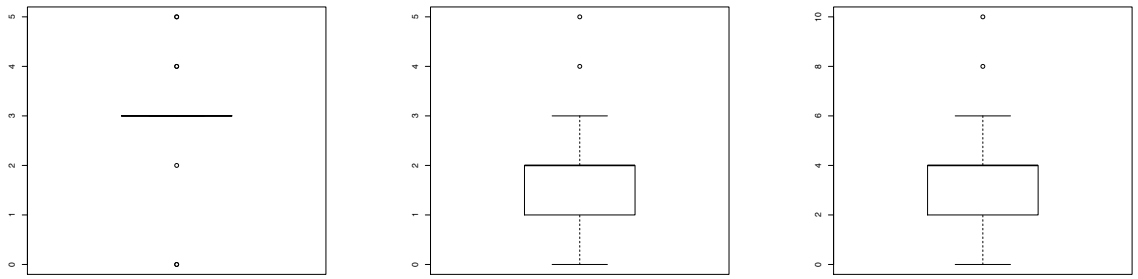
Operação	VEE	Caract.	Caract.*	Blocos	Blocos*
lua_checkstack	3	7	2	9	4
lua_copy	4	6	2	8	4
lua_insert	3	6	2	8	4
lua_pop	3	6	2	8	4
lua_pushboolean	3	6	2	8	4
lua_pushinteger	3	6	2	8	4
lua_pushnil	2	5	1	6	2
lua_pushnumber	3	6	2	8	4
lua_pushvalue	3	6	2	8	4
lua_remove	3	6	2	8	4
lua_replace	3	6	2	8	4
lua_setglobal	3	6	1	7	2
lua_settop	3	8	3	11	6
lua_status	0	0	0	0	0
lua_arith	4	10	5	15	10
lua_absindex	3	6	1	7	2
lua_compare	5	8	4	12	8
lua_gettop	0	0	0	0	0
lua_rawequal	5	7	3	10	6
lua_toboolean	3	6	1	7	2
lua_isboolean	3	6	1	7	2
lua_isnil	3	6	1	7	2
lua_isnone	3	6	1	7	2
lua_isnoneornil	3	6	1	7	2
lua_type	3	6	1	7	2
Média	**2.96**	**5.88**	**1.76**	**7.64**	**3.52**

Tabela 5.1 – Informações sobre as operações consideradas no estudo de caso

Legenda: VEE - Variáveis do Espaço de Entrada; Caract. - Características; * - Geram mais de mais de um bloco.

Valor Limite geraria os mesmos resultados. A Tabela 5.1 apresenta informações sobre o particionamento do espaço de entrada das operações consideradas no estudo de caso. Nela é possível ver o número de variáveis no espaço de entrada de cada operação (*VEE*); a quantidade total de características (*Caract.*); a quantidade de características que particionam o espaço de entrada em mais de um bloco (*Caract.**), que são as características que não são obtidas a partir do invariante e que não são apenas de tipagem; a quantidade total de blocos gerados (*Blocos*); e a quantidade de blocos considerando apenas os gerados a partir das características que particionam o espaço de entrada em mais de um bloco (*Blocos**), que são os blocos que influenciam na quantidade de casos de teste gerados.

Na Tabela 5.1 é possível ver que não foram identificadas variáveis no espaço de entrada das operações *lua_status* e *lua_gettop*, e, consequentemente, não foram identificadas características e não foram gerados blocos. Ambas as operações não possuem precondição ou alguma outra restrição no seu espaço de entrada, e o comportamento de ambas é apenas o retorno de um valor, a operação *lua_status* retorna um valor não determinístico e a operação *lua_gettop* retorna o valor de *stack_top*, por isso a ferramenta BETA não conseguiu identificar variáveis e características, não gerando nenhum bloco. Dessa forma, BETA não gerou nenhum caso de teste para essas operações, como poderá ser visto a seguir. Na versão atual de BETA não existe essa limitação, sendo gerado ao menos um caso de teste.



(a) Variáveis E. E. (b) Características* (c) Blocos*

Figura 5.4 – *Boxplot* dos dados das operações consideradas no estudo de caso

Na Tabela 5.1 é possível ver que a maioria das operações possuem três variáveis no seu espaço de entrada, possuem seis características no total e possuem entre uma e duas características considerando apenas as que geram mais de um bloco. O número de blocos possui uma maior variação, mas, mesmo assim, a maioria das operações possuem entre sete e oito blocos considerando a quantidade total e possuem entre dois e quatro blocos considerando apenas os que foram gerados a partir de características que geraram mais de um bloco. Nos pontos analisados, é possível ver que a maioria das operações possuem quantidades próximas da média. Este fato pode ser visualizado na Figura 5.4 que apresenta os gráficos *boxplot* das quantidades de variáveis no espaço de entrada, características, considerando apenas as que geram mais de um bloco, e blocos, considerando apenas os que foram gerados a partir características que geraram mais de um bloco.

Muitas variáveis e características eram comuns em todas as operações, como as variáveis *stack_top* e *max_stack_top* que estavam presentes no espaço de entrada de quase todas as operações e a característica $stack\_top < max\_stack\_top$, por exemplo, que também era comum em várias operações, algo que contribuiu para a semelhança nos dados das operações. Esse fato também impactou em semelhanças na quantidade de casos de teste gerados para as operações, dados que serão apresentados a seguir.

A Tabela 5.2 apresenta um resumo sobre os casos de teste gerados para as operações consideradas no estudo de caso. Todos os critérios de combinação suportados por BETA foram utilizados: *All Combinations* (AC), *Each Choice* (EC) e *Pairwise* (PW). A Tabela 5.2 apresenta a quantidade de casos de teste gerados ao utilizar cada um dos critérios de combinação. Algumas combinações podem gerar situações insatisfatíveis, que são as situações em que os dados de teste não podem ser gerados. A quantidade de casos insatisfatíveis é apresentada nas colunas $AC*$, $EC*$ e $PW*$.

Na Tabela 5.2 é possível ver que existe uma regularidade na quantidade de casos de teste gerados com o critério *All Combinations*, em que são gerados 2^c casos de teste, em que C é a quantidade de características que influenciam na quantidade de casos de teste,

Operação	**EC**	**EC***	**PW**	**PW***	**AC**	**AC***
lua_checkstack	2	0	4	1	4	1
lua_copy	2	0	4	0	4	0
lua_insert	2	0	4	1	4	1
lua_pop	2	0	4	0	4	0
lua_pushboolean	2	0	4	2	4	2
lua_pushinteger	2	0	4	0	4	0
lua_pushnil	2	0	2	0	2	0
lua_pushnumber	2	0	4	0	4	0
lua_pushvalue	2	0	4	0	4	0
lua_remove	2	0	4	1	4	1
lua_replace	2	0	4	1	4	1
lua_setglobal	2	0	2	0	2	0
lua_settop	2	1	4	2	8	3
lua_status	0	0	0	0	0	0
lua_arith	2	1	8	5	32	26
lua_absindex	2	0	2	0	2	0
lua_compare	2	1	6	3	16	8
lua_gettop	0	0	0	0	0	0
lua_rawequal	2	1	4	1	8	3
lua_toboolean	2	0	2	0	2	0
lua_isboolean	2	0	2	0	2	0
lua_isnil	2	0	2	0	2	0
lua_isnone	2	0	2	0	2	0
lua_isnoneornil	2	0	2	0	2	0
lua_type	2	0	2	0	2	0
Média	**1.84**	**0.16**	**3.2**	**0.68**	**4.88**	**1.84**

Tabela 5.2 – Casos de teste gerados para as operações consideradas no estudo de caso

Legenda: EC - *Each Choice*; PW - *Pairwise*; AC - *All Combinations*; * - Insatisfatíveis.

não considerando os casos das operações *lua_status* e *lua_gettop*. O critério *Pairwise* gera quantidades iguais ao critério *All Combinations* quando a operação possui uma ou duas características e quantidades inferiores a partir de três características. Já com o critério de combinação *Each Choice*, é possível ver uma uniformidade, com uma média próxima de dois casos de teste para cada operação, mesmo nos casos em que o número de blocos é superior a média.

Os resultados apresentados na Tabela 5.2 podem ser melhor visualizados nos gráficos *boxplot* apresentados nas figuras 5.5 e 5.6. A Figura 5.5 apresenta um *boxplot* da quantidade total de casos de teste para cada critério de combinação. Nela é possível ver que o critério de combinação *Each Choice* gerou uma quantidade constante de casos de teste e que os critérios *All Combinations* e *Pairwise* geraram quantidades superiores. Devido a semelhança entre as operações, com a maioria tendo em torno de uma ou duas características que influenciam na quantidade de casos de teste, os critérios *All Combinations* e *Pairwise* geraram quantidades semelhantes, como pode ser observado no gráfico. O gráfico da Figura 5.6 apresenta semelhança com o gráfico da Figura 5.5, mostrando que a quantidade de casos de teste insatisfatíveis também é influenciada pelos mesmos fatores que a quantidade total.

Considerando apenas os casos de teste satisfatíveis, todos os casos de teste gerados para cada uma das operações foram analisados e foram divididos em casos de teste

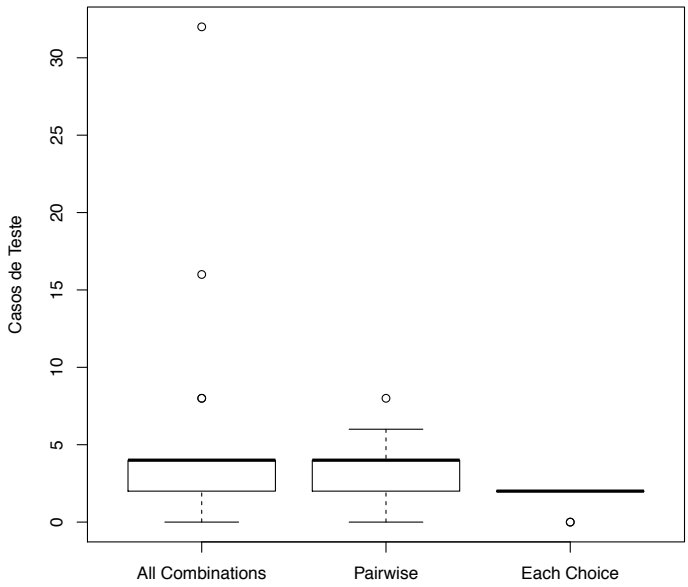


Figura 5.5 – *Boxplot* da quantidade total de casos de teste para cada critério de combinação

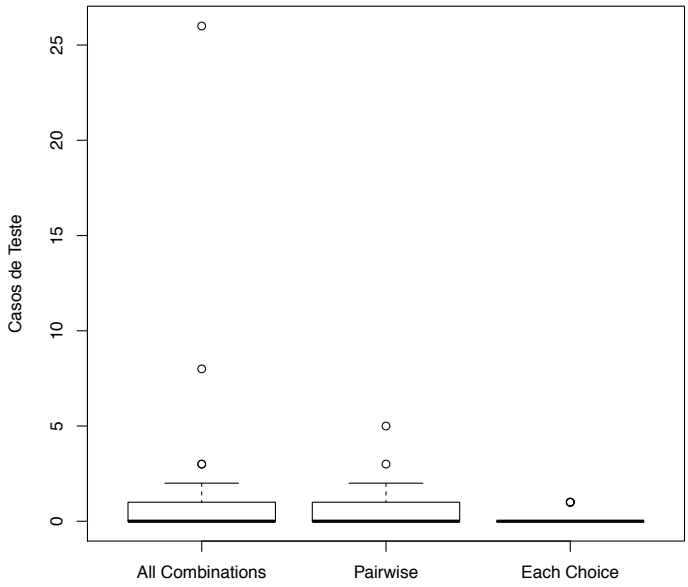


Figura 5.6 – *Boxplot* da quantidade de casos de teste insatisfatíveis para cada critério de combinação

Operação	EC P	EC N	PW P	PW N	AC P	AC N
lua_checkstack	1	1	2	1	2	1
lua_copy	1	1	1	3	1	3
lua_insert	1	1	1	2	1	2
lua_pop	1	1	1	3	1	3
lua_pushboolean	1	1	1	1	1	1
lua_pushinteger	1	1	1	3	1	3
lua_pushnil	1	1	1	1	1	1
lua_pushnumber	1	1	1	3	1	3
lua_pushvalue	1	1	1	3	1	3
lua_remove	1	1	1	2	1	2
lua_replace	1	1	1	2	1	2
lua_setglobal	1	1	1	1	1	1
lua_settop	0	1	0	2	2	3
lua_status	0	0	0	0	0	0
lua_arith	1	0	1	2	2	4
lua_absindex	1	1	1	1	1	1
lua_compare	1	0	1	2	2	6
lua_gettop	0	0	0	0	0	0
lua_rawequal	1	0	1	2	2	3
lua_toboolean	1	1	1	1	1	1
lua_isboolean	1	1	1	1	1	1
lua_isnil	1	1	1	1	1	1
lua_isnone	1	1	1	1	1	1
lua_isnoneornil	1	1	1	1	1	1
lua_type	1	1	1	1	1	1
Média	**0.88**	**0.8**	**0.92**	**1.6**	**1.12**	**1.92**

Tabela 5.3 – Casos de teste positivos e negativos gerados para as operações consideradas no estudo de caso

Legenda: EC - *Each Choice*; PW - *Pairwise*; AC - *All Combinations*; P - Positivos; N - Negativos.

positivos e negativos. A Tabela 5.3 apresenta a quantidade de casos de teste positivos e negativos gerados por cada um dos critérios de combinação. A Figura 5.7 apresenta o gráfico *boxplot* dos dados apresentados na tabela.

Ao analisar os dados da Tabela 5.3 e o gráfico da Figura 5.7, é possível ver que existe uma uniformidade na quantidade de casos de teste positivos para todos os critérios de combinação, em geral, a média é de um caso de teste positivo, com apenas algumas poucas operações em que foram gerados dois casos positivos com os critérios *All Combinations* e *Pairwise*, como pode ser observado na tabela e nas discrepâncias nos gráficos *boxplot*. Como já mencionado anteriormente, a semelhança no espaço de entrada e características entre as operações, além do fato de poucas operações utilizarem substituições condicionais, contribuiu para que fossem gerados poucos casos de teste positivos.

Já com relação aos casos de teste negativos, é possível ver que existe uma maior variação na quantidade gerada e que existem mais diferenças entre os critérios de combinação. Como esperado, o critério *All Combinations* gerou o maior número de casos de teste negativos, pois, uma vez que o critério gera todas as combinações de blocos possíveis, o número de casos com blocos negativos é maior. O critério *Pairwise* gerou uma quantidade intermediária de casos de teste negativos, levando em consideração os outros critérios de combinação. O critério *Each Choice* gerou uma quantidade uniforme de casos de teste

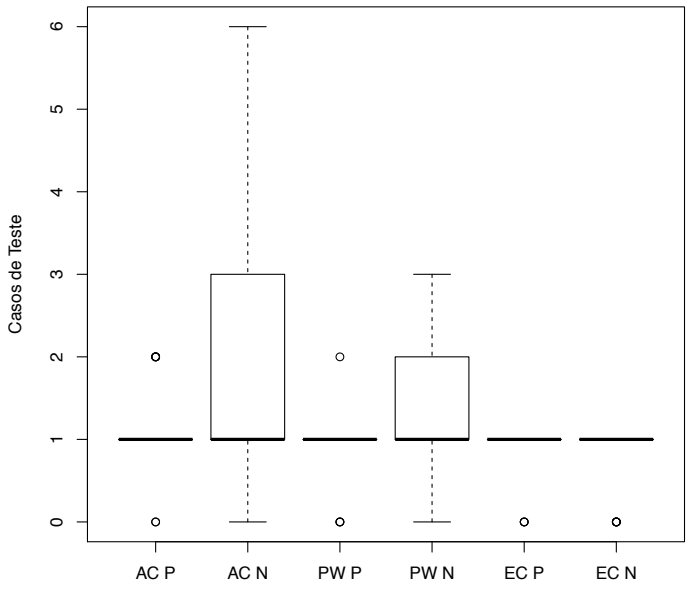


Figura 5.7 – *Boxplot* de casos de testes positivos e negativos

Legenda: EC - *Each Choice*; PW - *Pairwise*; AC - *All Combinations*; P - Positivos; N - Negativos.

negativos, com aproximadamente um caso para cada operação, e a menor quantidade em relação aos outros critérios, como também já era esperado. Em geral, uma vez que o critério *Each Choice* apenas exige que cada bloco esteja presente em ao menos um caso de teste, a ferramenta BETA procura minimizar a quantidade de casos de teste gerados ao combinar todos os blocos positivos em um único caso e todos os blocos negativos em um único caso, algo que explica o fato do critério gerar em média um caso de teste positivo e um negativo.

5.1.4.2 Implementação dos testes da API de Lua

Geradas as especificações de casos de teste, os passos seguintes consistem em definir os oráculos de teste e implementar os testes concretos. Apesar da linguagem Lua já se encontrar na versão *5.3*, nesta etapa e na etapa seguinte, a de execução dos testes, foi considerada a versão *5.2* da linguagem, pois a especificação em B da API de Lua foi desenvolvida com base na documentação da versão *5.2*, como foi mencionado na Seção 5.1.2, portanto, este trabalho optou por fazer uso da mesma versão que foi utilizada como referência na especificação que originou os casos de teste.

Os oráculos de teste para os casos de teste positivos foram projetados utilizando as estratégias de verificação de variáveis de estado e valores de retorno. Como foi discutido no Capítulo 4, essas duas estratégias de oráculos de teste são as mais precisas, por isso

apresentam os melhores resultados na verificação dos resultados dos testes. Além disso, utilizar as estratégias de verificação das variáveis de estado e valores de retorno em casos de teste positivos possui a vantagem de ser automatizado com o *ProB*, fazendo uso do método para a obtenção dos resultados esperados proposto na abordagem BETA, apresentado no Capítulo 3. Neste método, o *ProB* é utilizado, como ferramenta auxiliar, para calcular os valores retornados e os valores das variáveis de estado após a execução da operação sob teste, dessa forma, o engenheiro de testes não precisa calcular esses valores manualmente.

Apesar de casos de teste negativos serem importantes para testes de segurança e verificação da robustez de um sistema, estes não foram considerados nesta etapa do estudo de caso. Uma vez que casos negativos testam situações não previstas na especificação, não é possível, considerando apenas a especificação, determinar qual deve ser o comportamento esperado do sistema nesses casos, além disso, não se pode, necessariamente, esperar algum comportamento específico do sistema. Nessa situação, são necessários detalhes externos à especificação para determinar qual deve ser o comportamento esperado do sistema e, assim, poder projetar os oráculos de teste.

No caso da API de Lua, especificamente, é informado, em seu manual de referência, que o usuário é responsável por garantir a consistência da API, assim sendo, o usuário é responsável por garantir que as precondições serão respeitadas. O manual não especifica qual deve ser o comportamento da API nos casos em que as precondições não são respeitadas, dessa forma, não era possível projetar oráculos de teste específicos para os casos negativos, por isso, foram desconsiderados.

Para implementar os testes concretos foi necessário realizar um estudo da implementação da API de Lua, pois era necessário entender alguns detalhes internos da API para poder utilizar os dados gerados por BETA. Uma vez que a especificação apenas descreve o comportamento da API, detalhes de implementação, como alocação de memória e tratamento de tipagem por exemplo, são abstraídos e, além disso, dados abstratos são utilizados, o que faz com que os dados de teste gerados por BETA não possam ser utilizados de forma direta, sendo necessário refiná-los para os testes concretos.

Dada a sua grande extensão, além de não ser escopo deste trabalho, não era viável estudar em detalhes a implementação da API de Lua. Entretanto, era necessário entender algumas partes da API para poder refinar e utilizar os dados gerados por BETA. Dessa forma, foi realizado um estudo para mapear as variáveis da especificação em variáveis da implementação. Isso foi feito seguindo uma abordagem *top-down*, em que primeiro foi analisada a implementação das operações até se chegar na implementação do estado. Com isso, as variáveis *stack*, *stack_top* e *max_stack_top* da especificação foram mapeadas em variáveis da implementação.

A variável *stack* representa a pilha e é definida como uma função total entre o conjunto dos números naturais ($NAT1$) e o conjunto dos valores Lua (LUA_VALUES) na

especificação. Essas construções não são suportadas em uma linguagem de programação, por isso, a implementação da pilha é feita através de estruturas dinâmicas que armazenam várias informações, como o endereço na memória e referências para outras estruturas, por exemplo. Entre as estruturas que armazenam o estado da API e, consequentemente, da pilha, umas das mais importantes é a estrutura *lua_State*, que deve ser passada como parâmetro para quase todas as operações da API (menos as operações que criam novos estados). Dentro desta estrutura existe uma variável chamada *stack* que, diferentemente da especificação, é apenas uma referência para o início da pilha. Os elementos e espaços da pilha estão nos espaços de memória seguintes ao espaço da variável *stack*.

De forma análoga à variável *stack_top* na especificação, existe a variável *top* na estrutura *lua_State* que representa o topo da pilha, mas, diferentemente de *stack_top* que armazena o índice para o elemento que se encontra no topo, a variável *top* armazena a referência para o primeiro espaço livre disponível na pilha. Dessa forma, quando a pilha se encontra vazia, o valor da variável *stack_top*, de acordo com a especificação, deve ser zero, mas, na implementação, o valor da variável *top* deve ser o endereço que é logo seguinte ao endereço da variável *stack* na memória, que é a base da pilha, indicando que este é o primeiro espaço disponível. Uma vez que a variável *top* é manipulada em termos de endereço de memória, enquanto *stack_top* trabalha em cima de índices da pilha, e a abordagem BETA trabalha apenas com manipulação direta de variáveis, como foi apresentado no capítulo 3, foram desenvolvidas as funções *get_stack_top* e *set_stack_top* que abstraem o acesso e alteração, respectivamente, de *stack_top* na implementação para serem utilizadas nos testes.

A variável *max_stack_top* define o valor máximo que *stack_top* pode assumir na especificação. Na implementação, a variável *ci* da estrutura *lua_State*, que também é uma estrutura, possui uma variável chamada *top* que, diferentemente da variável *top* de *lua_State*, é correspondente à variável *max_stack_top* da especificação, guardando a referência para o último espaço disponível na pilha e, assim, limitando o seu tamanho. Assim como foi feito com a variável *stack_top*, foram desenvolvidas as funções *get_max_stack_top* e *set_max_stack_top* para abstrair o acesso e alteração, respectivamente, de *max_stack_top* nos testes. A Figura 5.8 apresenta a implementação das funções que abstraem o acesso e modificação de *stack_top* e *max_stack_top*.

Em todos os testes foram definidas duas variáveis do tipo *lua_State* para armazenar o estado da API, a primeira variável (*L*) foi o estado padrão utilizado na execução das operações sob teste, e a segunda (*L_control*) foi utilizada para servir como referência do estado inicial em alguns testes em que foi necessário comparar o estado final com o inicial. Ambas as variáveis eram inicializadas antes da execução dos testes e finalizadas após a execução.

Na parte inicial de cada teste, o estado da API era levado ao estado requerido

```
1  void set_stack_top(lua_State *L, int stack_top){
2      L->top = L->stack + 1 + stack_top;
3  }
4
5  int get_stack_top(lua_State *L){
6      return L->top - (L->stack + 1);
7  }
8
9  void set_max_stack_top(lua_State *L, int max_stack_top){
10     L->ci->top = L->stack + 1 + max_stack_top;
11 }
12
13 int get_max_stack_top(lua_State *L){
14     return L->ci->top - (L->stack + 1);
15 }
```

Figura 5.8 – Funções que abstraem o acesso das variáveis *stack_top* e *max_stack_top* nos testes

pelo caso de teste, fazendo uso dos dados gerados por BETA. Como em alguns casos era necessário comparar o estado final da API com o estado inicial, ambas as variáveis de estado, L e *L_control*, eram levadas ao mesmo estado. Além das funções *set_stack_top* e *set_max_stack_top*, apresentadas anteriormente, também foram utilizadas algumas funções da própria API para modificar o estado, nos casos em que era necessário inserir um valor específico na pilha por exemplo.

A segunda parte consistia em fazer chamada à operação sob teste. Antes da chamada da operação sob teste eram declaradas variáveis com os valores dos parâmetros de entrada da operação e, quando havia retorno, era declarado a variável que receberia o valor retornado pela operação. Em seguida, a operação sob teste era chamada, passando-se como parâmetro de entrada o estado padrão (L) e os valores de entrada. Apenas a variável de estado L era utilizada na execução das operações, pois a variável *L_control* não podia ser alterada após ser levada ao estado requisitado pelo caso de teste.

A terceira, e última, parte consiste nos oráculos de teste, que determinam o sucesso ou a falha do teste. Como já foi mencionado, foram escolhidas as estratégias de verificação de variáveis de estado e valores de retorno para projetar os oráculos de teste. Fazendo uso das funções *get_stack_top* e *get_max_stack_top*, apresentadas anteriormente, os valores das variáveis *stack_top* e *max_stack_top* foram verificados através de assertivas, para verificar se estas ficaram com os valores esperados após a execução da operação sob teste. Nos casos em que apenas uma parte da pilha era modificada, as outras partes que não deveriam ser modificadas eram comparadas com o valor inicial para garantir que ainda eram os mesmos, isso era feito através de comparação, índice a índice da pilha, do estado final, armazenado em L, com o estado inicial, armazenado na variável de controle *L_control*. Nos casos em que era necessário verificar algum valor específico da pilha foram utilizadas operações da própria API. Os valores de retorno também foram

verificados através de assertivas.

Para a implementação dos testes foi utilizado o *framework* de teste unitário *Check*. O gerador de *scripts* de teste desenvolvido neste trabalho foi utilizado na implementação dos testes neste estudo de caso. Para gerar os *scripts* de teste com a notação do *Check* foi desenvolvido um módulo de *template* para o *framework*. Como foi discutido no Capítulo 4, o gerador de *scripts* de teste pode ser adaptado para as necessidades de qualquer projeto, com isso, os módulos já desenvolvidos podem ser adaptados ou utilizados como base para o desenvolvimento de outros. Dessa forma, o módulo para o *Check* foi desenvolvido com base no que já havia sido feito para o *framework* de teste unitário *CuTest*, que tinha sido desenvolvido primeiro. Como ambos os *frameworks* de teste são em linguagem *C*, e a forma básica dos testes unitários é igual, a adaptação do gerador de *scripts* de teste foi rápida, não sendo necessário muito esforço para isso.

A Figura 5.9 apresenta a implementação de um caso de teste da operação *lua_arith* que realiza operações aritméticas sob elementos na pilha. O teste foi gerado parcialmente pelo gerador de *scripts* de teste e adaptado para que pudesse ser executado. Esse caso de teste exigia que os dois elementos que se encontravam no topo da pilha fossem do tipo *number* e que a operação aritmética a ser realizada sob esses elementos, que é passada como parâmetro, não fosse a negação unária. Para satisfazer esse caso de teste, BETA gerou os valores 2 e 2 para as variáveis *stack_top* e *max_stack_top*, respectivamente, gerou um valor para a pilha em que seus dois primeiros elementos eram do tipo *number*, ambos possuindo o valor -1, e gerou o valor LUA_OPADD para o parâmetro de entrada, que representa a operação aritmética de adição.

Entre as linhas 3 e 10, pode ser vista a primeira parte do teste, em que a API é levada ao estado exigido pelo caso de teste utilizando os valores gerados por BETA. Entre as linhas 3 e 5 é atribuído o valor 2 para *max_stack_top*, fazendo uso da função *set_max_stack_top*. Pode-se observar que a função é chamada duas vezes, uma passando o estado padrão L e outra passando o estado de controle *L_control*, fazendo com que ambos os estados fiquem com os mesmos valores. Entre as linhas 7 e 10 são inseridas duas vezes o valor -1 na pilha, através da operação *lua_pushnumber* da própria API e, assim como o valor de *max_stack_top*, os valores foram inseridos nos dois estados. Uma vez que dois elementos foram inseridos na pilha e que inicialmente ela se encontrava vazia, não foi necessário alterar de forma direta o valor de *stack_top* para 2, pois isso já foi feito ao se chamar duas vezes a operação *lua_pushnumber*.

Entre as linhas 12 e 13, pode ser vista a segunda parte do teste. Na linha 12, é atribuído o valor LUA_OPADD para a variável *op1* que será passada como parâmetro, e na linha 13 é feita a chamada à operação sob teste. A chamada de *lua_arith* é feita apenas passando-se o estado padrão L, pois o estado de controle *L_control* deve se manter igual ao inicial para em caso de ser necessário fazer alguma comparação.

```
1  START_TEST (PseudoStackOperations_lua_arith_test_case_4)
2  {
3      int max_stack_top = 2;
4      set_max_stack_top(L, max_stack_top);
5      set_max_stack_top(L_control, max_stack_top);
6
7      lua_pushnumber(L, -1);
8      lua_pushnumber(L_control, -1);
9      lua_pushnumber(L, -1);
10     lua_pushnumber(L_control, -1);
11
12     int op1 = LUA_OPADD;
13     lua_arith(L, op1);
14
15     ck_assert_int_eq(get_max_stack_top(L), get_max_stack_top(
           L_control));
16
17     int expected_stack_top = 1;
18     ck_assert_int_eq(get_stack_top(L), expected_stack_top);
19
20     int expected_result = -2;
21     int actual_result = lua_tonumber(L, 1);
22     ck_assert_int_eq(expected_result, actual_result);
23 }
24 END_TEST
```

Figura 5.9 – Implementação de um caso de teste da operação *lua_arith*

Entre as linhas 15 e 22, pode ser vista a terceira, e última, parte do teste, em que estão os oráculos de teste. Na linha 15 é verificado o valor de *max_stack_top*, que, uma vez que a operação sob teste não deve alterar o seu valor, é esperado que tenha o mesmo valor atribuído inicialmente, dessa forma, é verificado se continua com o mesmo valor que se encontra no estado de controle. Entre as linhas 17 e 18 é verificado o valor de *stack_top*. A operação *lua_arith* realiza a operação aritmética passada como parâmetro sob os elementos que estão no topo da pilha, depois desempilha esses elementos e empilha o resultado da operação aritmética, dessa forma, é esperado que os dois valores do tipo *number* que foram inseridos tenham sido removidos e que o resultado da soma entre os dois tenha sido inserido, ficando, portanto, com apenas um elemento na pilha, ou seja, é esperado que o valor de *stack_top* seja um. Entre as linhas 20 e 22 é verificado se o valor que se encontra no topo da pilha é o esperado, fazendo uso da operação *lua_tonumber* para se obter o valor do topo e verificando se este é igual a -2, resultado da soma entre -1 e -1.

Os casos de teste positivos de todas as operações consideradas no estudo foram implementados, com exceção da operação *lua_setglobal*, pois esta operação não foi completamente modelada em B, fazendo com que sua especificação não correspondesse com a operação original. Os resultados dos testes são apresentados, avaliados e discutidos a seguir.

5.1.5 Avaliação e Discussões

Após implementados, todos os testes positivos foram executados. Apesar de ser de responsabilidade do usuário garantir a consistência da API, é possível executar as suas operações em um modo protegido em que são verificadas a validade e a consistência dos argumentos passados e, em caso de haver algumas inconsistência, mensagens de erros específicas são apresentadas. Todos os testes foram executados com a API em modo protegido e seus resultados foram analisados. Uma vez que apenas os testes positivos foram considerados, era esperado sucesso em todos os testes. Entretanto, alguns testes das operações *lua_compare*, *lua_isnone* e *lua_type* falharam. Os testes dessas operações falharam devido a incompatibilidades entre a especificação B e a implementação da API de Lua, sendo o comportamento obtido com a implementação diferente do que era esperado segundo a especificação.

A operação *lua_compare* faz a comparação entre dois valores na pilha, dado dois índices aceitáveis e o tipo da comparação a ser feita. Para a operação, BETA gerou um mesmo caso de teste positivo com os três critérios de combinação e um segundo caso de teste positivo apenas com o critério *All Combinations*. Esse segundo caso de teste passou com sucesso, mas o caso de teste comum aos três critérios de combinação falhou porque os valores gerados por BETA para os índices passados como argumento foram considerados inaceitáveis.

Segundo a documentação da API de Lua, um índice é considerado aceitável quando se encontra dentro de um espaço alocado na pilha, como os índices que se encontram em uma posição real da pilha, chamados de índices válidos, e os pseudo-índices, além de qualquer índice que se encontra entre o topo da pilha e o tamanho da pilha. Os valores para os índices que foram gerados por BETA se encaixam na definição de índice aceitável. Ao comparar a especificação B com a documentação da API também não foram identificadas diferenças relacionadas a definição de índices aceitáveis. Dessa forma, não foi possível identificar a causa da incompatibilidade entre a especificação B e a implementação da API de Lua com relação a operação *lua_compare*.

A operação *lua_isnone* recebe como parâmetro um índice aceitável e verifica se o valor no índice correspondente não pertence a um tipo de Lua, e a operação *lua_type* recebe como parâmetro um índice aceitável e retorna o tipo do valor no índice correspondente. A ferramenta BETA gerou o mesmo caso de teste positivo para ambas devido ao fato de possuírem as mesmas precondições. Nos testes de ambas, os valores na pilha nos índices gerados por BETA eram do tipo *nil* de Lua, entretanto, era esperado, segundo a especificação B, o tipo *none*, que indica que o valor não deve ser tratado como um tipo de Lua. Devido a esse comportamento diferente do que era esperado, a especificação B foi comparada com a documentação da API.

Operação	Comandos	Each Choice	Pairwise	All Combinations
lua_checkstack	10	50%	60%	60%
lua_copy	3	100%	100%	100%
lua_insert	6	100%	100%	100%
lua_pushboolean	3	100%	100%	100%
lua_pushinteger	3	100%	100%	100%
lua_pushnil	3	100%	100%	100%
lua_pushnumber	3	100%	100%	100%
lua_pushvalue	3	100%	100%	100%
lua_remove	5	100%	100%	100%
lua_replace	4	100%	100%	100%
lua_settop	10	0%	70%	90%
lua_arith	13	92.3%	69.2%	92.3%
lua_absindex	1	100%	100%	100%
lua_compare	9	0%	0%	66.6%
lua_rawequal	4	100%	100%	100%
lua_toboolean	3	100%	100%	100%
lua_type	2	100%	100%	100%
Média		**70.6%**	**76.5%**	**85.9%**

Tabela 5.4 – Cobertura de comandos das operações da API de Lua

Segundo a documentação, os elementos que se encontram dentro dos índices válidos, que são os índices que se referem a posições reais na pilha, ou seja, que se encontram entre o índice 1 e o topo da pilha, devem ser de algum tipo de Lua e os elementos que não se encontram dentro dos índices válidos devem ser tratados como se não fossem de um tipo de Lua. Na comparação foi observado que essa restrição não havia sido inserida na especificação B, sendo, portanto, a causa das falhas nos testes. Em (MOREIRA; IERUSALIMSCHY, 2013) os autores decidiram não incluir essa restrição porque ela aumentaria de forma considerável o esforço de prova de consistência da especificação B.

As falhas nos testes das operações *lua_isnone* e *lua_type* foram importantes porque contribuíram para identificar incompatibilidades na especificação B. Para efeito de análise dos testes gerados por BETA, os resultados esperados nos testes de ambas as operações foram corrigidos segundo a documentação da API e, após, os testes foram executados com sucesso. Os testes gerados por BETA foram avaliados através de análise de cobertura de comandos e ramificações. Os resultados dessa análise fornecem informações importantes que contribuem na avaliação dos testes gerados por BETA. Para essa análise, foi utilizado, como ferramenta auxiliar, o *Gcov*. Uma vez que BETA trabalha com testes de unidade, apenas o aquivo *lapi.c*, que contém a implementação principal das operações da API de Lua, foi analisado.

As tabelas 5.4 e 5.5 apresentam, respectivamente, a cobertura de comandos e a cobertura de ramificações de cada operação testada neste estudo de caso, considerando os testes positivos gerados por cada critério de combinação. Na Tabela 5.4 é possível ver o número de comandos existentes em cada operação e a porcentagem de cobertura obtida com cada critério de combinação. Na Tabela 5.5 é possível ver o número de ramificações de cada operação e a porcentagem de cobertura obtida com os testes gerados por cada critério de combinação.

Operação	Ramificações	Each Choice	Pairwise	All Combinations
lua_checkstack	8	37.5%	50%	50%
lua_copy	0	–	–	–
lua_insert	4	75%	75%	75%
lua_pushboolean	2	50%	50%	50%
lua_pushinteger	2	50%	50%	50%
lua_pushnil	2	50%	50%	50%
lua_pushnumber	2	50%	50%	50%
lua_pushvalue	2	50%	50%	50%
lua_remove	4	75%	75%	75%
lua_replace	2	50%	50%	50%
lua_settop	8	0%	37.5%	62.5%
lua_arith	12	41.6%	41.6%	66.6%
lua_absindex	2	50%	50%	50%
lua_compare	8	0%	0%	37.5%
lua_rawequal	4	25%	25%	75%
lua_toboolean	4	75%	75%	75%
lua_type	2	50%	50%	50%
Média		**38.2%**	**44.1%**	**58.8%**

Tabela 5.5 – Cobertura de *ramificações* das operações da API de Lua

Ao se analisar a implementação, foi observado que algumas operações testadas não possuíam uma implementação direta, pois eram definidas através de macros em *C*, sendo definidas em termos de outras operações. As operações definidas através de macros são *lua_isboolean*, *lua_isnil*, *lua_isnone* e *lua_isnoneornil*, que foram definidas em termos da operação *lua_type*, e *lua_pop*, que foi definida em termos de *lua_settop*. As operações definidas através de macros não foram inseridas nas tabelas 5.4 e 5.5.

Na Tabela 5.4 é possível ver que a maioria das operações ficou com 100% de cobertura, mesmo com um número pequeno de casos de teste positivos por operação, uma média de um caso de teste como apresentado anteriormente. Esta cobertura foi possível porque muitas dessas operações não continham estruturas de seleção ou repetição na sua implementação, apenas estruturas sequenciais que eram facilmente cobertas. As operações que não tiveram 100% de cobertura (*lua_chekstack*, *lua_settop*, *lua_arith* e *lua_compare*) continham estruturas de seleção e repetição. Essas operações são analisadas a seguir.

A operação *lua_checkstack* recebe como parâmetro um número e verifica se a pilha contém esse número de espaços livres, caso não tenha, a operação aumenta o espaço da pilha quando é possível, retornando 1 para indicar que a pilha possui esse número de espaços livres ou 0 em caso contrário. Na especificação, são verificados o caso em que não é necessário aumentar o tamanho da pilha e o caso em que é necessário aumentar o tamanho da pilha. Para essa operação, foram gerados os mesmos dois casos de teste positivos com os critérios *All Combinations* e *Pairwise* e um caso de teste positivo com o critério *Each Choice*. O primeiro caso de teste de todos os critérios testava o caso em que não era necessário aumentar o tamanho da pilha, e o segundo, apenas para os critérios *All Combinations* e *Pairwise*, testava o caso em que era necessário e era possível aumentar o tamanho da pilha.

A Figura 5.10 apresenta um trecho do relatório de cobertura de código em que é

```
 92           : LUA_API int lua_checkstack (lua_State *L, int size) {
 93           :   int res;
 94         4 :   CallInfo *ci = L->ci;
 95           :   lua_lock(L);
 96         4 :   if (L->stack_last - L->top > size)  /* stack large enough? */
 97         4 :     res = 1;  /* yes; check is OK */
 98           :   else {  /* no; need to grow stack */
 99         0 :     int inuse = cast_int(L->top - L->stack) + EXTRA_STACK;
100         0 :     if (inuse > LUAI_MAXSTACK - size)  /* can grow without overflow? */
101         0 :       res = 0;  /* no */
102           :     else  /* try to grow stack */
103         0 :       res = (luaD_rawrunprotected(L, &growstack, &size) == LUA_OK);
104           :   }
105         8 :   if (res && ci->top < L->top + size)
106         2 :     ci->top = L->top + size;  /* adjust frame top */
107           :   lua_unlock(L);
108         4 :   return res;
109           : }
```

Figura 5.10 – Cobertura de código da operação *lua_checkstack*

apresentado o código e a cobertura da operação *lua_checkstack*. Os comandos nas linhas que se encontram destacadas em vermelho são os que não foram cobertos pelos testes. Na figura é apresentada a cobertura obtida com os testes gerados com o critério *All Combinations* e *Pairwise*. Se for considerada apenas a cobertura obtida com o teste gerado com o critério *Each Choice*, o comando na linha 106 não é coberto. Os casos de teste gerados por BETA não conseguiram testar as situações em que era necessário aumentar o tamanho da pilha mas que não era possível devido a falta de espaço na pilha. Essa situação era prevista na especificação B e era verificada através de substituições condicionais aninhadas do tipo *IF*. Na caracterização do espaço de entrada a ferramenta BETA considera apenas as substituições condicionais no primeiro nível, desconsiderando os condicionais internos a outros. Dessa forma, as situações que são verificadas em condicionais internos não são testadas. Esse fato prejudica o processo de teste, como foi observado nos resultados dos testes da operação *lua_checkstack*, o que mostra que é necessário realizar mudanças em BETA para que todas as estruturas condicionais sejam consideradas na geração dos casos de teste.

Para a análise de cobertura da operação *lua_compare* foi considerado apenas o segundo caso de teste positivo gerado com o critério *All Combinations*, pois o primeiro caso de teste, que é o mesmo para os outros critérios, não foi considerado uma vez que falhou, já explicado no início desta seção. Neste caso de teste todas as precondições eram respeitadas e era exigido que os dois índices passados como parâmetro fossem índices válidos. A Figura 5.11 apresenta um trecho do relatório de cobertura de código em que é apresentado o código de *lua_compare* e cobertura obtida com o caso de teste.

No relatório é possível ver que as linhas da operação *lua_compare* que não foram cobertas pelo teste correspondem às que tratam os possíveis valores de entrada para o tipo de comparação a ser feita. A operação *lua_compare* pode receber como parâmetro os valores *LUA_OPEQ*, *LUA_OPLT* e *LUA_OPLE*, e cada um desses valores corresponde a um tipo de comparação que vai ser feita pela operação. Nos casos de teste gerados por

```
316           : LUA_API int lua_compare (lua_State *L, int index1, int index2, int op) {
317           :   StkId o1, o2;
318           :   int i = 0;
319           :   lua_lock(L);  /* may call tag method */
320         2 :   o1 = index2addr(L, index1);
321         2 :   o2 = index2addr(L, index2);
322         2 :   if (isvalid(o1) && isvalid(o2)) {
323         2 :     switch (op) {
324         6 :       case LUA_OPEQ: i = equalobj(L, o1, o2); break;
325         0 :       case LUA_OPLT: i = luaV_lessthan(L, o1, o2); break;
326         0 :       case LUA_OPLE: i = luaV_lessequal(L, o1, o2); break;
327         0 :       default: api_check(L, 0, "invalid option");
328           :     }
329           :   }
330           :   lua_unlock(L);
331         2 :   return i;
332           : }
```

Figura 5.11 – Cobertura de código da operação *lua_compare*

BETA apenas o valor *LUA_OPEQ* foi considerado, por isso as outras possibilidades não foram testadas e, consequentemente, não foram cobertas no código.

Na especificação da operação *lua_compare* é utilizada uma substituição condicional *CASE* para definir o comportamento da operação para cada um dos valores do tipo de comparação a ser feita. Em uma substituição *CASE* é verificado se uma expressão é igual a uma constante ou pertence a um grupo de constantes, e para cada constante ou grupo de constantes pode ser definido um comportamento diferente para a operação. Na ferramenta BETA, as características obtidas a partir de uma substituição *CASE* consideram apenas a primeira constante ou grupo de constantes que são verificadas na substituição, dessa forma, as outras constantes ou grupos são desconsiderados. Na operação *lua_compare*, a primeira verificação é feita apenas com o valor *LUA_OPEQ*, sendo assim, os outros dois valores verificados (*LUA_OPLT* e *LUA_OPLE*) são desconsiderados.

Assim como *lua_compare*, a especificação da operação *lua_arith* também utiliza uma substituição condicional *CASE* para definir o comportamento da operação de acordo com o argumento passado. Como já foi dito anteriormente, a operação *lua_arith* realizar operações aritméticas sob os elementos que se encontram no topo da pilha e recebe como argumento apenas o tipo da operação aritmética a ser feita. Nos casos de teste gerados para a operação *lua_arith* apenas as operações de adição e negação unária foram consideradas, por isso as outras operações aritméticas não foram testadas. Isso pode ser visto nos trechos do relatório de cobertura apresentados nas figuras 5.12 e 5.13.

Na Figura 5.12 é possível ver a cobertura do código de *lua_arith* obtida com os casos de teste gerados pelo critério de combinação *All Combinations*. Apesar de o código estar relativamente bem coberto (92.3%), o tratamento do tipo de operação aritmética a ser realizada não é feito na própria função, mas sim na função *luaO_arith* que é evocada na linha 307. Esta função é definida no arquivo *lobject.c* e sua cobertura pode ser vista na Figura 5.13. No relatório é possível ver que apenas os casos em que a operação era igual a *LUA_OPADD* ou *LUA_OPUNM*, que correspondem adição e negação unária respectivamente, foram cobertos, o que confirma que apenas esses dois casos foram considerados

```
293            : LUA_API void lua_arith (lua_State *L, int op) {
294            :   StkId o1;  /* 1st operand */
295            :   StkId o2;  /* 2nd operand */
296            :   lua_lock(L);
297          4 :   if (op != LUA_OPUNM) /* all other operations expect two operands */
298          3 :     api_checknelems(L, 2);
299            :   else {  /* for unary minus, add fake 2nd operand */
300          1 :     api_checknelems(L, 1);
301          1 :     setobjs2s(L, L->top, L->top - 1);
302          1 :     L->top++;
303            :   }
304          2 :   o1 = L->top - 2;
305          2 :   o2 = L->top - 1;
306          4 :   if (ttisnumber(o1) && ttisnumber(o2)) {
307          2 :     setnvalue(o1, luaO_arith(op, nvalue(o1), nvalue(o2)));
308          2 :   }
309            :   else
310          0 :     luaV_arith(L, o1, o1, o2, cast(TMS, op - LUA_OPADD + TM_ADD));
311          2 :   L->top--;
312            :   lua_unlock(L);
313          2 : }
```

Figura 5.12 – Cobertura de código da operação *lua_arith*

```
73             : lua_Number luaO_arith (int op, lua_Number v1, lua_Number v2) {
74           2 :   switch (op) {
75           1 :     case LUA_OPADD: return luai_numadd(NULL, v1, v2);
76           0 :     case LUA_OPSUB: return luai_numsub(NULL, v1, v2);
77           0 :     case LUA_OPMUL: return luai_nummul(NULL, v1, v2);
78           0 :     case LUA_OPDIV: return luai_numdiv(NULL, v1, v2);
79           0 :     case LUA_OPMOD: return luai_nummod(NULL, v1, v2);
80           0 :     case LUA_OPPOW: return luai_numpow(NULL, v1, v2);
81           1 :     case LUA_OPUNM: return luai_numunm(NULL, v1);
82           0 :     default: lua_assert(0); return 0;
83             :   }
84           2 : }
```

Figura 5.13 – Cobertura de código da operação *luaO_arith*

nos casos de teste.

Em ambos os casos, *lua_compare* e *lua_arith*, alguns comportamentos não foram testados porque nem todos os casos verificados nos condicionais do tipo *CASE* foram considerados na caracterização do espaço de entrada, o que impactou na cobertura do código. Esses resultados mostram que é necessário realizar mudanças em BETA para que todos os casos verificados em um condicional *CASE* sejam considerados e, assim, a qualidade do conjunto de testes não seja afetada.

A operação *lua_settop* recebe um índice como parâmetro e o atribui como o topo da pilha. Quando o índice passado é maior que o anterior os valores nos índices acima passam a ser *nil* e quando o índice passado é 0 todos os elementos da pilha são removidos. A Figura 5.14 apresenta o trecho do relatório de cobertura em que é possível ver o código de *lua_settop* e a cobertura obtida com os casos de teste gerados com o critério de combinação *All Combinations*. Considerado apenas o caso de teste gerado com o critério de combinação *Pairwise*, os comandos nas linhas 171, 175 e 176 não foram cobertos, e não foram gerados testes positivos com o critério *Each Choice*, por isso a operação não teve cobertura com o critério. No relatório é possível ver que a situação em que o índice passado como argumento é maior que o anterior, fazendo com que os

```
165           : LUA_API void lua_settop (lua_State *L, int idx) {
166         3 :   StkId func = L->ci->func;
167           :   lua_lock(L);
168         3 :   if (idx >= 0) {
169         1 :     api_check(L, idx <= L->stack_last - (func + 1), "new top too large");
170         1 :     while (L->top < (func + 1) + idx)
171         0 :       setnilvalue(L->top++);
172         1 :     L->top = (func + 1) + idx;
173         1 :   }
174           :   else {
175         2 :     api_check(L, -(idx+1) <= (L->top - (func + 1)), "invalid new top");
176         2 :     L->top += idx+1;  /* `subtract' index (index is negative) */
177           :   }
178           :   lua_unlock(L);
179         3 : }
```

Figura 5.14 – Cobertura de código da operação *lua_settop*

valores nos índices acima passem a ser *nil*, não foi testada. Essa situação é prevista na especificação B da operação, entretanto, é verificada em uma operação de uma outra máquina B que é utilizada pela máquina que define a operação *lua_settop*. Na geração dos casos de teste, a ferramenta BETA considera apenas as restrições que estão presentes na operação a ser testada, logo, as restrições das operações que são chamadas por ela não são consideradas. Esse resultado mostra que é necessário levar em consideração as operações que são chamadas na geração dos testes.

Na Tabela 5.5 é possível ver que existe uma discrepância entre a cobertura de ramificações e a cobertura de comandos mostrada na Tabela 5.4. Na análise de cobertura de ramificações, todas as ramificações, ou possíveis decisões, das estruturas de controle são levadas em consideração. Dessa forma, mesmo que um teste tenha atingido todos os comandos, não significa que este conseguiu testar todos os possíveis comportamentos da operação sob teste. Este resultado mostra que os testes gerados por BETA não estão conseguindo testar de forma satisfatória as possíveis decisões dos códigos, indicando que é necessário fazer mudanças na abordagem para gerar testes com mais qualidade.

O conjunto de testes desenvolvido por BETA foi executado com a versão *5.2* da linguagem Lua, uma vez que a especificação apresentada em (MOREIRA; IERUSALIMSCHY, 2013) foi desenvolvida com base na documentação desta versão. Entretanto, a atual versão da linguagem, a *5.3*, não possui grandes mudanças em relação a sua versão anterior, a *5.2*, possuindo poucas incompatibilidades entre uma versão e outra, como é explicado na documentação da versão *5.3* (IERUSALIMSCHY; FIGUEIREDO; CELES, 2015). Com relação a API, nenhuma das mudanças afetaram as funções que foram especificadas em B e as funções que foram testadas neste estudo de caso, logo, todos os testes desenvolvidos puderam ser reexecutados com a versão *5.3* de Lua sem necessitar de quaisquer alterações nos códigos. Os mesmos resultados obtidos ao realizar os testes com a versão *5.2* de Lua foram obtidos com a versão *5.3*.

A linguagem de programação Lua possui um conjunto de testes próprio que avalia a implementação da linguagem e da API[8]. Diferentemente dos testes gerados por BETA,

[8] Os testes da linguagem de programação Lua e sua API em C são abertos e disponíveis no site

o conjunto de testes de Lua foi desenvolvido de maneira *ad hoc* pelos desenvolvedores da linguagem com o intuito específico de tentar "quebrar" Lua. Dessa forma, os testes foram desenvolvidos a partir de uma perspectiva estrutural (caixa branca) da linguagem, sendo necessário ter conhecimentos detalhados da implementação de Lua para entender o seu conjunto de testes. Como é dito no site, os testes não foram desenvolvidos para o uso geral, portanto, não foram desenvolvidos para serem portáteis e fáceis de usar. Uma vez que os testes gerados por BETA não são desenvolvidos a partir da implementação e sim da especificação (testes funcionais ou caixa preta), não requerem o mesmo nível de conhecimento da implementação da linguagem para o seu entendimento e são mais fáceis de usar.

5.1.6 Conclusões

O estudo de caso apresentado nesta seção consistiu em aplicar a abordagem e ferramenta BETA na geração e implementação de casos de teste para operações da API de Lua a partir do modelo em B apresentado em (MOREIRA; IERUSALIMSCHY, 2013). A especificação da API de Lua apresentava complexidade e situações que ainda não haviam sido exploradas em BETA, o que contribuiu para identificar algumas limitações e problemas na abordagem e ferramenta.

Na primeira tentativa de gerar casos de teste a partir da especificação da API de Lua, a ferramenta informou que não conseguiu gerar a especificação de casos de teste porque não havia nenhuma combinação válida, que significa que todos os casos de teste gerados eram insatisfatíveis. Após investigação, foi identificado que a ferramenta não havia conseguido gerar os dados de teste devido a limitações do *ProB*, como já foi explicado, e não porque só haviam casos insatisfatíveis, como a mensagem informada pela ferramenta BETA sugeria. Como um primeiro resultado, este caso mostrou que é importante melhorar o *feedback* da ferramenta para o usuário, pois a mensagem que foi apresentada pela ferramenta BETA não contribuiu para a identificação do problema e poderia levar a uma conclusão errônea sobre a situação, dado que era uma mensagem afirmativa.

Com relação ao *ProB*, foi identificado que ele não conseguiu verificar e animar o modelo em B da API de Lua dado a complexidade dos seus dados que levou a uma explosão combinatória. Nesta situação, é preciso deixar claro para o usuário que a geração de dados pela ferramenta BETA é limitada pelo *ProB*, mas que isso não impede que os outros resultados da ferramenta possam ser utilizados, como a caracterização do espaço de entrada e a geração dos casos de teste (fórmulas), passos que são automatizados e que dependem somente da ferramenta BETA. Utilizar o *ProB* como solucionador de restrições é conveniente uma vez que a notação do Método B é utilizada em todo o processo, não sendo necessário adaptações na notação. Entretanto, fazer uso de um outro

<http://www.lua.org/tests/>.

solucionador de restrições em BETA pode ser uma possível solução para este problema a ser explorada futuramente. Em (CRISTIÁ; ROSSI; FRYDMAN, 2013) foi realizado um estudo comparativo entre o $\{log\}$ (DOVIER et al., 1996), uma linguagem para programação lógica de restrições, e o *ProB* para verificar qual possuía a melhor performance para ser utilizado como solucionador de restrições no *Fastest* (CRISTIÁ; MONETTI, 2009), ferramenta com proposta semelhante à BETA apresentada na Seção 3.3 do Capítulo 3. Os resultados desse estudo mostraram que o $\{log\}$ teve um melhor desempenho que o *ProB*. Dessa forma, o $\{log\}$ se mostra como um candidato a substituir o *ProB* como solucionador de restrições em BETA. Outra alternativa é fazer uso de solucionadores *SMT* (*Satisfiability Modulo Theories*), que apresentam resultados promissores quando utilizados para o mesmo propósito, como foi mostrado em (CRISTIÁ; FRYDMAN, 2012).

Para dar continuidade ao estudo de caso, este trabalho optou por reduzir o escopo da especificação, considerando apenas um subconjunto em que menos tipos de Lua foram considerados e o estado foi simplificado. Com isso, o número de máquinas abstratas da especificação diminuiu de 23 para 11 e o número de operações finais especificadas da API diminuiu de 71 para 25. Esta redução possibilitou o *ProB* a animar e verificar a máquina e, consequentemente, gerar os dados de teste para a ferramenta BETA. Apesar da especificação reduzida possuir uma complexidade consideravelmente menor que a original, ainda assim possui um nível de complexidade que até então não havia sido explorado em BETA.

Os resultados da ferramenta foram apresentados e analisados. Em todos os pontos considerados foi observado uma certa uniformidade, em que a maioria das operações apresentou resultados semelhantes, como a quantidade de variáveis no espaço de entrada, número de características e blocos, e quantidade de casos de teste gerados. Um dos fatores que contribuíram para isso foi o fato de muitas operações apresentarem semelhanças em suas precondições.

Na caracterização do espaço de entrada foi identificada uma limitação da ferramenta BETA. Segundo a abordagem, todas as restrições que influenciam o comportamento da operação devem ser consideradas no particionamento do espaço de entrada. Entretanto, foi observado que a ferramenta não consegue identificar todas as restrições contidas na operação, apenas as que se encontram nas substituições do primeiro nível. Esse fator influencia negativamente a qualidade dos testes, pois, uma vez que nem todas as restrições são consideradas, o número de situações a serem testadas diminui, o que pode fazer com que nem todos os possíveis comportamentos da operação sejam testados, algo que foi observado nos casos que utilizavam substituições condicionais aninhadas do tipo *IF* ou *CASE*, por exemplo. Dessa forma, é necessário melhorar a caracterização do espaço de entrada na ferramenta, passando a considerar todas as restrições existentes na operação, para, assim, não comprometer a qualidade dos testes.

Apesar das semelhanças entre as operações, foi possível observar a diferença na geração de casos de teste entre os critérios de combinação suportados por BETA. A análise quantitativa dos casos de teste gerados para cada um dos critérios de combinação apresentou resultados que já eram esperados e que já haviam sido observados nos estudos de caso realizados anteriormente. O critério *All Combinations* gerou a maior quantidade de casos de teste, assim como a maior quantidade de casos insatisfatíveis e casos negativos. O critério *Pairwise* gerou uma quantidade intermediária de casos de teste e o critério *Each Choice* gerou a menor quantidade, sendo em média apenas dois casos de teste, um positivo e um negativo, por operação.

A implementação dos testes ocorreu após um estudo da implementação da API de Lua porque era necessário entender detalhes de refinamento para poder utilizar os dados de teste gerados por BETA. Dada a sua complexidade, o estudo da implementação da API foi limitado apenas a mapear as variáveis do espaço de entrada da especificação em variáveis na implementação. Uma vez que BETA trabalha com atribuição direta das variáveis de estado para atingir o estado requisitado no teste, apenas as variáveis mapeadas foram modificadas na implementação dos testes, fazendo uso dos dados gerados pela ferramenta.

Apesar de não ter apresentado problemas, foi observado, após análise dos testes implementados e resultados, que a atribuição direta de variáveis, solução adotada por BETA, não pode ser aplicada em todos os casos, uma vez que nem sempre se tem livre acesso às variáveis, e em algumas situações não é confiável, podendo levar a inconsistências no estado. Nesse estudo de caso, uma vez que a implementação do estado da API não era conhecida em detalhes, fazer modificação direta de apenas algumas variáveis pode implicar em alguma inconsistência interna não observada nos testes. Uma solução para este problema é fazer uso de operações do próprio sistema sob teste para levar o sistema ao estado solicitado pelo caso de teste, sendo uma forma mais segura de modificar o estado. Na versão atual de BETA isso já está sendo feito, de forma que a especificação de casos de teste traz a sequência de operações que devem ser executadas para levar o sistema ao estado solicitado pelo caso de teste.

Os testes implementados no estudo de caso foram avaliados através de análise de cobertura de linhas de código e ramificações. Os resultados da cobertura de comandos foram positivos, obtendo 100% de cobertura para a maioria das operações, com exceção das operações *lua_chekstack*, *lua_settop*, *lua_arith* e *lua_compare*. Ao analisar essas quatro operações foi observado que os casos de teste não conseguiram atingir todas as estruturas condicionais contidas no código. A limitação da ferramenta de não considerar todas as restrições da operação na caracterização do espaço de entrada impactou de forma direta nesse resultado, uma vez que nem todos os possíveis casos da operação foram testados. Este resultado evidencia que é necessário melhorar a caracterização do espaço de entrada na ferramenta.

Diferentemente da cobertura de comandos, os resultados da cobertura de ramificações mostraram que os testes gerados por BETA não conseguiram atingir todas as ramificações, ou possíveis decisões, das operações. Este resultado mostra que é necessário revisar a atual abordagem de geração de testes para melhorar a qualidade dos testes gerados. Uma solução para este problema, que está atualmente em desenvolvimento, é incluir outros critérios de cobertura na abordagem e ferramenta, como critérios de *cobertura lógica* (AMMANN; OFFUTT, 2010), ampliando, assim, a possibilidade de testes com BETA e melhorando a qualidade dos testes gerados.

Os testes, inicialmente, foram executados com a versão *5.2* da linguagem Lua. Mas, após a avaliação, foram reexecutados com a versão mais atual da linguagem, a *5.3*, uma vez que não tiveram mudanças entre uma versão e outra que alterassem as funções que foram testadas neste trabalho. Os resultados obtidos com a versão *5.3* de Lua foram os mesmos obtidos com a versão *5.2*. Esse resultado mostra que os testes gerados por BETA podem ser utilizados futuramente em *testes de regressão* nas próximas versões da linguagem Lua. Os testes gerados por BETA são funcionais, diferente dos testes feitos pelos desenvolvedores da linguagem Lua. Dessa forma, os testes gerados por BETA não são dependentes da implementação, são mais fáceis de serem entendidos e, portanto, são mais portáteis. Apesar disso, os testes gerados por BETA não são completos, ou seja, não testam toda a API e todo o código, e não substituem os testes feitos pelos desenvolvedores. Entretanto, os testes gerados por BETA podem ser ver vistos como complementares aos testes dos desenvolvedores da linguagem, uma vez que avaliam a API através de uma perspectiva diferente e, além disso, podem ser utilizados como referência em outras possíveis implementações da linguagem Lua.

5.2 Segundo estudo de caso: b2llvm e C4B

Neste estudo de caso a abordagem e ferramenta BETA foi utilizada para contribuir com a validação de dois geradores de código para o Método B, o *b2llvm* e o *C4B*. Em ambos os geradores, foi aplicado um conjunto de módulos B, e os códigos gerados foram verificados através de testes gerados por BETA. Este estudo de caso foi apresentado no artigo (MATOS et al., 2015), que foi aceito na conferência *TAP 2015* (*9th International Conference on Tests & Proofs*). No início desta seção é feita uma apresentação dos geradores *b2llvm* e *C4B*. Em seguida, é feita uma breve discussão sobre validação de compiladores e geradores de código. Por último, são apresentados o processo de execução do estudo de caso, resultados obtidos e avaliação dos resultados.

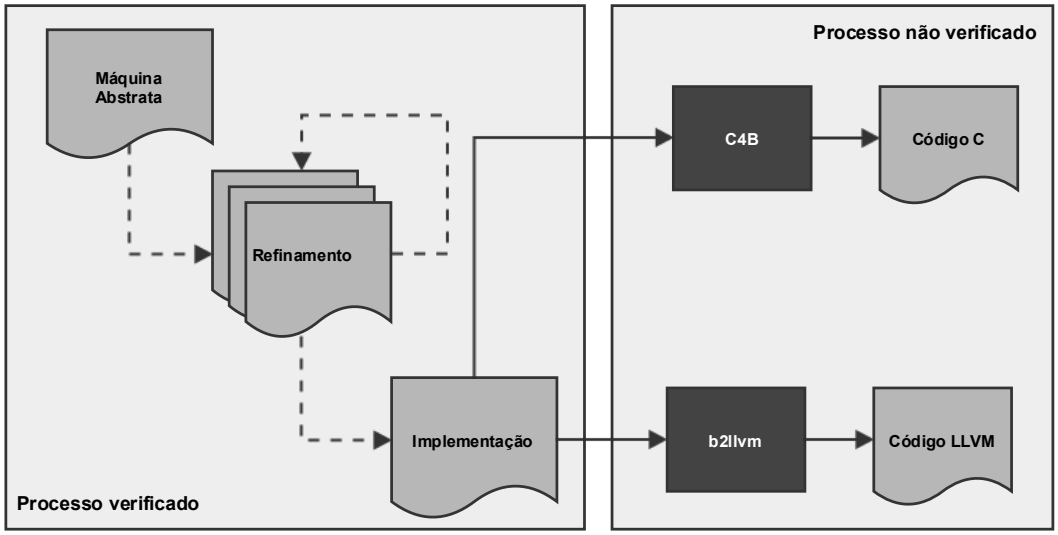


Figura 5.15 – Processo do Método B e dos geradores *b2llvm* e *C4B*

5.2.1 b2llvm e C4B

O *b2llvm*[9] (DÉHARBE; MEDEIROS JR., 2013) é um gerador de código *LLVM* (LATTNER; ADVE, 2004) para o Método B desenvolvido no grupo de pesquisas *ForAll*. O *LLVM* é uma infraestrutura de compilador de código-aberto, que fornece uma linguagem *assembly* intermediária, permitindo que técnicas de otimização, análise estática, geração de código e depuração possam ser utilizadas.

O *C4B* é um gerador de código *C* para o Método B que é integrado à *IDE Atelier B*, uma ferramenta para desenvolvimento formal com Método B. Ambos os geradores, *b2llvm* e *C4B*, recebem como entrada uma implementação B que deve estar seguindo a notação B0, que é um subconjunto do Método B que contém apenas construções diretamente implementáveis.

A Figura 5.15 apresenta o processo do Método B e dos geradores *b2llvm* e *C4B*. No primeiro quadro é apresentado o processo do Método B. Este processo é formalmente verificado, iniciando com a especificação de uma Máquina Abstrata, passando por sucessíveis refinamentos até se obter uma Implementação B. No segundo quadro é apresentado o processo de geração dos códigos *LLVM* e *C* pelos geradores *b2llvm* e *C4B*, respectivamente. Este último processo não se beneficia do mesmo rigor matemático que é aplicado no processo do Método B e também se encontra fora de seu escopo. A verificação formal de geradores de código apresenta uma série de dificuldades inerentes a sua complexidade. Testes podem ser aplicados como uma alternativa à verificação formal, a seção seguinte discute sobre a verificação de geradores de código.

[9] O projeto *b2llvm* continua em desenvolvimento e seu código pode ser acompanhado em: <https://www.b2llvm.org/b2llvm>.

5.2.2 Verificação de geradores de código

A verificação de compiladores e geradores de código é uma tarefa complexa. As abordagens de verificação, em geral, seguem duas linhas, a verificação formal ou teste de software, que podem ter diferentes objetivos, como a verificação do compilador ou gerador em si ou a verificação de sua saída, ou seja, o código gerado.

Verificação formal consiste em aplicar métodos que provam que o compilador ou gerador de código é correto para todos os possíveis modelos ou programas de entrada (GOERIGK et al., 1996). Esta abordagem exige muita experiência e esforço para provar a semântica da linguagem e as regras de tradução que são aplicadas pelo compilador ou gerador. Verificação formal em compiladores ou geradores que trabalham com linguagens *assembly* exige um esforço ainda maior. Por exemplo, em (LEROY, 2009) a ferramenta de prova formal *Coq* (The Coq development team, 2004) foi utilizada para verificar o compilador *CompCert*. Neste trabalho, foram desenvolvidas mais de 40,000 linhas de especificação *Coq* em um esforço estimado em 3 pessoas-anos de trabalho. Este trabalho demonstra o grande esforço que é necessário para aplicar uma verificação formal, o que faz com que técnicas de teste de software sejam mais empregadas.

Testes baseados em gramáticas consistem em gerar programas ou modelos de teste para o compilador ou gerador de código com base na gramática do programa ou modelo fonte (HOFFMAN et al., 2011). Os programas ou modelos de teste são derivados de forma sistemática de uma gramática e têm como objetivo exercitar o compilador ou gerador com as diferentes construções descritas na gramática. Testes baseados em gramáticas são uma maneira tradicional de testar softwares que têm suas entradas descritas por gramáticas e, em geral, apresentam resultados positivos (LÄMMEL; SCHULTE, 2006; HÄRTEL; HÄRTEL; LÄMMEL, 2014). Entretanto, testes baseados em gramáticas não tem foco em questões semânticas, o que pode dificultar a verificação do código resultante.

Validação de tradução consiste em verificar a correta tradução de um programa ou modelo passado como entrada para o compilador ou gerador (NECULA, 2000). Neste tipo de abordagem, o código gerado é validado ao invés do próprio compilador ou gerador de código em si, verificando se este foi corretamente traduzido dada uma entrada específica. A validação de tradução pode ser utilizada como teste funcional para o compilador ou gerador sob teste, o que faz com que um bom conjunto de testes (programas ou modelos de entrada) sejam necessários para que o compilador ou gerador seja bem exercitado. Diferentes níveis de rigor podem ser aplicados, como uma verificação formal ou teste do código gerado.

Em (ZUCK et al., 2003) é apresentada uma metodologia para validação de tradução em compiladores otimizados. Nesta abordagem, a correspondência entre o código fonte e o alvo é formalmente provada utilizando uma representação intermediária especí-

fica. Em (GOGOLLA; VALLECILLO, 2011) é apresentada uma abordagem que utiliza o conceito de *Tracts* para testar transformações de modelo. Nesta abordagem, um conjunto de restrições para o modelo fonte, o modelo alvo e para a transformação é definido em *OCL* (Object Management Group, 2014). Um conjunto de modelos fonte é, então, gerado automaticamente a partir das restrições, e é utilizado para testar as transformações. Por último, é verificado se o modelo gerado atende às restrições. Esta abordagem foi adaptada em (WIMMER; NO, 2013) para testar transformações de modelo para texto, como a transformação de UML para Java, por exemplo.

Neste trabalho, os códigos gerados pelos geradores *b2llvm* e *C4B* foram testados para verificar "equivalências funcionais" com os modelos de origem, utilizado para isso os testes gerados por BETA. Esta abordagem se encaixa como uma validação de tradução. A Figura 5.16 apresenta a abordagem aplicada neste trabalho. Na figura, é possível ver que a mesma máquina abstrata que deu origem a implementação B, que será aplicada nos geradores *b2llvm* e *C4B*, é aplicada na ferramenta BETA. Então, a ferramenta BETA gera como saída uma especificação de casos de teste para cada operação da máquina. Em seguida, a especificação gerada por BETA é aplicada no gerador de *scripts* de teste desenvolvido neste trabalho, que gera como saída um *script* de testes.

O *script* de testes deve ser codificado para adaptar refinamentos nos dados abstratos e para a inserção de valores nos oráculos de teste. Além disso, são excluídos do *script* os casos de teste negativos, uma vez que apenas os comportamentos especificados estão sendo verificados, o que não é possível testar com casos negativos. Adaptado o *script* de testes, os códigos gerados pelo *b2llvm* e *C4B* são executados com o *script* e os seus resultados são avaliados. Em caso de todos os testes passarem, significa que nenhum erro foi detectado no código gerado, em caso contrário, o código gerado é investigado para verificar possíveis defeitos na sua geração.

Os geradores *b2llvm* e *C4B* também foram verificados através de testes baseados em gramáticas. O artigo (MATOS et al., 2015) apresenta o trabalho em conjunto do grupo de pesquisas *Forall* na verificação dos geradores *b2llvm* e *C4B*. Neste trabalho, foi aplicada uma abordagem de validação de tradução, utilizando a ferramenta BETA (apresentada no presente trabalho), e uma abordagem de testes baseados em gramáticas, utilizando a ferramenta *LGen*[10](*Lua Language Generator*) (HENTZ, 2010; MOREIRA; HENTZ; RAMALHO, 2013), também desenvolvida no grupo de pesquisas *Forall*, para gerar módulos de entrada para os geradores *b2llvm* e *C4B*.

Ambos os geradores de código são testados por suas respectivas equipes de desenvolvimento. Nos testes do *b2llvm*[11] é utilizado um conjunto de módulos B para testar

[10] A ferramenta LGen pode ser obtida no site <http://lgen.wikidot.com>.

[11] Os testes do *b2llvm* estão disponíveis no mesmo repositório do código da ferramenta, que pode ser obtido no site <https://github.com/DavidDeharbe/b2llvm>.

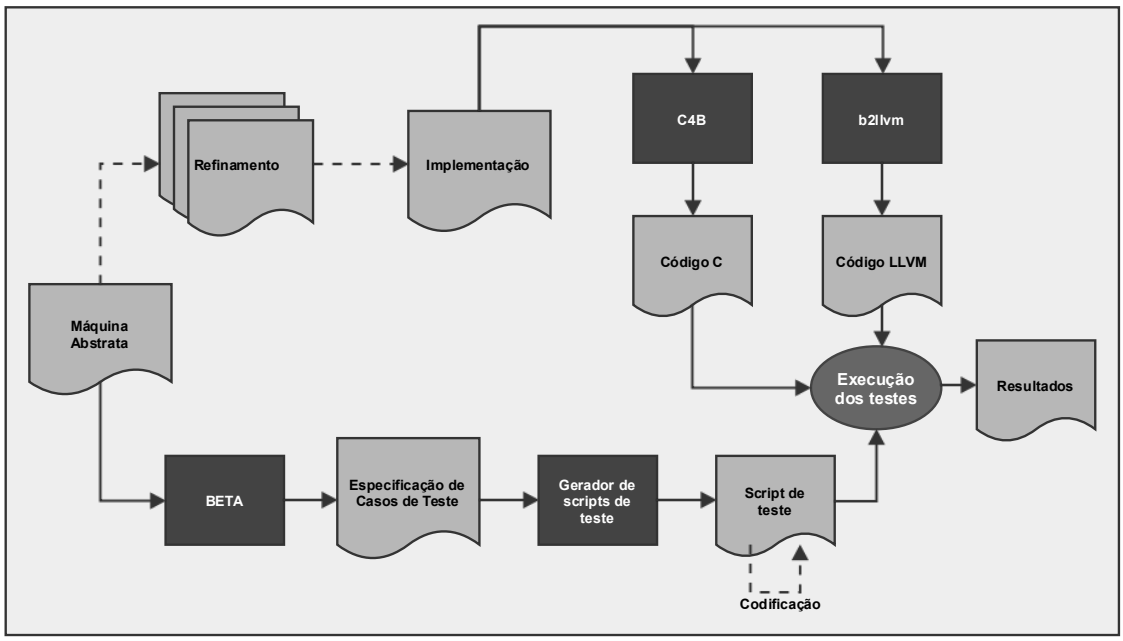


Figura 5.16 – Abordagem de verificação dos geradores *b2llvm* e *C4B* utilizando BETA

as traduções das construções na notação do Método B para código em *LLVM* feitas pelo *b2llvm*. O conjunto de módulos B foi selecionado de maneira *ad hoc* para testar a tradução de cada uma das construções B que são suportadas. A verificação do código gerado é feita de maneira manual através da inspeção do código. Os testes do *b2llvm* também se encaixam como uma validação de tradução, uma vez que possuem a finalidade de verificar se determinados módulos B foram traduzidos da maneira correta pela ferramenta. Diferentemente do que é feito pela equipe de desenvolvimento do *b2llvm*, neste trabalho, os testes gerados por BETA foram utilizados para fazer a verificação do código gerado. Com isso, os testes gerados por BETA foram utilizados como oráculos de teste automatizados, algo que reduziu o esforço nos testes. O *C4B* também possui um conjunto de testes com o mesmo propósito dos testes do *b2llvm*, mas estes não estão disponíveis no repositório do código da ferramenta.

5.2.3 Objetivos

O objetivo deste estudo de caso era utilizar BETA para contribuir com a validação dos geradores *b2llvm* e *C4B*, além de fazer uma avaliação da abordagem e ferramenta. Também era objetivo deste estudo de caso realizar todas as etapas da abordagem, relatar todas as dificuldade encontradas no processo e avaliar a qualidade dos testes gerados por BETA, através de análise de cobertura de comandos e ramificações, e teste de mutação.

5.2.4 Execução do Estudo de Caso

Para este estudo de caso foram selecionados um conjunto de 13 módulos B que apresentam cenários comuns encontrados em projetos que utilizam o Método B. De maneira sucinta, os 13 módulos são descritos a seguir:

- *ATM*: especifica o comportamento de um caixa eletrônico de um banco, possuindo operações para sacar e depositar dinheiro, e verificar o saldo da conta;
- *Sort*: especifica a operação de ordenação de um vetor;
- *Calculator*: especifica operações aritméticas comuns em uma calculadora, que são a adição, subtração, divisão, multiplicação, módulo e exponenciação;
- *Calendar*: especifica uma operação que dado um número retorna o nome do mês que este número representa;
- *Counter*: especifica operações de um contador, que são incrementar ou decrementar o contador, reiniciar o valor do contador e obter o valor atual do contador;
- *Division*: especifica uma operação de divisão;
- *Fifo*: especifica a estrutura de dados fila, possuindo uma operação para inserir um elemento na fila e uma operação para remover um elemento da fila;
- *Prime*: especifica uma operação que verifica se um número é primo;
- *Swap*: especifica operações que manipulam duas variáveis internas e uma operação que troca o valor entre as variáveis;
- *Team*: especifica operações que manipulam jogadores em um time de futebol;
- *TicTacToe*: especifica um jogo popular conhecido como “Jogo da Velha”;
- *Timetracer*: especifica operações que manipulam um cronômetro.
- *Wd*: especifica operações que manipulam um contador regressivo.

A máquina abstrata de cada um dos módulos foi aplicada na ferramenta BETA. Para gerar a especificação de casos de teste para cada uma das operações, foram utilizadas todas as configurações possíveis da ferramenta BETA, que são uma estratégia de particionamento (Classe de Equivalência ou Análise do Valor Limite) e um critério de combinação (*Each Choice*, *Pairwise* ou *All Combinations*). Para cada configuração possível, a ferramenta conseguiu gerar a especificação de casos de teste de todas as operações dos módulos B selecionados para o estudo de caso. Detalhes sobre a geração dos casos de teste e uma análise quantitativa são feitas a seguir.

5.2.4.1 Casos de teste gerados por BETA

No estudo de caso foi possível aplicar todas as combinações de estratégia de particionamento e critério de combinação que a ferramenta BETA permite. A Tabela 5.6 apresenta informações sobre o particionamento do espaço de entrada de cada operação dos módulos selecionados para o estudo de caso. Nela é possível ver o número de variáveis no espaço de entrada de cada operação (*VEE*); a quantidade total de características (*C*); a quantidade de características que particionam o espaço de entrada em mais de um bloco (*C**), que são as características que não são obtidas a partir do invariante e que não são apenas de tipagem; a quantidade total de blocos obtidos ao utilizar a estratégia de particionamento em Classe de Equivalência (*BCE*); a quantidade de blocos obtidos ao utilizar a estratégia de particionamento em Classe de Equivalência considerando apenas os gerados a partir das características que particionam o espaço de entrada em mais de um bloco (*BCE**), que são os blocos que influenciam na quantidade de casos de teste gerados; a quantidade total de blocos obtidos ao utilizar a estratégia de particionamento de Análise de Valor Limite (*BAL*); e a quantidade de blocos obtidos ao utilizar a estratégia de particionamento de Análise de Valor Limite considerando apenas os gerados a partir das características que particionam o espaço de entrada em mais de um bloco (*BAL**).

Na Tabela 5.6 é possível ver que a maioria das operações possuem entre duas e quatro variáveis no seu espaço de entrada, possuem entre quatro e seis características no total e possuem entre uma e duas características considerando apenas as que geram mais de um bloco. Com a estratégia de particionamento em Classe de Equivalência, foram gerados entre cinco e oito blocos no total e entre dois e quatro blocos considerando apenas os que foram gerados a partir de características que geram mais de um bloco. Com a estratégia de particionamento de Análise de Valor Limite, foram gerados entre seis e oito blocos no total e entre três e quatro blocos considerando os que foram gerados a partir de características que geram mais de um bloco. Estes dados podem ser melhor visualizados na Figura 5.17 que apresenta os gráficos *boxplot* das quantidades de variáveis no espaço de entrada (a), características (b), considerando apenas as que geram mais de um bloco, e quantidade de blocos com a estratégia de particionamento em Classe de Equivalência (c) e Análise de Valor Limite (d), considerando apenas os que foram gerados a partir de características que geram mais de um bloco.

Analisando os dados da Tabela 5.6 e os gráficos (c) e (d) da Figura 5.17, é possível ver que a estratégia de particionamento de Análise de Valor Limite gera uma quantidade superior de blocos e, consequentemente, uma quantidade superior de casos de teste, como será visto a seguir. Entretanto, é possível ver que a estratégia de particionamento em Classe de Equivalência gerou a mesma quantidade de blocos para os módulos *ATM*, *Sort*, *Fifo*, *Swap*, *Team*, *Timetracer* e *Wd*. Isso ocorreu porque a estratégia de particionamento de Análise de Valor Limite gera uma quantidade superior de blocos apenas quando são

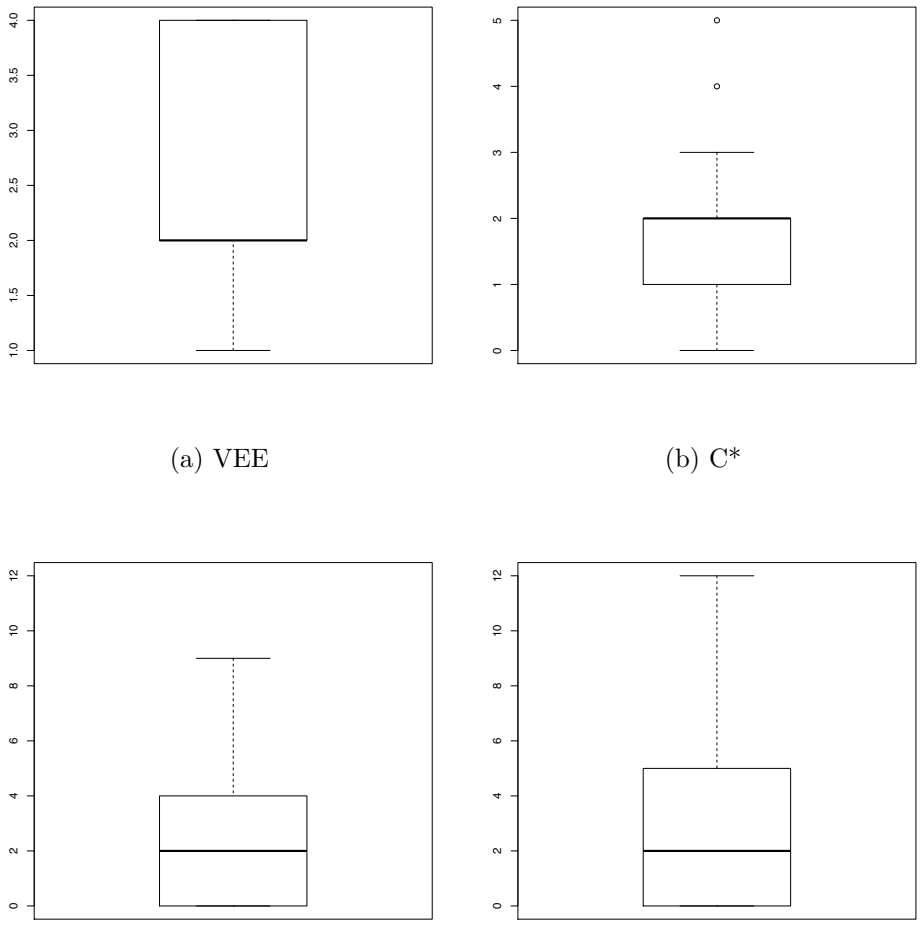
Módulo	Operação	VEE	C	C*	BCE	BCE*	BAL	BAL*
ATM	balance	2	4	0	4	0	4	0
	deposit	3	6	1	7	2	7	2
	withdraw	3	7	2	9	4	9	4
Sort	op_sort	2	5	1	6	2	6	2
Calculator	add	2	2	0	2	0	2	0
	sub	2	2	0	2	0	2	0
	mult	2	2	0	2	0	2	0
	div	2	3	1	4	2	4	2
	modulo	2	3	1	4	2	4	2
	pow	2	3	1	5	3	8	6
Calendar	month	1	2	2	5	5	8	8
Counter	zero	3	7	0	7	0	7	0
	inc	3	9	2	12	5	15	8
	dec	3	9	2	12	5	15	8
	get	3	7	0	7	0	7	0
Division	div	2	2	2	4	4	12	12
Fifo	input	2	4	1	5	2	5	2
	output	1	3	1	4	2	4	2
Prime	isPrime	1	3	3	6	6	10	10
Swap	step	2	2	0	2	0	2	0
	set	4	4	0	4	0	4	0
	get	2	2	0	2	0	2	0
Team	substitute	3	6	2	8	4	8	4
	in_team	2	4	1	5	2	5	2
TicTacToe	BlueMove	4	12	4	17	9	20	12
	RedMove	4	12	4	17	9	20	12
	GameResult	4	11	3	14	6	14	6
Timetracer	enable	4	5	1	6	2	6	2
	run	4	6	2	8	4	8	4
	stop	4	6	2	8	4	8	4
	disable	4	5	1	6	2	6	2
	tick	4	6	2	8	4	8	4
	pick	4	4	0	4	0	4	0
Wd	start	1	3	0	3	0	3	0
	tick	1	4	1	5	2	5	2
	expired	1	4	1	5	2	5	2
Média		**2.6**	**5**	**1.2**	**6.7**	**2.6**	**7.2**	**3.4**

Tabela 5.6 – Informações sobre o particionamento do espaço de entrada dos módulos testados

Legenda: VEE - Variáveis do Espaço de Entrada; C - Características; BCE - Blocos obtidos com a estratégia de particionamento em Classes de Equivalência; BAL - Blocos obtidos com a estratégia de particionamento de Análise de Valor Limite; * - Geram mais de mais de um bloco.

utilizados intervalos numéricos nas restrições do espaço de entrada, quando isso não ocorre, a quantidade gerada é a mesma que é gerada pela estratégia de particionamento em Classe de Equivalência. Neste caso, apenas os módulos *Calculator*, *Calendar*, *Counter*, *Division*, *Prime* e *TicTacToe* que fazem uso de intervalos numéricos tiveram a quantidade de casos de teste diferentes em relação as duas estratégias de particionamento. Dessa forma, nas tabelas e análises seguintes, os módulos que fazem uso de intervalos numéricos serão sublinhados para diferenciar dos módulos que não fazem uso, e a comparação entre as duas estratégias de particionamento será feita considerando apenas os módulos que fazem uso de intervalos numéricos, uma vez que para os que não utilizam as duas estratégias geram os mesmos resultados.

A Tabela 5.7 apresenta um resumo dos casos de teste gerados para as operações



(a) VEE

(b) C*

(c) BCE*

(d) BAL*

Figura 5.17 – *Boxplot* dos dados dos módulos selecionados para o estudo de caso

Legenda: VEE - Variáveis do Espaço de Entrada; C - Características; BCE - Blocos obtidos com a estratégia de particionamento em Classes de Equivalência; BAL - Blocos obtidos com a estratégia de particionamento de Análise de Valor Limite; * - Geram mais de um bloco.

dos módulos selecionados para o estudo de caso. Na tabela, é possível ver a quantidade total de casos de teste satisfatíveis e insatisfatíveis (sinalizados com *) gerados com cada uma das duas estratégias de particionamento (Classe de Equivalência e Análise de Valor Limite) e um critério de combinação (*Each Choice*, *Pairwise* e *All Combinations*). Para os módulos que não fazem uso de intervalos numéricos, os valores das colunas referentes à estratégia de particionamento de Análise de Valor Limite foram substituídos pelo símbolo "=" para indicar que estes possuem os mesmos valores exibidos nas colunas referentes à estratégia de Classe de Equivalência, considerando os mesmos critérios de combinação.

Os resultados apresentados na Tabela 5.7 podem ser melhor visualizados nos gráficos *boxplot* apresentados nas Figura 5.18. Os gráficos (a) e (b) apresentam a quantidade

Módulo	Operação	CEEC	CEEC*	CEPW	CEPW*	CEAC	CEAC*	ALEC	ALEC*	ALPW	ALPW*	ALAC	ALAC*
ATM	balance	1	0	1	0	1	0	=	=	=	=	=	=
	deposit	2	0	2	0	2	0	=	=	=	=	=	=
	withdraw	2	1	4	1	4	1	=	=	=	=	=	=
Sort	op_sort	2	0	2	0	2	0	=	=	=	=	=	=
Calculator	add	1	0	1	0	1	0	1	0	1	0	1	0
	sub	1	0	1	0	1	0	1	0	1	0	1	0
	mult	1	0	1	0	1	0	1	0	1	0	1	0
	div	2	0	2	0	2	0	2	0	2	0	2	0
	modulo	2	0	2	0	2	0	2	0	2	0	2	0
	pow	3	1	3	1	3	1	6	1	6	1	6	1
Calendar	month	3	1	6	2	6	2	6	2	12	6	12	6
Counter	zero	1	0	1	0	1	0	1	0	1	0	1	0
	inc	3	2	6	4	6	4	6	5	12	8	12	8
	dec	3	2	6	4	6	4	6	5	12	8	12	8
	get	1	0	1	0	1	0	1	0	1	0	1	0
Division	div	2	0	4	0	4	0	-	-	-	-	-	-
Fifo	input	2	0	2	0	2	0	=	=	=	=	=	=
	output	2	0	2	0	2	0	=	=	=	=	=	=
Prime	isPrime	2	0	4	2	8	3	6	4	-	-	-	-
Swap	step	1	0	1	0	1	0	=	=	=	=	=	=
	set	1	0	1	0	1	0	=	=	=	=	=	=
	get	1	0	1	0	1	0	=	=	=	=	=	=
Team	substitute	2	0	4	0	4	0	=	=	=	=	=	=
	in_team	2	0	2	0	2	0	=	=	=	=	=	=
TicTacToe	BlueMove	3	1	8	2	24	8	6	1	12	2	48	8
	RedMove	3	1	7	2	24	10	6	1	12	4	48	11
	GameResult	2	1	4	1	8	2	2	1	4	1	8	2
Timetracer	enable	2	0	2	0	2	0	=	=	=	=	=	=
	run	2	0	4	0	4	0	=	=	=	=	=	=
	stop	2	0	4	0	4	0	=	=	=	=	=	=
	disable	2	0	2	0	2	0	=	=	=	=	=	=
	tick	2	0	4	0	4	0	=	=	=	=	=	=
	pick	1	0	1	0	1	0	=	=	=	=	=	=
Wd	start	1	0	1	0	1	0	=	=	=	=	=	=
	tick	2	0	2	0	2	0	=	=	=	=	=	=
	expired	2	0	2	0	2	0	=	=	=	=	=	=
Média		**1.86**	**0.28**	**2.81**	**0.53**	**3.94**	**0.97**	**2.42**	**0.58**	**3.42**	**0.86**	**5.53**	**1.25**

Tabela 5.7 – Quantidade de casos de teste gerados no estudo de caso

Legenda: CEEC - Classe de Equivalência e *Each Choice*; CEPW - Classe de Equivalência e *Pairwise*; CEAC - Classe de Equivalência e *All Combinations*; ALEC - Análise de Valor Limite e *Each Choice*; ALPW - Análise de Valor Limite e *Pairwise*; ALAC - Análise de Valor Limite e *All Combinations*; * - Insatisfatíveis.

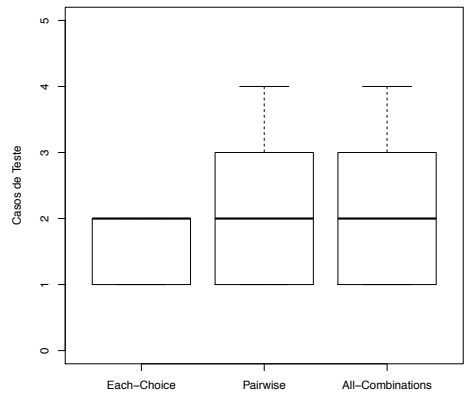
total de casos de teste satisfatíveis e insatisfatíveis, respectivamente, gerados para os módulos que não utilizam intervalos numéricos, sendo, portanto, o resultado que seria obtido com a aplicação de qualquer uma das duas estratégias de particionamento. Os gráficos (c) e (e) apresentam a quantidade total de casos de teste satisfatíveis e insatisfatíveis, respectivamente, gerados com a estratégia de particionamento em Classe de Equivalência para os módulos que fazem uso de intervalos numéricos, e os gráficos (d) e (f) apresentam a quantidade total de casos de teste satisfatíveis e insatisfatíveis, respectivamente, gerados com a estratégia de particionamento de Análise de Valor Limite para os módulos que fazem uso de intervalos numéricos.

Como pode ser observado na Figura 5.18, ao comparar os gráficos (c) e (d), que apresentam a quantidade total de casos de teste satisfatíveis, e os gráficos (e) e (f), que apresentam a quantidade total de casos de teste insatisfatíveis, considerando apenas os módulos que fazem uso de intervalos numéricos, é possível ver que a estratégia de particionamento de Análise de Valor Limite gera uma quantidade superior de casos de teste à estratégia de Classe de Equivalência em ambos os casos (satisfatíveis e insatisfatíveis).

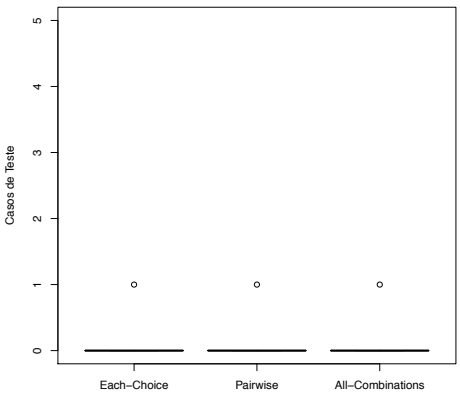
As Tabelas 5.8 e 5.9 apresentam a quantidade de casos de teste positivos e negativos, considerando os casos satisfatíveis e insatisfatíveis (sinalizados com *), gerados com as estratégias de particionamento em Classe de Equivalência e Análise de Valor Limite, respectivamente. Esses resultados podem ser melhor visualizados nos gráficos *bloxplot* apresentados nas Figuras 5.19, 5.20 e 5.21. A Figura 5.19 apresenta os resultados obtidos para os módulos que não fazem uso de intervalos numéricos, a Figura 5.20 apresenta os resultados obtidos com a estratégia de particionamento em Classe de Equivalência para os módulos que fazem uso de intervalos numéricos, e a Figura 5.21 apresenta os resultados obtidos com a estratégia de Análise de Valor Limite para os módulos que fazem uso de intervalos numéricos. Ao analisar os dados das tabelas e os gráficos, é possível ver que a estratégia de particionamento de Análise de Valor Limite também gera uma quantidade superior de casos de teste positivos e negativos em relação à estratégia de particionamento em Classe de Equivalência.

5.2.4.2 Implementação dos testes concretos

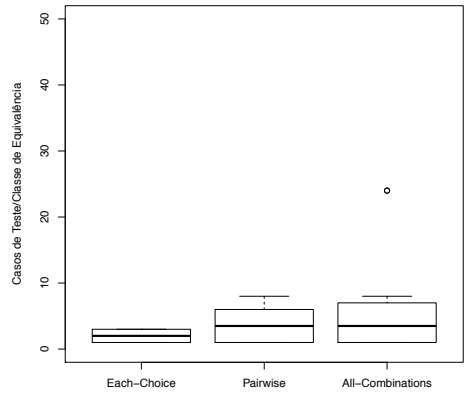
Para a implementação dos testes concretos dos códigos gerados pelo *b2llvm* e *C4B*, foi utilizado o *framework* de teste unitário *CuTest*. O gerador de *scripts* de teste desenvolvido neste trabalho foi utilizado como ferramenta auxiliar na implementação. O gerador já possuía um módulo que gerava *scripts* de teste na notação do *CuTest*, mas como existiam pequenas diferenças entre as notações no *b2llvm* e no *C4B*, como o padrão de nomenclatura das variáveis e chamada das operações, por exemplo, foi necessário fazer pequenas adaptações para gerar os *scripts* para os dois. Primeiro, o gerador foi adaptado para gerar *scripts* de teste para o código gerado pelo *b2llvm*, e, em seguida, foi adaptado



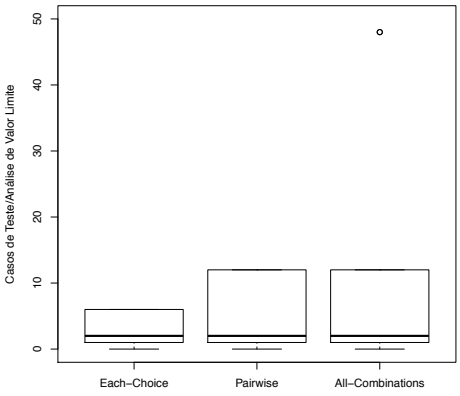
(a) Geral



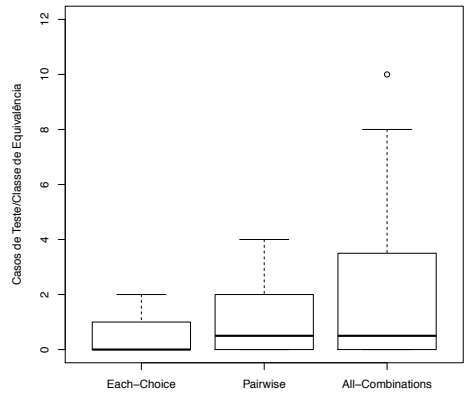
(b) Geral*



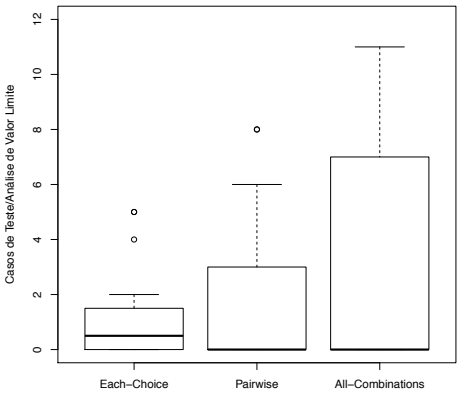
(c) Classe de Equivalência



(d) Análise de Valor Limite



(e) Classe de Equivalência*



(f) Análise de Valor Limite*

Figura 5.18 – *Boxplot* da quantidade total de casos de teste gerados para os módulos selecionados para o estudo de caso

Legenda: * - Insatisfatíveis.

Módulo	Operação	EC P	EC N	PW P	PW N	AC P	AC N	EC* P	EC* N	PW* P	PW* N	AC* P	AC* N
ATM	balance	1	0	1	0	1	0	0	0	0	0	0	0
	deposit	1	1	1	1	1	1	0	0	0	0	0	0
	withdraw	1	0	1	2	1	2	0	1	0	1	0	1
Sort	op_sort	1	1	1	1	1	1	0	0	0	0	0	0
Calculator	add	1	0	1	0	1	0	0	0	0	0	0	0
	sub	1	0	1	0	1	0	0	0	0	0	0	0
	mult	1	0	1	0	1	0	0	0	0	0	0	0
	div	1	1	1	1	1	1	0	0	0	0	0	0
	modulo	1	1	1	1	1	1	0	0	0	0	0	0
	pow	1	1	1	1	1	1	0	1	0	1	0	1
Calendar	month	1	1	2	2	2	2	0	1	0	2	0	2
Counter	zero	1	0	1	0	1	0	0	0	0	0	0	0
	inc	1	0	2	0	2	0	0	2	0	4	0	4
	dec	1	0	2	0	2	0	0	2	0	4	0	4
	get	1	0	1	0	1	0	0	0	0	0	0	0
Division	div	1	1	1	3	1	3	0	0	0	0	0	0
Fifo	input	1	1	1	1	1	1	0	0	0	0	0	0
	output	1	1	1	1	1	1	0	0	0	0	0	0
Prime	isPrime	1	1	1	1	2	3	0	0	0	0	0	0
Swap	step	1	0	1	0	1	0	0	0	0	0	0	0
	set	1	0	1	0	1	0	0	0	0	0	0	0
	get	1	0	1	0	1	0	0	0	0	0	0	0
Team	substitute	1	1	1	3	1	3	0	0	0	0	0	0
	in_team	2	0	2	0	2	0	0	0	0	0	0	0
TicTacToe	BlueMove	0	2	1	5	1	15	0	1	0	2	0	8
	RedMove	0	2	0	5	1	13	0	1	0	2	0	10
	GameResult	0	1	2	1	5	1	1	0	1	0	2	0
Timetracer	enable	1	1	1	1	1	1	0	0	0	0	0	0
	run	1	1	1	3	1	3	0	0	0	0	0	0
	stop	1	1	1	3	1	3	0	0	0	0	0	0
	disable	1	1	1	1	1	1	0	0	0	0	0	0
	tick	1	1	2	2	2	2	0	0	0	0	0	0
	pick	1	0	1	0	1	0	0	0	0	0	0	0
Wd	start	1	0	1	0	1	0	0	0	0	0	0	0
	tick	1	1	1	1	1	1	0	0	0	0	0	0
	expired	1	1	1	1	1	1	0	0	0	0	0	0
Média		**0.94**	**0.64**	**1.14**	**1.14**	**1.29**	**1.69**	**0.03**	**0.25**	**0.03**	**0.44**	**0.06**	**0.83**

Tabela 5.8 – Casos de teste positivos e negativos gerados com a estratégia de Classes de Equivalência

Legenda: EC - *Each Choice*; PW - *Pairwise*; AC - *All Combinations*; P - Positivos; N - Negativos; * - Insatisfatíveis.

Módulo	Operação	EC P	EC N	PW P	PW N	AC P	AC N	EC* P	EC* N	PW* P	PW* N	AC* P	AC* N
ATM	balance	1	0	1	0	1	0	0	0	0	0	0	0
	deposit	1	1	1	1	1	1	0	0	0	0	0	0
	withdraw	1	0	1	2	1	2	0	1	0	1	0	1
Sort	op_sort	1	1	1	1	1	1	0	0	0	0	0	0
Calculator	add	1	0	1	0	1	0	0	0	0	0	0	0
	sub	1	0	1	0	1	0	0	0	0	0	0	0
	mult	1	0	1	0	1	0	0	0	0	0	0	0
	div	1	1	1	1	1	1	0	0	0	0	0	0
	modulo	1	1	1	1	1	1	0	0	0	0	0	0
	pow	4	1	4	1	4	1	0	1	0	1	0	1
Calendar	month	3	1	4	2	4	2	0	2	0	6	0	6
Counter	zero	1	0	1	0	1	0	0	0	0	0	0	0
	inc	1	0	4	0	4	0	3	2	4	4	4	4
	dec	1	0	4	0	4	0	3	2	4	4	4	4
	get	1	0	1	0	1	0	0	0	0	0	0	0
Division	div	-	-	-	-	-	-	-	-	-	-	-	-
Fifo	input	1	1	1	1	1	1	0	0	0	0	0	0
	output	1	1	1	1	1	1	0	0	0	0	0	0
Prime	isPrime	0	2	-	-	-	-	2	2	-	-	-	-
Swap	step	1	0	1	0	1	0	0	0	0	0	0	0
	set	1	0	1	0	1	0	0	0	0	0	0	0
	get	1	0	1	0	1	0	0	0	0	0	0	0
Team	substitute	1	1	1	3	1	3	0	0	0	0	0	0
	in_team	2	0	2	0	2	0	0	0	0	0	0	0
TicTacToe	BlueMove	0	5	0	10	4	36	0	1	0	2	0	8
	RedMove	0	5	0	8	4	33	0	1	0	4	0	11
	GameResult	0	1	2	1	5	1	1	0	1	0	2	0
Timetracer	enable	1	1	1	1	1	1	0	0	0	0	0	0
	run	1	1	1	3	1	3	0	0	0	0	0	0
	stop	1	1	1	3	1	3	0	0	0	0	0	0
	disable	1	1	1	1	1	1	0	0	0	0	0	0
	tick	1	1	2	2	2	2	0	0	0	0	0	0
	pick	1	0	1	0	1	0	0	0	0	0	0	0
Wd	start	1	0	1	0	1	0	0	0	0	0	0	0
	tick	1	1	1	1	1	1	0	0	0	0	0	0
	expired	1	1	1	1	1	1	0	0	0	0	0	0
Média		**1.03**	**0.81**	**1.31**	**1.25**	**1.61**	**2.67**	**0.25**	**0.33**	**0.25**	**0.61**	**0.28**	**0.97**

Tabela 5.9 – Casos de teste positivos e negativos gerados com a estratégia de Análise de Valor Limite

Legenda: EC - *Each Choice*; PW - *Pairwise*; AC - *All Combinations*; P - Positivos; N - Negativos; * - Insatisfatíveis.

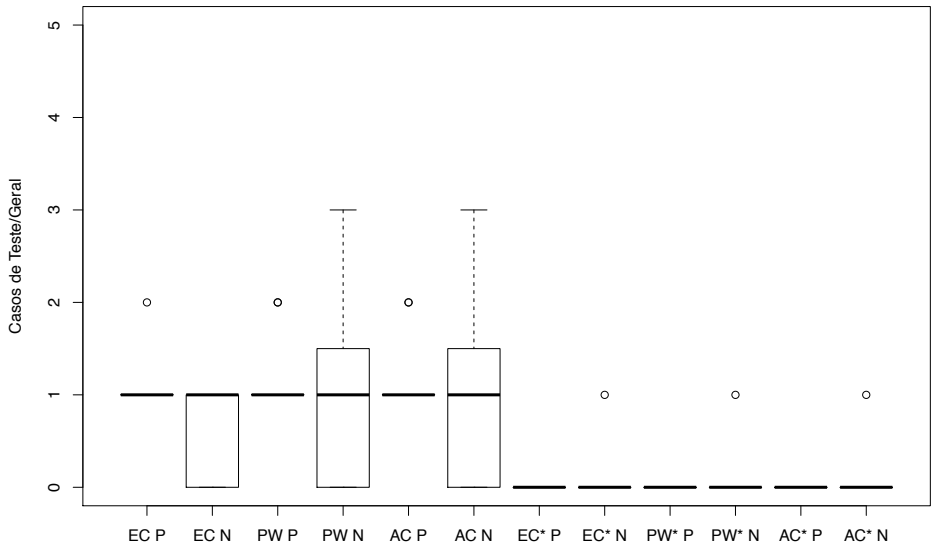


Figura 5.19 – *Boxplot* da quantidade de casos de teste positivos e negativos gerados para os módulos que não utilizam intervalos numéricos

Legenda: EC - *Each Choice*; PW - *Pairwise*; AC - *All Combinations*; P - Positivos; N - Negativos; * - Insatisfatíveis.

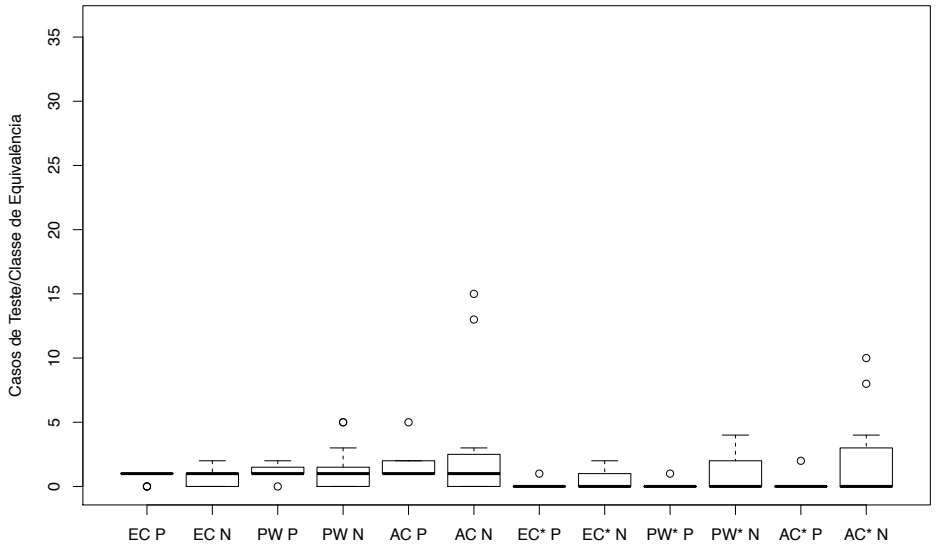


Figura 5.20 – *Boxplot* da quantidade de casos de teste positivos e negativos gerados com a estratégia de particionamento em Classe de Equivalência

Legenda: EC - *Each Choice*; PW - *Pairwise*; AC - *All Combinations*; P - Positivos; N - Negativos; * - Insatisfatíveis.

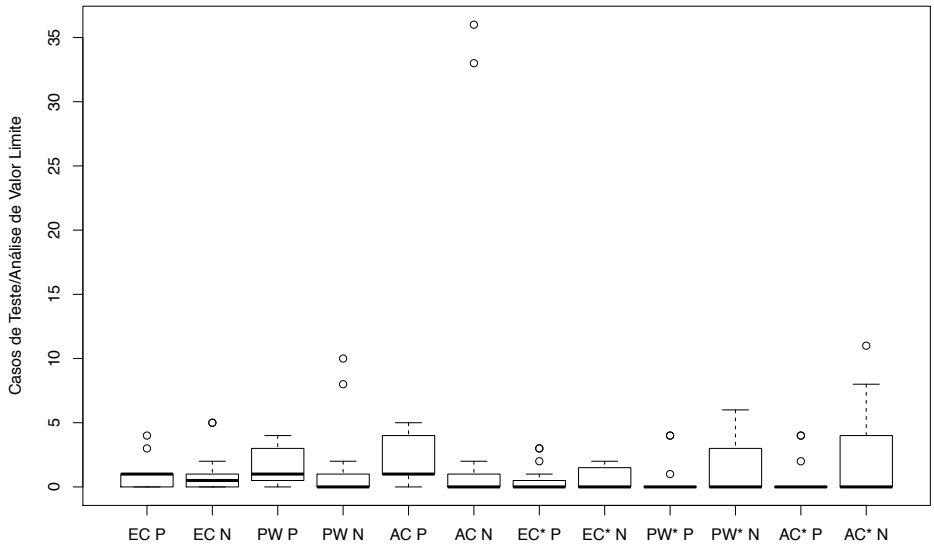


Figura 5.21 – *Boxplot* da quantidade de casos de teste positivos e negativos gerados com a estratégia de particionamento de Análise de Valor Limite

Legenda: EC - *Each Choice*; PW - *Pairwise*; AC - *All Combinations*; P - Positivos; N - Negativos; * - Insatisfatíveis.

para gerar *scripts* para o código gerado pelo *C4B*. A adaptação do gerador para o *C4B* foi mais rápida porque apenas algumas poucas adaptações foram necessárias para adequar o modelo de *script* desenvolvido para o *b2llvm*, reutilizando a maior parte do código. Isso mostra que o gerador de *scripts* de teste desenvolvido neste trabalho pode facilmente ser adaptado para as necessidades do projeto.

Para projetar os oráculos de teste, foram utilizadas as estratégias de verificação das variáveis de estado e valores de retorno. Como já havia sido discutido no primeiro estudo de caso, essas duas estratégias apresentam os melhores resultados e podem ser automatizadas pelo *ProB*, que foi utilizado para a obtenção dos valores esperados nos oráculos de teste. Apesar das estratégias de verificação das variáveis de estado e valores de retorno possuírem uma maior precisão, como foi discutido no Capítulo 4, neste estudo de caso também foi utilizada a estratégia de verificação do invariante de estado nos oráculos de teste. A estratégia de verificação do invariante de estado é parcialmente automatizada pelo gerador de *scripts* de teste, dessa forma, o esforço para implementar os oráculos com essa estratégia é reduzido ou, até mesmo, eliminado, quando o módulo sob teste possui um invariante de estado simples, algo que era o caso da maioria dos módulos B utilizados neste estudo de caso.

A adaptação dos *scripts* também seguiu a mesma ordem de adaptação do gerador de *scripts* de teste, primeiro foram adaptados os *scripts* gerados para o código do *b2llvm*

e, em seguida, foram adaptados os *scripts* gerados para o código do *C4B*. A adaptação consistiu em remover os testes negativos, uma vez que estes não puderam ser aplicados, refinar os dados de teste gerados por BETA, seguindo o refinamento feito na implementação B, e inserir os valores esperados nos oráculos de teste.

O gerador de código *b2llvm* não gerou código *LLVM* para os módulos *Fifo*, *Calendar*, *ATM* e *Timetracer*, pois este ainda se encontra em desenvolvimento e não suporta algumas das construções que foram utilizadas por esses módulos, por isso, apenas o código *C* gerado pelo *C4B* para esses módulos foi testado. Os resultados da execução dos testes são apresentados a seguir, assim como uma avaliação e discussão dos testes gerados por BETA.

5.2.5 Avaliação e Discussões

Após a implementação, todos os testes implementados para os códigos gerados pelo *b2llvm* e *C4B* foram executados e seus resultados foram analisados. Uma vez que apenas os testes positivos foram executados, era esperado, caso os códigos tenham sido gerados corretamente, que todos os testes passassem. Em caso contrário, os códigos gerados eram analisados para a identificação de possíveis erros na sua geração, seguindo a abordagem apresentada na Figura 5.16. Com exceção de um teste gerado para o módulo *Wd*, que falhou para ambos os códigos *LLVM* e *C*, e alguns testes gerados para o módulo *Timetracer*, que falhou apenas para o código *C* uma vez que o código *LLVM* não foi gerado, todos os testes dos outros módulos passaram. Os dois módulos que tiveram falhas nos testes, *Wd* e *Timetracer*, são analisados e discutidos a seguir.

Com a falha de um dos testes do módulo *Wd* para ambos os códigos *LLVM* e *C*, os dois códigos gerados foram analisados e comparados com a implementação B que os originaram. A análise não detectou nenhum erro na geração dos códigos, pois ambos estavam em conformidade com a implementação B. Dessa forma, a implementação B foi analisada e foi detectado que esta não estava correspondendo à máquina abstrata de origem. A implementação B do módulo *Wd* não foi propriamente validada, isso fez com que um erro de refinamento em uma das operações não fosse percebido, sendo a causa da falha dos testes que foram gerados a partir da máquina abstrata. Este resultado mostrou que a abordagem e ferramenta BETA pode ajudar a encontrar erros de refinamento em implementações quando estas não forem propriamente validadas. O erro de refinamento na implementação B do módulo *Wd* foi corrigido para ser utilizado na avaliação de BETA.

O código *C* gerado para o módulo *Timetracer* foi analisado após as falhas de alguns testes. Após a análise, foram detectados alguns erros no código gerado pelo *C4B*. A implementação B do módulo *Timetracer* faz a importação de outro módulo B. O *C4B* conseguiu gerar o código corretamente para o módulo importado, entretanto, o código *C* gerado para o módulo *Timetracer* não fez a importação do outro módulo, dessa forma,

quando alguma operação do módulo importado era chamada, esta chamada não aparecia no código gerado, sendo a causa do erro no código. Após uma análise mais profunda do *C4B*, foi detectado que o gerador não suporta a renomeação e a importação múltipla de um mesmo módulo, algo que é permitido no Método B e que foi utilizado no módulo *Timetracer*. Este problema foi reportado para *ClearSy*[12], a empresa que desenvolve o *Atelier B* e o *C4B*. Uma vez que o *C4B* gerou incorretamente o código do módulo *Timetracer*, este módulo não foi considerado na avaliação dos testes gerados por BETA.

Os testes gerados por BETA foram avaliados através de análise de cobertura de comandos e ramificações, e teste de mutação. Para esta avaliação, não foram considerados os códigos gerados pelo *b2llvm* devido a dificuldade em encontrar ferramentas que automatizassem a análise de código *LLVM* e por ser inviável fazer de forma manual, sendo considerado apenas os códigos *C* gerados pelo *C4B*. Para a análise de cobertura de comandos e ramificações foi utilizado, como ferramenta auxiliar, o *Gcov*, e para o teste de mutação foi utilizada a ferramenta *Milu* (JIA; HARMAN, 2008).

As Tabelas 5.10 e 5.11 apresentam, respectivamente, a cobertura de comandos e ramificações de cada módulo considerado nesta avaliação, considerando os testes positivos gerados por cada estratégia de particionamento e cada critério de combinação. Na Tabela 5.10 é possível ver o número de comandos de cada módulo, e a porcentagem de comandos cobertos por cada combinação de estratégia de particionamento e critério de combinação. De forma semelhante, na Tabela 5.11 é possível ver o número de ramificações de cada módulo e a cobertura obtida com cada combinação de estratégia de particionamento e critério de combinação. Em ambas as tabelas, para os módulos que não utilizam intervalos numéricos (módulos que não estão sublinhados), os resultados obtidos com a estratégia de particionamento de Análise de Valor Limite foram substituídos pelo símbolo "=" para indicar que estes possuem os mesmos valores exibidos nas colunas referentes a estratégia de Classe de Equivalência, considerando os mesmos critérios de combinação.

Na Tabela 5.10 é possível ver que, com exceção de dois casos para os módulos *Prime* e *TicTacToe*, os testes gerados com a estratégia de particionamento de Análise de Valor Limite conseguem uma melhor cobertura de comandos que a estratégia de Classe de Equivalência, considerando os módulos que utilizam intervalos numéricos. Os testes gerados para o módulo *Prime* com o critério de combinação *Each Choice* e para o módulo *TicTacToe* com o critério *Pairwise* tiveram uma cobertura melhor com a estratégia de particionamento em Classe de Equivalência. Isso ocorreu porque, em algumas ocasiões, o maior número de blocos gerados com a estratégia de Análise de Valor Limite pode propiciar um maior número de casos de teste insatisfatíveis, resultando em menos testes. De maneira geral, os testes gerados com o critério de combinação *All Combinations* obtiveram uma melhor cobertura, seguido dos critérios *Pairwise*, que em alguns casos obteve o

[12] *ClearSy* é uma empresa especializada em sistemas de alta integridade localizada na França.

Módulo	Comandos	CEEC	CEPW	CEAC	ALEC	ALPW	ALAC
ATM	14	100%	14 100%	100%	=	=	=
Sort	41	100%	100%	100%	=	=	=
Calculator	39	69.2%	69.2%	69.2%	100%	100%	100%
Calendar	30	20%	26.7%	26.7%	33.3%	40%	40%
Counter	23	73.9%	100%	100%	73.9%	100%	100%
*Division**	9	66.7%	66.7%	66.7%	-	-	-
Fifo	18	100%	100%	100%	=	=	=
*Prime**	15	66.7%	66.7%	80%	0%	-	-
Swap	16	100%	100%	100%	=	=	=
Team	41	97.6%	97.6%	97.6%	=	=	=
TicTacToe	34	0%	88.2%	100%	0%	79.4%	100%
Wd	27	100%	100%	100%	=	=	=
Média		**72.8%**	**86.2%**	**87.6%**	**78.4%**	**90.8%**	**93.3%**

Tabela 5.10 – Análise de cobertura de comandos

Legenda: CEEC - Classe de Equivalência e *Each Choice*; CEPW - Classe de Equivalência e *Pairwise*; CEAC - Classe de Equivalência e *All Combinations*; ALEC - Análise de Valor Limite e *Each Choice*; ALPW - Análise de Valor Limite e *Pairwise*; ALAC - Análise de Valor Limite e *All Combinations*; * - Não considerado na média.

Módulo	Ramificações	CEEC	CEPW	CEAC	ALEC	ALPW	ALAC
*ATM**	0	-	-	-	-	-	-
Sort	8	75%	75%	75%	=	=	=
Calculator	12	33.3%	33.3%	33.3%	75%	75%	75%
Calendar	13	7.7%	15.4	15.4	23.1%	30.8%	30.8%
Counter	4	50%	100%	100%	50%	100%	100%
*Division**	2	50%	50%	50%	-	-	-
*Fifo**	0	-	-	-	-	-	-
*Prime**	4	50%	50%	100%	0%	-	-
*Swap**	0	-	-	-	-	-	-
Team	8	87.5%	87.5%	87.5%	=	=	=
TicTacToe	96	0%	18.8%	39.6%	0%	18.8%	39.6%
Wd	6	100%	100%	100%	=	=	=
Média		**17.7%**	**32%**	**45.6%**	**22.4%**	**36.7%**	**50.3%**

Tabela 5.11 – Análise de cobertura de ramificações

Legenda: CEEC - Classe de Equivalência e *Each Choice*; CEPW - Classe de Equivalência e *Pairwise*; CEAC - Classe de Equivalência e *All Combinations*; ALEC - Análise de Valor Limite e *Each Choice*; ALPW - Análise de Valor Limite e *Pairwise*; ALAC - Análise de Valor Limite e *All Combinations*; * - Não considerado na média.

mesmo resultado, e *Each Choice*. Ao se analisar as médias, é possível ver que os resultados obtidos com o critério *Pairwise* são próximos dos obtidos com *All Combinations*. Isso mostra que o critério *Pairwise* possui uma boa relação de custo/benefício, uma vez que ele consegue obter quase os mesmos resultados do critério *All Combinations* com menos testes.

De forma semelhante à cobertura de comandos, a cobertura de ramificações apresentada na Tabela 5.11 manteve o mesmo padrão em relação a cobertura obtida com cada estratégia de particionamento e critério de combinação. Entretanto, é possível observar que para os módulos que possuem um maior número de ramificações o nível de cobertura foi baixo. Este resultado mostra que nos casos mais complexos em que existe um maior número de ramificações, ou possíveis decisões, os testes gerados por BETA não conseguem um nível satisfatório de cobertura. Este mesmo resultado foi observado no primeiro

estudo de caso, o que reforça que é necessário revisar a abordagem atual para melhorar a qualidade dos testes gerados. Como foi dito anteriormente, uma possível solução para este problema, que já se encontra em desenvolvimento, é inserir critérios de cobertura lógica na abordagem e ferramenta BETA, possibilitando testar melhor as possíveis decisões no módulo sob teste.

Para o teste de mutação, foram aplicados os seguintes operadores de mutação: *OAAN* (substituição de operador aritmético), *OAAA* (substituição de operador de atribuição aritmética), *ORRN* (substituição de operador relacional), *OLLN* (substituição de operador lógico), *OLNG* (negação lógica), *OCNG* (negação de contexto lógico), *CRCR* (substituição de constante), *SSDL* (eliminação de comando), *ABS* (inserção de valor absoluto) e *UOI* (inserção de operador unário). Todos os operadores de mutação aplicados são suportados pela ferramenta *Milu* e foram apresentados na Subseção 2.1.3 do Capítulo 2. A ferramenta *Milu* automatizou o processo de geração dos mutantes. A verificação dos mutantes para detectar os equivalentes foi feita de forma manual fazendo uso do Modelo *RIP* apresentado na subseção 2.1.1 do capítulo 2. A execução e verificação dos mutantes mortos foram automatizadas através do desenvolvimento de *scripts* de execução em *shell script*.

A Tabela 5.12 apresenta os resultados do teste de mutação. Nela é possível ver o número total de mutantes gerados, o número de mutantes equivalentes, que foram detectados manualmente, e o escore de mutação (razão entre o número de mutantes mortos e o número de mutantes não equivalentes) obtido com os testes positivos gerados por cada combinação de estratégia de particionamento e critério de combinação. Devido ao número elevado de mutantes gerados para o módulo *TicTacToe*, não foi viável analisar todos os seus mutantes manualmente para detectar os equivalentes, por isso foi considerado, dada a análise de uma pequena amostra, que não existiam mutantes equivalentes para o módulo.

Assim como foi observado na análise de cobertura de comandos e ramificações, os resultados do teste de mutação mostraram que os testes gerados com a estratégia de particionamento de Análise de Valor Limite possuem um melhor resultado que os testes gerados com a estratégia de Classe de Equivalência, além do melhor resultado com o critério de combinação *All Combinations*, seguido dos critérios *Pairwise* e *Each Choice*. Também foi possível observar a proximidade entre os resultados obtidos com os critérios *All Combinations* e *Pairwise*. Os resultados do teste de mutação mostram, de maneira geral, que é necessário melhorar a qualidade dos testes gerados por BETA. Esta melhoria poderá ser obtida com uma revisão da abordagem, melhorando a caracterização do espaço de entrada e inserindo critérios mais rigorosos de cobertura, como já mencionado.

De forma comum nos resultados da análise de cobertura de linhas de código e ramificações, e no teste de mutação, os testes gerados por BETA não apresentaram bons resultados para os módulos *Calendar* e *TicTacToe*. O módulo *Calendar* especifica uma

Módulo	**M**	**ME**	**CEEC**	**CEPW**	**CEAC**	**ALEC**	**ALPW**	**ALAC**
ATM	13	2	81.8%	81.8%	81.8%	=	=	=
Sort	133	10	89.4%	89.4%	89.4%	=	=	=
Calculator	122	2	47.5%	47.5%	47.5%	74.2%	74.2%	74.2%
Calendar	67	0	9%	19.4%	19.4%	25.4%	35.8%	35.8%
Counter	87	0	41.4%	85.1%	85.1%	41.4%	94.2%	94.2%
*Division**	29	0	31%	31%	31%	-	-	-
Fifo	40	0	90%	90%	90%	=	=	=
*Prime**	66	0	34.8%	34.8%	53%	0%	-	-
Swap	8	0	100%	100%	100%	=	=	=
Team	111	22	68.5%	68.5%	68.5%	=	=	=
TicTacToe	764	0	0%	21.9%	40.3%	0%	20.4%	40.3%
Wd	72	4	91.2%	91.2%	91.2%	=	=	=
Média			**61.9%**	**69.5%**	**71.3%**	**66.2%**	**74.5%**	**76.5%**

Tabela 5.12 – Resultados do Teste de Mutação

Legenda: M - Quantidade de mutantes gerados; ME - Quantidade de mutantes equivalentes; CEEC - Classe de Equivalência e *Each Choice*; CEPW - Classe de Equivalência e *Pairwise*; CEAC - Classe de Equivalência e *All Combinations*; ALEC - Análise de Valor Limite e *Each Choice*; ALPW - Análise de Valor Limite e *Pairwise*; ALAC - Análise de Valor Limite e *All Combinations*; * - Não considerado na média.

operação que dado um número indica qual mês este número representa. Para fazer a verificação dos meses o módulo *Calendar* utiliza uma substituição do tipo *CASE*. Atualmente, a ferramenta BETA considera apenas a primeira constante ou grupo de constantes que são verificadas em uma substituição *CASE*, desconsiderando as outras constantes ou grupos. Dessa forma, o espaço de entrada não é totalmente considerado na caracterização, prejudicando os testes. Este problema relacionado a substituição *CASE* também foi observado no primeiro estudo de caso, o que reforça que é necessário melhorar a caracterização do espaço de entrada em BETA e passar a considerar todas as condições verificadas em substituições *CASE* ou outras condicionais, não comprometendo, assim, a qualidade dos testes.

O módulo *TicTacToe* especifica o "Jogo da Velha". Dentre todos os módulos testados, o módulo *TicTacToe* é o que possui o maior número de ramificações, ou possíveis decisões, e maior número de mutantes gerados, como pode ser observado nas tabelas 5.11 e 5.12, respectivamente. As abstrações feitas para representar o jogo na máquina abstrata do módulo *TicTacToe* simplificam o tabuleiro e a verificação do resultado de uma partida. Entretanto, na implementação B do módulo é feito um maior número de verificações para determinar este resultado, uma vez que todas as possibilidades devem ser verificadas. Essas verificações são feitas através de estruturas condicionais que não são utilizadas na máquina abstrata. Dessa forma, a geração dos testes não se beneficia das informações contidas nas implementações B que, possivelmente, seriam interessantes para gerar testes com mais qualidade. Portanto, uma das possíveis melhorias para BETA seria gerar testes a partir de implementações B. Além disso, outro benefício dessa abordagem seria gerar dados de teste concretos, uma vez que implementações B não permitem dados abstratos, não sendo necessário, portando, refinar os dados manualmente.

Um dos fatores que influenciaram nos resultados das análises de cobertura e teste de mutação foi a pouca variação nos dados de teste gerados por BETA. De maneira geral, foi observado que a ferramenta BETA não varia muito os valores gerados para as variáveis do espaço de entrada, algo que fez com que algumas variáveis sempre ficassem com os mesmos valores em diferentes casos de teste. Isso ocorre porque o solucionador de restrições utilizado pela ferramenta, o *ProB*, sempre calcula os valores mínimos, ou primeiros valores, que satisfazem as restrições dos casos de teste, fazendo com que os casos de teste que possuem restrições em comum ficassem sempre com os mesmos valores em algumas variáveis. Uma maior variação nos dados ajudaria a cobrir mais o código nos testes, principalmente nos casos em que existem mais situações a serem testadas, como o módulo *TicTacToe*. Isso mostra que é necessário melhorar a etapa de obtenção dos dados de teste de BETA para que sejam gerados dados mais variados.

5.2.6 Conclusões

Neste estudo de caso a abordagem e ferramenta BETA foi utilizada para contribuir com a verificação e validação de dois geradores de código para o Método B, o *b2llvm* e o *C4B*. Os testes gerados por BETA ajudaram com a verificação de equivalências funcionais entre o modelo de origem e o código gerado. Como resultado, foram encontradas limitações e problemas em ambos os geradores de código que foram reportados para os seus respectivos desenvolvedores, contribuindo para a melhoria do *b2llvm* e *C4B*. Os testes gerados por BETA também ajudaram a identificar erros em uma implementação B que não havia sido devidamente verificada, o que mostra outra possível aplicação de BETA.

O estudo de caso também mostrou a aplicabilidade de BETA, além da viabilidade da abordagem adotada para fazer a verificação de geradores de código. Além disso, este estudo também mostrou a aplicabilidade do gerador de scripts de teste desenvolvido neste trabalho, que ajudou a diminuir o esforço do processo de teste ao automatizar parte da implementação. A avaliação dos testes gerados por BETA, através da análise de cobertura de linhas de código e ramificações, e teste de mutação, revelou que é necessário revisar a abordagem para melhorar a qualidade dos testes gerados por BETA. As possíveis melhorias levantadas são:

- *Melhorar a caracterização do espaço de entrada e a obtenção dos dados de teste*: Atualmente, a caracterização do espaço de entrada feita pela ferramenta não considera todas as condições utilizadas, o que prejudica a qualidade dos testes. Isso pode ser observado nos casos que são utilizadas substituições condicionais aninhadas e em substituições do tipo *CASE*. Para melhorar a qualidade dos testes é necessário considerar todas as substituições condicionais e verificações que são feitas no módulo sob teste. Também foi observado que os dados de teste gerados pela ferramenta não apresentam muita variação, algo que pode prejudicar a qualidade dos testes. Para

melhorar isso é necessário modificar as configurações do solucionador de restrições utilizado, o *ProB*, ou fazer otimizações em BETA para forçar uma maior variação nos dados de teste gerados;

- *Inserir critérios de cobertura lógica na abordagem*: A análise de cobertura de ramificações, principalmente, revelou que os testes gerados por BETA não conseguem cobrir bem o módulo sob teste quando existem um grande número de ramificações, ou possíveis decisões. Uma das formas de melhorar isso é através da utilização de critérios de cobertura lógica na geração dos testes, uma vez que esses critérios consideram todas as condições lógicas existentes no módulo e podem, dependendo do critério escolhido, cobrir todos os possíveis resultados dessas condições, o que faz com que todas as possíveis decisões sejam testadas. Esta melhoria já se encontra em desenvolvimento;

- *Gerar testes também a partir de implementações B*: A ferramenta e abordagem BETA gera testes a partir de máquinas abstratas. Como foi observado no estudo de caso, o refinamento da máquina abstrata traz uma série de informações que seriam relevantes para a geração dos testes. Dessa forma, gerar testes a partir de implementações B seria uma forma de gerar testes com mais qualidade, além do benefício de gerar dados de teste mais concretos.

6 Considerações Finais

A demanda crescente por sistemas robustos e seguros faz com que seja imprescindível utilizar técnicas de validação e verificação para desenvolver software com qualidade. Métodos Formais e Teste de Software estão entre as principais abordagens que visam a qualidade. A união entre essas duas abordagens vem ganhando muito interesse, pois ambas podem se complementar e trazer vários benefícios para a qualidade do software. Nesse sentido, vários trabalhos foram desenvolvidos com o intuito de aliar teste de software e métodos formais.

No trabalho iniciado em (SOUZA, 2009) e aprimorado em (MATOS; MOREIRA, 2012) e (MATOS; MOREIRA, 2013), foi desenvolvida a abordagem e ferramenta BETA. Com a abordagem, é possível gerar testes de unidade positivos e negativos a partir de uma especificação formal escrita na notação do Método B. A ferramenta automatiza parte da abordagem, ficando apenas as etapas finais, de definição dos oráculos de teste e implementação dos testes concretos, para serem realizadas de forma manual.

O objetivo deste trabalho era contribuir com a evolução e aperfeiçoamento da abordagem e ferramenta BETA. Com esse intuito, este trabalho agiu em duas linhas de ação. Na primeira linha, foram propostas e implementadas melhorias para as duas últimas etapas da abordagem BETA. Para a etapa de definição dos oráculos de teste, este trabalho propôs estratégias de oráculos que trouxeram mais flexibilidade e uma melhor definição para a etapa. Para a etapa de implementação dos testes concretos, este trabalho desenvolveu um gerador de *scripts* de teste que automatizou parte do processo de implementação, diminuindo os custos desta etapa.

Na segunda linha de ação, foi realizada uma avaliação aprofundada da abordagem e ferramenta com o intuito de identificar as limitações e qualidades de BETA. Para isso, a abordagem e ferramenta foi avaliada em dois estudos de caso. No primeiro estudo de caso, foi aplicada na ferramenta uma especificação parcial em B da API de Lua. Essa especificação oferecia complexidade e situações que ainda não haviam sido exploradas em BETA. No segundo estudo de caso, BETA foi utilizada para contribuir com a validação de dois geradores de código para o Método B, o *b2llvm* e o *C4B*. Em ambos os estudos de caso, foram realizadas todas as etapas da abordagem BETA, algo que não havia sido feito até então.

As avaliações feitas em ambos os estudos de caso mostraram que os testes gerados por BETA são promissores, mas que ainda precisam ser melhorados para que possa ser feito um bom teste do sistema que está sendo desenvolvido. Além disso, os estudos de caso revelaram limitações na abordagem e ferramenta que não haviam sido detectadas em

trabalhos anteriores. Como resultado, este trabalho conseguiu mostrar vários pontos que podem ser melhorados e desenvolvidos em BETA. Dessa forma, foram estabelecidas uma série de diretivas para trabalhos futuros que podem contribuir com o aperfeiçoamento da abordagem e ferramenta. Os possíveis trabalhos futuros são listados a seguir:

- *Inserir na abordagem BETA uma etapa de especificação do comportamento excepcional em B*: como foi apresentado, o comportamento excepcional ou negativo de uma operação (em que as precondições da operação não são respeitadas) não pode ser determinado a partir da máquina abstrata B. Dessa forma, projetar oráculos de teste que avaliam os testes negativos se torna uma tarefa mais difícil e que é dependente de informações externas a especificação, como decisões da equipe de desenvolvimento por exemplo. Especificar o comportamento excepcional em B é uma possível solução para este problema. Portanto, especificar o comportamento excepcional em B poderia ser incluído como uma etapa complementar da abordagem BETA;

- *Estudar a aplicação dos testes negativos gerados por BETA*: os casos de teste negativos são importantes para testar o comportamento do sistema em situações excepcionais, avaliando, assim, a sua robustez. Nos estudos de caso realizados neste trabalho não foram aplicados os casos de teste negativos na etapas finais do processo de teste. Dessa forma, não foi possível avaliar a qualidade dos testes negativos. Por causa disso, será importante realizar novos estudos que tenham como foco principal avaliar os testes negativos gerados por BETA;

- *Automatizar a adaptação dos scripts de teste*: os *scripts* gerados pelo gerador de *scripts* de teste desenvolvido neste trabalho precisam ser adaptados para poderem ser executados. Esta adaptação envolve o refinamento de dados e a inserção de valores nos oráculos de teste. Esta adaptação pode ser automatizada com a criação de um método de refinamento para os testes, semelhante a linguagem *FTCRL* apresentada em (CRISTIÁ et al., 2011), ou fazendo uso de um módulo de implementação B;

- *Melhorar o feedback da ferramenta BETA*: em um primeiro momento no estudo de caso da API de Lua, a ferramenta BETA apresentou mensagens de erro que não indicavam corretamente o problema que estava ocorrendo. As mensagens apresentadas confundiram e não contribuíram para a solução do problema. Dessa forma, é preciso melhorar o *feedback* da ferramenta para o usuário, fazendo com que a ferramenta possa dar mensagens mais precisas, ajudando o usuário a identificar um problema quando ocorrer;

- *Pesquisar outros solucionadores de restrições ou otimizações no ProB*: a ferramenta BETA utiliza o *ProB* para gerar os dados dos casos de teste. Especificações muito

complexas, como a especificação da API de Lua, podem sobrecarregar o *ProB*, fazendo com que ele não consiga verificar e animar a especificação. Quando isso ocorre, o processo de teste é prejudicado porque os dados não conseguem ser gerados. Uma possível solução para este problema seria fazer otimizações para tentar fazer com que o *ProB* consiga animar e verificar a máquina. Outra solução seria pesquisar um solucionador de restrições mais eficiente para ser utilizado na ferramenta BETA;

- *Melhorar as etapas de caracterização do espaço de entrada e obtenção dos dados de teste*: em ambos os estudos de caso foi possível observar que a ferramenta BETA não considera todas as restrições presentes em uma operação, como nos casos de estruturas condicionais aninhadas e no caso de uma estrutura *CASE*, por exemplo. Com isso, algumas restrições não são utilizadas na geração dos testes, o que prejudica a qualidade. Dessa forma, é necessário melhorar a caracterização do espaço de entrada das operações para gerar testes com mais qualidade. Nos estudos também foi observado que os dados de teste gerados pela ferramenta BETA não apresentavam muita variação, ou seja, em muitos casos de teste eram gerados os mesmos dados de teste para algumas variáveis, algo que prejudica a qualidade dos testes. Isso ocorre porque o solucionador de restrições utilizado pela ferramenta BETA, o *ProB*, sempre retorna os valores mínimos, ou os primeiros valores, que satisfazem as restrições dos casos de teste. Para se ter uma maior variação nos dados de teste é necessário modificar as configurações do *ProB* ou fazer otimizações em BETA para forçar o solucionador a gerar dados mais variados;

- *Flexibilizar a etapa de particionamento do espaço de entrada*: em BETA o particionamento é feito seguindo uma das duas estratégias que são implementadas (*Classe de Equivalência* e *Análise de Valor Limite*). Esta etapa não apresenta muita variação uma vez que as duas estratégias se diferenciam apenas quando são utilizados intervalos numéricos na especificação. Para oferecer uma maior variação e não limitar a etapa, é necessário flexibilizar o particionamento em BETA, permitindo que o usuário possa utilizar as suas próprias estratégias de particionamento e que possa realizar vários particionamentos, semelhante ao que é feito na abordagem *TTF* (STOCKS; CARRINGTON, 1996; MACCOLL; CARRINGTON, 1998).

- *Gerar casos de teste a partir de implementações B*: atualmente, a abordagem e ferramenta BETA gera os casos de teste a partir de máquinas abstratas B. Quando essas máquinas são refinadas, são inseridas uma série de informações que poderiam ser utilizadas na geração dos testes. Gerar testes a partir de implementações B, ou módulos de refinamento, seria uma maneira de aproveitar as informações inseridas nesses módulos para gerar testes com mais qualidade. Além disso, permitiria gerar testes concretos sem a necessidade de adaptação dos dados abstratos. Com essa

melhoria, BETA seria capaz de gerar testes estruturais (caixa branca), além dos testes funcionais (caixa preta) que ela gera atualmente.

Referências

ABRAN, A.; BOURQUE, P.; DUPUIS, R.; MOORE, J. W.; TRIPP, L. L. *Guide to the Software Engineering Body of Knowledge - SWEBOK*. 2004 version. ed. Piscataway, NJ, USA: IEEE Press, 2004. 1–202 p. Disponível em: <http://www.swebok.org/ironman/pdf/SWEBOK_Guide_2004.pdf>.

ABRIAL, J. *The B-Book: Assigning Programs to Meanings*. [S.l.]: Cambridge U. Press, 2005.

ABRIAL, J.-R. *Modeling in Event-B: System and Software Engineering*. 1st. ed. New York, NY, USA: Cambridge University Press, 2010.

AGRAWAL, H.; DEMILLO, R. A.; HATHAWAY, B.; HSU, W.; HSU, W.; KRAUSER, E. W.; MARTIN, R. J.; MATHUR, A. P.; SPAFFORD, E. *Design of Mutant Operators for the C Programming Language*. West Lafayette, Indiana, 1989.

AMMANN, P.; OFFUTT, J. *Introduction to Software Testing*. First. New York, NY: Cambridge University Press, 2010.

AYDAL, E.; PAIGE, R.; UTTING, M.; WOODCOCK, J. Putting Formal Specifications under the Magnifying Glass: Model-based Testing for Validation. In: *International Conference on Software Testing, Verification and Validation, 2009*. [S.l.: s.n.], 2009. p. 131–140.

BARBOSA, H. *Desenvolvendo um sistema crítico através de formalização de requisitos utilizando o Método B*. Monografia (Graduação) — Universidade Federal do Rio Grande do Norte, Natal, 2010.

BRIAND, L. C.; PENTA, M. D.; LABICHE, Y. Assessing and Improving State-Based Class Testing: A Series of Experiments. *IEEE Trans. Softw. Eng.*, IEEE Press, Piscataway, NJ, USA, v. 30, n. 11, p. 770–793, nov. 2004. ISSN 0098-5589.

CARRINGTON, D.; STOCKS, P. A tale of two paradigms: Formal methods and software testing. In: BOWEN, J.; HALL, J. (Ed.). *Z User Workshop, Cambridge 1994*. [S.l.]: Springer London, 1994, (Workshops in Computing). p. 51–68.

CLEARSY. *B Language Reference Manual*. [S.l.], 2011. Version 1.8.7.

COSTA, U. S. da; MOREIRA, A. M.; MUSICANTE, M. A.; NETO, P. A. S. JCML: A Specification Language for the Runtime Verification of Java Card Programs. *Sci. Comput. Program.*, Elsevier North-Holland, Inc., Amsterdam, The Netherlands, The Netherlands, v. 77, n. 4, p. 533–550, abr. 2012. ISSN 0167-6423.

CRISTIA, M.; ALBERTENGO, P.; MONETTI, P. Pruning Testing Trees in the Test Template Framework by Detecting Mathematical Contradictions. In: *Software Engineering and Formal Methods (SEFM), 2010 8th IEEE International Conference on*. [S.l.: s.n.], 2010. p. 268–277.

CRISTIÁ, M.; FRYDMAN, C. S. Applying SMT Solvers to the Test Template Framework. In: *Proceedings 7th Workshop on Model-Based Testing, MBT 2012, Tallinn, Estonia, 25 March 2012.* [S.l.: s.n.], 2012. p. 28–42.

CRISTIÁ, M.; HOLLMANN, D.; ALBERTENGO, P.; FRYDMAN, C.; MONETTI, P. R. A Language for Test Case Refinement in the Test Template Framework. In: *Proceedings of the 13th International Conference on Formal Methods and Software Engineering*. Berlin, Heidelberg: Springer-Verlag, 2011. (ICFEM'11), p. 601–616.

CRISTIÁ, M.; MONETTI, P. Implementing and Applying the Stocks-Carrington Framework for Model-Based Testing. In: BREITMAN, K.; CAVALCANTI, A. (Ed.). *Formal Methods and Software Engineering*. [S.l.]: Springer Berlin Heidelberg, 2009, (Lecture Notes in Computer Science, v. 5885). p. 167–185.

CRISTIÁ, M.; ROSSI, G.; FRYDMAN, C. {log} as a Test Case Generator for the Test Template Framework. In: HIERONS, R.; MERAYO, M.; BRAVETTI, M. (Ed.). *Software Engineering and Formal Methods*. [S.l.]: Springer Berlin Heidelberg, 2013, (Lecture Notes in Computer Science, v. 8137). p. 229–243.

DIJKSTRA, E. Classics in Software Engineering. In: YOURDON, E. N. (Ed.). Upper Saddle River, NJ, USA: Yourdon Press, 1979. cap. The Humble Programmer, p. 111–125.

DOVIER, A.; OMODEO, E. G.; PONTELLI, E.; ROSSI, G. {log}: A language for programming in logic with finite sets. *The Journal of Logic Programming*, v. 28, n. 1, p. 1 – 44, 1996.

DÉHARBE, D.; MEDEIROS JR., V. Proposal: Translation of B Implementations to LLVM-IR. *Brazilian Symposium on Formal Methods*, 2013.

FREERTOS. *The FreeRTOS project*. 2010. Disponível em: <http://www.freertos.org/>. Acesso em Julho 18, 2014.

GALVÃO, S. d. S. L. *Especificação do micronúcleo FreeRTOS utilizando Método B*. Dissertação (Mestrado) — Universidade Federal do Rio Grande do Norte, Natal, 2010.

GOERIGK, W.; DOLD, A.; GAUL, T.; GOOS, G.; HEBERLE, A.; HENKE, F.; HOFFMANN, U.; LANGMAACK, H.; PFEIFER, H.; RUESS, H.; ZIMMERMANN, W. Compiler Correctness and Implementation Verification: The *Verifix* Approach. In: FRITZSON, P. (Ed.). *Proceedings of the Poster Session of CC '96 – International Conference on Compiler Construction*. IDA Technical Report LiTH-IDA-R-96-12, Linkøping, Sweden: [s.n.], 1996. p. 65 – 73.

GOGOLLA, M.; VALLECILLO, A. Tractable Model Transformation Testing. In: FRANCE, R.; KUESTER, J.; BORDBAR, B.; PAIGE, R. (Ed.). *Modelling Foundations and Applications*. [S.l.]: Springer Berlin Heidelberg, 2011, (Lecture Notes in Computer Science, v. 6698). p. 221–235. ISBN 978-3-642-21469-1.

HÄRTEL, J.; HÄRTEL, L.; LÄMMEL, R. Test-data generation for Xtext. In: *Software Language Engineering*. [S.l.]: Springer, 2014. p. 342–351.

HENTZ, C. *Automatic Generation of Tests from Language Descriptions*. Dissertação (Mestrado) — UFRN, Natal, Brazil, 2010.

HIERONS, R. M.; BOGDANOV, K.; BOWEN, J. P.; CLEAVELAND, R.; DERRICK, J.; DICK, J.; GHEORGHE, M.; HARMAN, M.; KAPOOR, K.; KRAUSE, P.; LüTTGEN, G.; SIMONS, A. J. H.; VILKOMIR, S.; WOODWARD, M. R.; ZEDAN, H. Using Formal Specifications to Support Testing. *ACM Comput. Surv.*, ACM, New York, NY, USA, v. 41, n. 2, p. 9:1–9:76, 2009.

HOARE, C. A. R. Communicating Sequential Processes. *Commun. ACM*, ACM, New York, NY, USA, v. 21, n. 8, p. 666–677, ago. 1978. ISSN 0001-0782.

HOFFMAN, D. M.; LY-GAGNON, D.; STROOPER, P.; WANG, H.-Y. Grammar-based test generation with YouGen. *Software: Practice and Experience*, Wiley Online Library, v. 41, n. 4, p. 427–447, 2011.

IERUSALIMSCHY, R. *Programming in Lua, Third Edition*. 3rd. ed. [S.l.]: Lua.Org, 2013. ISBN 859037985X, 9788590379850.

IERUSALIMSCHY, R.; FIGUEIREDO, L.; CELES, W. Lua-an extensible extension language. *Softw., Pract. Exper.*, Citeseer, v. 26, n. 6, p. 635–652, 1996.

IERUSALIMSCHY, R.; FIGUEIREDO, L.; CELES, W. The evolution of Lua. In: ACM. *Proceedings of the third ACM SIGPLAN conference on History of programming languages*. [S.l.], 2007. p. 2–1.

IERUSALIMSCHY, R.; FIGUEIREDO, L. H.; CELES, W. *Lua 5.2 Reference Manual*. 2014. Disponível em: <http://www.lua.org/manual/5.2/>.

IERUSALIMSCHY, R.; FIGUEIREDO, L. H.; CELES, W. *Lua 5.3 Reference Manual*. 2015. Disponível em: <http://www.lua.org/manual/5.3/manual.html>.

JACKSON, D. *Software Abstractions: Logic, Language, and Analysis*. [S.l.]: MIT Press, 2012. ISBN 0262017156, 9780262017152.

JIA, Y.; HARMAN, M. MILU: A Customizable, Runtime-Optimized Higher Order Mutation Testing Tool for the Full C Language. In: *Practice and Research Techniques, 2008. TAIC PART '08. Testing: Academic Industrial Conference*. [S.l.: s.n.], 2008. p. 94–98.

KLINT, P.; STORM, T. van der; VINJU, J. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. *2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, IEEE Computer Society, Los Alamitos, CA, USA, v. 0, p. 168–177, 2009.

LÄMMEL, R.; SCHULTE, W. Controllable Combinatorial Coverage in Grammar-Based Testing. In: UYAR, M. Ü.; DUALE, A. Y.; FECKO, M. A. (Ed.). *TestCom*. New York, NY, USA: Springer, 2006. (LNCS, v. 3964), p. 19–38.

LATTNER, C.; ADVE, V. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*. Palo Alto, California: [s.n.], 2004.

LEGEARD, B.; PEUREUX, F.; UTTING, M. Automated Boundary Testing from Z and B. In: *Proceedings of the International Symposium of Formal Methods Europe*. [S.l.]: Springer-Verlag, 2002. (FME '02), p. 21–40. ISBN 3-540-43928-5.

LEROY, X. Formal verification of a realistic compiler. *Commun. ACM*, v. 52, n. 7, p. 107–115, 2009.

LEUSCHEL, M.; BUTLER, M. ProB: A Model Checker for B. In: KEIJIRO, A.; GNESI, S.; DINO, M. (Ed.). *FME*. [S.l.]: Springer-Verlag, 2003. (Lecture Notes in Computer Science, v. 2805), p. 855–874.

LI, N.; OFFUTT, J. An Empirical Analysis of Test Oracle Strategies for Model-based Testing. *IEEE 7th International Conference on Software Testing, Verification and Validation*, April 2014.

LIU, J.; YANG, Z. Test generation from StateChart and B method for flight control software of unmanned aerial vehicle. In: *Computer Science Education, 2009. ICCSE '09. 4th International Conference on*. [S.l.: s.n.], 2009. p. 851–856.

MA, Y. S.; OFFUTT, J.; KWON, Y. R. MuJava: an automated class mutation system. *Software Testing, Verification and Reliability*, John Wiley & Sons, Ltd., v. 15, n. 2, p. 97–133, 2005.

MACCOLL, I.; CARRINGTON, D. Extending the Test Template Framework. In: *Proceedings of the 3rd BCS-FACS Conference on Northern Formal Methods*. Swinton, UK, UK: British Computer Society, 1998. (3FACS'98), p. 12–12.

MALIK, Q. A.; LILIUS, J.; LAIBINIS, L. Methods, Models and Tools for Fault Tolerance. In: BUTLER, M.; JONES, C.; ROMANOVSKY, A.; TROUBITSYNA, E. (Ed.). [S.l.]: Springer-Verlag, 2009. cap. Model-Based Testing Using Scenarios and Event-B Refinements, p. 177–195.

MALIK, Q. A.; LILIUS, J.; LAIBINIS, L. Scenario-based Test Case Generation Using Event-B Models. In: *In proceedings of International IEEE Conference on Advances in System Testing and Validation Lifecycle (VALID 2009)*. [S.l.]: IEEE Computer Society, 2009.

MATOS, E. C. B.; DÉHARBE, D.; MOREIRA, A. M.; HENTZ, C.; JR, V. M.; NETO, J. B. S. Verifying Code Generation Tools for the B-Method Using Tests: a Case Study. In: BLANCHETTE, J. C.; KOSMATOV, N. (Ed.). *Tests and Proofs*. [S.l.]: Springer International Publishing, 2015. (Lecture Notes in Computer Science, v. 9154), p. 76–91. ISBN 978-3-319-21214-2.

MATOS, E. C. B.; MOREIRA, A. M. BETA: A B Based Testing Approach. In: *Formal Methods: Foundations and Applications*. [S.l.]: Springer Berlin, 2012, (LNCS, v. 7498). p. 51–66.

MATOS, E. C. B.; MOREIRA, A. M. BETA: a tool for test case generation based on B specifications. In: *Proceedings of Congresso Brasileiro de Software: Teoria e Prática, CBSoft Tools*. Brasília - DF: [s.n.], 2013.

MATOS, E. C. B.; MOREIRA, A. M.; SOUZA, F. M.; COELHO, R. S. Generating test cases from B specifications: An industrial case study. In: *Proceedings of 22nd IFIP International Conference on Testing Software and Systems*. [S.l.: s.n.], 2010.

MILLER, J. C.; MALONEY, C. J. Systematic Mistake Analysis of Digital Computer Programs. *Commun. ACM*, ACM, New York, NY, USA, v. 6, n. 2, p. 58–63, fev. 1963. ISSN 0001-0782.

MOREIRA, A. M.; HENTZ, C.; RAMALHO, V. Application of a Syntax-based Testing Method and Tool to Software Product Lines. In: *7th Brazilian Workshop on Systematic and Automated Software Testing*. Brasília - DF: [s.n.], 2013.

MOREIRA, A. M.; IERUSALIMSCHY, R. Modeling the Lua API in B. Draft. 2013.

MUHAMMAD, H.; IERUSALIMSCHY, R. C APIs in Extension and Extensible Languages. *Journal of Universal Computer Science*, v. 13, n. 6, p. 839–853, 2007.

MYERS, G.; SANDLER, C.; BADGETT, T.; THOMAS, T. *The Art of Software Testing*. [S.l.]: Wiley, 2004. (Business Data Processing: A Wiley Series). ISBN 9780471678359.

NECULA, G. C. Translation validation for an optimizing compiler. *ACM sigplan notices*, ACM, v. 35, n. 5, p. 83–94, 2000.

Object Management Group. *Object Constraint Language - Version 2.4*. [S.l.], 2014. Disponível em: <http://www.omg.org/spec/OCL/2.4/>.

ROSE, G.; DUKE, R. *Formal Object Oriented Specification Using Object-Z* . [S.l.]: Palgrave Macmillan, 2000. 240 p. (Cornerstones of Computing).

SATPATHY, M.; LEUSCHEL, M.; BUTLER, M. ProTest: An automatic test environment for B specifications. *Electronic Notes in Theoretical Computer Science*, Elsevier, v. 111, p. 113–136, 2005.

SCHNEIDER, S. *The B-Method: An Introduction*. [S.l.]: Palgrave Macmillan Limited, 2001. (Cornerstones of computing).

SHRESTHA, K.; RUTHERFORD, M. An Empirical Evaluation of Assertions as Oracles. In: *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*. [S.l.: s.n.], 2011. p. 110–119.

SOUZA, F. M. *Geração de Casos de Teste a partir de Especificações B*. 100 p. Dissertação (Mestrado) — Universidade Federal do Rio Grande do Norte, Natal, 2009.

SPIVEY, J. M. *The Z Notation: A Reference Manual*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1989.

STAATS, M.; WHALEN, M.; HEIMDAHL, M. Better testing through oracle selection: (NIER track). In: *Software Engineering (ICSE), 2011 33rd International Conference on*. [S.l.: s.n.], 2011. p. 892–895.

STOCKS, P.; CARRINGTON, D. A Framework for Specification-Based Testing. *IEEE Trans. Softw. Eng.*, IEEE Press, Piscataway, NJ, USA, v. 22, n. 11, p. 777–793, nov. 1996. ISSN 0098-5589.

TANENBAUM, A. S. In Defense of Program Testing or Correctness Proofs Considered Harmful. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 11, n. 5, p. 64–68, 1976.

The Coq development team. *The Coq proof assistant reference manual*. [S.l.], 2004. Version 8.0. Disponível em: <http://coq.inria.fr>.

WIMMER, M.; NO, L. B. Testing M2T/T2M Transformations. In: MOREIRA, A.; SCHäTZ, B.; GRAY, J.; VALLECILLO, A.; CLARKE, P. (Ed.). *Model-Driven Engineering Languages and Systems*. [S.l.]: Springer Berlin Heidelberg, 2013, (Lecture Notes in Computer Science, v. 8107). p. 203–219. ISBN 978-3-642-41532-6.

XIE, Q.; MEMON, A. M. Designing and Comparing Automated Test Oracles for GUI-based Software Applications. *ACM Trans. Softw. Eng. Methodol.*, ACM, New York, NY, USA, v. 16, n. 1, fev. 2007. ISSN 1049-331X.

YOUNG, M.; PEZZE, M. *Software Testing and Analysis: Process, Principles and Techniques*. [S.l.]: John Wiley & Sons, 2005. ISBN 0471455938, 9780471455936.

ZUCK, L.; PNUELI, A.; FANG, Y.; GOLDBERG, B. VOC: A Methodology for the Translation Validation of Optimizing Compilers. *Journal of Universal Computer Science*, v. 9, n. 3, p. 223–247, mar 2003.

APÊNDICE A – Códigos

Neste Apêndice é apresentado o código em *Rascal* (KLINT; STORM; VINJU, 2009) do gerador de *scripts* de teste desenvolvido neste trabalho, apresentado no Capítulo 4. Na Figura 4.1 foi apresentada a visão geral da arquitetura do gerador de *scripts* de teste. O gerador recebe como entrada uma especificação de casos de teste gerada por BETA. Essa especificação é passada para um *Parser* que extrai as informações da especificação e as convertem para um modelo definido em *Rascal*. Este modelo é apresentado na Listagem A.1. Em seguida, este modelo é passado para um módulo de tradução que traduz alguns predicados e expressões para uma linguagem de programação. Para fazer a tradução foi necessário definir a sintaxe de predicados e expressões do Método B em *Rascal*. Essa sintaxe é apresentada na Listagem A.2. A Listagem A.3 apresenta o módulo de tradução para a linguagem de programação *C*. Após alguns predicados e expressões serem traduzidos, estes são passados para um módulo de *template* de teste que transforma os dados da especificação de casos de teste em um código de teste. A Listagem A.4 apresenta o módulo de *template* de teste para a linguagem de programação *C* seguindo a notação do *framework* de testes *CuTest*.

Listagem A.1 – Modelo da Especificação de Casos de Teste em *Rascal*

```
1  module parser::Model
2
3  data TestSuite = TestSuite(str machineName, list[str] machineInvariant,
4                             str operationUnderTest, str testingStrategy,
5                             str coverageCriteria, list[TestCase] testCases,
6                             list[OracleStrategy] oracleStrategies);
7
8  data TestCase = TestCase(int id, str formula, bool negative,
9                             list[Variable] stateVariables,
10                            list[Parameter] operationParameters,
11                            list[Variable] expectedStateVariables,
12                            list[Variable] returnVariables
13
14 data OracleStrategy = StateInvariant()
15                     | ReturnValues()
16                     | StateVariables()
17                     | Exception();
18
19 data Variable = Variable(str identifier, list[str] values);
20 data Parameter = Parameter(str identifier, list[str] values);
```

Listagem A.2 – Sintaxe de Predicados e Expressões do Método B em *Rascal*

```
1  module B::Syntax
2
3  layout Whitespaces = [\t\n\ \r\f]*;
```

```
4  lexical Ident = ([a-z A-Z 0-9 _] !<< [a-z A-Z][a-z A-Z 0-9 _]* !>> [a-z A-Z
       0-9 _]) \ Keywords;
5  lexical Integer_literal = [0-9]+;
6  keyword Keywords = "not" | "MAXINT" | "MININT" | "TRUE" | "FALSE"
7                   | "NAT" | "NAT1" | "INT"| "BOOL" | "STRING"
8                   | "bool" | "succ" | "pred" | "max" | "min" | "card"
9                   | "SIGMA" | "PI" | "POW" | "POW1" | "FIN" | "FIN1"
10                  | "union" | "inter" | "UNION" | "INTER";
11
12 start syntax Predicate
13    = bracket Bracketed_predicate: "(" Predicate p ")"
14    | Negation_predicate: "not" "(" Predicate p ")"
15    | left Conjunction_predicate: Predicate p1 "&" Predicate p2
16    | left Disjunction_predicate: Predicate p1 "or" Predicate p2
17    | left Implication_predicate: Predicate p1 "=\>" Predicate p2
18    | left Equivalence_predicate: Predicate p1 "\<=\>" Predicate p2
19    | Predicate_universal: "!" List_ident li "." "(" Predicate p1 "=\>"
          Predicate p2 ")"
20    | Existential_predicate: "#" List_ident li "." "(" Predicate p ")"
21    | left Equals_predicate: Expression e1 "=" Expression e2
22    | left Predicate_unequal: Expression e1 "/=" Expression e2
23    | left Less_than_or_equal_predicate: Expression e1 "\<=" Expression e2
24    | left Strictly_less_than_predicate: Expression e1 "\<" Expression e2
25    | left Preedicate_greater_than_or_equal: Expression e1 "\>=" Expression
          e2
26    | left Strictly_greater_predicate_than: Expression e1 "\>" Expression e2
27    | left Belongs_predicate: Expression e1 ":" Expression e2
28    | left Non_belongs_predicate: Expression e1 "/:" Expression e2
29    | left Predicate_includes: Expression e1 "\<:" Expression e2
30    | left Predicate_includes_strictly: Expression e1 "\<\<:" Expression e2
31    | left Non_inclusion_predicate: Expression e1 "/\<:" Expression e2
32    | left Non_inclusion_predicate_strict: Expression e1 "/\<\<:" Expression
          e2
33    ;
34 syntax List_ident
35    = ID: Ident id
36    | List_id: "(" {Ident ","}+ li ")"
37    ;
38 syntax Expression
39    = Expressions_primary
40    | Expressions_Boolean
41    | Expressions_arithmetical
42    | Expressions_of_couples
43    | Expressions_of_sets
44    | Construction_of_sets
45    | Expressions_of_relations
46    | Expressions_of_functions
47    | Construction_of_functions
48    | Expressions_of_sequences
49    | Construction_of_sequences
50    ;
51 syntax Expressions_primary
52    = Data: Ident+ id
53    | bracket Expr_bracketed: "(" Expression e ")"
54    ;
55 syntax Expressions_Boolean
56    = TRUE: "TRUE"
```

```
57      | FALSE: "FALSE"
58      | Conversion_Bool: "bool" "(" Predicate p ")"
59      ;
60  syntax Expressions_arithmetical
61      = Integer_lit
62      | left Addition: Expression e1 "+" Expression e2
63      | left Difference: Expression e1 "-" Expression e2
64      | Unary_minus: "-"Expression  e1
65      | left Product: Expression e1 "*" Expression e2
66      | left Division: Expression e1 "/" Expression e2
67      | left Modulo: Expression e1 "mod" Expression e2
68      | left Power_of: Expression e1 "**" Expression e2
69      | Successor: "succ" "(" Expression e ")"
70      | Predecessor: "pred" "(" Expression e ")"
71      | Maximum: "max"  "(" Expression e ")"
72      | Minimum: "min"  "(" Expression e ")"
73      | Cardinal: "card"  "(" Expression e ")"
74      | Generalized_sum: "SIGMA" List_ident li "." "(" Predicate p "|"
            Expression e ")"
75      | Generalized_product: "PI" List_ident li "." "(" Predicate p "|"
            Expression e ")"
76      ;
77  syntax Integer_lit
78      = Integer_literal il
79      | MAX_INT: "MAXINT"
80      | MIN_INT: "MININT"
81      ;
82  syntax Expressions_of_couples
83      = left Couple: Expression e1 "|-\>" Expression e2
84      | left Couple2: Expression e1 "," Expression e2
85      ;
86  syntax Expressions_of_sets
87      = Empty_set: "{" "}"
88      | Boolean_set: "BOOL"
89      | Strings_set: "STRING"
90      | Integer_set
91      ;
92  syntax Integer_set
93      = Natural: "NAT"
94      | Natural1: "NAT1"
95      | Integer: "INT"
96      ;
97  syntax Construction_of_sets
98      = Comprehension_set: "{" {Ident ","}+ li "|" Predicate p "}"
99      | Subset: "POW" "(" Expression e ")"
100     | Subset1: "POW1" "(" Expression e ")"
101     | Finite_subset: "FIN" "(" Expression e ")"
102     | Finite_subset1: "FIN1" "(" Expression e ")"
103     | Set_extension: "{" {Expression ","}+ li "}"
104     | left Interval: Expression e1 ".." Expression e2 // Partly
105     | left Union: Expression e1 "\\/" Expression e2
106     | left Intersection: Expression e1 "/\\" Expression e2
107     | Generalized_union: "union" "(" Expression e ")"
108     | Generalized_intersection: "inter" "(" Expression e ")"
109     | Quantified_union: "UNION" List_ident li "." "(" Predicate p "|"
            Expression e ")"
```

```
110     | Quantified_intersection: "INTER" List_ident li "." "(" Predicate p "|"
          Expression e ")"
111     ;
112  syntax Expressions_of_relations
113     = left Relations: Expression e1 "\<-\>" Expression e2
114     | Identity: "id" "(" Expression e ")"
115     | Reverse: Expression e "~"
116     | First_projection: "prj1" "(" Expression e1 "," Expression e2 ")"
117     | Second_projection: "prj2" "(" Expression e1 "," Expression e2 ")"
118     | left Composition: Expression e1 ";" Expression e2
119     | left Direct_product: Expression e1 "\>\<" Expression e2
120     | left Parallel_product:  Expression e1 "||" Expression e2
121     | Iteration: "iterate" "(" Expression e1 "," Expression e2 ")"
122     | Reflexive_closure: "closure" "(" Expression e ")"
123     | Closure: "closure1" "(" Expression e ")"
124     | Domain: "dom" "(" Expression e ")"
125     | Range: "ran" "(" Expression e ")"
126     | left Image: Expression e1 "[" Expression e2 "]"
127     | left Domain_restriction: Expression e1 "\<|" Expression e2
128     | left Domain_subtraction: Expression e1 "\<\<|" Expression e2
129     | left Range_restriction: Expression e1 "|\>" Expression e2
130     | left Range_subtraction:  Expression e1 "|\>\>" Expression e2
131     | left Overwrite: Expression e1 "\<+" Expression e2
132     ;
133  syntax Expressions_of_functions
134     = Partial_functions: Expression e1 "+-\>" Expression e2
135     | Total_functions: Expression e1 "--\>" Expression e2
136     | Partial_injections: Expression e1 "\>+\>" Expression e2
137     | Total_injections: Expression e1 "\>-\>" Expression e2
138     | Partial_surjections: Expression e1 "+-\>\>" Expression e2
139     | Total_surjections: Expression e1 "--\>\>" Expression e2
140     | Partial_bijections: Expression e1 "\>+\>\>" Expression e2
141     | Total_bijections: Expression e1 "\>-\>\>" Expression e2
142     ;
143  syntax Construction_of_functions
144     = Lambda_expression: "%" List_ident li "." "(" Predicate p "|" Expression
          e ")"
145     | Evaluation_functions: Expression e "(" {Expression ","}+ li ")"
146     | Transformed_function: "fnc" "(" Expression e ")"
147     | Transformed_relation: "rel" "(" Expression e ")"
148     ;
149  syntax Expressions_of_sequences
150     = Sequences: "seq" "(" Expression e ")"
151     | Non_empty_sequences: "seq1" "(" Expression e ")"
152     | Injective_sequences: "iseq" "(" Expression e ")"
153     | Non_empty_inj_sequences: "iseq1" "(" Expression e ")"
154     | Permutations: "perm" "(" Expression e ")"
155     ;
156  syntax Construction_of_sequences
157     = Empty_sequence: "\<" "\>"
158     | Empty_sequence2: "[" "]"
159     | Sequence_extension: "\<" {Expression ","}* li "\>"
160     | Sequence_extension2: "[" {Expression ","}* li "]"
161     | Sequence_size: "size" "(" Expression e ")"
162     | Sequence_first_element: "first" "(" Expression e ")"
163     | Sequence_last_element: "last" "(" Expression e ")"
164     | Sequence_front: "front" "(" Expression e ")"
```

```
165     | Sequence_tail: "tail" "(" Expression e ")"
166     | Reverse_sequence: "rev" "(" Expression e ")"
167     | left Concatenation: Expression e1 "^" Expression e2
168     | left Insert_front: Expression e1 "-\>" Expression e2
169     | left Insert_tail: Expression e1 "\<-" Expression e2
170     | left Restrict_front: Expression e1 "/|\\" Expression e2
171     | left Restrict_tail: Expression e2 "\\|/" Expression e2
172     | Generalized_concat: "conc" "(" Expression e ")"
173     ;
```

Listagem A.3 – Módulo de tradução dos Predicados e Expressões do Método B para a linguagem *C*

```
1  module B::CTranslate
2
3  import B::Syntax;
4  import String;
5  import ParseTree;
6  import IO;
7
8  public str translate(str txt) = translatePred(parse(#Predicate, txt));
9
10 public str translatePred((Predicate) `(<Predicate p>)`) = "(<translatePred(
       p)>)";
11 public str translatePred((Predicate) `not(<Predicate p>)`) = "!(<
       translatePred( p)>)";
12 public str translatePred((Predicate) `<Predicate p1>&<Predicate p2>`) = "<
       translatePred( p1)> && <translatePred( p2)>";
13 public str translatePred((Predicate) `<Predicate p1>or<Predicate p2>`) = "<
       translatePred( p1)> || <translatePred( p2)>";
14 public str translatePred((Predicate) `<Predicate p1>=\><Predicate p2>`) = "
       !(<translatePred( p1)>) || <translatePred( p2)>";
15 public str translatePred((Predicate) `<Predicate p1>\<=\><Predicate p2>`) =
       "(<translatePred( p1)> && <translatePred( p2)>) || !(<translatePred( p1)
       > || <translatePred( p2)>)";
16 public str translatePred((Predicate) `<Expression e1>=<Expression e2>`) = "<
       translateExp(e1)> == <translateExp(e2)>";
17 public str translatePred((Predicate) `<Expression e1>/=<Expression e2>`) = "
       <translateExp(e1)> != <translateExp(e2)>";
18 public str translatePred((Predicate) `<Expression e1>\<=<Expression e2>`) =
       "<translateExp(e1)> \<= <translateExp(e2)>";
19 public str translatePred((Predicate) `<Expression e1>\<<Expression e2>`) = "
       <translateExp(e1)> \< <translateExp(e2)>";
20 public str translatePred((Predicate) `<Expression e1>\>=<Expression e2>`) =
       "<translateExp(e1)> \>= <translateExp(e2)>";
21 public str translatePred((Predicate) `<Expression e1>\><Expression e2>`) = "
       <translateExp(e1)> \> <translateExp(e2)>";
22 public str translatePred((Predicate) `<Expression e1>:<Expression e2>`) {
23    if((Expression) `<Expression i1>..<Expression i2>` := e2){
24       return "<translateExp(e1)> \>= <translateExp(i1)> && <translateExp(e1)
             > \<= <translateExp(i2)>";
25    } else {
26       return "";
27    }
28 }
29 public default str translatePred( Predicate p) = "";
```

```
30
31 public str translateExp((Expression) `<Ident id>`) = "<id>";
32 public str translateExp((Expression) `(<Expression e>)`) = "(<translateExp(
       e)>)";
33 public str translateExp((Expression) `TRUE`) = "true";
34 public str translateExp((Expression) `FALSE`) = "false";
35 public str translateExp((Expression) `bool(<Predicate p>)`) = translatePred(
       p);
36 public str translateExp((Expression) `<Integer_literal i>`) = "<i>";
37 public str translateExp((Expression) `MAXINT`) = "INT32_MAX";
38 public str translateExp((Expression) `MININT`) = "INT32_MIN";
39 public str translateExp((Expression) `<Expression e1>+<Expression e2>`) = "<
       translateExp(e1)> + <translateExp(e2)>";
40 public str translateExp((Expression) `<Expression e1>-<Expression e2>`) = "<
       translateExp(e1)> - <translateExp(e2)>";
41 public str translateExp((Expression) `-<Expression e>`) = "-<translateExp( e
       )>";
42 public str translateExp((Expression) `<Expression e1>*<Expression e2>`) = "<
       translateExp(e1)> * <translateExp(e2)>";
43 public str translateExp((Expression) `<Expression e1>/<Expression e2>`) = "<
       translateExp(e1)> / <translateExp(e2)>";
44 public str translateExp((Expression) `<Expression e1>mod<Expression e2>`) =
       "<translateExp(e1)> % <translateExp(e2)>";
45 public str translateExp((Expression) `<Expression e1>**<Expression e2>`) = "
       pow(<translateExp(e1)>, <translateExp(e2)>)";
46 public str translateExp((Expression) `succ(<Expression e>)`) = "(<
       translateExp( e)> + 1)";
47 public str translateExp((Expression) `pred(<Expression e>)`) = "(<
       translateExp( e)> - 1)";
48 public str translateExp((Expression) `<Expression e1>|-\><Expression e2>`) =
        "<translateExp(e2)>";
49 public default str translateExp(Expression e) = "<e>";
```

Listagem A.4 – Módulo de *template* para a linguagem *C* seguindo a notação do *CuTest*

```
1  module template::CuTest
2
3  import parser::Model;
4  import B::Syntax;
5  import B::CTranslate;
6  import ParseTree;
7  import String;
8  import List;
9  import IO;
10
11 // Function that returns the oracle strategies names of the test suite.
12 private str oracleStrategies(TestSuite testSuite){
13    OracleStrategy first = head(testSuite.oracleStrategies);
14    str oss = "";
15    switch(first){
16       case StateInvariant(): oss = "State Invariant";
17       case ReturnValues(): oss = "Return Values";
18       case StateVariables(): oss = "State Variables";
19       case Exception(): oss = "Exception";
20    }
21    for(OracleStrategy os <- tail(testSuite.oracleStrategies)){
22       switch(os){
```

```
23          case StateInvariant(): oss = oss + ", State Invariant";
24          case ReturnValues(): oss = oss + ", Return Values";
25          case StateVariables(): oss = oss + ", State Variables";
26          case Exception(): oss = oss + ", Exception";
27       }
28    }
29    return oss;
30 }
31
32 // Function that generate a variable declaration.
33 private str variableDeclaration(TestSuite testSuite, str formula, str
       identifier, list[str] values) {
34    // get invariant and formula predicates
35    list[str] predicates = testSuite.machineInvariant + [trim(pr) | str pr <-
          split("&", formula)];
36
37    str ty = "";
38    bool isSet = false;
39
40    for(str p <- predicates) {
41       try{
42          Predicate predicate = parse(#Predicate, p);
43          Expression expIdent = parse(#Expression, identifier);
44          switch(predicate){
45             // if the predicate is a belong the variable isn't a set
46             case (Predicate) `<Expression e1>:<Expression e2>` : {
47                if(e1 == expIdent){
48                   ty = "<e2>"; // get the variable type
49                   break;
50                }
51             }
52             // if the predicate is an include the variable is a set (array)
53             case (Predicate) `<Expression e1>\<:<Expression e2>` : {
54                if(e1 == expIdent){
55                   ty = "<e2>"; // get the variable type
56                   isSet = true;
57                   break;
58                }
59             }
60             // if the predicate is a strictly include the variable is a set
                  (array)
61             case (Predicate) `<Expression e1>\<\<:<Expression e2>` : {
62                if(e1 == expIdent){
63                   ty = "<e2>"; // get the variable type
64                   isSet = true;
65                   break;
66                }
67             }
68          }
69       } catch: ;
70    }
71
72    isSet = isSet || size(values) > 1; // if the number of values is greater
          than 1 the variable is a set (array)
73
74    if(isEmpty(ty)){
75       ty = identifier; // if ty is empty takes identifier value
```

```
76      }
77
78      str declaration = "";
79
80      if (ty == "INT" || ty == "NAT" || ty == "NAT1"
81              || ty == "INTEGER" || ty == "NATURAL"
82              || contains(ty, "MAXINT") || contains(ty, "MININT")
83              || contains(ty, "..")) {
84          declaration = "int32_t";
85      } else if (ty == "BOOL") {
86          declaration = "bool";
87      } else if(ty == "STRING") {
88          declaration = "char*";
89      } else {
90          declaration = ty;
91      }
92
93      if (isSet) {
94          declaration = declaration + "[]";
95      }
96
97      declaration = declaration + " " + identifier;
98
99      return declaration;
100 }
101
102 // Function that generate a variable attribution.
103 private str variableAttribution(TestSuite testSuite, str formula, str
        identifier, list[str] values) {
104     // get invariant and formula predicates
105     list[str] predicates = testSuite.machineInvariant + [trim(pr) | str pr <-
            split("&", formula)];
106
107     bool isSet = false;
108
109     for(str p <- predicates) {
110         try {
111             Predicate predicate = parse(#Predicate, p);
112             Expression expIdent = parse(#Expression, identifier);
113             switch(predicate){
114                 // if the predicate is an include the variable is a set
115                 case (Predicate) `<Expression e1>\<:<Expression e2>` : {
116                     if(e1 == expIdent){
117                         isSet = true;
118                         break;
119                     }
120                 }
121                 // if the predicate is a strictly include the variable is a set
122                 case (Predicate) `<Expression e1>\<\<:<Expression e2>` : {
123                     if(e1 == expIdent){
124                         isSet = true;
125                         break;
126                     }
127                 }
128             }
129         } catch: ;
130     }
```

```
131
132     isSet = isSet || size(values) > 1; // if the number of values is greater
            than 1 the variable is a set (array)
133
134     str attribution = "";
135
136     if (isSet) {
137        if (values[0] == "{-}" || isEmpty(values)) { // empty array
138           attribution = attribution + "{}";
139        } else {
140           Expression exp = parse(#Expression, head(values)); // array with
               the values
141           attribution = attribution + "{" + translateExp( exp);
142           for (str s <- tail(values)) {
143              exp = parse(#Expression, s);
144              attribution = attribution + ", " + translateExp( exp);
145           }
146           attribution = attribution + "}";
147        }
148     } else {
149        if(!isEmpty(values)){
150           str val = head(values);
151           if(val == "{-}")
152              val = "{}";
153           Expression exp = parse(#Expression, val);
154           attribution = attribution + translateExp( exp); // simple variable
               attribution
155        }
156     }
157     return attribution;
158 }
159
160 // Function that generate the operation call with the operation parameters
       and return variables.
161 private str operationCall(TestSuite testSuite, TestCase testCase){
162     str call = testSuite.machineName + "__" + testSuite.operationUnderTest +
          "(";
163     if(!isEmpty(testCase.operationParameters)){
164        Parameter first = testCase.operationParameters[0];
165        for(Parameter p <- testCase.operationParameters){
166           if(first.identifier == p.identifier){
167              call = call + p.identifier;
168           } else {
169              call = call + ", " + p.identifier;
170           }
171        }
172     }
173     if(!isEmpty(testCase.returnVariables)){
174        if(!isEmpty(testCase.operationParameters)){
175           call = call + ", ";
176        }
177        Variable first = testCase.returnVariables[0];
178        for(Variable v <- testCase.returnVariables){
179           if(first.identifier == v.identifier){
180              call = call + "&" + v.identifier;
181           } else {
182              call = call + ", " + "&" + v.identifier;
```

```
183             }
184          }
185       }
186       call = call + ")";
187       return call;
188    }
189
190    // Function that verify if the check invariant function content isn't empty.
191    private bool hasCheckInvariant(TestSuite testSuite){
192       bool has = false;
193       for(str p <- testSuite.machineInvariant){
194          if(!isEmpty(translate(p))){
195             has = true;
196          }
197       }
198       return has;
199    }
200
201    // Auxiliary function that change the variables names to the pattern in the
          check invariant function.
202    private str replaceVariablesNamesInCheckInvariant(TestSuite testSuite, str
          predicate){
203       str r = predicate;
204       TestCase tc = testSuite.testCases[0];
205       for(Variable variable <- tc.stateVariables){
206          if(!contains(r, testSuite.machineName + "__" + variable.identifier)){
207             r = replaceAll(r, variable.identifier, testSuite.machineName + "__"
                    + variable.identifier);
208          }
209       }
210       return r;
211    }
212
213    // Function that create the check invariant function content. Call functions
          to translate the invariant predicates.
214    private str templateCheckInvariant(TestSuite testSuite){
215       return
216          "void check_invariant(CuTest* tc) {
217          '  <for(str p <- testSuite.machineInvariant){>
218          '  <if(!isEmpty(translate(p))){>
219          '  if(!(<replaceVariablesNamesInCheckInvariant(testSuite, translate(p)
                   )>)){
220          '     CuFail(tc, \"The invariant \'<p>\' was unsatisfied\");
221          '  }
222          '  <} else {>
223          '  // Predicate \'<p>\' can\'t be automatically translated
224          '  <}> <}>
225          '}
226          ";
227    }
228
229    // Function that create the test case content. Call other functions to
          generate variable declaration and operation call.
230    private str templateTestCase(TestSuite testSuite, TestCase testCase){
231       return
232          "/**
233          '* Test Case <testCase.id>
```

```
234         '* Formula: <testCase.formula>
235         '* <if(testCase.negative){>Negative Test<}else{>Positive Test<}>
236         '*/
237         'void <testSuite.machineName>_<testSuite.operationUnderTest>
                _test_case_<testCase.id>(CuTest* tc)
238         '{
239         '  <testSuite.machineName>__INITIALISATION();
240         '  <for(Variable variable <- testCase.stateVariables){>
241         '  <variableDeclaration(testSuite, testCase.formula, variable.
               identifier, variable.values)> = <variableAttribution(testSuite,
               testCase.formula, variable.identifier, variable.values)>;
242         '  <testSuite.machineName>__<variable.identifier> = <variable.
               identifier>;
243         '  <}>
244         '  <for(Parameter parameter <- testCase.operationParameters){>
245         '  <variableDeclaration(testSuite, testCase.formula, parameter.
               identifier, parameter.values)> = <variableAttribution(testSuite,
               testCase.formula, parameter.identifier, parameter.values)>; <}>
246         '  <for(Variable v <- testCase.returnVariables){>
247         '  // <v.identifier> return variable declaration
248         '  <}>
249         '  <operationCall(testSuite, testCase)>;
250         '  <if(!isEmpty(testCase.returnVariables) && ReturnValues() in
               testSuite.oracleStrategies){>
251         '  <for(Variable v <- testCase.returnVariables){>
252         '  CuAssertTrue(tc, <v.identifier> == /* Add expected value here */);
253         '  <}><}>
254         '  <if(StateVariables() in testSuite.oracleStrategies){>
255         '  <if(isEmpty(testCase.expectedStateVariables)){>
256         '  <for(Variable variable <- testCase.stateVariables){>
257         '  <variableDeclaration(testSuite, testCase.formula, variable.
               identifier, [])>Expected; // Add expected value here.
258         '  CuAssertTrue(tc, <testSuite.machineName>__<variable.identifier> ==
               <variable.identifier>Expected);
259         '  <}> <}else{>
260         '  <for(Variable variable <- testCase.expectedStateVariables){>
261         '  <variableDeclaration(testSuite, testCase.formula, variable.
               identifier, variable.values)>Expected = <variableAttribution(
               testSuite, testCase.formula, variable.identifier, variable.values)
               >;
262         '  CuAssertTrue(tc, <testSuite.machineName>__<variable.identifier> ==
               <variable.identifier>Expected);
263         '  <}><}><}>
264         '  <if(StateInvariant() in testSuite.oracleStrategies &&
               hasCheckInvariant(testSuite)){>check_invariant(tc);<}>
265         '}
266         ";
267 }
268
269 // Function that returns the test file name.
270 str fileName(TestSuite testSuite){
271    return "<testSuite.machineName>_<testSuite.operationUnderTest>_test.c";
272 }
273
274 // Main function that create the test content template and call another
       template functions.
275 public str template(TestSuite testSuite){
```

```
276     return
277         "#include \<stdio.h\>
278         '#include \<string.h\>
279         '#include \<stdint.h\>
280         '#include \"CuTest.h\"
281         '#include \"<testSuite.machineName>.h\"
282         '
283         '<if(StateInvariant() in testSuite.oracleStrategies &&
                hasCheckInvariant(testSuite)){>
284         '<templateCheckInvariant(testSuite)> <}>
285         '<for(TestCase testCase <- testSuite.testCases){>
286         '<templateTestCase(testSuite, testCase)> <}>
287         '/**
288         '* Test Suite
289         '* Machine: <testSuite.machineName>
290         '* Operation: <testSuite.operationUnderTest>
291         '*
292         '* Testing Strategy: <testSuite.testingStrategy>
293         '* Coverage Criteria: <testSuite.coverageCriteria>
294         '* Oracle Strategy: <oracleStrategies(testSuite)>
295         '*/
296         'CuSuite* <testSuite.machineName>_<testSuite.operationUnderTest>
                _test_suite(void)
297         '{
298         '  CuSuite* suite = CuSuiteNew();
299         '  <for(TestCase tc <- testSuite.testCases){>
300         '  SUITE_ADD_TEST(suite, <testSuite.machineName>_<testSuite.
                operationUnderTest>_test_case_<tc.id>);<}>
301         '
302         '  return suite;
303         '}
304         "
305         ;
306 }
```