Universidade Federal do Rio Grande do Norte
Centro de Ciências Exatas e da Terra
Departamento de Informática e Matemática Aplicada

Programa de Pós-Graduação em Sistemas e Computação

BTestBox: uma ferramenta de teste para implementações B

Diego de Azevedo Oliveira

Dissertação de Mestrado

Natal
Fevereiro de 2017

Universidade Federal do Rio Grande do Norte
Centro de Ciências Exatas e da Terra
Departamento de Informática e Matemática Aplicada

Diego de Azevedo Oliveira

BTestBox: uma ferramenta de teste para implementações B

Trabalho apresentado 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 obtenção do grau de Mestre em Sistemas e Computação.

Orientador: *David Boris Paul Déharbe*

Natal
Fevereiro de 2017

Universidade Federal do Rio Grande do Norte - UFRN
Sistema de Bibliotecas - SISBI
Catalogação de Publicação na Fonte. UFRN - Biblioteca Setorial Prof. Ronaldo Xavier de Arruda - CCET

Oliveira, Diego de Azevedo.
BTestBox: uma ferramenta de teste para implementações B / Diego de Azevedo Oliveira. - 2017.
73f.: il.

Dissertação (mestrado) - Universidade Federal do Rio Grande do Norte, Centro de Ciências Exatas e da Terra, Departamento de Informática e Matemática Aplicada, Programa de Pós-Graduação em Sistemas e Computação. Natal, 2017.
Orientador: David Boris Paul Déharbe.

1. Engenharia de software - Dissertação. 2. Método B - Dissertação. 3. Teste de software - Dissertação. 4. Ferramenta - Dissertação. 5. Atelier-B - Dissertação. 6. Cobertura de código - Dissertação. I. Déharbe, David Boris Paul. II. Título.

RN/UF/CCET CDU 004.41

Elaborado por Joseneide Ferreira Dantas - CRB-15/324

Resumo

Softwares precisam ser seguros e corretos. Partindo desse pressuposto, novas tecnologias e técnicas são desenvolvidas para comprovar as competências de um programa. Essa necessidade de segurança se torna mais relevante ao tratar de softwares que atuam em sistemas críticos, como os sistemas ferroviário e aeroviário. A utilização de métodos formais na construção de software busca solucionar o problema. Ao utilizar o método formal B através da plataforma Atelier-B, e após provar os componentes de um projeto é necessária a tradução para a linguagem desejada. Essa tradução ocorre por meio de tradutores e compiladores B. Habitualmente, o processo de compilação em compiladores maduros é seguro, porém não estão completamente livres de falhas e eventualmente erros são encontrados. Ao expandir essa afirmação para tradutores B é necessário cautela, uma vez que esses não são tão comuns e utilizados quanto compiladores que circulam há mais tempo no mercado. Testes de software podem ser utilizados para realizar a análise da tradução. Através de critérios de cobertura é possível inferir o nível de qualidade do software e facilitar a detecção de falhas. Realizar a checagem da cobertura e testes em software pode exigir bastante esforço e tempo, principalmente ao serem realizados manualmente. Para sanar essa demanda, a ferramenta BTestBox visa analisar, de maneira automática, a cobertura atingida por implementações B desenvolvidas através do Atelier-B. BTestBox também testa automaticamente as traduções feitas a partir de implementações B. Para isso, BTestBox utiliza os casos de teste gerados para a verificação de cobertura e compara os valores esperados de saída com os encontrados após a tradução. O processo feito por BTestBox é todo automático e pode ser utilizado a partir do Atelier-B via instalação de plugin com uma interface simples.

Essa dissertação apresenta a ferramenta BTestBox. BTestBox foi testado através de pequenas implementações B com os elementos gramaticais possíveis da linguagem B. BTestBox apresenta funcionalidade e vantagens para programadores que utilizam o método formal B.

Palavras-chave: Método B, Teste de Software, Ferramenta, Atelier-B, Cobertura de Código.

Abstract

Software needs to be safe and run smoothly. From that assumption, new technologies and techniques are developed to test the quality of a program. This is more relevant when developing critical systems, such as railways and avionics control systems. Formal methods try to adress this need. When using B in Atelier-B, after proving the components of a project is necessary to translate to the desired language, a translation is made by using B translators and compilers. Usually, the process of compilation is safe when perfomed by mature compilers although they are not free of errors and bugs often crop up. The same reliability is not always observed in B translators since they have been on the market for less time. Software testing may solve and be used to perform the analyses of the translated code. From coverage criteria, it is possible to infer quality of a piece of software and detect bugs. This process is hard and time-consuming, mainly if it is perfomed manually. To address this problem, the BTestBox tool aims to analyze automatically the coverage of B implementations built through Atelier-B. BTestBox also automatically tests the translation from B implementations. For this, BTestBox uses the same test case generated to verify the coverage and compare the output expected values with the values found in the translation. This process is fully automatic and may be started from Atelier-B with a plugin.

This thesis presents the tool BTestBox. The tool is a solution to the problems exposed in the previous paragraph. BTestBox was tested with small B implementations and all gramatical elements from B language. It offers various functionalities and advantages to developers that use the B-Method.

Keywords: B-Method, Software Test, Tool, Atelier-B, Code Coverage.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Definições

CAPÍTULO 1

Introdução

O desenvolvimento de software cria e evolui tecnologias que auxiliam a expandir múltiplas áreas da ciência, fazendo da Ciência da Computação um dos pilares fundamentais da nossa sociedade, permitindo que a medicina, os estudos aeroespaciais e diversas outras avançarem. Além de permitir o desenvolvimento de outras áreas, softwares são utilizados para garantir a segurança de pessoas e de produtos. Em ambientes críticos, como na indústria automobilística, por exemplo, uma falha do software pode ocasionar danos físicos e materiais irreversíveis.

A exemplo dos acidentes ocasionados por falha de software, podem ser citados os com o foguete Ariane-5 [35] e das máquinas Therac-25 [34]. Em 1996, o foguete Ariane-5 explodiu devido a uma falha no software que ativou a auto-destruição em pleno ar. No acidente ocorrido com as máquinas Therac-25, os pacientes recebiam quantidades indevidas de radiação. Em ambos os casos a falha no software poderia ter sido evitada se medidas de segurança tivessem sido tomadas antes de sua utilização.

Em diversos meios são resguardados às agências reguladoras a responsabilidade legal de averiguar a segurança e a eficiência de um produto, visando, com isso, que os riscos de acidentes diminuam. Essas agências têm o respeito da sociedade e podem decidir o que pode, ou não, ser comercializado. Desta forma, são criados os certificados, que impõem a um produto quais normas devem ser respeitadas.

Cada vez mais softwares são utilizados nos mais diversos cenários da sociedade. Sendo assim, a sua regulamentação é mais abrangente e exigida. Quando utilizados para sistemas críticos, como na indústria aérea ou ferroviária, são requisitados certificados mais exigentes para a utilização do software. Diante disto, a Sociedade Brasileira de Computação (SBC) enumerou diversos desafios para o desenvolvimento de software de qualidade na computação, dentre eles sistemas seguros, disponíveis, corretos, escaláveis, persistentes e ubíquos [19].

Portanto, a fiscalização das agências faz com que a indústria deseje a integração de novas abordagens para conseguir atingir o necessário para a obtenção dos certificados. Dentre estes meios estão os métodos formais e o teste de software. Primeiramente, métodos formais não são visíveis para a indústria ao menos que tenham ferramentas de suporte adequadas [5]. Segundo, uma vez que problemas reais necessitam de diversas abordagens, e nenhum método formal é apropriado para solucionar todos os problemas, é necessário tomar vantagem dos pontos fortes de diferentes métodos formais.

Dentre os métodos formais, existe B, um método formal consolidado e que é utilizado há anos no desenvolvimento de sistemas para ambientes críticos [38]. Tal método se baseia na anotação de máquinas abstratas e na substituição generalizada. B suporta o desenvolvimento modular, isto é, inicialmente se tem uma máquina abstrata e através de subsequente refinamentos, que adicionam detalhes, tornam o modelo concreto. O último e mais concreto refinamento

é chamado de implementação. O objetivo desse processo é obter uma implementação provada do modelo abstrato [6]. Estas provas garantem propriedades sobre os modelos. A etapa final do processo é o elo mais fraco, a geração de código, através de compiladores B, que será efetivamente executado no produto em operação.

Existe uma preocupação no que diz respeito aos compiladores. Erros podem ocorrer e silenciosamente adicionar bugs a partir do código fonte correto [32]. Até mesmo quando o código alcança os requisitos funcionais desejados, o que pode ser demonstrado através de métodos formais, o código final pode não atingir os mesmo requisitos. Isso é confirmado pelo fato de que, em onze compiladores C foram descobertos mais de 325 erros de compilação [50]. Dentre esses compiladores havia cinco compiladores de uso livre (GCC, LLVM, CIL, TCC e Open64), cinco comerciais e o CompCert [31]. Compiladores e tradutores que são usados em comunidades menores têm maiores chances de risco tecnológico que ferramentas utilizadas em larga escala [44]. Na comunidade de B, o Atelier-B é uma ferramenta utilizada para criar componentes B, suportando diversos geradores de código. Além disso, esses geradores demandam uma maior atenção, pois são utilizados em ambientes de risco. Esses geradores de código, também chamados de compiladores B, apresentam os mesmos riscos de outros compiladores.

De maneira que os riscos sejam mitigados, os certificados podem exigir diferentes técnicas para um mesmo produto, tais como a utilização de testes de software. Também é importante ressaltar que os métodos formais não conseguem suprir toda a demanda necessária dos certificados. Atualmente, os métodos utilizados para checar a consistência de programas é baseado em simulações, testes, certificados e a verificação formal do código [26].

Nesse tocante, a técnica mais popular é o teste de software [9]. Como precaução, para realizar testes em sistemas críticos de segurança, um certificado é necessário para assegurar que os requerimentos do teste garantam a cobertura do código. Por exemplo, a certificação ”DO-178C”, requerida pela Administração Federal de Aviação (*Federal Aviation Administration* - FAA) provê um guia para softwares utilizados na aviação norte americana e, entre seus requisitos, demanda também a utilização da cobertura *Modified Condition/Decisive Condition* (MC/DC), uma cobertura de nível A [23], altamente confiável e consistente.

Em vista desse cenário, é proposto o BTestBox como uma ferramenta capaz de checar a tradução B através de teste. Na prática, o proposto para esse trabalho é uma contra-medida que é aceita por alguns certificados, uma checagem dupla que explora uma redundância. Desenvolver e aplicar duas diferentes medidas para um objeto em produção. Primeiramente o programa é desenvolvido em método formal B utilizando o Atelier-B e em seguida é posto em teste utilizando a ferramenta BTestBox. A tradução realizada através de compiladores B é executada e o resultado checado com o esperado pelo teste. E, caso uma diferença seja detectada, o usuário é avisado.

Entre as coberturas de código, MC/DC é a mais procurada. Porém, BTestBox contempla outras coberturas, tais como: análise de código morto através da Cobertura de Código, Cobertura de Galhos e Cobertura de Predicados para analisar as tomadas de decisões como um todo, além da cobertura de Caminhos para conferir se todas as opções possíveis de escolha são tomadas, e a cobertura de Cláusulas para observar se cada cláusula pode assumir um valor dentro da decisão.

Os testes realizados com BTestBox levam os critérios de cobertura em consideração e ten-

tam aplicá-los durante o teste da tradução. Esta prática permite a ferramenta verificar a cobertura do código e a tradução simultaneamente. Em um primeiro momento, BTestBox verifica o programa com relação a algum critério de cobertura, dessa forma, gera os casos de testes necessários para analisar a cobertura. Em um segundo momento, após a tradução do programa, BTestBox executa o código gerado e aplica os casos de teste apresentados no primeiro momento. Com isso, a ferramenta testa a tradução de acordo com os critérios de cobertura.

A apresentação dessa dissertação é organizada na seguinte estrutura: o capítulo 2 exibe os trabalhos relacionados ao BTestBox. Os capítulos 3 e 4 são responsáveis por apresentar o conteúdo necessário para entender o que o BTestBox realiza. O terceiro capítulo concede um breve resumo do que é o método B, as ferramentas que o suportam e introduz um pouco sobre a sintaxe e linguagem B. O quarto introduz testes de software, algumas das terminologias utilizadas nesta dissertação e cobertura de código. No capítulo 5 a ferramenta é explicada, focando a metodologia, suas funções e abordagem para o teste, além de conter a execução de um exemplo passo-a-passo. Finalmente, o capítulo 6 apresenta as considerações finais.

CAPÍTULO 2

Trabalhos Relacionados

Teste de software e métodos formais, os tópicos utilizados nessa dissertação, são correlacionados e podem ser utilizados em conjunto, uma vez que ambos são ferramentas para garantir a segurança do software. Usualmente, métodos formais e teste de software são utilizados de maneira separada, porém atualmente existe o esforço em combinar essas duas técnicas. O teste automático de casos de teste a partir de métodos formais já vem sendo estudado por alguns grupos de pesquisa.

Em uma pesquisa contendo o estado da arte para ferramentas de teste de modelo [36] é apresentado BZ-TT [1] e ProTest [43] duas ferramentas utilizadas pelo método B. A primeira ferramenta auxilia o usuário na geração de casos de teste a partir de modelos B e Z. O objetivo é testar cada operação declarada ao atingir o estado limite das variáveis com relação a resolução de restrições. O estado limite significa que pelo menos uma variável do sistema esteja no valor máximo ou mínimo. BZ-TT é privado e não é disponível para o público. A segunda ferramenta testa automaticamente o ambiente para especificações B. A ferramenta é baseada no que o usuário quer testar, ele define os requisitos do teste e então os casos de teste são gerados. Com os casos de teste, a ferramenta usa técnicas de checagem de modelo para encontrar sequências que satisfaçam os parâmetros. A ferramenta somente gera testes de casos abstratos que necessitam ser implementados antes de serem executados.

BETA é outra ferramenta utilizada para testes [37]. Esta, por sua vez, depende do particionamento do espaço de entrada e dos critérios de cobertura lógica para a geração de testes unitários a partir de máquinas abstratas B. A ferramenta automatiza todos os passos do processo de geração de teste, a partir da geração dos dados de teste até a concretização do teste e a geração de script.

BTestBox já foi apresentado em 2016 a partir de um artigo [15]. Nesta ocasião, BTestBox era um pouco diferente do que é atualmente. Ele gerava casos de teste utilizando um script de teste e blocos de comandos que eram gerados randômicamente. Mas já era possível testar uma implementação através de testes criados diretamente da linguagem de programação final.

Diferentemente do *BETA*, os casos de teste do BTestBox são gerados a partir da implementação do modelo, o que é mais próximo do software real. Outra diferença é que o *BETA* é focado no teste de unidade, enquanto BTestBox testa o módulo inteiro e suas funções.

Outro esforço da comunidade de pesquisa em teste é como aplicar eficientemente as coberturas com o menor tempo de execução. Isso também é um esforço do BTestBox. Na literatura diferentes algoritmos são propostos para auxiliar nesse problema.

Search-Based Test Data Generation (SBTDG) é uma maneira de gerar dados para testes e, usualmente, composto por técnicas de otimização. Várias pesquisas utilizam SBTDG, algumas delas são: Ghani com *Simulated Annealing* [20] que visava a geração de testes para alcançar a

cobertura MC/DC; Awediking e o seu algoritmo *Hill Climb* [4]; algoritmos genéticos utilizados por Awediking [4] e Minj [39]; *Concolic Testing* para alcançar alta cobertura MC/DC [45]; Rungta e a avaliação parcial [49]. A maioria desses trabalhos tem o foco de solucionar o MC/DC, pois este é demandado pela FAA, mas não é somente essa cobertura o objetivo. Gupta desenvolveu uma técnica de geração de dados para a cobertura de galhos [22]. Para cobertura de código foram utilizados algoritmos gananciosos para reduzir o grupo de teste [29] e algoritmos genéticos [42]. BTestBox também apresenta um algoritmo para a solução de coberturas, porém usa da lógica de Hoare para encontrar um predicado que apresenta um estado possível para um caso de teste.

Uma vez que MC/DC parece ser a cobertura mais difícil a ser implementada, mais pesquisas são feitas para compreender como e onde utilizar essa cobertura. MC/DC é largamente utilizado na indústria, sendo mais aplicada na aviação, por causa do certificado DO-178B, mas é empregada em diversos outros setores industriais, como o ferroviário e a indústria aeroespacial, sistemas eletrônicos ou sistemas programáveis, produção de software e indústria militar. Essa cobertura também é largamente estudada na academia. Neste ambiente, existe comparações entre coberturas, desenvolvimento de ferramentas, estudos para identificar a relação entre cobertura de código e cobertura de modelo, entre outros. A tabela 2.1 mostra cada campo onde o MC/DC é encontrado.

Tabela 2.1 Artigos por assunto e área de trabalho.

	Artigos
Aviação	[40], [30], [21]
Ferrovias	[40]
Aeroespacial	[30]
Sistemas Eletrônicos	[27], [41]
Militar	[40], [30]
Comparação entre Coberturas	[28], [47]
Estudos Acadêmicos	[13], [21], [8], [18], [9]

Com foco no campo da aviação e da indústria ferroviária, Jan Peleska [40] apresenta uma abordagem unificada para a interpretação abstrata, verificação formal e teste de módulos C/C++. As técnicas utilizadas são implementadas à ferramenta RT-Tester desenvolvida pelo autor e sua equipe de pesquisa na Universidade de Bremem em cooperação com a *Verified Systems International GmbH* [46]. A abordagem destes testes foca pequenos programas unitários (funções e métodos) e devem ser guiados através da perícia de um desenvolvedor ou especialista em verificação. Além disso, RT-Tester concede mecanismos para a especificação das pré-condições sobre os dados de entrada aceitáveis ou esperados para a unidade em inspeção, como também gera de maneira semi-automática um *mock* apresentando ao usuário o comportamento da unidade a ser analisada sempre que esta é executada. Esta ferramenta é utilizada na indústria e tem a capacidade de aplicar testes para a verificação da cobertura de predicado, cobertura de código e a cobertura MC/DC. Em comparação com o BTestBox, o RT-Tester se difere por realizar uma interpretação abstrata enquanto a ferramenta apresentada nessa dissertação utiliza de componentes concretos, além disso BTestBox é automatizado. Porém, se

assemelha em realizar uma verificação formal.

Andrew Korneci e Janusz Zalewski [30] descrevem o estado da arte sobre o tema *Certification of software for real-time safety-critical systems: state of the art*. Eles discutem os diferentes certificados com foco na indústria aeroespacial, na aviação e nos usos militares. Além de explanar quais são aceitos nos Estados Unidos da América e quais são adotados no resto do mundo. Adicionalmente, relatam quais são as linguagens aceitas pelos certificados e como as ferramentas desenvolvidas por pesquisadores devem ser qualificadas para serem utilizadas. A ideia do BTestBox nasceu a partir da necessidade de obter certificados para os componentes desenvolvidos com o método B.

Formal testing for separation assurance [21] apresenta testes formais realizados para um algoritmo de aproximação de aeronaves. Para estes testes Dimitra Giannakopoulou et al. utilizam a plataforma TSAFE, desenvolvida pela mesma pesquisadora. Neste artigo, apresenta-se três diferentes abordagens de geração de caso de teste, a aplicação deles em um protótipo de segurança, assim como os pontos positivos e negativos. A plataforma TSAFE suporta geração automática de casos de testes que são construídos a partir de execução simbólica. Para finalizar, os autores apresentam uma grande quantidade de análise estatística dos testes realizados com a plataforma. Para a geração de teste os autores utilizam de execução simbólica, checagem de modelos e teste combinatorial. BTestBox realiza os testes de maneira semelhante, através de um software de checagem de modelos são obtidos os casos de testes.

A fim de aplicar testes de software em sistemas eletrônicos, Susanne Kandl et al. [27] apresentam uma plataforma capaz de testar sistemas de ambientes críticos com base em conceitos de teste baseado em modelos. Na abordagem dos autores, o sistema automaticamente extrai o modelo a partir do código C. A plataforma gera os dados de entrada randomicamente, porém com métodos formais. Em seguida, para selecionar os casos válidos de teste são utilizados requerimentos definidos na especificação do sistema. Adicionalmente, para cobrir caminhos diferentes de execução os autores desenvolveram um novo método baseado em um critério de cobertura estrutural especial. Para finalizar os autores apresentam resultados de um estudo de caso de um modelo industrial concreto. Em sua primeira versão, BTestBox também utilizava de testes randomicos para a geração dos dados de entrada, mas atualmente depende da definição de um critério de cobertura para a geração de casos de teste.

Alexander Pretschner et al. [41] discutem teste baseado em modelos de uma maneira geral e aplicam os conceitos em um estudo de caso com *smart card*. Eles apresentam a ferramenta AUTOFOCUS, que utiliza conceitos de modelagem, e uma abordagem de geração de teste de caso utilizando a execução simbólica com um programador de restrições lógicas. Os autores também explanam sobre diferentes estratégias e algoritmos utilizados para a geração do caso de teste. Um ponto forte da abordagem que os autores escolheram para seus testes é a utilização de regras de restrições, porém é necessário que um especialista interaja com a ferramenta.

No domínio B, Frédéric Dadeau et al. [13] discutem um caminho para a produção de casos de teste para o mini-desafio POSIX, baseado em um modelo formal de um administrador de arquivo, os autores escrevem uma máquina B. Dadeau et al. apresentam um estudo de caso no qual ilustram as limitações de um processo de teste totalmente automático, o que justifica os diferentes cenários que complementam a abordagem clássica de teste. Eles utilizam um motor de animação simbólica para computar parâmetros das operações, nos quais são baseados no

critério de cobertura desejado. Em relação ao BTestBox, a utilização da lógica de Hoare assim como a tentativa de obter caminhos por quais o software é executado acaba resultando no uso de execução simbólica. Com a execução simbólica é obtido um predicado que determina se é possível a execução daquele caminho ou não existe estado satisfatório dentro da máquina para que a operação ocorra daquela maneira.

Matthieu Carlier e Catherine Dubois apresentam a geração e a execução de casos de teste com a plataforma Focal em *Functional Testing in the Focal Environment* [8]. Focal é uma linguagem de programação na qual todas as propriedades do programa são escritas no código fonte. Os autores testam o código em função dessas propriedades e separam o teste em dois estágios. No primeiro eles dividem a propriedade em diversas propriedades elementares, que por sua vez são uma tupla composta de algumas pré-condições e uma conclusão. Por último, cada propriedade elementar é testada separadamente. As pré-condições geram os casos de testes de maneira randomica. Todo o processo é realizado automaticamente.

Devido a necessidade de testar os modelos gerados através do Simulink do MATLAB, Yunwei Dong et al. [18] tentam confirmar uma relação entre a cobertura de modelo e a cobertura do código. Os autores utilizam um método que identifica as restrições impostas através do modelo do MATLAB e geram casos de teste capazes de alcançar uma grande porcentagem de cobertura.

Test-data generation for control coverage by proof [9] discute como a caracterização formal de um critério de cobertura pode ser utilizado para a geração de dados de teste. Ana Cavalcanti et al. apresentam um procedimento baseado nas tradicionais técnicas de programação, tais como normalização e o cálculo da pré-condição mais fraca. Eles tentam usufruir de um provador algébrico de teoremas para automatizar o processo. Caso o provador falhe em produzir um teste, os autores somente conseguem a especificação de um conjunto de teste complacente.

Como pode ser observado através dos trabalhos citados acima, vários softwares e algoritmos podem ser utilizados para verificar a cobertura de softwares, alguns, como em [8], utilizam em conjunto o teste de software e os métodos formais. As técnicas utilizadas são variadas, como a checagem de modelo para a geração de casos de teste [21], ou a utilização execução simbólica [41], ou testes randômicos [27], dentre outros. Em sua maioria, os autores desenvolvem diferentes maneiras para a verificação, podendo ser necessária a presença de um profissional [41] ou totalmente automático [8]. BTestBox é mais um software que aproveita do conjunto de teste de software com métodos formais, realizando consigo, de maneira automática, testes em códigos traduzidos a partir de B com base em critérios de cobertura.

CAPÍTULO 3

Método B

O método B é uma metodologia de especificação, validação e construção de componentes de software, sendo utilizado para desenvolver programas para ambientes de condições críticas. Tal método apresenta uma linguagem simples porém expressiva, incluindo lógica de primeira ordem, teoria dos conjuntos e aritmética. Além disso, suporta a modelagem modular, isto é, cada módulo é responsável pela especificação de um componente do software em um diferente nível de abstração [26]. A ideia é começar com um modelo abstrato do sistema e, gradualmente, em subsequentes refinamentos adicionar estruturas concretas. O último refinamento é chamado de implementação e este é o mais concreto componente em B. A figura 3.1 apresenta uma visão geral do processo de desenvolvimento em B. Nesta figura é possível observar as etapas de construção formal através dos pontos 1 até o 4. Ao chegar na implementação, o objetivo é que todo o componente esteja provado: a máquina para provar a consistência da especificação e os refinamentos para expor a conformidade com o nível anterior. Essas provas asseguram que o modelo satisfaz propriedades que ratificam o bom funcionamento e implementação do modelo abstrato. Ainda assim, existe um passo fraco dentro do processo de desenvolvimento: a geração do código binário a partir da implementação B, isto é, a tradução para uma linguagem de programação e a aplicação em um compilador próprio.

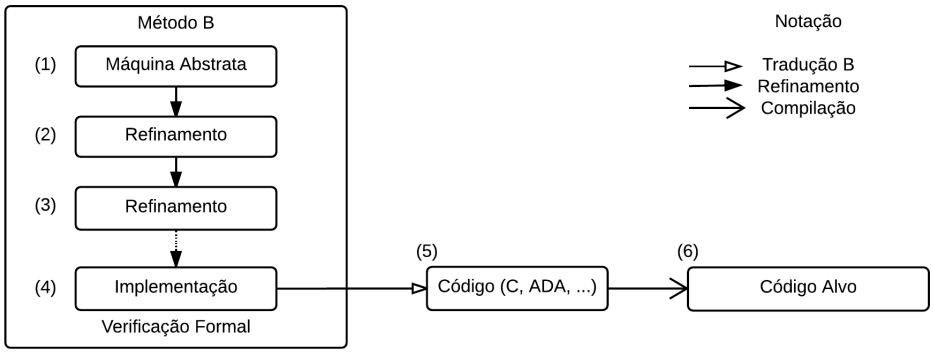


Figura 3.1 Método B [38].

Como pode ser observado na figura 3.1, cada módulo B consiste dos seguintes componentes: uma máquina abstrata, uma implementação e pode ter vários ou nenhum refinamento. É possível verificar a consistência de cada componente através de obrigações de prova geradas partindo das cláusulas B utilizadas pelo componente.

Nas próximas seções será apresentado o método B e suas definições.

3.1 Máquinas, Refinamentos e Implementações

A linguagem utiliza de módulos que correspondem ao software a ser desenvolvido. Cada módulo contém um estado que é descrito através das variáveis, que são modificadas por meio de operações. As operações apresentam a sua própria seção para serem declaradas e podem ter entradas e saídas. As operações são especificadas de forma que sua saída deve ser calculada levando em consideração o valor das entradas, além do estado atual. Além de calcular a saída, a aplicação da operação pode modificar o valor das variáveis do componente, alterando seu estado. Logo, para descrever um componente é necessário especificar todos os estados legais deste. A especificação dos estados legais das variáveis é declarada no invariante.

Para verificar se um componente B está bem estruturado é necessário mostrar que para toda operação que é aplicada a um estado que satisfaz ao invariante, ela precisa encerrar em um estado no qual o invariante continue sendo respeitado.

Os diferentes níveis de abstração dos modelos B são definidos em: máquinas, refinamentos e implementações. Cada tipo de componente tem suas próprias particularidades. Estas estão descritas abaixo:

- Máquinas: estado abstrato do modelo. A máquina contêm um conjunto de variáveis que são o estado e operações que podem alterar o estado atual. A máquina deve apresentar o invariante. Esta cláusula define todos os estados legais das varíaveis, como informações de tipagem ou qualquer outra restrição. Além disso, cada operação contém uma precondição que precisa ser respeitada para garantir a correta execução da operação, o estado das variáveis ao término e o início de cada operação deve estar de acordo com o invariante. Ademais, os componentes B incluem a cláusula da inicialização que permite especificar as restrições que a máquina deve satisfazer inicialmente, estas condições devem estar de acordo com o invariante e para observar este fato, obrigações de prova são geradas. Adicionalmente, B permite que as máquinas importem outros modelos a partir de diferentes cláusulas: *Uses* concede a capacidade de utilizar as operações da outra máquina; *Sees* autoriza ver todas as entidades de outra máquina; *Includes* cuja utilização permite o uso de operações de outro componente mas sendo necessário obedecer o invariante; Por fim, *Extends* tem a mesma função do *Sees* e *Includes* juntos. Outras cláusulas que permitem adicionar condições, restrições e definições são: *Sets* que permite a definição de conjuntos; *Constants* na qual é possível declarar constantes; *Properties* que descreve as condições impostas sobre as contantes e os conjuntos declarados nas duas cláusulas anteriores; *Assertions* que compreende uma lista de predicados referentes as variáveis da máquina, que por sua vez auxiliam na prova do componente; *Constraints* é usada para tipar parâmetros escalares da máquina e expressar propriedades complementares aplicadas a esses parâmetros. A descrição de uma máquina abstrata é uma coleção de pedaços de informação organizada em diferentes cláusulas, para assumir que esta é coerente, se faz necessário provar que o invariante é consistente, que a inicialização estabelece um estado inicial em que o invariante é respeitado e que a execução de qualquer operação preserva o invariante.
- Refinamento: o nível intermediário entre o componente abstrato e o concreto, podendo assim utilizar-se dos operadores de ambos os tipos. Um refinamento mandatoriamente

tem de refinar uma máquina ou outro refinamento. Ainda, os refinamentos precisam ser provados para validar a relação entre o refinamento e o componente refinado.

- Implementação: estado concreto do modelo. Somente permite a notação B0, uma estrutura de linguagem subconjunta do B. A notação garante que o programa B seja construído com cláusulas concretas, fazendo com que seja mais adequado para a tradução do código B. Assim como no nível abstrato, o modelo concreto têm suas cláusulas particulares, tais como: *Refines* que define qual componente está sendo refinado e a cláusula *Values* a qual determina valores para as constantes concretas. Adicionamente, este componente também necessita de prova para validar a relação entre a implementação e o componente refinado, além de provar que a execução das operações sempre termina, quando essas possuem laços.

3.2 Substituições B

O método B usa o conceito de substituições generalizadas, que são baseadas nos trabalhos de Dijkstra [17] e Hoare [25].

As substituições visam auxiliar a verificação por prova uma vez que tornam simples determinar o que deve ser analisado, ou seja, facilitam a geração das obrigações de prova. As operações são escritas a partir das substituições, e então, é possível de maneira mecânica, aplicar estas a um predicado. Em outras palavras, as substituições são utilizadas para reescrever variáveis livres em um predicado.

As substituições usam a notação $[E/x]P$. Isso significa que uma variável x em um predicado P é reescrita com a expressão E. Sendo assim é possível calcular a partir de um estado inicial um outro estado final que satisfaz o invariante.

Dessa maneira, se o invariante for o predicado I e S a substituição, para provar que uma operação está correta é preciso mostrar que a seguinte fórmula é verdadeira:

$$I \Rightarrow [S]I$$

B apresenta outras substituições e algumas delas serão explicadas nessa seção. A semântica das substituições é de lógica de primeira ordem. Algumas das mais usadas substituições estão descritas abaixo. E para informações mais completas e variadas, ClearSy mantêm um manual da linguagem B e apresenta a lista completa de substituições [10].

- **SKIP**: substituição que não altera o estado das variáveis. Em $[skip]Q \equiv Q$ é possível observar que a substituição *skip* não altera a expressão Q.

$$[skip]Q \equiv Q$$

- **BEGIN**: declaração de um bloco de comandos, dentro deste bloco as outras cláusulas podem ser utilizadas. Na expressão $[\text{BEGIN } S \text{ END}]Q \equiv [S]Q$, é relatado que a cadeia de comandos, *BEGIN* S *END* é equivalente a uma substituição $[S]$.

$$[BEGIN\ S\ END\ ]Q \equiv [S]Q$$

- **ASSIGNMENT**: substituição do a por b dentro de Q. Caso a ocorra dentro de Q a expressão é modificada acontecendo a substituição por b. Se a está ausente em Q, a substituição não altera o predicado. Outro evento é a substituição com uma conjunção de predicados, o que aparece no terceiro caso, se ocorre uma substituição a essa união, é equivalente a substituição aplicada separadamente aos predicados e em seguida a união destes, o que é evidenciado em $[a := b]Q_1 \wedge Q_2 \equiv [a := b]Q_1 \wedge [a := b]Q_2$.

$$[a := b]Q \equiv b \text{ (Se } Q = a)$$
$$[a := b]Q \equiv Q \text{ (Se } a \text{ não ocorre em } Q)$$
$$[a := b](Q_1 \wedge Q_2) \equiv [a := b]Q_1 \wedge [a := b]Q_2$$

- **PRE**: significa que uma precondição é necessária para que a substituição suceda. Na expressão abaixo é preciso que a precondição P seja satisfeita para que a substituição S possa ser executada dentro da expressão Q.

$$[\text{PRE } P \text{ THEN } S \text{ END}]Q \equiv (P \wedge [S]Q)$$

- **IF**: aplica condicionalmente uma substituição. Na expressão abaixo, se a condição P é verdadeira, então $S1$ acontece, senão $S2$ ocorre. O *ELSE* é opcional, caso não for definido ele é equivalente a uma substituição *SKIP*.

$$[\text{IF } P \text{ THEN } S_1 \text{ ELSE } S_2 \text{ END}]Q \equiv (P \Rightarrow [S_1]Q) \wedge (\neg P \Rightarrow [S_2]Q)$$

- **CASE**: uma série de escolhas possíveis é oferecida e somente uma escolhida. E é a expressão a qual deve ser enquadrada em um dos casos do *CASE*. Primeiro é testado se E tem o mesmo valor da expressão e_1 e em caso afirmativo S_1 sucede, senão é checado se E apresenta o mesmo valor da expressão e_2 e, em caso afirmativo, S_2 acontece.

$$[\text{CASE } E \text{ OF EITHER } e_1 \text{ THEN } S_1 \text{ OR } e_2 \text{ THEN } S_2 \text{ END }]Q \equiv$$
$$(E = e_1 \Rightarrow [S_1]Q) \wedge (E = e_2 \Rightarrow [S_2]Q)$$

- **ANY**: cria localmente dentro do bloco da declaração do *ANY* uma variável x que respeita o predicado P e o aplica em S. x pode ser qualquer valor que satisfaz o predicado P.

$$[\text{ANY } x \text{ WHERE } P \text{ THEN } S \text{ END}]I \equiv \forall x(P \Rightarrow [S]I)$$

- **PARALLEL**: substituição em paralelo, então as substituições ocorrem ao mesmo tempo. Não se pode substituir a mesma variável duas vezes, ao mesmo tempo. Na fórmula $[x := E \,||\, y := F]Q \equiv [x, y := E, F]Q$ a substituição das variáveis x e y acontece simultaneamente por E e por F, respectivamente.

$$[x := E \,||\, y := F]Q \equiv [x, y := E, F]Q$$

- **SEQUENCING**: substituição em sequência. Nesta regra uma cadeia de comandos pode ser definida e então aplicadas de maneira sequencial. Como pode ser especificado com a fórmula $[S_1;\, S_2]Q \equiv [S_1][S_2]Q$, a substituição S_1 ocorre antes de S_2. No entanto, para sa-

ber se Q é satisfeito ao fim da execução, é necessário primeiro calcular $[S_2]Q$, resultando em Q', se a condição Q' for satisfeita, então a execução de S_2 garante Q. Em seguida deve-se calcular $[S_1]Q'$, resultando na condição Q'', que se por sua vez satisfeita, implica que a execução de S_1 garante Q'. Dessa forma, ao término da execução de $[S_1;\ S_2]Q$ é possível observar se a condição é Q é s Esta ordem deve ser respeitada, senão a expressão final pode ser diferente.

$$[S_1;\ S_2]Q \equiv [S_1][S_2]Q$$

- **WHILE**: Para provar o laço é necessário provar dois requisitos:
  1. A execução do laço sempre finaliza, ou seja, existe um número finito de execuções dentro do corpo do laço.
  2. Ao término da execução do laço, a condição Q é verdadeira.

  Para auxiliar a prova, o laço apresenta um invariante I e um variante v. I é um predicado, que deve ser verdadeiro cada vez que a condição do laço é avaliada, e v é uma expressão. Com isso, é possível representar os dois requisitos em cinco condições.

  1. A execução do corpo do laço S deve manter o invariante I quando a condição do laço P é verdadeira, ou seja, deve-se verificar $(I \wedge P \Rightarrow [S]I)$
  2. Quando a condição do laço P é falso, então a condição Q é verdadeira, ou seja, deve-se provar $(I \wedge \neg P \Rightarrow Q)$
  3. A cada execuçao do laço, o variante é um número inteiro positivo, deve-se provar $(I \wedge P \Rightarrow v \in \mathbb{N})$
  4. Cada vez que o corpo do laço é executado, o valor do variante deve diminuir, deve-se então provar $(I \wedge P \wedge v_i = v \Rightarrow [S](v < v_i)$
  5. Na primeira iteração, verifica-se o invariante I.

  Através das condições 1, 3, 4 e 5 se é capaz de satisfazer o primeiro requisito, com a condição 2 é póssivel satisfazer o segundo requisito.

$$\begin{gathered}[\text{WHILE } P \text{ DO } S \text{ INVARIANT } I \text{ VARIANT } v \text{ END}]Q \equiv \\ (1)(I \wedge P \Rightarrow [S]I) \\ (2)(I \wedge \neg P \Rightarrow Q) \\ (3)(I \wedge P \Rightarrow v \in \mathbb{N}) \\ (4)(I \wedge P \wedge v_i = v \Rightarrow [S](v < v_i) \\ (5)I\end{gathered}$$

Algumas regras são somente aplicadas sobre máquinas abstratas, enquanto outras somente no domínio concreto. A tabela 3.1 sintetiza quando uma substituição pode ser utilizada.

3.3 Expressões em B

B tem sua própria notação para escrever os predicados, asserções e condições. Em B existe sintaxe para representar relações, funções, lógicas de conjuntos e operadores aritméticos. As tabelas 3.2, 3.3, 3.4 e 3.5 apresentam essa sintaxe.

Tabela 3.1 Substituições permitidas para cada tipo de componente.

Cláusula	Permitido em:		
	Máquina	Refinamento	Implementação
SKIP	√	√	√
BEGIN	*X*	√	√
ASSIGNMENT	√	√	√
PRE	√	√	*X*
IF	√	√	√
CASE	√	√	√
ANY	√	√	*X*
PARALLEL	√	√	*X*
SEQUENCING	*X*	√	√
WHILE	*X*	√	√

Tabela 3.2 Notação B: Funções e relações [37].

Símbolo ASCII	Descrição
+->	Função parcial
+-»	Subjeção parcial
>+>	Injeção parcial
–»	Subjeção
–>	Função total
>->	Injeção total
>-»	Bijeção total
<->	Relação
|->	Mapeamento

Com estes operadores é possível construir os componentes B e através da mesma sintaxe são realizadas as provas dos componentes criados. A notação aqui apresentada é parcial, para mais informações a ClearSy, mais uma vez, tem um documento com todos os símbolos B, palavras-chave e operadores [11].

3.4 Obrigações de Prova

Todo componente em B deve respeitar obrigações de prova a fim de demonstrar a sua consistência. Diferentes componentes apresentam diferentes obrigações de prova, assim como máquinas e refinamentos. Uma observação pertinente é que por serem refinamentos, as implementações compartilham das mesmas obrigações de prova.

As obrigações de prova são geradas a partir dos diferentes elementos do modelo e utilizam as substituições B já apresentadas. As obrigações de prova apresentadas nessa seção são: Consistência do invariante, inicialização, obrigações de prova das operações, as obrigações de prova de laço, além das obrigações de prova para refinamentos.

Tabela 3.3 Notação B: Operadores lógicos [37].

Símbolo ASCII	Descrição
&	Conjunção
or	Disjunção
#	Quantificação Existencial
!	Quantificação Universal
=	Igualdade
/=	Desigualdade
=>	Implicação Lógica
<=>	Equivalência
not(P)	Negação

Tabela 3.4 Notação B: Operadores de conjuntos [37].

Símbolo ASCII	Descrição
:	Pertence
/:	Não pertence
<:	Inclusão
/<:	Não incluso
/\	Interseção
\/	União
{ }	Conjunto vazio
POW	Conjunto de subconjuntos

3.4.1 Consistência do invariante

O requerimento para a consistência do invariante é que algum estado da máquina seja consistente com o invariante, isto é simplismente expressado por:

$$H \Rightarrow \exists v.I$$

No qual v significa o estado das variável, I o invariante e H é o conjunto de hipóteses composto pelo conteúdo das cláusulas que condicionam as varíaveis, como as cláusulas *CONTRAINTS* e *PROPERTIES*. É obrigatório que o invariante seja respeitado no início e ao término de uma operação, além da inicialização, o que será demonstrado nas obrigações de prova seguintes. O Atelier-B não produz a obrigação de prova para a consistência do invariante.

3.4.2 Prova de Inicialização

A inicialização do sistema também demanda uma prova. Quando o componente é inicializado, ele deve respeitar o invariante. Para isso, a fórmula abaixo demonstra o que é necessário:

$$H \Rightarrow [T]I$$

Nesta fórmula, $[T]$ representa a substituição que ocorre na inicialização, I o invariante e H

Tabela 3.5 Notação B: Operadores aritméticos [37].

Símbolo ASCII	Descrição
+	Adição
-	Subtração
*	Multiplicação
/	Divisão
<	Menor que
>	Maior que
<=	Menor ou igual que
>=	Maior ou igual que

é o conjunto de hipóteses. Desta forma, a prova de inicialização implica que as condições e parâmetros impostos em H implica sobre a substituição inicial T no invariante I. No caso a checagem de $[T]I$ requer confirmar que todos os valores fornecidos na cláusula da inicialização são adequadas ao invariante.

Geralmente é uma má prática utilizar as variáveis da máquina nas expressões da cláusula *INITIALIZATION*, pois isto corresponde em tornar a inicialização dependente de um estado randômico da máquina quando esta é ligada. É preferível ter um maior controle sobre o estado inicial da máquina.

3.4.3 Obrigações de prova para operações

As obrigações de prova para operações levam em consideração as operações declaradas na máquina e sua precondição, elas provam que a operação não quebra o invariante quando é chamada, além de determinar os possíveis estados para a chamada da operação. Para uma operação ser executada, o invariante e a precondição devem ser satisfeitos. Após a execução de uma operação, o novo estado das variáveis também deve satisfazer o invariante. O que pode ser caracterizado a partir da seguinte fórmula:

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

Na fórmula, I é o invariante, P a precondição da operação, H o conjunto de hipóteses e S é o corpo da operação. A precondição P expressa as condições no estado da máquina e os valores de entrada que realizarão S corretamente. O usuário da máquina é o responsável por assegurar que P é verdadeiro quando a operação é invocada. A máquina não garante nenhum comportamento quando P é falso, nem mesmo que a máquina termina. A expressão aci;a afirma que a máquina está em um estado no qual o invariante I, a precondição P e as hipóteses contidas em H são verdadeiras. Além disso, a operação é abastecida com uma entrada em acordância com P, então o comportamento, descrito em S, deve garantir o restabelecimento de I. Então, a operação PRE P THEN S END é consistente com o invariante I se é garantido em preservar I sempre que é invocado corretamente.

3.4.4 Obrigações de Prova para Refinamentos

3.4.4.1 Inicialização do Refinamento

Para *T1* ser um refinamento de *T*, é necessário que $\neg[T]\neg J$ seja verdadeiro para qualquer estado que *T1* possa alcançar, onde *T1* é a inicialização do refinamento, *T* é inicialização do componente refinado e *J* é o invariante de ligação entre os componentes. Isso simplesmente expressa que *T1* deve garantir alcançar um estado no qual $\neg[T]\neg J$ seja verdadeiro. Esse fato é observado através da fórmula a seguir:

$$[T1]\neg[T]\neg J$$

Em outras palavras, qualquer transição em *T1* deve alcançar um estado no qual alguma transição de *T* possa estabelecer o invariante de ligação *J*. Isso significa que qualquer transição de *T1* tem uma equivalente transição em *T*, o que é exatamente o requerido para a inicialização *T1* ser um refinamento válido de *T*.

3.4.4.2 Operações do refinamento

Diferentemente da inicialização, as operações do refinamento são executadas a partir de um estado da máquina, e as execuções anteriores devem ser levadas em consideração. É necessaário que $[S1]\neg[S]\neg[J]$ seja verdadeiro em qualquer estado no qual a máquina e seu refinamento estejam em conjunto, onde $S1$, S e J são o corpo da operação refinada, o corpo da operação que está sendo refinada e o invariante de ligação, respectivamente. O estado dessas variáveis deve estar de acordo com os invariantes I e J. Além disso, a operação só pode ser chamada quando o estado das variáveis satisfaz a precondição P. Então, a obrigação de prova para o refinamento de uma operação é:

$$I \wedge J \wedge P \Rightarrow [S1]\neg[S]\neg J$$

O predicado $[S1]\neg[S]\neg$ é um predicado sobre o estado da máquina e seu refinamento. Então, geralmente será verdadeiro para alguns pares de estado e falso para outros. Será verdadeiro para qualquer par de estados em toda execução de *S1* que é equivalente com uma execução de *S*, e será falso para o par de estados no qual a execução de *S1* não equivale a execução de *S*.

No caso geral, uma operação refinada contêm uma precondição $P1$. Em alguns casos pode ser proveitoso manter uma precondição para estruturar as provas do refinamento, uma vez que a verificação de um refinamento mais profundo da operação PRE $P1$ THEN $S1$ END pode fazer uso direto de $P1$ em vez da precondição abstrata P. O requerimento de $P1$ é que seja verdadeiro sempre que a operação esteja de acordo com operação abstrata. Em outras palavras, $P1$ não deve rejeitar nenhuma invocação que podem ser realizadas pela especificação. Quando os componentes estiverem em pares de estado relacionados, se a precondição P é satisfeita então $P1$ deve estar satisfeita. Isso é expresso formalmente a seguir:

$$I \wedge J \wedge P \Rightarrow P1$$

3.4.4.3 Operações do refinamento com saída

Expressar o requerimento de operações com saída deve levar em consideração o fato de que as operações atualizam a mesma variável. De fato, as variáveis de saídas não são parte do estado da máquina ou do refinamento, portanto elas não são mencionadas no invariante de ligação *J*. Saídas devem ser explicitamente mencionadas como parte da obrigações de prova do refinamento.

Para exprimir o requerimento de operações com saída é necessário renomear a saída de uma das operações para algo diferente da outra, se determinarmos que a saída da operação na máquina é *out* devemos diferenciar a saída do refinamento de *out*. Portanto, a saída da execução de *S1* é chamada de *out'* em vez de somente *out*. Modificando cada aparição de *out* em *S1* temos a substituição *S1*[*out*/*out*']. O requerimento nas operações com saída relaciona as saídas de forma que elas devam ter o mesmo valor, e para isso é utilizado o predicado *out* = *out'*, ou $out'_1 = out_1 \wedge ... \wedge out'_n = out_n$ para uma lista de variáveis. O requerimento para saídas relacionadas é expresso de uma maneira similar aos estados relacionados: $[S1[out'/out]]\neg[S]\neg(out' = out)$. Todo possível estado para *out'* deve coincidir com algum estado para *out*.

As operações com saída só podem ser executadas a partir de estados relacionados, além de satisfazer a precondição *P1*. Os estados resultantes também devem estar relacionados por *J*. Resultando na seguinte obrigação de prova:

$$I \wedge J \wedge P \Rightarrow [S1[out'/out]]\neg[S]\neg(J \wedge out' = out)$$
$$I \wedge J \wedge P \Rightarrow P1$$

No qual *I* representa o invariante da máquina, *J* o invariante de ligação, *P* a precondição da operação, *S*1 o corpo da operação na máquina, out' a saída da operação na máquina, *out* a saída da operação no refinamento e *P*1 é a precondição da operação refinada.

3.4.4.4 Obrigações de Prova com constantes e conjuntos

Em adição às obrigações de prova sobre comportamentos dinâmicos da máquina refinada, é necessário assumir que novos conjuntos e constantes são introduzidos no refinamento e isso deve ser instanciado e consistente com a cláusula *PROPERTIES*. A expressão do requerimento é inteiramente similar para as máquinas abstratas. Parâmetros não podem ser introduzidos em um refinamento, logo não existe a cláusula *CONSTRAINTS*. Portanto as obrigações de prova introduzidas por um refinamento são:

- Conjuntos e Constantes: St_2, k_2, B_2 são conjuntos, constantes e propriedades do refinamento, enquanto St_1, k_1, B_1, C_1 são conjuntos, constantes, propriedades e restrições da máquina abstrata refinada. Então, a obrigação de prova é:

$$C_1 \Rightarrow \exists St_1, k_1, St_2, k_2 . B_1 \wedge B_2$$

- Inicialização: se *T* é a inicialização da máquina abstrata, *T*1 a inicialização do refinamento e *J* o invariante de ligação. Assim sendo, a obrigação de prova é:

$$C_1 \wedge B_1 \wedge B_2 \Rightarrow [T1]\neg[T]\neg J$$

- Operações: as obrigações de prova para operações devem ser satisfeitas para todas as operações de um refinamento. Para uma operação **PRE** P **THEN** S **END** que apresenta uma saída out e um refinamento descrito através de **PRE** $P1$ **THEN** $S1$ **END** é necessária a análise das seguintes obrigações de prova:

$$C_1 \wedge B_1 \wedge B_2 \wedge I \wedge J \wedge P \Rightarrow [S1[out'/out]]\neg[S]\neg(J \wedge out' = out)$$
$$C_1 \wedge B_1 \wedge B_2 \wedge I \wedge J \wedge P \Rightarrow P1$$

3.5 Ferramentas B

O método B também apresenta ferramentas que fornecem suporte para os usuário. As ferramentas mais conhecidas e utilizadas são *Atelier-B* e *ProB*.

3.5.1 Atelier-B

Atelier-B é um software desenvolvido e mantido pela empresa ClearSy[1] e visa a criação de especificações para sistemas críticos. De acordo com o site da ClearSy, Atelier-B provê as seguintes funções:

- Provas: ajuda aos engenheiros a provar, demonstrando obrigações de prova utilizando as ferramentas adequadas.
- Desenvolvimento: auxilía no desenvolvimento de componentes, mostrando a dependência entre esses e analisa a sintaxe B do código.
- Ferramentas: provê diversos instrumentos para o engenheiro, tais como representação gráfica dos projetos, visão do estado do projeto, estatísticas e arquivamento.
- Automação: permite ao engenheiro automatizar a verificação de sintaxe, a criação de refinamentos e a tradução para as linguagens C, ADA e entre outras.

Atelier-B é uma plataforma utilizada para o desenvolvimento de especificações B, além de permitir ao usuário provar os componentes criados. A ferramenta gera as obrigações de prova dos componentes e concede ao usuário a possibilidade de realizar as provas de maneira automática ou manual, apresentando um provador iterativo.

3.5.1.1 Refinamento Automático

B utiliza uma ferramenta de refinamento automático (B Automatic Refinement Tool - BART). BART possibilita a geração de refinamentos e implementações a partir de uma base de regras de refinamento que pode ser expandida pelo usuário. Adicionalmente, as regras de refinamento podem ser personalizadas para certos componentes de forma a acelerar e personalizar o refinamento automático.

[1] http://www.ClearSy.com/

3.5.1.2 Analisador de Sintaxe

Um editor de modelo está integrado ao Atelier-B. Ele é responsável pela análise sintática, complementação automática e funções de navegação pelo modelo. Atelier-B também checa se a linguagem é B0 (uma sub-divisão da linguagem B) a fim de assegurar que as especificações podem ser traduzidas. Por fim, a ferramenta também apresenta uma verificação de projeto, que verifica todos os componentes de um projeto e controla a sua arquitetura (as relações entre os componentes).

3.5.1.3 Ferramentas de Prova

O Atelier-B conta com ferramentas que auxiliam na prova de um projeto, tais como:

- O gerador automático de obrigações de prova.
- Provador interativo: possibilita ao usuário guiar a prova do componente através de comandos iterativos (adição de lemas, provas por caso e outros).
- Provador de predicado: permite provar as regras de prova adicionadas pelo usuário.
- A observação de obrigações de prova, sua origem e seu estado (trivial, provado, não provado).
- Administração de mais 2200 regras base utilizadas para as provas.

A ferramenta apresenta duas versões, uma comercial que é paga e uma livre. Ambas estão disponíveis para Windows, Mac e Linux.

3.5.2 ProB

ProB [33] é um animador, solucionador de restrições e dá suporte na checagem do modelo [33]. Ele tem a capacidade de animar especificações B e solucionar condições para verificar erros, intertravamentos e geração de caso de testes. Ele pode ser acoplado ao Atelier-B através de um plugin.

ProB cobre grande parte da linguagem B, e está avançando para cobrir todos os construtores utilizados no Atelier-B. ProB suporta características B tais como operações não determinísticas, declarações do tipo *ANY*, operações com argumentos complexos, conjuntos, sequências, funções, abstrações lambda, gravações, constantes, propriedades e outros. Ainda não suporta algumas definições do Atelier-B sobre árvores, restrições e definições. ProB suporta múltiplas máquinas, refinamentos e implementações. Além disso, também pode ser utilizado para refinamento automático e checagens do modelo. Na ferramenta, os estados das especificações são graficamente visíveis, além de ser disponível através de uma calculadora lógica online.

ProB dá suporte às linguagens EventB, CSP-M, TLA+, e Z. Ele é grátis e está disponível para Windows, Mac, e Linux.

3.5.3 Tradutores B

Um outro tipo de plugin para o Atelier-B são os compiladores B. Estes compiladores servem como tradutores, compilando a implementação B em uma linguagem de programação.

Os tradutores B são importantes pois a partir da linguagem B não se pode utilizar o software, é necessário que o código desenvolvido seja traduzido para uma linguagem de programação. Essa etapa não é verificada pelo Atelier-B. Portanto, é dito que esse passo é fraco dentro da criação formal. Além disso, nem toda a linguagem B pode ser traduzida por todos os tradutores. Cada tradutor consegue traduzir uma parte do subconjunto B0, um subconjunto do B contendo somente as cláusulas concretas. É necessário que o usuário saiba o que o tradutor consegue traduzir, pois isto pode resultar em diversos erros.

No pequeno esquema a seguir está sintetizado alguns tradutores B. Todos apresentam documentação, no caso do B2LLVM ele foi desenvolvido por David Déharbe e Valério Medeiros e C4B e ADA têm manual documentado pela ClearSy.

- B2LLVM: traduz a linguagem B para LLVM (Low Level Virtual Machine). [14]
- C4B: traduz de B para C. [12]
- ADA: traduz de B para ADA. [12]

A tradução em B não gera um arquivo principal, sendo necessário que o usuário o adicione, só então será possvel compilar o código traduzido. Esta etapa não é verificada pelo Atelier-B.

CAPÍTULO 4

Testes de Software

Quando um produto é criado, testes são realizados a fim de averiguar a qualidade e funcionalidades desse. Estes são necessários para assegurar que o produto consiga alcançar o nível de qualidade e segurança desejados. Com o software não é diferente. Existem meios para avaliar se um software é seguro e bem escrito. B e outros métodos formais são exemplos de abordagens para a avaliação. Ao descrever um componente utilizando a linguagem B e provando a implementação, é garantido o bom funcionamento daquele componente. Porém, métodos formais não são suficientes para alguns certificados. Entre as técnicas utilizadas, o teste de software continua liderando como método mais utilizado pela indústria [37]. Esta seção apresenta o tema teste de software, definindo conceitos utilizados neste trabalho.

A principal ideia é conseguir com sucesso alcançar requisitos de teste (*Test Requirements* - TR). Os requisitos funcionais determinam como o software deve se comportar, isto permite inferir o nível de qualidade do software e facilita a detecção de falhas. Os TR são definidos a partir do critério de cobertura que o engenheiro deseja alcançar. Para um software completar um teste com sucesso podem ser necessários diversos cenários que produzam diferentes estados. Cada um desses cenários é um caso de teste e apresenta um resultado esperado após ser avaliado.

Uma vez que os casos de testes estão gerados, eles são executados. Terminada a execução os dados são coletados e o engenheiro de teste deve checar se os resultados coletados condizem com os esperados. Se os resultados são diferentes, o teste é considerado falho e algum erro aconteceu durante a execução do programa.

4.1 Terminologia Básica

Essa seção apresenta a terminologia utilizada em teste de software, como também a usada nesse documento.

- Defeito: uma implementação feita incorretamente. Um erro do código implementado.
- Erro: um estado incoerente do programa durante a execução. Estado diferente do esperado. Provável em ocorrência de defeito.
- Falha: o estado do erro mostrado para o usuário, isto é, o valor errado mostrado como um resultado.
- Teste: o ato de fazer um teste. Avaliar um sistema em teste.
- Depuração: o ato de seguir a execução passo-a-passo de um sistema a fim de observar o comportamento do programa.

- Critério de Cobertura: são diretrizes para a realização de um teste, o qual exercita determinada estrutura do código. Por exemplo, o critério de cobertura de arestas busca que todas as diferentes arestas do grafo dirigido do fluxo de execução de um código sejam alcançadas.
- Requisitos de Teste (TR): um objetivo do teste que deve ser completado a fim de que o teste seja considerado um sucesso. O TR depende do critério de cobertura que o engenheiro quer definir.
- Caso de Teste: um dos conjuntos de entrada e saída esperadas para realizar o teste.
- Resultados Esperados: as saídas esperadas que devem ser produzidas por um sistema em teste. Esses resultados implicam se o programa é bem formado e a execução não apresentou nenhum erro.

Os conceitos de defeito, erro e falha são similares. Geralmente, um defeito acaba gerando uma falha e um erro, mas não é mandatório. Casos de teste não devem ser confundidos com uma execução, pois um caso de teste é um cenário no qual o teste pode ser realizado mas não implica que este será realmente executado. Diferentes casos de teste podem exercitar a mesma coisa, por isso casos de testes que não acrescentam resultados podem ser descartados e não serão executados.

Os requisitos de teste são uma conjunção do ”o que” um teste deve cobrir e ”como” as especificações de testes devem ser cobertas.

4.2 Critérios de Cobertura

Um critério de cobertura é uma regra ou coleção de regras que impõem requerimentos de teste. Para alcançar tal cobertura, é necessário elaborar um conjunto de casos de teste. O critério descreve o requerimento de teste de maneira completa e inequívoca. [3]

Os critérios de cobertura podem ser divididos em dois grupos: as coberturas estruturais e as coberturas lógicas. No primeiro grupo estão coberturas nas quais o teste se baseia no controle de estruturas, definições de dados e referências [48]. No último grupo o foco são os operadores lógicos do programa. Abaixo está um pequeno sumário de algumas coberturas.

- Coberturas Estruturais:
  - Cobertura de Linhas (Statement Coverage).
  - Cobertura de Arestas (Branch Coverage).
  - Cobertura dos Caminhos (Path Coverage).
- Coberturas Lógicas:
  - Cobertura de Predicado (Predicate Coverage).
  - Cobertura de Cláusula (Clause Coverage).
  - Cobertura Combinada (Combinatorial Coverage).
  - Cobertura de Cláusula Ativa (Active Clause Coverage).
  - Modified Condition/Decisive Condition Coverage (MC/DC) ou Cobertura de Cláusula Ativa Correlacionada (Correlated Active Clause Coverage - CACC).

4.2.1 Coberturas Estruturais

Antes de definir coberturas estruturais, é preciso ter uma básica noção de grafos, pois o conceito de grafos dirigidos está na fundação destas coberturas, que por sua vez podem também ser chamadas de coberturas baseada em grafos [2].

Uma grafo de maneira formal é:

- um conjunto N de nós.
- um conjunto N_0 de nós iniciais, onde $N_0 \subseteq N$.
- um conjunto N_f de nós finais, onde $N_f \subseteq N$.
- um conjunto E de arestas, onde E é um subconjunto de $N \times N$

Para um grafo G ser útil para a geração de testes, é necessário que N, N_0 e N_f contenham pelo menos um nó cada.

De maneira geral, mais de um nó inicial pode ser apresentado, ou seja, N_0 é um conjunto. Ter vários nós iniciais é utilizado em alguns artefato de software, como por exemplo, uma classe que apresenta múltiplos pontos de entrada. No método B, isto não ocorre, portanto neste trabalho os grafos somente apresentam um nó inicial.

Arestas são consideradas elementos que partem de um nó até outro e são escritas como (n_i,n_j), dessa forma é demonstrado que a aresta conecta o nó n_i ao nó n_j. O nó inicial da aresta, n_i, pode ser chamado de predecessor e o nó final, n_j, de sucessor.

Apesar de utilizar o termo nó, este apresenta sinônimos que não modificam o significado, como por exemplo, o termo vértice. Sendo assim, um nó, ou um vértice, é uma estrutura que representa uma declaração ou um bloco de comandos. Similarmente, o termo aresta pode ser apresentado de outras formas, como por exemplo através do termo arco.

Um caminho de um grafo é uma sequência de nós [n_1,n_2,...,n_M], onde cada par adjacente de nós (n_i, $n_{(i+1)}$), $1 \leq i < M$, está no conjunto E de arestas. O tamanho de um caminho é definido a partir da quantidade de arestas que ele é composto. Caminhos podem apresentar tamanho zero quando não contiverem nenhuma aresta. Seguindo a notação de arestas, pode ser dito que um caminho flui do primeiro nó até o último nó.

Muitos critérios de cobertura exigem que o grafo deve começar em um nó e acabar em outro. Isto é somente possível se aqueles nós são conectados por um caminho. Quando esses critérios de cobertura são aplicados em grafos específicos, algumas vezes é possível encontrar caminhos que não podem ser executados. Por exemplo um grafo que apresenta uma tomada de decisão, em que esta decisão só pode ser tomada para um único valor booleano, então uma das arestas que saem do nó de decisão não pode ser alcançada, como também não se pode testar a tomada de decisão para outro valor. Quando isto ocorre é dito que aquela aresta não pode ser exercitada. Fatidicamente, se um grafo tem nós, ou arestas que não podem ser exercitadas ou tomadas de decisões inúteis nas quais a avaliação não é modificada, algumas coberturas também são inatingíveis.

Para exemplificar as coberturas estruturais, o fluxograma 4.1 será usado como exemplo principal. Ele apresenta um nó inicial, uma tomada de decisão, uma instrução e um nó final.

Definição 4.1 (Cobertura de Linhas)**.** *O requerimento de teste da cobertura de linhas é que todo nó do grafo do fluxo de controle do código seja executado pelo menos uma vez. [2]*

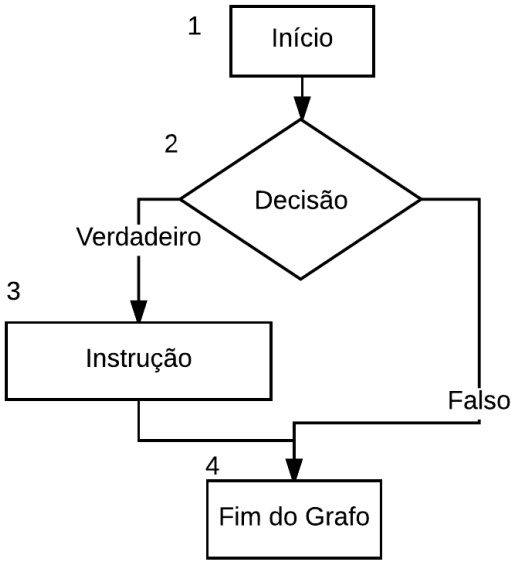


Figura 4.1 Fluxograma para exemplo.

Para o exemplo principal alcançar a cobertura de linhas, somente um caso de teste é necessário, pois esse consegue visitar todas as instruções do programa. O nó que contêm a decisão deve ser avaliado como verdadeiro e todos os nós são visitados.

A cobertura de linhas é a mais simples e fácil de ser alcançada dentre as apresentadas neste trabalho.

Definição 4.2 (Cobertura de Arestas)**.** *O requerimento de teste da cobertura de arestas é que toda aresta do grafo do fluxo de controle de controle do código seja executado pelo menos uma vez. [2]*

Para alcançar a cobertura de arestas no exemplo principal dois casos de teste são necessários. É necessário um caso de teste no qual a condição do nó 2 é avaliada como verdadeira, portanto visitando o galho que leva para a instrução (nó 3 do grafo), e um outro caso de teste que avalia a condição do nó 2 para falso, o que leva o fluxo de execução diretamente ao fim do grafo.

Além disso, se utilizarmos esses dois casos de teste a cobertura de linhas também é alcançada. A fim de comparar duas coberturas diferentes pode ser utilizado o conceito de incorporação. Uma cobertura $C1$ incorpora a cobertura $C2$ se, e somente se, todos os conjuntos de testes que satisfazem C1 também satisfazem C2. Por exemplo, a cobertura de arestas incorpora a cobertura de linhas, porém o contrário não é verdade. Em outras palavras, isto significa que ao alcançar a cobertura de arestas de um programa é também alcançada a cobertura de linhas.

Definição 4.3 (Cobertura de Caminhos)**.** *O requerimento de teste da cobertura de caminhos é que todo o caminho do fluxo do gráfico de controle do código seja executado pelo menos uma vez. [2]*

Figura 4.2 Relação de incorporação entre coberturas estruturais.

Cobertura de caminhos também demanda dois casos de teste para o grafo exemplo, assim como a cobertura de arestas, um dos casos passa pela decisão verdadeira e a instrução 1 e o outro a decisão é avaliada para a negação. Os casos de teste são os mesmos da cobertura de arestas, mas apenas porque o grafo apresenta somente uma decisão em sua estrutura, se um outro nó lógico ocorresse após o encontro dos caminhos e antes do término do grafo, os casos de testes seriam diferentes.

Um fato que ocorre com a cobertura de caminhos pode ser observado quando um grafo cíclico é o alvo do teste. Por apresentar um laço, cada vez que a condição do laço é satisfeita isso gera um novo caminho a ser testado. Dessa forma, existem repetições que podem gerar uma grande quantidade de caminhos a serem testados.

A cobertura de caminhos incorpora a cobertura de arestas e exige no mínimo a mesma quantidade de casos de teste. A figura 4.2 representa a incorporação das coberturas estruturais.

4.2.2 Coberturas Lógicas

O objetivo deste tipo de cobertura é testar o código com base nas condições de execução encontradas nas instruções condicionais e de repetição. Os requerimentos de teste dessas coberturas são relacionados com as tomadas de decisões dentro do programa e variam de acordo com a cobertura desejada.

Para utilizar a ideia de critério de teste para coberturas lógicas é preciso introduzir a definição de cláusulas e predicados.

Definição 4.4 (Cláusula)**.** *"Uma cláusula C é uma condição atômica ou a negação de uma condição atômica."[7]*

Definição 4.5 (Predicado)**.** *Predicado é uma composição de cláusulas conectadas a partir de operadores lógicos da linguagem de programação.*

Em outras palavras, uma cláusula é cada comparação dentro de um predicado e predicados é o conjunto de cláusulas. No predicado $P = ((A \vee B) \wedge C)$, é possível observar as cláusulas A, B e C, relacionadas a partir de operadores lógicos de disjunção e conjunção. Ao definir valores para as cláusulas é possível avaliar este predicado e assim verificar se ele apresenta uma afirmação verdadeira ou falsa.

Para exemplificar a definição de cada cobertura lógica o predicado $P = ((A \vee B) \wedge C)$ será utilizado como exemplo.

Definição 4.6 (Cobertura de Predicado (PC)). *O requerimento de teste da cobertura de predicados é que todo o predicado do conjunto de predicados a serem avaliados seja analisado uma vez para verdadeiro e uma vez para falso. [2]*

Em palavras simples, um predicado necessita ser avaliado ao menos uma vez para verdadeiro e outra para falso. PC é equivalente à cobertura de arestas, onde cada aresta do grafo do programa deve ser coberto.

Para o predicado exemplo, dois casos de teste são suficientes para satisfazer a cobertura: (A = Verdadeiro, B = Falso, C = Verdadeiro) e (A = Falso, B = Falso, C = Verdadeiro).

O PC falha em testar a influência de cada cláusula sobre o predicado. No parágrafo anterior, os valores das cláusulas B e C não mudam, e o predicado mesmo assim alcança a cobertura. Para solucionar esse problema, outros tipos de cobertura observaram as mudanças provocados por cada cláusula do predicado.

Definição 4.7 (Cobertura de Cláusula (CC)). *O requerimento de teste da cobertura de cláusula é que todo a cláusula do conjunto de cláusulas a serem avaliadas seja analisada uma vez para verdadeiro e uma vez para falso. [2]*

Isso significa que, para o predicado exemplo, outros valores são necessários para satisfazer a CC, porém dois casos de teste continuam suficientes: (A = Verdadeiro, B = Verdadeiro, C = Verdadeiro) e (A = Falso, B = Falso, C = Falso).

Tanto em CC quanto em PC não é possível de observar a influência da cláusula sobre o predicado. A maneira mais direta e robusta de observar o comportamento é testando todas as combinações.

Definição 4.8 (Cobertura Combinatorial (CoC)). *O requerimento de teste da cobertura combinatorial é que para todo o predicado do conjunto de predicados, as cláusulas presentes nesse predicado devem ser avaliadas de maneira que todas as combinações de resultados para as cláusulas do predicado sejam analisadas. [2]*

A partir do exemplo padrão, CoC pede por 8 casos de teste para ter sucesso, a tabela verdade 4.1 contêm todos eles:

Infelizmente, utilizar CoC é impraticável na maioria dos casos, para um predicado p com n cláusulas independentes, existem 2^n colunas na tabela verdade. A cobertura combinatorial é um exemplo de cobertura que captura o efeito da cláusula, mas demanda muitos casos de testes. Para solucionar esse problema, a ideia de cláusula 'ativa' é definida.

Definição 4.9 (Cláusula Ativa). *A cláusula ativa, ou cláusula majoritária, é a cláusula que determina o resultado do predicado, após ela ser escolhida, todas as outras são consideradas cláusulas minoritárias. As cláusulas minoritárias, têm valores tais que, ao modificar o valor da cláusula ativa, a avaliação do predicado também é modificado. [2]*

A definição não requer que a cláusula majoritária e o predicado tenham o mesmo valor. Uma vez definida a cláusula ativa, a cláusula majoritária e a minoritária, é possível definir a cobertura de cláusula ativa.

Tabela 4.1 Tabela Verdade para Cobertura Combinatorial.

	A	B	C	$((A \vee B) \wedge C)$
1	V	V	V	V
2	V	V	F	F
3	V	F	V	V
4	V	F	F	F
5	F	V	V	V
6	F	V	F	F
7	F	F	V	F
8	F	F	F	F

Definição 4.10 (Cobertura de Cláusula Ativa (ACC)). *O requerimento de teste para a cobertura de cláusula ativa é que para cada predicado no conjunto de predicados e a cada cláusula majoritária no conjunto de cláusulas do predicado, deve se escolher os valores das cláusulas minoritárias de forma que a cláusula ativa determina o valor do predicado. Dessa forma, para cada cláusula ativa o requerimento de teste necessita que essa cláusula seja avaliada para verdadeiro e outra para falso. [2]*

Para explicar o ACC é mais simples se for utilizado um predicado com duas cláusulas, o exemplo utilizado é $Q = A \vee B$. Escolhendo a cláusula A como determinante de Q, então A é a cláusula majoritária; A precisa ser avaliado para verdadeiro e em seguida para falso, de uma maneira que modifique o valor do predicado Q. Para conseguir isso B precisa ser sempre falso. Caso contrário B irá mascarar o valor de A. Portanto, o requerimento de teste pode ser solucionado em dois casos: $\{(A = verdadeiro, B = falso), (A = falso, B = falso)\}$. Todos os casos de teste para o exemplo estão na tabela verdade 4.2.

Tabela 4.2 Todos os casos de teste para $p = A \vee B$.

	A	B
$c_i = A$	**V**	f
	F	f
$c_i = B$	f	**V**
	f	**F**

Os valores em negrito são as cláusulas majoritárias. Observando a tabela verdade, dois casos de teste são iguais, esse sobreposto sempre ocorre no ACC. Então a quantidade de conjuntos para para n cláusulas independentes é $n+1$ no melhor caso.

Para o exemplo principal, a tabela verdade 4.3 contém o conjunto de testes que satisfaz o ACC. São apresentadas seis entradas capazes de satisfazer esta cobertura. Porém, alguns dos casos são iguais, fazendo com que o conjunto de seis se torne somente quatro casos efetivos.

A cobertura de cláusula ativa é similar com o MC/DC que foi descrito em seus artigos originais [3]. O problema é que a definição do MC/DC é dúbia e leva a confusões de interpretação. O cerne da questão é saber se as cláusulas minoritárias precisam ter o mesmo valor

Tabela 4.3 Conjunto de casos de teste que satisfazem ACC para o exemplo padrão.

	A	B	C	$((A \vee B) \wedge C)$
$c_i = A$	**V**	f	v	V
	F	f	v	F
$c_i = B$	f	**V**	v	V
	f	**F**	v	F
$c_i = C$	f	v	**V**	V
	v	f	**F**	F

quando a cláusula majoritária está sendo avaliada, essa questão encaminha para três diferentes interpretações do MC/DC.

Definição 4.11 (Cobertura de Cláusula Ativa Correlacionada (CACC))**.** *: O requerimento de teste para a cobertura de cláusula ativa é que para cada predicado no conjunto de predicados e a cada cláusula majoritária no conjunto de cláusulas do predicado, deve se escolher os valores das cláusulas minoritárias de forma que a cláusula ativa determina o valor do predicado. Dessa forma, para cada cláusula ativa o requerimento de teste necessita que essa cláusula seja avaliada para verdadeiro e outra para falso, porém é necessário que o valor do predicado seja diferente para quando a cláusula majoritária for verdadeira e para quando for falsa. [2]*

O CACC é a definição de MC/DC utilizada pela norma DO-178B. Utilizando o exemplo $p = ((A \vee B) \wedge C)$ e escolhendo A como cláusula majoritária, então A define o valor de p. Sendo assim, as cláusulas minoritárias B e C devem ocorrer ao menos uma vez falso e uma vez verdadeiro. Para satisfazer a CACC em relação a A dois casos de teste podem ser utilizados $\{(A = \mathit{verdadeiro}, B = \mathit{falso}, C = \mathit{verdadeiro}), (A = \mathit{falso}, B = \mathit{verdadeiro}, C = \mathit{falso})\}$. Essa descrição soluciona o CACC para A. Estes casos de teste estão representados na tabela 4.4.

Tabela 4.4 Tabela verdade para a cláusula majoritária A.

A	B	C	$((A \vee B) \wedge C)$
V	F	V	V
F	F	V	F

Para obter o CACC com a cláusula majoritária sendo A, é necessário escolher entre as linhas 3 e 7 correspondente à tabela verdade 4.1 que demonstra os casos de teste para a cobertura combinatorial. Para C ser a cláusula majoritária e determinar p, a tabela verdade 4.5 é mais extensa.

C como cláusula majoritária apresenta 6 possíveis colunas da tabela 4.1 a serem escolhidas, a combinação das colunas 1,3 ou 5, com as 2,4,6, geram 9 conjuntos soluções para alcançar o CACC em relação a C.

A relação de incorporação entre coberturas lógicas traz que o CACC não é a cobertura que incorpora todas as outras, e essa posição é ocupada pelo CoC. A figura 4.3 apresenta a relação de incorporação entre todas as coberturas lógicas deste trabalho.

A partir das coberturas lógicas apresentadas, o foco do BTestBox é o CACC, que é o MC/DC aceito pelos certificados DO-178C/ED-12C [3].

Tabela 4.5 Tabela verdade para a cláusula majoritária C.

	A	B	C	$((A \vee B) \wedge C)$
1	V	V	V	V
3	V	F	V	V
5	F	V	V	V
2	V	V	F	F
4	V	F	F	F
6	F	V	F	F

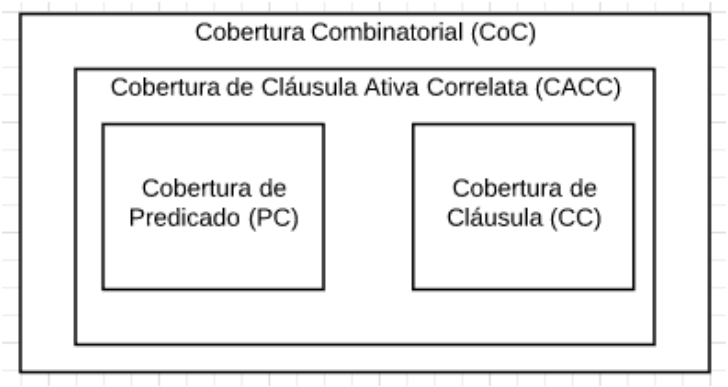


Figura 4.3 Relação de incorporação de coberturas lógicas.

CAPÍTULO 5

BTestBox

BTestBox é uma ferramenta para testar o código traduzido de implementações B. Como entradas ele demanda ao usuário qual o gerador de código e a implementação, além do critério de cobertura desejado e parâmetros necessários para a compilação. No estado atual, BTestBox suporta testes para as coberturas de: Cláusulas, Código, Predicado, Arestas e Caminhos, como também aceita toda a linguagem B0. Nessa seção, será apresentado um exemplo de execução da ferramenta, além de discutida e explicada a metodologia.

5.1 Metodologia do BTestBox

BTestBox realiza dois procedimentos. Primeiro, a ferramenta gera os casos de teste para verificar se a implementação B a ser testada pode atingir o critério de cobertura desejado. Segundo, BTestBox testa a tradução B com os testes calculados para a cobertura, aferindo se a tradução obtém os mesmos resultados dos casos de teste gerados. Para isso, BTestBox apresenta um fluxo de execução para testar uma implementação B:

1. O usuário escolhe uma implementação a ser coberta, o critério de cobertura que deseja atingir, o tradutor alvo.
2. O usuário deve escolhe as operações que serão cobertas. Se mais de uma for escolhida, os testes seguem a ordem de declaração das operações na implementação.
3. O grafo dirigido da operação em análise é gerado.
4. Todos os guias são calculados. Os guias são caminhos do grafo a partir do início da operação até o encerramento desta, calculados a partir do fluxo de controle da operação. A diferença de um guia para um caminho é que caso exista um laço dentro do grafo, ele é contabilizado somente com um ciclo.
5. BTestBox gera o predicado que determina as entradas. Este é dependente da cobertura escolhida. Por exemplo, para cobertura de arestas o foco são os guias com maior número de arestas não cobertas.
6. BTestBox chama o ProB para avaliar o predicado. Se obtiver uma resposta positiva, isso significa que o ProB achou um estado válido, os valores das variáveis nesse estado são utilizados como entrada em um caso de teste.
7. Um novo predicado é gerado a partir do mesmo guia, mas dessa vez busca as saídas da execução com as entradas obtidas no ponto anterior.
8. Repete a partir da instrução quatro até não existir mais objetivos.
9. Repete a partir da instrução três para a próxima operação da implementação.
10. Cria uma cópia das máquinas e implementações utilizadas para os testes em uma pasta

escolhida pelo usuário. Além disso, adiciona operações *get* e *set* nesses componentes. Os modelos B são copiados para que os arquivos do usuário não sejam modificados, além disso as operações adicionadas permitem a fácil manipulação e visualização das variáveis do modelo durante o teste.

11. Cria o conjunto de componentes de teste, máquinas e implementações que exercitam os casos de teste.
12. Traduz todas as máquinas e implementações para a linguagem escolhida.
13. Gera um componente na linguagem da tradução para compilação e execução do código.
14. Cria um HTML que sumariza todo o teste, com todos os componentes gerados e os originais, permitindo ao usuário facilmente interpretar o teste.

Os passos enumerados podem ser visualizados na figura 5.1.

5.2 Escolhendo o Critério de Cobertura

O critério de cobertura é escolhido dependendo do que o usuário deseja testar, se é a estrutura do programa ou as decisões lógicas. As diferentes instruções são detalhadas nas próximas subseções.

Para exemplificar e clarificar cada um dos 15 passos da metodologia do BTestBox, um exemplo de execução será aplicado à implementação encontrada na Figura 5.2. A implementação é simples, sem a utilização de um laço, este último é explicado e exemplificado separadamente na seção 5.6 por razão da sua complexidade e diferenciada solução.

No primeiro passo da execução, a cobertura e o tradutor do código são escolhidos, para o nosso exemplo é a cobertura de arestas e o tradutor C4B que traduz de B para C. Nem todos os tradutores pedem por um perfil de tradução, porém este demanda, e o perfil utilizado será o C9X. Escolhendo uma cobertura diferente, altera como o BTestBox atua. Estas diferenças serão explicadas mais adiante nesta seção.

BTestBox utiliza uma interface gráfica simples, a qual pode ser executada a partir do Atelier-B, observada na Figura 5.3.

Além da escolha do tradutor no campo “language”, é necessário escolher um nome para a pasta que conterá os arquivos do teste, como também o compilador e argumentos opcionais para o BTestBox. Os argumentos opcionais da ferramenta são as preferências utilizadas pelo solucionador de predicados.

5.2.1 Montando o Grafo Dirigido da Operação

O grafo é baseado na operação a ser coberta, o início do grafo é a declaração da operação e o nó final é a declaração do *END* que indica o término da operação. O grafo pode ser cíclico se apresentar uma estrutura *while*.

Enquanto realiza a construção do grafo, BTestBox também salva informações importantes que o ajudarão a calcular os guias. A ferramenta separa as diferentes instruções B0 do código em cinco tipos de nós. Cada tipo de nó está descrito a seguir:

- Substituição: estrutura de substituição, qualquer substituição em uma operação.

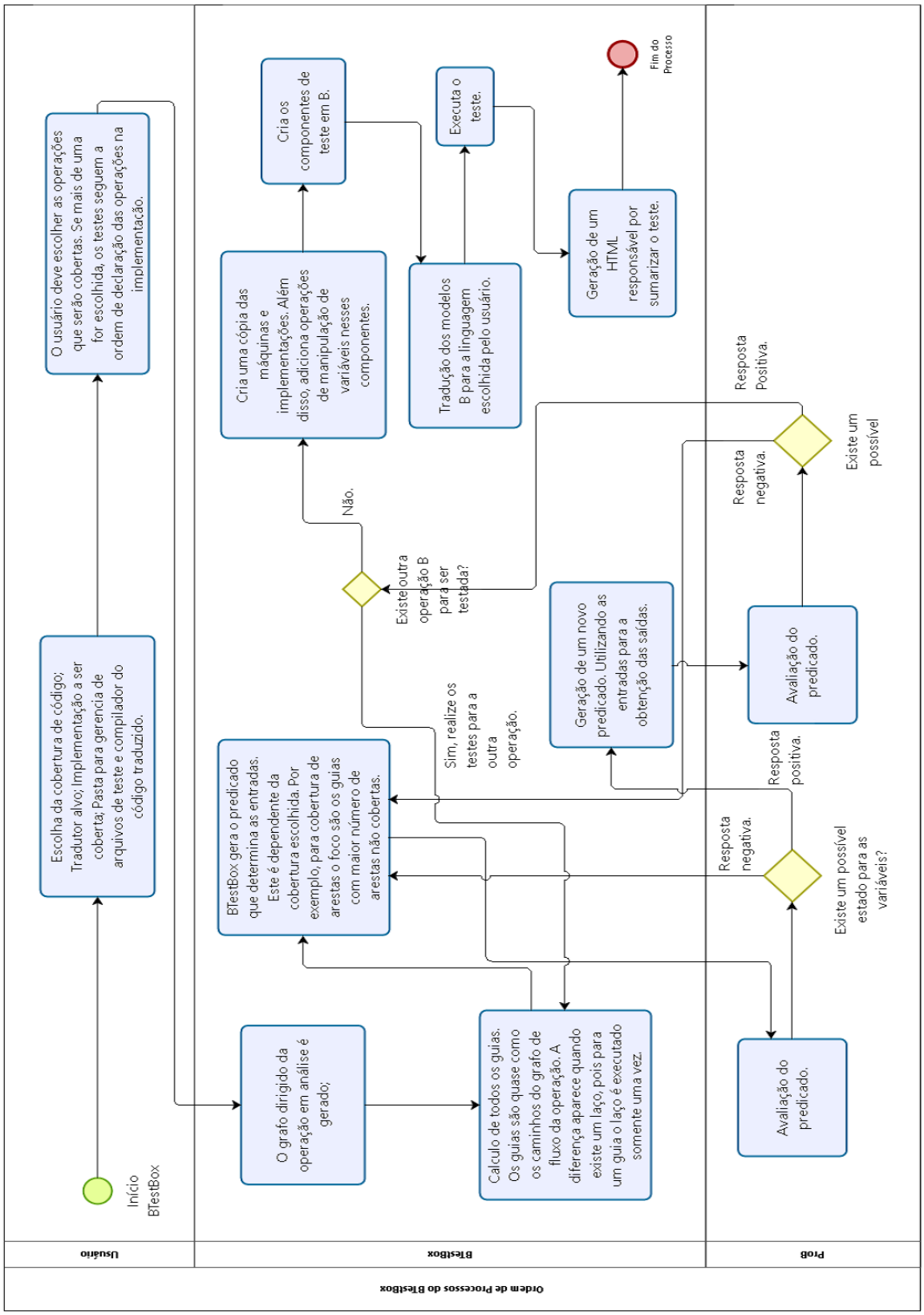


Figura 5.1 Metodologia BTestBox.

```
1  IMPLEMENTATION Example_i
2  REFINES Example
3
4  CONCRETE_VARIABLES
5      count
6
7  INVARIANT
8      count : NAT & count <= 10
9
10 INITIALISATION
11     count := 0
12
13 OPERATIONS
14     res <-- op1(xx, yy) =
15     BEGIN
16         VAR var1 IN
17             var1 := count * 2;
18             IF var1 < 20 THEN
19                 count := count + 1
20             ELSE
21                 count := count - 1
22             END;
23             res := count;
24             IF xx > 0 & yy > 0 THEN
25                 res := count + xx + yy
26             END
27         END
28     END
29 END
```

Figura 5.2 Implementação Exemplo.

- Condição: esse tipo de nó significa que uma decisão é tomada. É utilizada para as estruturas *IF* e *CASE* em uma operação. Os nós do início e do fim da operação são considerados deste tipo. O nó inicial contém toda informação sobre os condicionais para que a operação ocorra: precondição, invariante, propriedades, restrições e asserções de cada componente que é utilizado pela implementação na qual a operação é declarada. O nó final sempre contém uma comparação verdadeira (true = true).
- Chamada de Operação: quando uma chamada de operação é encontrada enquanto o grafo é construído, ela é armazenada em um nó deste tipo.
- Skip: a instrução *skip* é armazenada em um tipo separado.
- Condição While: nó do tipo condição que indica o início de um laço while.

BTestBox ainda armazena as condições necessárias para se alcançar o nó, isto é, se para tomar aquele caminho o nó anterior precisa tomar uma decisão falsa ou verdadeira. Esta informação é um componente importante para construção e manipulação do predicado. Usualmente, essa propriedade contém o valor verdadeiro e só é modificada caso o nó anterior seja do tipo condição.

Para o exemplo, neste ponto a operação “op1” foi marcada para o teste. Então, o grafo da Figura 5.4 é gerado.

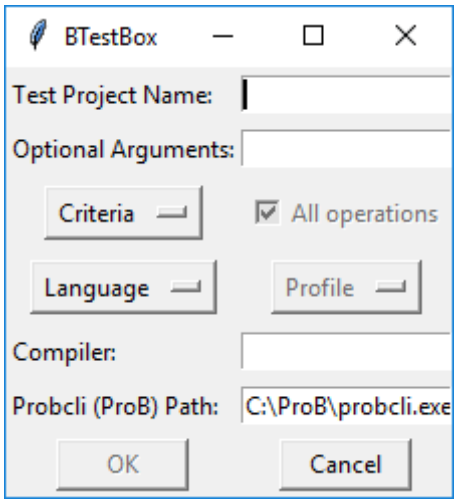


Figura 5.3 Interface BTestBox.

5.2.2 Calculando os Guias

Os guias não são todos os caminhos possíveis dentro de uma operação, pois em grafos cíclicos isto resultaria em infinitas possibilidades. Para gerar um guia, BTestBox utiliza o grafo de fluxo da operação. Cada nó apresenta uma propriedade que indica os nós conectados a ele, e com isso é realizada a criação dos guias. Caso o grafo seja cíclico, o guia somente contabiliza uma vez os nós dentro do laço.

Assim como o grafo, os guias também contém propriedades que assistem na geração do predicado. Os requerimentos de teste de um critério de cobertura determinam quais os objetivos para que o critério seja satisfeito. Sendo assim, BTestBox gera um mapa dos objetivos necessários para isso. Por exemplo, para a cobertura de linhas os nós são os objetivos, já as arestas são o alvo da cobertura de arestas, diferentemente da cobertura de caminhos que tem como mapa os caminhos, e para as coberturas lógicas são mapeadas as estruturas lógicas. Como os guias são as possíveis execuções da operação, para que um objetivo possa ser cumprido é preciso ser tomado um guia que permita que o objetivo seja alcançado. Estes objetivos são monitorados e quando todos são realizados a geração de predicados termina. Outro caso de parada acontece quando não existem mais guias possíveis a serem tomados para tentar cumprir o requerimento de teste.

BTestBox gera os guias abaixo para operação exemplo “op1”:

- Guia 1: 1, 2, 3, 4, 6, 7, 8, 9.
- Guia 2: 1, 2, 3, 5, 6, 7, 8, 9.
- Guia 3: 1, 2, 3, 4, 6, 7, 9.
- Guia 4: 1, 2, 3, 5, 6, 7, 9.

Estes quatro guias representam as possíveis tentativas que BTestBox fará para obter a cobertura da operação. A ferramenta tenta obter sucesso em alcançar todos os objetivos da cobertura escolhida. No caso exemplo, os objetivos são as arestas. Inicialmente não existe nenhum objetivo cumprido, por isso:

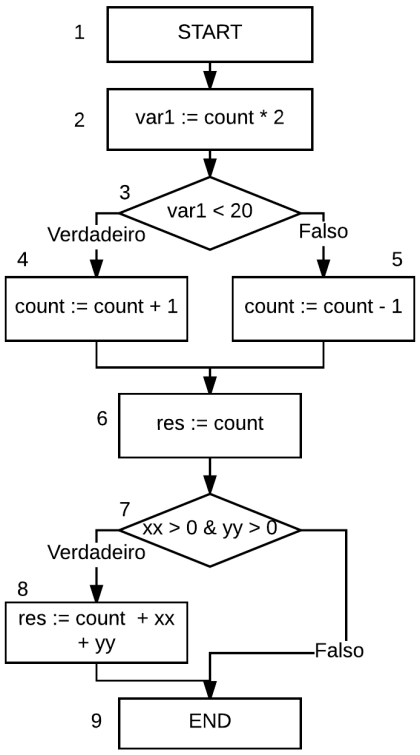


Figura 5.4 Grafo dirigido da operação “op1”.

Objetivos restantes = (1,2),(2,3),(3,4),(3,5),(4,6),(5,6),(6,7),(7,8),(7,9),(8,9)
Objetivos concluídos = ∅

Com a combinação dos objetivos e os guias gerados, BTestBox é capaz de analisar quais os guias podem cumprir um objetivo e somente realizar a geração de predicado para esses guias.

5.3 Geração dos Casos de Teste

A geração de casos de teste abrange do ponto 5 até o 12 da metodologia do BTestBox. Ela consiste na geração do predicado, recebimento das entradas e das saídas de cada caso de teste, e na criação das máquinas e implementações para a execução dos casos.

Esses passos são quase os mesmos para qualquer cobertura. Usualmente, primeiro é calculado o predicado para as entradas de um guia e depois pede-se para que o *ProB* avalie se existe algum estado no qual esse predicado pode ser satisfeito. Caso uma resposta positiva seja obtida, outro predicado é criado para definir as saídas e então avaliado. Esse processo ocorre para as coberturas de código, arestas, predicado e caminhos. Entretanto, o processo para a cobertura de cláusulas é diferente. Neste caso, inicialmente são calculadas todas as entradas possíveis através de vários predicados e em seguida é calculado o predicado para as saídas. Isto acontece pois a cobertura de cláusulas tem que avaliar todas as cláusulas de um predicado para

verdadeiro ou falso, ao se expandir para todos os predicados da operação isso leva a muitos casos de teste.

5.3.1 Calculando o Predicado Para um Guia

Uma vez que o guia para criar o predicado é escolhido, BTestBox começa a percorrer o grafo começando do fim até o início, manipulando o predicado e utilizando a lógica de Hoare, iniciando a partir do nó final do guia, com a condição verdadeira, que fornece à ferramenta o predicado inicial.

Os tipos de nós são decisivos no que BTestBox faz a seguir. Se uma substituição for apresentada, essa é aplicada ao predicado. Para nós do tipo condição, essa condição é adicionada ao predicado atual. Skip não faz nada.

Encontrando uma condição *while*, BTestBox transforma o predicado em duas cláusulas quantificadas.

- Uma quantificação para o predicado atual, adicionando a ele o invariante e a negação da condição da guarda do *while*.
- Uma quantificação para os nós dentro do laço, e para esse predicado é adicionado a guarda sem modificação e o invariante.

A quantificação ocorre sobre as variáveis que são modificadas dentro do laço. Isto permite calcular um predicado no qual o laço é exercitado. Por serem mais complexos, um exemplo de resolução de laço é dado separadamente a seguir na seção 5.6.

Para chamadas de operação é aplicada uma substituição complexa. Neste caso, a operação chamada é escrita dentro da operação que está sendo avaliada, e para cada decisão dentro da chamada uma implicação é gerada para o predicado. Este tipo é solucionado com o auxílio de funções da geração de obrigações prova do Atelier-B.

5.3.1.1 Calculando o predicado da operação “op1”

Voltando para o exemplo, o guia que tem o maior número de pontos não cobertos é escolhido, como os guias 1 e 2 obtiveram a mesma quantidade de pontos não cobertos, o guia com o menor índice é escolhido. Então o predicado será calculado para o guia 1.

A notação utilizada para calcular o predicado é a tripla de Hoare [24]. A tripla define como a execução de um comando modifica o estado das variáveis. Pode-se expressar a tripla de Hoare através da seguinte notação:

$$\{P\}C\{Q\}$$

Nesta fórmula, C é o comando, P é a pré-condição e Q a pós-condição. Para o caso do BTestBox, é sabido a pós-condição Q e o comando C, e se deseja obter a pré-condição P. Assim, é possível obter o estado das variáveis necessário para que a pré-condição seja estabelecida e a execução do comando ocorra, de maneira que, a pós-condição, quando o código for executado, será tal que o caminho tomado representa o apontado pelo guia.

A geração de predicado começa no fim do guia e percorre o caminho contrário até o nó inicial. Desse modo, para o primeiro guia, a geração de predicado começa no nó 9 e no fim alcança o nó 1. O nono nó contem a condição *TRUE = TRUE*, que sempre é verdadeira. Então, na primeira iteração o predicado $P1$ contem esses dados.

A segunda iteração alcança o oitavo nó, ele é do tipo substituição e BTestBox tenta inserir a instrução dentro do predicado. Porém, como pode ser observado, ele não modificará o predicado atual. No contexto do BTestBox em relação à tripla de Hoare, o nono nó é a pós-condição, a condição é a instrução presente dentro do nó atual (oitavo nó) e a pré-condição é o que será obtido quando terminado o cálculo da tripla. Desta forma *P2*, res := count + xx + yy e *P1* são respectivamente equivalentes à P, C e Q, da tripla de Hoare.

$$\{P\}\ C\ \{Q\}$$
$$\equiv \{P2\}\ res\ :=\ count\ +\ xx\ +\ yy\ \{P1\}$$
$$\equiv \{P2\}\ res\ :=\ count\ +\ xx\ +\ yy\ \{TRUE\ =\ TRUE\}$$
$$\equiv P2:\ TRUE\ =\ TRUE$$

Na terceira iteração, o sétimo nó é uma condição e vem a partir da avaliação positiva do nó, neste caso, nada é adicionado à condição. Se viesse do lado negativo, uma negação seria adicionada à condição do nó antes de ser adicionada ao predicado.

$$\{P3\}\ [xx\ >\ 0\ \&\ yy\ >\ 0\ ]\ \{P2\}$$
$$\equiv \{P3\}\ [xx\ >\ 0\ \&\ yy\ >\ 0\ ]\ \{TRUE\ =\ TRUE\}$$
$$\equiv P3:\ xx\ >\ 0\ \&\ yy\ >\ 0\ \&\ TRUE\ =\ TRUE$$

Desde que a informação *TRUE = TRUE* é trivial, o predicado $P3$ pode ser simplificado em: $P3:\ xx\ >\ 0\ \&\ yy\ >\ 0$.

Na quarta iteração, outra substituição não afetará o predicado, de maneira similar à anterior, alcançando o nó 6:

$$\{P4\}\ res\ :=\ count\{P3\}$$
$$\equiv \{P4\}\ res\ :=\ count\{xx\ >\ 0\ \&\ yy\ >\ 0\}$$
$$\equiv P4:\ xx\ >\ 0\ \&\ yy\ >\ 0$$

Na quinta iteração, temos outra substituição, percorrendo até o nó 4:

$$\{P5\}\ count\ :=\ count\ +\ 1\{P4\}$$
$$\equiv \{P5\}\ count\ :=\ count\ +\ 1\{xx\ >\ 0\ \&\ yy\ >\ 0\}$$
$$\equiv P5:\ xx\ >\ 0\ \&\ yy\ >\ 0$$

Na sexta iteração, temos mais uma outra condição a partir do lado positivo. O nó atual é o 3:

$$\{P6\}\ [var1\ <\ 20]\ \{P5\}$$
$$\equiv \{P6\}\ [var1\ <\ 20]\ \{xx\ >\ 0\ \&\ yy\ >\ 0\}$$
$$\equiv P6:\ xx\ >\ 0\ \&\ yy\ >\ 0\ \&\ var1\ <\ 20$$

Na próxima iteração, o estado do predicado é alterado novamente. Atravessando o segundo nó:

$$\{P7\}\ var1\ :=\ count\ *\ 2\ \{P6\}$$
$$\equiv \{P7\}\ var1\ :=\ count\ *\ 2\ \{xx\ >\ 0\ \&\ yy\ >\ 0\ \&\ var1\ <\ 20\}$$
$$\equiv P7:\ xx\ >\ 0\ \&\ yy\ >\ 0\ \&\ count\ *\ 2\ <\ 20$$

Na oitava iteração: a última iteração antes da pré-condição, invariantes, propriedades e qualquer outra informação sobre as condições que a operação necessita para a formação do predicado. Para o exemplo, a pré-condição da operação é $xx\ :\ NAT\ \&\ yy\ :\ NAT$ e o invariante da implementação: $count\ :\ NAT\ \&\ count\ \leq 10$.

$$\{P8\}\ [xx\ :\ NAT\ \&\ yy\ :\ NAT\ \&\ count\ :\ NAT\ \&\ count\ \leq\ 10]\ \{P7\}$$
$$\equiv \{P8\}[xx : NAT\ \&\ yy : NAT\ \&\ count : NAT\ \&\ count \leq 10]\{xx > 0\ \&\ yy > 0\ \&\ count * 2 < 20\}$$
$$\equiv P8:\ xx > 0\ \&\ yy > 0\ \&\ count * 2 < 20\ \&\ xx : NAT\ \&\ yy : NAT\ \&\ count : NAT\ \&\ count \leq 10$$

Finalmente, as iterações acabaram para esse guia, e o resultado final do predicado é:

$$P8:\ xx > 0\ \&\ yy > 0\ \&\ count * 2 < 20\ \&\ xx : NAT\ \&\ yy : NAT\ \&\ count : NAT\ \&\ count \leq 10$$

.

5.3.2 Avaliando o Predicado

Terminada a geração do predicado e todas as condições adicionadas, BTestBox pede para o *ProB* avaliar o predicado. Se o resultado da análise é positivo e existe um estado no qual é possível exercer os nós que geraram o predicado, *ProB* retorna os valores dos estados das variáveis. Caso contrário, o predicado e o guia são descartados e nenhum dos nós é marcado como coberto.

Alguns argumentos são utilizados para chamar, através da linha de comando, o *ProB*. Na primeira tentativa as preferências são editadas para desabilitar o tempo máximo de avaliação, para modificar o máximo e o mínimo de inteiro para o qual existe o calculo, para que o ProB expanda os valores dentro de declarações "para todos", indica que deve realizar a expansão simbólica. O usuário pode modificar os argumentos ao utilizar argumentos opcionais no BTestBox.

Para o predicado do exemplo o *ProB* retorna: $xx\ =\ 1\ \&\ yy\ =1\ \&\ count\ =\ 0$.

Quando um predicado retorna falso, a execução da avaliação do predicado gerado pelo guia acaba neste ponto e passa para um novo guia. Entretanto, para o exemplo foi retornado verdadeiro, neste caso, outro predicado é calculado a fim de determinar a saída para a execução

da operação com as entradas. Para este predicado é adicionado os valores das variáveis da máquina e da implementação.

Para obter as saídas o predicado é preparado antes da primeira iteração, então em vez de iniciar com *TRUE = TRUE* como dado do nó final, as saídas da operação são utilizadas, logo o predicado inicial é: $P1 : output\_\_res = res \,\&\, output\_\_count = count$.

Utilizando o mesmo guia, depois de todas as operações o predicado se torna:

$$P8 : output\_\_res = count + xx + yy \,\&\, output\_\_count = count + 1 \,\&$$

$$xx > 0 \,\&\, yy > 0 \,\&\, count * 2 < 20 \,\&\, xx : NAT \,\&\, yy : NAT \,\&\, count : NAT \,\&\, count \leq 10$$

Para avaliar um predicado de uma saída, as entradas que foram previamente encontradas são adicionadas ao predicado, temos então:

$$P : output\_\_res = count + xx + yy \,\&\, output\_\_count = count + 1 \,\&$$
$$xx > 0 \,\&\, yy > 0 \,\&\, count * 2 < 20 \,\&\, xx : NAT \,\&\, yy : NAT \,\&$$
$$count : NAT \,\&\, count \leq 10 \,\&\, xx = 1 \,\&\, yy = 1 \,\&\, count = 0$$

Quando esse predicado é avaliado por *ProB*, o retorno é $output\_\_res = 3 \,\&\, output\_\_count = 1$. Então o primeiro caso de teste está preparado com as entradas e saídas definidas. Logo, todas as arestas que o guia passa são marcadas como cobertas. Os objetivos resultantes podem ser observados abaixo:

Objetivos restantes = (3,5),(5,6),(8,9)
Objetivos concluídos = (1,2),(2,3),(3,4),(4,6),(6,7),(7,8),(7,9)

Quando o primeiro guia está todo percorrido, BTestBox escolhe o próximo guia com a maior quantidade de arestas não cobertas. Portanto, o guia 4 é escolhido e assim se o predicado gerado obtiver uma resposta positiva, os casos de teste estão completos. Realizando todo o processo iterativo, o predicado gerado para as entradas é:

$$P7 : not(xx > 0 \,\&\, yy > 0) \,\&\, not(count * 2 < 20) \,\&\, xx : NAT \,\&$$
$$yy : NAT \,\&\, count : NAT \,\&\, count \leq 10$$

Avaliando o predicado, são obtidas as seguintes entradas: $xx = 0 \,\&\, yy = 0 \,\&\, count = 10$. Depois de obter as entradas, BTestBox gera um novo predicado para as saídas e pede ao *ProB* para avaliá-lo. Este retorna os valores $output\_\_res = 9 \,\&\, output\_\_count = 9$. Em seguida, as arestas que o guia passa são marcadas. Com os casos de teste obtidos é possível cobrir a operação ”op1” e a cobertura deve ser satisfeita. Não é necessário avaliar mais nenhum guia uma vez que todos os objetivos foram alcançados. O conjunto de teste pode ser observado na tabela 5.1.

Tabela 5.1 Casos de teste para cobertura de arestas da operação ”op1”.

Casos de Teste	Entradas	Saídas
1	xx = 1 yy = 1 count = 0	res = 3 count = 1
2	xx = 0 yy = 0 count = 10	res = 9 count = 9

5.3.2.1 Limitações do ProB

ProB apresenta limitações no que ele pode avaliar [33] e essas são herdadas pelo BTestBox. No estado atual, BTestBox utiliza todos os termos que são condições sobre a implementação, durante a fase de preparação do predicado. Para o *ProB* avaliar um predicado, ele analisa os estados das variáveis em busca de um estado no qual o predicado é verdadeiro. Quando isso ocorre para termos um número significativo de estados o tempo de avaliação pode ser muito grande. Além disso, *ProB* não consegue trabalhar muito bem sobre funções recursivas. Outra limitação é a da chamada de operações. Quando ela ocorre, BTestBox utiliza funções disponíveis no Atelier-B que permitem trocar uma chamada de operação por um predicado. Assim, para solucionar a chamada de operação, o BTestBox usufrui dessa função e adiciona o predicado retornado pelo Atelier-B ao predicado gerado pelo próprio BTestBox. Com isso, é possível determinar o predicado de uma chamada de operações. Infelizmente, a API do Atelier-B não disponibiliza esta funcionalidade atualmente.

5.3.3 Gerando os Conjuntos de Teste

Com todas as possibilidades de entradas e suas correspondentes saídas já calculadas, o conjunto de teste é gerado seguindo os passos a seguir:

1. Os componentes do usuário são copiados, dessa forma os arquivos originais não são modificados mantendo o projeto intacto
2. As operações *get* e *set* necessárias são adicionadas. As modificações adicionam ao código operações que consigam manipular as variáveis, com isso é possível manipular o estado das variáveis para se adequar ao teste, além de possibilitar obter o valor após a execução.
3. Quatro componentes B são criados. Um conjunto de componentes para todos os casos de teste, no qual são geradas operações equivalentes aos casos de teste. E outro conjunto para executar e verificar o resultados de todos os casos de teste para cada operação.
4. Por fim, todos esses componentes são traduzidos e então BTestBox produz um arquivo principal na linguagem desejada, uma vez que o tradutor B não gera esse arquivo para a compilação.

A fim de não modificar os arquivos do usuário, uma pasta deve ser escolhida para receber a cópia dos arquivos e então as funções de manipulação são adicionadas. Quando uma máquina

```
GetcountForTest <-- OperationForTestGetcountExample =
    ANY aux WHERE aux : INTEGER THEN GetcountForTest := aux END;

SetVariablesForTestExample(nn1) =
PRE nn1 : NAT
THEN
    skip
END;
```

Figura 5.5 As operações *Set* e *Get* na cópia da máquina.

```
GetcountForTest <-- OperationForTestGetcountExample =
    GetcountForTest := count;

SetVariablesForTestExample(nn1) =
BEGIN
    count := nn1
END;
```

Figura 5.6 As operações *Set* e *Get* na cópia da implementação.

ou implementação tem uma variável, o teste precisa preparar o estado inicial e, ao executar a operação em análise, verificar o valor final.

Operações *set* assinalam valores para todas as variáveis da implementação de uma vez só, excluindo *arrays* que por serem mais complexos são feitos de modo separado. No nível abstrato, o corpo da operação é somente um *skip*. Unicamente em nível concreto, ou seja, na implementação, é possível atribuir os valores para as variáveis.

Operações *get* retornam o estado das variáveis, portanto cada variável apresenta uma operação correspondente. Para a abstração, o valor retornado pode ser qualquer um do tipo da variável. Entretanto, para a implementação somente é retornado o valor do estado atual, o que respeita as possíveis condições impostas sobre a variável.

As operações para variáveis do tipo *array* são singulares, sendo declaradas de forma diferente dos outros tipos de variáveis. As operações de um *array* recebem o valor do índice e o da substituição. É importante ressaltar que a linguagem B0 exclui os parâmetro de tipo arranjo. Para a forma abstrata, a operação *set* é um *skip* e a operação *get* recebe qualquer valor dentro do tipo do *array*. Na implementação, os valores do *set* são definidos e do *get* retorna o valor armazenado no índice escolhido.

Retornando ao exemplo, com toda a informação já calculada, BTestBox copia os arquivos do usuário, máquinas, implementações e qualquer outro componente utilizado pela implementação que está sendo avaliada. Em cada cópia são adicionadas as funções *get* e *set*. A Figura 5.5 apresenta as funções criadas na máquina. Para o *get*, somente é retornado o tipo da variável e o *set* é um *skip*. Na Figura 5.6 são as operações da implementação, onde o *set* é definido e o *get* retorna a variável. É importante ressaltar que o *skip* no corpo da operação *set* não representará um refinamento correto, a escolha do autor para utilizar o *skip* leva em consideração a facilidade de não precisar definir a operação ao utilizar essa abordagem.

5.3.3.1 Componentes do Conjunto de Teste

As máquinas de teste são as mais simples, para cada conjunto de entrada e saída uma operação é criada. Essas operações retornam um veredito, verdadeiro para quando as variáveis têm os valores esperados e falso caso contrário. A abstração dessas operações possui somente como retorno qualquer valor booleano.

Criar a implementação de um conjunto de teste exige mais passos. Cada operação declarada na máquina precisa ser desenvolvida na implementação. Primeiramente, o conjunto importa a máquina do componente que está sendo testado. Se a máquina receber qualquer parâmetro, este é inicializado com qualquer valor que satisfaça a sua condição.

A geração da implementação segue sempre os mesmos passos. Começa com a modificação dos estados das variáveis até o estado definido para o teste, utilizando das operações *set* criadas na máquina importada. Então, a operação que está em análise é executada com as entradas do caso de teste. Se a operação retorna qualquer valor, esse é salvo em uma variável específica para uma futura comparação com a saída esperada, assim também é feito com todas as variáveis de estado. Em seguida, os valores atuais são comparados com os esperados fornecidos pelo *ProB*; se todos são os mesmos, a operação retorna o veredito verdadeiro, senão retorna falso.

Quando se testa um vetor, as operações *get* e *set* são realizadas para todo os índices do vetor, checando o vetor por completo.

Uma vez que as variáveis do exemplo estão aptas a serem manipuladas, então o conjunto de teste, máquina e implementação são gerados baseados no caso de teste. A máquina é apenas um escopo para a implementação. As operações somente aceitam o retorno de qualquer booleano e nada mais é definido, como pode ser observado na figura 5.7.

No entanto, a implementação é bem definida e mais complexa, como observado na Figura 5.8. Esta importa a máquina sendo testada e qualquer outro componente que ela vê através da cláusula *SEES*, permitindo a mudança das variáveis. Adicionalmente, a operação é desenvolvida: primeiro é necessário utilizar a operação *set* criada para definir o estado da máquina antes do teste. No exemplo, somente precisamos de uma chamada de operação, pois nenhuma outra máquina é importada, vista ou estendida. Em seguida, a operação testada é utilizada e o estado das variáveis modificado. Então, essas variáveis são armazenadas e testadas com os valores esperados. Finalmente, se todos os valores forem iguais aos esperados, a operação retorna verdadeiro, informando que para aquele caso de teste tudo está correto, ou então retorna falso.

5.3.3.2 Componentes de Execução do Teste

Para executar os componentes declarados na subseção anterior, é gerada uma implementação e sua respectiva máquina, que executa todas as operações equivalentes aos casos de teste de uma operação da implementação em teste. A máquina é criada de maneira a retornar um booleano.

O modelo concreto importa o conjunto de teste, então ele é capaz de chamar cada caso de teste. Operações locais são criadas para que todos os casos de testes de uma operação sejam avaliados ao mesmo tempo. A implementação destas utiliza as operações e retorna verdadeiro se todas as chamadas também retornarem o mesmo valor.

```
MACHINE
    TestSet_BRANCH_Example

OPERATIONS
    verdict <-- TEST_0_op1 =
        ANY kk WHERE kk : BOOL THEN verdict := kk END;

    verdict <-- TEST_1_op1 =
        ANY kk WHERE kk : BOOL THEN verdict := kk END
END
```

Figura 5.7 Máquina preparada com os casos de teste.

```
IMPLEMENTATION
    TestSet_BRANCH_Example_i

REFINES
    TestSet_BRANCH_Example

IMPORTS
    Example

OPERATIONS
    verdict <-- TEST_0_op1 =
    BEGIN
        SetVariablesForTestExample(10);
        VAR aux1, aux2 IN
            aux1 <-- op1(1, 1);
            aux2 <-- OperationForTestGetcountExample;
            IF aux1 = 11 & aux2 = 9 THEN
                verdict := TRUE
            ELSE
                verdict := FALSE
            END
        END
    END;

    verdict <-- TEST_1_op1 =
    BEGIN
        SetVariablesForTestExample(0);
        VAR aux1, aux2 IN
            aux1 <-- op1(0, 0);
            aux2 <-- OperationForTestGetcountExample;
            IF aux1 = 1 & aux2 = 1 THEN
                verdict := TRUE
            ELSE
                verdict := FALSE
            END
        END
    END
END
```

Figura 5.8 Implementação preparada com os casos de teste.

```
MACHINE
    runTest_BRANCH_Example

OPERATIONS
    verdict <-- testAll =
        ANY kk WHERE kk : BOOL THEN verdict := kk END
END
```

Figura 5.9 Máquina de execução do teste.

Uma operação que testa todas as outras de uma vez então é criada. Essa obtém o retorno de todas as operações locais e checa o resultado da saída. Mais uma vez, se todas retornam verdadeiro então a operação retornará verdadeiro.

Para o exemplo, os componentes de execução de teste podem ser observados na Figura 5.9 para a máquina e na Figura 5.10 para a implementação. A máquina é criada com a operação *testAll*, assim como a parte abstrata do caso de testes, ela serve como escopo. A implementação dessa operação é responsável por chamar uma operação local, que contém a execução de todos os casos de teste para uma operação da implementação. No exemplo, apenas é testada a operação *op1*, logo, a implementação possui somente uma operação local. Para finalizar, a operação *testAll* contém os dois casos de teste.

5.3.3.3 Arquivo na linguagem traduzida

A tradução de B para qualquer linguagem gera definições, mas não a função principal fornecendo o ponto inicial da execução de um programa. Logo, é necessário que o BTestBox crie um arquivo na linguagem escolhida, para que seja possível executar o código traduzido. Esse arquivo é criado justamente para chamar a execução do teste e checar se o seu valor retorna verdadeiro, indicando que toda tradução está correta, ou falso, o que pode indicar falha durante a tradução. Além disso, um compilador é necessário para a criação do executável. Este é enviado como argumento pelo usuário.

Para o exemplo, a implementação de execução do teste é traduzida para a linguagem C, utilizando o tradutor C4B com o perfil de tradução C9X. BTestBox cria em C o arquivo *main* que não é gerado pelo tradutor. Este é apresentado na figura 5.11.

Primeiramente, é criada uma variável que armazena a saída da chamada da operação *testAll*. Para o teste da tradução são esperadas três saídas:

1. O sucesso da tradução e o alcance da cobertura escolhida, sendo apresentada uma mensagem de sucesso após a tradução. Isto indica que os valores esperados pelo BTestBox e os valores obtidos ao executar o código na linguagem traduzida estão iguais.
2. O sucesso da tradução e a falha em alcançar a cobertura escolhida, resultando em uma mensagem de sucesso da tradução mas que não foi possível alcançar a cobertura. Isso indica que os valores esperados pelo BTestBox e os valores obtidos ao executar o código, na linguagem traduzida, estão iguais, porém não é possível alcançar a cobertura desejada. Isto ocorre quando o BTestBox encontra testes para uma cobertura, mas eles não são

```
IMPLEMENTATION
    runTest_BRANCH_Example_i

REFINES
    runTest_BRANCH_Example

IMPORTS
    TestSet_BRANCH_Example

LOCAL_OPERATIONS
    verdict <-- testop1 =
        ANY kk WHERE kk : BOOL THEN verdict := kk END

OPERATIONS
    verdict <-- testop1 =
    BEGIN
        VAR v0, v1 IN
            v0 <-- TEST_0_op1;
            v1 <-- TEST_1_op1;
            IF v0 = TRUE & v1 = TRUE THEN
                verdict := TRUE
            ELSE
                verdict := FALSE
            END
        END
    END;

    verdict <-- testAll =
    BEGIN
        VAR v0 IN
            v0 <-- testop1;
            IF v0 = TRUE THEN
                verdict := TRUE
            ELSE
                verdict := FALSE
            END
        END
    END
END
```

Figura 5.10 Implementação de execução do teste.

```
#include <stdio.h>
#include <stdlib.h>

#include "runTest_BRANCH_Example.h"

int main(int argc, char **argv)
{
    bool result;
    runTest_BRANCH_Example__testAll(&result);
    if (result == true){
        printf("The translation of the implementation Example_i is well performed and achievedBranch Coverage");
    }
    else{
        printf("The translation of the implementation Example_i is NOT well performed");
    }
    return 0;
}
```

Figura 5.11 Arquivo ‘main’ criado.

suficientes para que a cobertura seja alcançada.
3. A falha na tradução. Os valores obtidos e os esperados são diferentes, não é afirmado nenhuma informação sobre a cobertura do código.

Na figura 5.11 é possível observar que o "se" do código permite a execução de dois casos, o caso no qual a tradução falhou e o caso no qual a tradução e a cobertura foram um sucesso. Isso acontece porque BTestBox já previamente define se a cobertura é obtida.

Para a compilação do código, GCC é o compilador escolhido para o exemplo. Executando o executável criado a partir da compilação é obtida a mensagem de sucesso, respondendo que a tradução é bem realizada e que a cobertura de arestas do código é alcançada.

5.3.3.4 Limitações sobre a Tradução

Como já comentado na seção 3, BTestBox utiliza de tradutores B, se estes apresentam qualquer limitação isto também se aplica sobre o BTestBox, portanto o usuário precisa saber previamente se é possível traduzir o código a partir do tradutor escolhido.

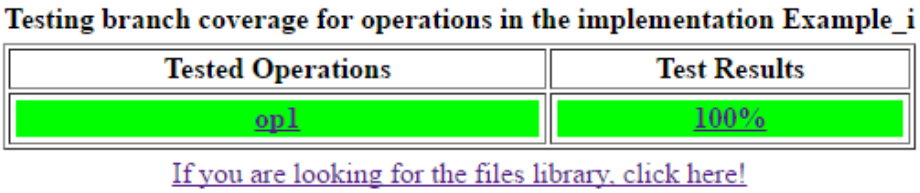
5.4 Avaliação e Relatório

Enquanto ocorre a criação dos guias os objetivos são mapeados para falso, isto significa que eles não foram cobertos. Quando, na avaliação de um predicado é encontrado um estado possível para um caso de teste, todos os nós dentro daquele guia, são avaliados para verdadeiro. Uma vez que todos os guias, ou todos os objetivos, têm seu predicado produzido e avaliado, a porcentagem de cobertura é calculada baseando-se na quantidade de objetivos não cobertos. Isso é indicado de maneira simples ao gerar o relatório para o usuário.

O relatório só é gerado após a execução, sendo um arquivo HTML que contêm um sumário de todo o teste. O relatório apresenta todos os arquivos criados a partir de uma biblioteca, contendo os casos de teste com suas entradas e saídas, além da percentagem da cobertura. Quando não é possível alcançar a cobertura de uma operação, o relatório a apresenta em vermelho e marca os locais que não foi possível cobrir. Quando a cobertura for satisfeita, aparece escrito em verde. A demarcação facilita ao usuário saber o que aconteceu e onde o código não atinge a cobertura esperada.

Para finalizar a exemplificação é criado o relatório. Na primeira página, demonstrada na figura 5.12, é apresentado o sumário, onde o usuário pode observar quais operações foram testadas, a porcentagem de cobertura e o critério escolhido.

Clicar no nome da operação leva para a implementação do usuário. O caminho na porcentagem mostra um sumário avançado do teste, no qual é possível observar todos os valores utilizados na análise da implementação, as entradas e saídas esperadas para cada caso de teste e estado final do objetivo, falha ou sucesso. Para o exemplo, a janela que aparece ao clicar na porcentagem leva para a página apresentada na figura 5.13, o texto em verde indica que a implementação está totalmente coberta, se qualquer problema tivesse sido encontrado a operação apareceria em vermelho. Nenhuma marcação ou alteração é realizada dentro do código do usuário, e se ele quer detalhes do teste, como quais linhas de código não conseguiram ser executadas, ele deve procurar nessa página do relatório.

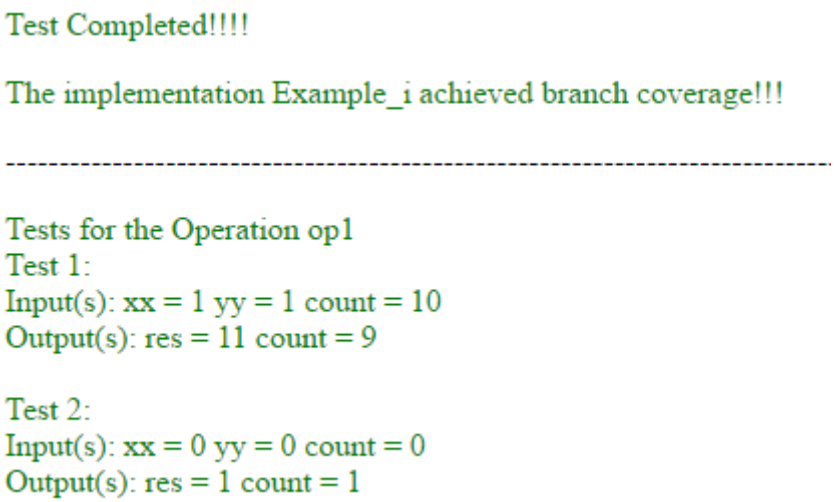


Testing branch coverage for operations in the implementation Example_i

Tested Operations	Test Results
op1	100%

If you are looking for the files library, click here!

Figura 5.12 Primeira página do relatório.



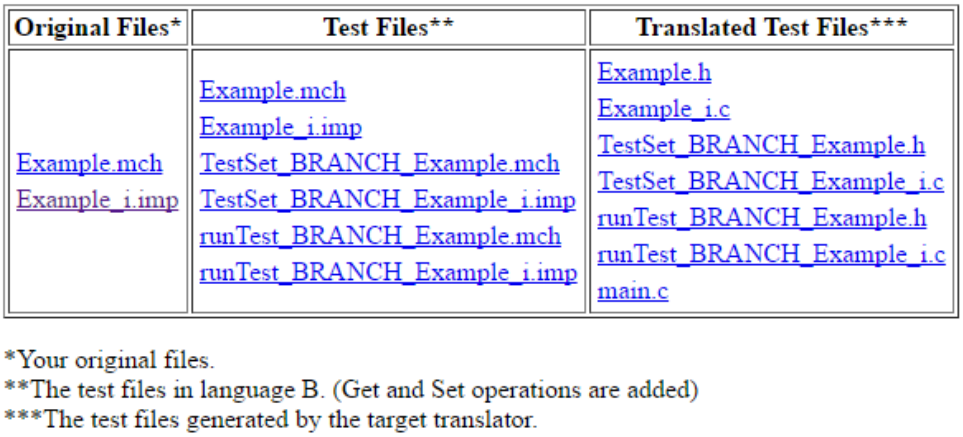
Test Completed!!!!

The implementation Example_i achieved branch coverage!!!

Tests for the Operation op1
Test 1:
Input(s): xx = 1 yy = 1 count = 10
Output(s): res = 11 count = 9

Test 2:
Input(s): xx = 0 yy = 0 count = 0
Output(s): res = 1 count = 1

Figura 5.13 Relatório para todas as entradas e saídas.



Original Files*	Test Files**	Translated Test Files***
Example.mch Example_i.imp	Example.mch Example_i.imp TestSet_BRANCH_Example.mch TestSet_BRANCH_Example_i.imp runTest_BRANCH_Example.mch runTest_BRANCH_Example_i.imp	Example.h Example_i.c TestSet_BRANCH_Example.h TestSet_BRANCH_Example_i.c runTest_BRANCH_Example.h runTest_BRANCH_Example_i.c main.c

*Your original files.
**The test files in language B. (Get and Set operations are added)
***The test files generated by the target translator.

Figura 5.14 Biblioteca de todos os arquivos utilizados no teste.

Além disso, no fim da primeira página é possível clicar em um link que segue para a biblioteca com todos os arquivos, disponível na imagem 5.14. Nessa tela são apresentados os arquivos originais, traduzidos e os utilizados no teste. A primeira coluna mostra os arquivos originais e eles tem um link que mostra o código do usuário. A segunda coluna disponibiliza os arquivos copiados e os criados para o teste. A terceira coluna contêm os arquivos traduzidos.

Isso conclui o teste, o usuário portanto obtém os detalhes do teste. Apesar de utilizar os arquivos importados, o teste é único da implementação escolhida, sendo necessário uma nova análise para outra implementação.

5.5 Exemplo de operação não coberta

Para exemplificar o procedimento realizado por BTestBox quando uma cobertura não pode ser alcançada, a seguinte máquina, figura 5.15, e a sua implementação, figura 5.16, serão utilizadas. Os componentes não descrevem nenhum modelo em particular, somente apresentam duas operações que modificam o valor da variável *pos_x*.

Se o BTestBox for executado para a implementação *Arm_i*, ele fará a análise das duas operações presentes no componente, porém, no intuito de manter a exemplificação somente na falha da cobertura, o BTestBox só fará a verificação da operação *wrongMove*. Começando a execução do BTestBox, é escolhida a cobertura que será verificada. Para demonstrar outra análise de cobertura, desta vez será escolhida a cobertura de linhas. Então, como próximo passo, é gerado o grafo da operação, disponível na figura 5.17.

Seguindo o fluxo da metodologia do BTestBox, são calculados os guias utilizados durante a operação e os objetivos necessários a serem atingidos pelo critério de cobertura, no caso são os nós:

- Guia 1: 1,2,3,4.
- Guia 2: 1,2,4.

Objetivos restantes = (1),(2),(3),(4)

```
MACHINE
  Arm
INCLUDES
  Motor
ABSTRACT_VARIABLES
  pos_x
INVARIANT
  pos_x ∈ 0 .. 100
INITIALISATION
  pos_x := 0
OPERATIONS
  randomMove =
    ANY xx WHERE
      xx : 0..100
    THEN
      pos_x := xx
    END

  wrongMove =
    IF pos_x < 0 THEN
      pos_x := 0
    END
END
```

Figura 5.15 Modelo B abstrato (máquina).

$$\text{Objetivos concluídos} = \emptyset$$

Como pode ser observado, o guia 1 remete a escolha positiva na decisão do nó 2, e o guia 2 se refere a escolha negativa na decisão do mesmo nó. Como o guia 1 tem um maior número de nós, este é analisado primeiro. Na primeira iteração temos o predicado:

$$P1 : TRUE = TRUE$$

Na segunda iteração, o predicado não é modificado:

$$P2 : TRUE = TRUE$$

Em seguida, a condição da decisão é adicionada ao predicado, portanto:

$$P3 : pos\_x < 0$$

Para finalizar, o invariante é adicionado ao predicado:

$$P4 : pos\_x < 0 \ \& \ pos\_x : 0..100$$

```
IMPLEMENTATION Arm_i
REFINES Arm
IMPORTS Motor
CONCRETE_VARIABLES
    pos_x
INVARIANT
    pos_x ∈ 0..100
INITIALISATION
    pos_x := 0
OPERATIONS
  randomMove =
    IF pos_x < 100 THEN
      pos_x := pos_x + 1
    ELSE
      pos_x := 0
    END

  wrongMove =
    IF pos_x < 0 THEN
       pos_x := 0
    END
 END
```

Figura 5.16 Modelo B concreto (implementação).

Esse predicado não tem solução, pois não é possível que a variável *pos_x* seja menor que zero.

Mesmo com a negativa e a observação de que o guia 1 não pode ser executado, BTestBox continua a verificação. Para isso, a ferramenta analisa o guia 2. No fim da iteração esse guia retorna o predicado:

$$P : \neg(pos\_x < 0)\ \&\ pos\_x : 0..100$$

Analisando o predicado, *proB* retorna 0 como possível valor para a variável *pos_x*. Em seguida, obtém a saída para essa entrada, resultando em $pos\_x = 0$. Então, o conjunto dos objetivos termina como:

Objetivos restantes = (3)
Objetivos concluídos = (1),(2),(4)

Continuando o fluxo da metodologia, BTestBox cria os arquivos de teste com o caso de teste que ele obteve, depois traduz todos os arquivos e gera um relatório de todo o teste. Como é possível observar na figura 5.18, a partir da primeira página do relatório, o usuário já descobre

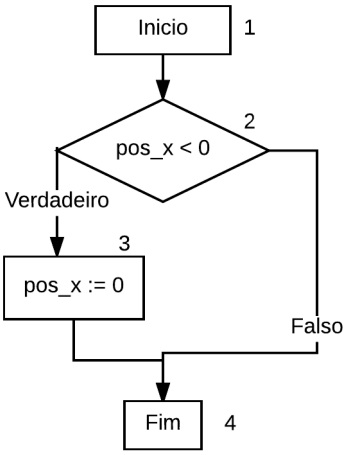


Figura 5.17 Grafo da operação *wrongMove*.

o que aconteceu no teste da implementação. Neste caso a operação *wrongMove* só é coberta em 75% em relação à cobertura de linhas.

Testing Code Coverage for operations in the implementation Arm_i

Tested Operations	Test Results
randomMove	100%
wrongMove	75%

If you are looking for the files library, click here!

Figura 5.18 Primeira página do relatório da implementação Arm_i.

Na segunda página do relatório, disponível na figura 5.19, o usuário consegue observar qual linha do código não foi executada.

BTestBox conclui o teste da implementação *Arm_i* informando ao usuário que apesar de poder ser traduzida, ela não alcança a cobertura de linhas, pois falha ao cobrir a operação *wrongMove*.

5.6 Exemplo de solução de laço

Dada a operação "RussianMultiplication" presente na figura 5.20 e com o intuito de produzir um exemplificação de geração de predicado para um laço, a cobertura escolhida será a simples cobertura de linhas.

Para a operação, é gerado o grafo apresentado na figura 5.21. A partir do grafo os seguintes guias são calculados:

Test Failed!!!!

Tests for the Operation randomMove
Test 1:
Input(s): pos_x = 100
Output(s): pos_x = 0

Test 2:
Input(s): pos_x = 0
Output(s): pos_x = 1

Tests for the Operation wrongMove
Test 1:
Input(s): pos_x = 0
Output(s): pos_x = 0

Test operation wrongMove did not achieved Code Coverage

BTestBox was unable to reach the instruction node:
pos_x := 0

Figura 5.19 Segunda página do relatório da implementação Arm_i.

- Guia 1: 1, 2, 3, 4, 5, 6, 7, 8, 9, 5, 10.
- Guia 2: 1, 2, 3, 4, 5, 6, 8, 9, 5, 10.
- Guia 3: 1, 2, 3, 4, 5, 10.

Uma vez que o critério de cobertura escolhido é o de linhas, todos os nós do grafo devem ser visitados pelo menos uma vez. Então, inicialmente os conjuntos de objetivos são:

Objetivos restantes = (1),(2),(3),(4),(5),(6),(7),(8),(9),(10)
Objetivos concluídos = $\emptyset$

Continuando com o fluxo de execução do BTestBox, o programa calcula o predicado para o guia com o maior número de objetivos não concluídos, portanto é realizada a geração de predicado para o primeiro guia.

Começando do décimo nó, a condição é $P : TRUE = TRUE$

```
IMPLEMENTATION RussianMult_i

REFINES RussianMult

CONCRETE_VARIABLES
    xx, yy, total
INVARIANT
    xx ∈ ℕ ∧ yy ∈ ℕ ∧ total ∈ ℕ
INITIALISATION
    xx, yy, total := 0, 0, 0

OPERATIONS
  RussianMultiplication(aa,bb) =
    xx := aa;
    yy := bb;
    total := 0;
    WHILE xx > 0
    DO
      IF xx mod 2 = 1 THEN
        total := total + yy
      END;
      xx := xx / 2;
      yy := yy * 2
    INVARIANT xx ∈ ℕ ∧ total + xx * yy = aa * bb
    VARIANT xx
    END
  END
END
```

Figura 5.20 Implementação da multiplicação russa.

Seguindo para o nó 5, a condição do nó é adicionada ao predicado, porém como alcança o nó a partir da falsidade da condição presente no nó, o conteúdo do nó é adicionado e então negado, de forma que:

$$P : \neg(xx > 0)$$

Ao alcançar o nó 9, é também enc contrado o laço, com isso, BTestBox divide o predicado em dois, uma condição para dentro do laço e outra para fora. Estas são calculadas isoladas e então são unidas ao término do cálculo do laço. O predicado que estava fora do laço é P2, para diferenciar do predicado durante o cálculo ele será chamado de P_e, já o predicado da parte interna será P_i. Inicialmente, o predicado P_i não apresenta condições, e segura $TRUE = TRUE$. As condições da parte interna do laço são calculadas separadamente e adicionadas a P_i, a parte interna é calculada até alcançar novamente o nó 5. O conteúdo presente nos nós 9, 8, 7 não

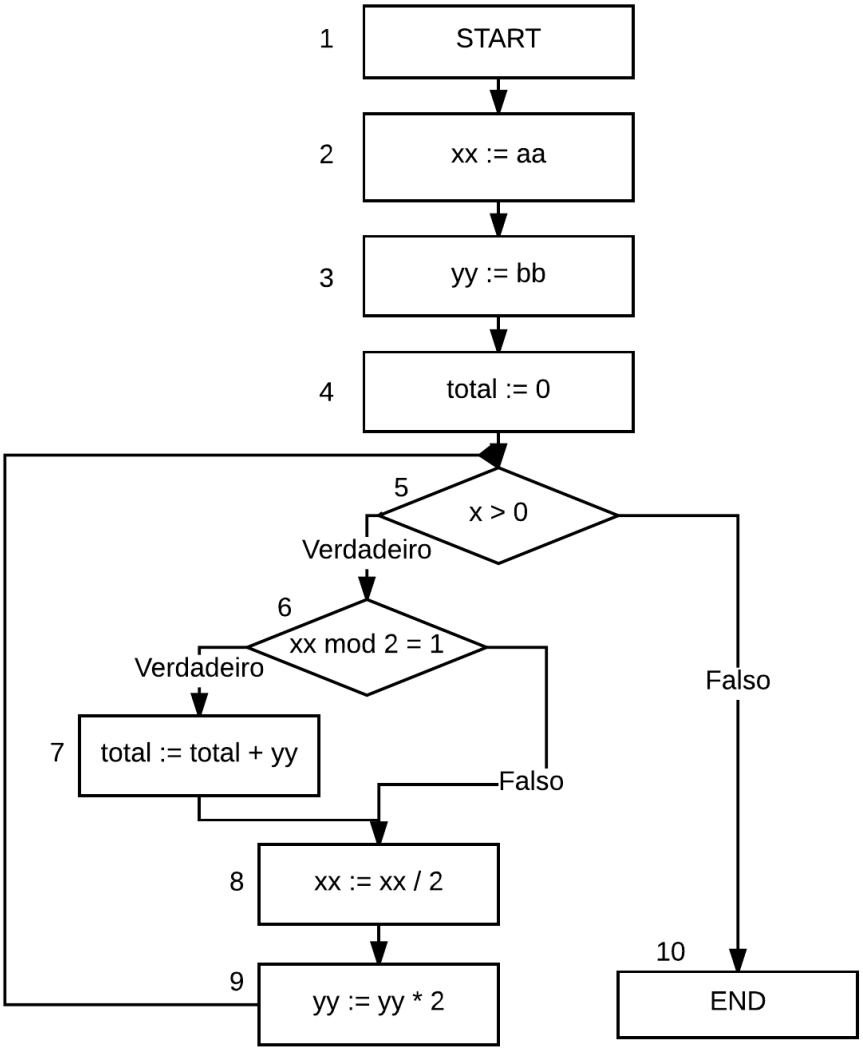


Figura 5.21 Grafo da operação “RussianMultiplication”.

modifica o predicado interno, porém ao alcançar o nó 6 este é modificado em:

$$P_i : xx \; mod \; 2 = 1$$

Chegando ao nó 5 o laço é terminado e então é adicionado ao predicado interno a condição de entrada no laço e o invariante do laço. Para o predicado externo do laço é somente adicionado o invariante do laço, o que resulta em:

$$P_i : xx \; mod \; 2 = 1 \wedge xx \in \mathbb{N} \wedge total + xx * yy = aa * bb$$

$$P_e : \neg(xx > 0) \wedge xx \in \mathbb{N} \wedge total + xx * yy = aa * bb$$

Ainda não terminado, uma quantificação sobre esses predicados é realizada, de maneira

que os valores alcançados dentro do laço não sejam determinantes para as condições presentes antes do laço ser executado no fluxo normal da operação. Essa quantificação é feita somente nos valores que foram modificados dentro do laço. Logo:

$$P_i : \#(xx, yy, total).(xx > 0 \wedge xx \; mod \; 2 = 1 \wedge xx \in \mathbb{N} \wedge total + xx * yy = aa * bb)$$

$$P_e : \#(xx, yy, total).(\neg(xx > 0) \wedge xx \in \mathbb{N} \wedge total + xx * yy = aa * bb)$$

A execução continua após a conjunção desses dois predicados, resultando em:

$$P : \#(xx, yy, total).(xx > 0 \wedge xx \; mod \; 2 = 1 \wedge xx \in \mathbb{N} \wedge total + xx * yy = aa * bb) \wedge$$

$$\#(xx, yy, total).(\neg(xx > 0) \wedge xx \in \mathbb{N} \wedge total + xx * yy = aa * bb)$$

O que está contido dentro da quantificação não é modificado por instruções que venham antes do laço no grafo de fluxo de controle da operação. Dessa forma, ao continuar a execução, como nenhuma outra condição é encontrada somente é adicionado ao predicado o invariante presente na máquina e a precondição da operação. Por fim:

$$P : \#(xx, yy, total).(xx > 0 \wedge xx \; mod \; 2 = 1 \wedge xx \in \mathbb{N} \wedge total + xx * yy = aa * bb) \wedge$$

$$\#(xx, yy, total).(\neg(xx > 0) \wedge xx \in \mathbb{N} \wedge total + xx * yy = aa * bb) \wedge$$

$$xx : NAT \wedge yy : NAT \wedge total : NAT \wedge aa : NAT \wedge bb : NAT$$

Ao avaliar o predicado com a ferramenta *ProB* este retorna: xx = 1, yy = 0, total = 0. Com isso, é obtido valores que satisfazem o critério de cobertura e passam pelo guia um. Dessa forma, todos os objetivos são concluídos e nenhum outro guia precisa ser analisado. Sendo assim, com o exemplo citado acima é possível observar como BTestBox realiza o cálculo de predicado para laços.

5.7 Validação

BTestBox foi testado em diversos componentes com diferentes cláusulas, com o objetivo de testar a sua funcionalidade e corretude. Mais de 120 implementações provadas foram utilizadas apresentando a linguagem B, além de uma implementação confidencial disponibilizada pela empresa *ClearSy* na qual simulava um programa real B.

Os componentes podem ser divididos em seis grupos, expostos na tabela 5.2:

Os grupos são escolhidos conforme é escrito o programa e o que está sendo testado da linguagem.

Tabela 5.2 Grupos de componentes da linguagem B.

	Quantidade de Implementações
Cláusulas	14
Chamada de Operações	26
Blocos Simples	6
Blocos Duplos	25
Blocos Triplos	58

No grupo de cláusulas são testados as estruturas B: SEES, EXTENDS, CONSTANTS, CONSTRAINTS, IMPORTS, PROMOTES, SETS, VARIABLES e LOCAL OPERATIONS. Os elementos são testados em conjunto e separadamente de forma a observar a mudança do predicado de acordo com cada um deles.

Chamada de operações, como o próprio nome indica, testa a chamada de operação em diferentes contextos que modificam o predicado. Por exemplo, um componente testa uma chamada de operação que ocorre depois de um laço while e outro uma chamada que ocorre antes.

Blocos simples são operações com as instruções: IF-ELSE, CASE, SKIP, WHILE e ASSIGNMENT. Estes são os blocos básicos, sem aninhamento ou sequenciamento.

Blocos duplos são operações com as mesmas instruções do bloco simples, mas com aninhamento ou sequenciamento de no máximo dois comandos.

Com aninhamento ou sequenciamento de três comandos o bloco de três comandos apresenta o maior número de implementações. Neste não foram atribuídos testes para comandos if sem um else correspondente.

Todos as implementações foram testadas com todas as coberturas disponíveis. Portanto foi testado com cobertura de linhas, arestas, caminhos e cláusula. A partir da análise do relatório produzido por cada teste é possível observar a coerência do BTestBox.

Para o caso industrial, diversas contribuições foram dadas ao BTestBox, como a adição de algumas estruturas B, que não haviam anteriormente sido testadas. O caso industrial foi o mais demorado para ser analisado pelo BTestBox, para uma cobertura simples era necessário dois dias para o ProB lançar uma resposta; isso ocorre porque o ProB precisa expandir muitos conjuntos e variáveis. Assim, não foi possível completar todos os testes no código.

Além disso, BTestBox não testa algumas estruturas do código, pois a ferramenta não foi implementada para matrizes. Atualmente, BTestBox consegue manipular vetores, porém não está implementado para estruturas de dados mais complexas. Os elementos suportados por BTestBox podem ser observados na tabela 5.3.

A ferramenta também foi testada a fim de explicitar o gargalo gerado por utilizar a ferramenta *ProB* e a escalabilidade do BTestBox. Para realizar isto, grande implementações foram analisadas por *BTestBox* e o tempo de execução de cada seção foi registrado. Estas implementações foram geradas a partir dos diversos elementos gramaticais da linguagem B e criadas de maneira randomica com as instruções aninhadas. Estas implementações já foram previamente utilizadas por Valério para testar a escalabilidade do compilador de B para LLVM [16], ele utiliza de implementações com tamanho colossais, com comandos aninhados, chegando a ocupar

Tabela 5.3 Estruturas suportadas por BTestBox.

Tipo		Suporta
Operadores Aritméticos		√
Operadores Lógicos		√
Operadores de Conjuntos		√
Estrutura de Dados	Vetores	√
	Matrizes	X
	Árvores	X

um espaço de mais de 100 megabytes de memória. Os arquivos para o teste de escabilidade podem ser classificados de acordo com o tamanho das operações, as varíaveis que definem a geração destes arquivos são a quantidade de aninhamentos presente em um bloco de comando e a quantidade de blocos de comando dentro de uma operação. O comandos visam por testar a gramática B. As implementações podem ter 1, 10, 50, 100 ou 500 blocos de comando e o aninhamento variou de 1 até 5.

Para os testes de escabilidade foi utilizado um computador com o sistema operacional Windows 10 64 bits, com um processador Intel Core i56300HQ 2300 GHz e 8 GB de memória RAM. Foi observado pelo autor que para a observação da escalabilidade pode se ater a implementações de menor porte, uma vez que o próprio Atelier-B, versão 4.3.1, presente no computador citado não consegue gerar as obrigações de prova e o arquivo BXML, para arquivos contendo mais que duplo aninhamento e dez blocos de comando por operação.

Além disso, as implementações apresentadas por Valério precisaram ser adaptadas para funcionar com o BTestBox, porém não perdendo a quantidade de instruções ou aninhamentos. Isto ocorreu porque Valério [16] utilizou de chamada de operação que não é um elemento suportado pela ferramenta em seu estado atual, uma vez que é necessario acessar um arquivo do Atelier-B que atualmente está indisponível para uso externo. Além disso, por não serem implementações provadas, elas apresentaram instruções e decisões que realizavam cálculos matemáticos sem solução. Nos primeiros testes esses problemas não foram notados e as avaliações podiam resultar em valores para as entradas que não resultavam em uma saída correspondente. Com a observação dos problemas essas instruções foram modificadas para instruções mais simples que não apresentam erros matemáticos.

Já para a menor implementação do teste de escalabilidade é notório que o tempo de análise do predicado pelo solucionador de predicados necessita muito mais tempo do que as outras seções da execução. Pois com esse exemplo é observado que para a cobertura de comandos 89.49% do tempo de execução é dedicado ao solucionamento de predicados. Para as outras implementações testadas o tempo necessário para a avaliação do provador também toma a maior parte do tempo como pode ser observado na tabela 5.4.

Para algumas operações do componente COMP_2seq10 o número de caminhos que foram calculados por BTestBox superou 750000. Por se tratar de uma operação realizada para o teste e gerada de maneira randômica, muitos dos caminhos são impossíveis de serem executados. Então para o BTestBox executar um caminho capaz de gerar entradas pode ser necessário descartar vários caminhos, para este caso a escolha de um caminho que é possível de ser executado

Tabela 5.4 Implementações para teste de Escalabilidade

Componente	COMP_1seq1	COMP_2seq1	COMP_3seq1
Quantidade de Operações	39	199	999
Aninhamento	1	2	3
Comandos por Bloco	1	1	1
Tempo Total de execução (segundos)	587.36	3102.89	18214.17
Tempo gasto com a avaliação de predicado (segundos)	525.63	2858.76	15274.43
Porcentagem da execução para a avaliação do predicado	89.49	92.13	83.86

reduziria o tempo de execução drasticamente. Na tabela 5.5 pode ser observado os valores obtidos para os tempos de execução das operações do componente. Os testes nas operações dos componentes COMP_2seq10 demonstram o gargalo apresentado pelo ProB mas também mostra que a escolha de caminhos inúteis também pode ser um fator crucial no tempo de teste de uma implementação. É importante resaltar que a ferramenta Atelier-B não conseguiu gerar obrigações de prova para o componente COMP_2seq10 porém é possível que a implementação seja testada com o BTestBox.

Tabela 5.5 Teste de escalabilidade com as operações da implementação COMP_2seq10

Componente	Operação	Quantidade de Caminhos	Tempo Total de execução (segundos)	Tempo gasto com a avaliação de predicado (segundos)	Porcentagem da execução para a avaliação do predicado
COMP_2seq10	ID00000	4320	8637.51	7647.99	88.54
	ID00001	87480	205552.43	168452.96	81.95
	ID00002	787320	1781627.72	1423590.45	79.90

Na tabela 5.5, também é possível observar que o tempo para a execução da operação "ID00002" foi mais de 20 dias e desse tempo a avaliação de predicado ocupa mais de 16 dias. É importante ressaltar que para as operações dos casos apresentados por Valério é necessário testar muitos guias para encontrar algum que possivelmente executa os nós do caminho. Com uma quantidade maior de nós e instruções randômicas, se torna mais difícil encontrar decisões que satisfaçam vários nós.

CAPÍTULO 6

Considerações Finais

Nessa dissertação foi apresentada a ferramenta BTestBox. A ferramenta é capaz de testar implementações B, de maneira que em primeiro momento verifica o alcance de um critério de cobertura e, em um segundo momento, a partir dos casos de teste gerados para essa verificação aplica a execução dos casos de teste na tradução. Desse modo, BTestBox verifica o comportamento de um software, através dos critérios de cobertura, e analisa se a tradução através de um compilador B está correta. Sendo assim, BTestBox tenta mitigar os riscos de erros através de uma verificação da tradução B e teste do compilador de forma a achar eventuais erros.

BTestBox é capaz de aplicar testes para uma grande variedade de implementações B, porém não é perfeito. BTestBox funciona rapidamente para pequenas implementações, mas, com grandes implementações, nas quais é necessário o cálculo de muitos estados e expansões para alcançar um resultado do predicado avaliado, a obtenção da resposta pode demorar dias. Isso ocorre porque o BTestBox utiliza o ProB para avaliar se o predicado gerado pode ter algum estado que o solucione; com esse objetivo, o avaliador expande todas as variáveis, o que pode demandar grande quantidade de tempo. Essa é a etapa mais demorada da ferramenta BTestBox e precisa apresentar uma melhor performance. Uma possível solução é simplificar o predicado antes de pedir a avaliação do mesmo, esta avaliação deve levar em conta quais variáveis, conjuntos e condições são importantes e têm relação com a operação em análise.

Outro problema encontrado é quando uma máquina não apresenta uma implementação, usualmente este é o caso da máquina base. Essas máquinas não apresentam implementação e podem apresentar estruturas abstratas. Uma vez que BTestBox atualmente só pode ser aplicado em componentes com implementação, essa restrição limita o alcance da ferramenta, geralmente não podendo ser aplicada para projetos reais. Isso se torna complicado ao tentar testar uma implementação e comparar o resultado esperado com o obtido, visto que o ProB retorna um resultado possível enquanto o programa pode resultar em um diferente. Portanto, atualmente BTestBox consegue calcular programas apenas para máquinas determinísticas ou que têm uma implementação.

Além disso, BTestBox utiliza tradutores B para realizar a tradução do método B para a linguagem desejada. Sendo assim, BTestBox também herda todas as limitações do tradutor escolhido, como somente poder traduzir o subconjunto do B0 que o tradutor é capaz de traduzir. A ferramenta também utiliza dos compiladores B para realizar a tradução dos casos de testes, isto pode ser um problema, uma vez o resultado dos compiladores são os objetos a serem testados, utilizar os compiladores como meio de tradução pode inserir erros no próprio teste. Uma solução, seria escrever os casos de teste na linguagem alvo da tradução.

Mais uma questão encontrada é no artifício utilizado durante a inserção da função *set*. Uma vez que a operação abstrata apresenta um *skip* em seu corpo e nenhuma precondição, essa

função não pode ser provada. Para que a implementação possa ser provada, é necessário que os possíveis estados das variáveis estejam discretos dentro da precondição.

Apesar de todas as limitações, BTestBox é operacional. Pode ser lançado a partir do Atelier-B via plugin, com uma interface simples e totalmente automático após as escolhas impostas pela interface. A ferramenta auxilia a comunidade que utiliza B, testando a tradução dos compiladores além de verificar a cobertura das implementações B. Dessa forma, pode ajudar a observar o comportamento da implementação como também na obtenção de certificados.

BTestBox está executável, porém não está finalizado. Com os trabalhos futuros o autor visa dar mais opções para a ferramenta, de modo a suportar mais compiladores B, uma vez que atualmente o único compilador suportado é o C4B. Para dar esse passo, é necessário que o autor forneça ao BTestBox meios de criar um arquivo na linguagem da tradução, além de uma maneira de compilar e checar o resultado. Os próximos compiladores a serem adicionados como opções são o ADA e o LLVM. Por enquanto, a ferramenta somente está disponível para o sistema operacional Windows, mas uma vez que as outras ferramentas utilizadas em conjunto com o BTestBox, ProB e o Atelier-B, estão também disponíveis em Linux e Mac, o BTestBox também poderá ser funcional nestas plataformas. Além disso, é necessária a solução dos problemas descritos acima.

Diante de outras ferramentas, como o ProTest [43], BZ-TT [1] e o BETA [37], BTestBox se diferencia por tratar de componentes concretos, desse modo, está mais próximo de um software real do que os componentes abstratos que são o objeto de teste das outras ferramentas. Outra diferença para os dois primeiros softwares é que BTestBox é automático na geração de casos de teste.

BTestBox já foi previamente idealizado e apresentado ao meio científico através do artigo "BTestBox: an automatic test generator for B method" [16]. Porém, estava limitado a testes randomicos. A ferramenta foi remodelada e então passou a realizar teste com base nos critérios de cobertura.

BTestBox também é mais um trabalho que tenta verificar a cobertura MC/DC. O esforço utilizado para a análise e verificação dessa cobertura foi grande e uma solução simples não foi encontrada, de forma que a melhor solução apresentada foi utilizar uma cobertura mais robusta do que a MC/DC, portanto, se a solução apresentada por BTestBox é satisfeita, o MC/DC também é satisfeito. Porém, apesar de não ter sido encontrada uma solução simples para o MC/DC, a utilização da lógica de Hoare permite analisar de maneira eficaz diversas outras coberturas, tais como as coberturas de linhas, arestas, caminhos, predicados, cláusulas e a cobertura combinada.

Os resultados deste trabalho demonstram a viabilidade da proposta, BTestBox é funcional e pode ser utilizado pela comunidade B para a checagem de cobertura além da verificação da corretude da tradução. BTestBox foi testado com várias pequenas implementações B e também com um projeto B fornecido pela empresa ClearSy. Dessa forma, BTestBox é uma colaboração entre academia e industria, um esforço para a obtenção de mais certificados pelos componentes desenvolvidos com o Atelier-B e o método B.

Por fim, BTestBox contribui com o avanço das tecnologias de métodos formais e de teste de software, pois utiliza dessas duas áreas da computação.

Referências Bibliográficas

[1] AMBERT, F., BOUQUET, F., CHEMIN, S., GUENAUD, S., LEGEARD, B., PEUREUX, F., VACELET, N., AND UTTING, M. BZ-TT: A tool-set for test generation from Z and B using constraint logic programming. *Proc. of FATES 2002* (2002), 105–120.

[2] AMMANN, P., AND OFFUTT, J. *Introduction to software testing*. Cambridge University Press, 2008.

[3] AMMANN, P., OFFUTT, J., AND HUANG, H. Coverage criteria for logical expressions. In *Software Reliability Engineering, 2003. ISSRE 2003. 14th International Symposium on* (2003), IEEE, pp. 99–107.

[4] AWEDIKIAN, Z., AYARI, K., AND ANTONIOL, G. Mc/dc automatic test input data generation. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation* (2009), ACM, pp. 1657–1664.

[5] BARJAKTAROVIC, M., AND NASSIFF, M. The state-of-the-art in formal methods. *AFOSR Summer Research Technical Report for Rome Research Site, Formal Methods Framework-Monthly Status Report, F30602-99-C-0166, WetStone Technologies* (1998).

[6] BJØRNER, D., AND HENSON, M. C. *Logics of specification languages*. Springer Science & Business Media, 2007.

[7] BUSS, S. R. An introduction to proof theory. *Handbook of proof theory 137* (1998), 1–78.

[8] CARLIER, M., AND DUBOIS, C. Functional testing in the focal environment. In *International Conference on Tests and Proofs* (2008), Springer, pp. 84–98.

[9] CAVALCANTI, A., KING, S., O'HALLORAN, C., AND WOODCOCK, J. Test-data generation for control coverage by proof. *Formal Aspects of Computing 26*, 4 (2014), 795–823.

[10] CLEARSY SYSTEMS ENGINEERING. *Atelier B User Manual*.

[11] CLEARSY SYSTEMS ENGINEERING. *B Language Keywords and Operators*.

[12] CLEARSY SYSTEMS ENGINEERING. *Traducteurs de l'Atelier B*.

[13] DADEAU, F., DE KERMADEC, A., AND TISSOT, R. Combining scenario-and model-based testing to ensure posix compliance. In *International Conference on Abstract State Machines, B and Z* (2008), Springer, pp. 153–166.

[14] DEHARBE, D. *Translation of B Implementations to the LLVM: Specification*.

[15] DEHARBE, D., AZEVEDO, D., DE MATOS, E. C., AND MEDEIROS JR, V. Btestbox: an automatic test generator for b method.

[16] DEHARBE, D., AZEVEDO, D., DE MATOS, E. C., MEDEIROS JR, V., NATAL, R., AND PARNAMIRIM, R. Btestbox: an automatic test generator for b method.

[17] DIJKSTRA, E. W., DIJKSTRA, E. W., DIJKSTRA, E. W., AND DIJKSTRA, E. W. *A discipline of programming*, vol. 1. prentice-hall Englewood Cliffs, 1976.

[18] DONG, Y., LI, Z., AND TOWEY, D. On the relationship between model coverage and code coverage using matlab's simulink. In *Software Quality, Reliability and Security (QRS), 2015 IEEE International Conference on* (2015), IEEE, pp. 213–218.

[19] GALVÃO, S. D. S. L. Especificação do micronúcleo freertos utilizando o método b.

[20] GHANI, K., AND CLARK, J. A. Automatic test data generation for multiple condition and mcdc coverage. In *Software Engineering Advances, 2009. ICSEA'09. Fourth International Conference on* (2009), IEEE, pp. 152–157.

[21] GIANNAKOPOULOU, D., BUSHNELL, D. H., SCHUMANN, J., ERZBERGER, H., AND HEERE, K. Formal testing for separation assurance. *Annals of Mathematics and Artificial Intelligence 63*, 1 (2011), 5–30.

[22] GUPTA, N., MATHUR, A. P., AND SOFFA, M. L. Generating test data for branch coverage. In *Automated Software Engineering, 2000. Proceedings ASE 2000. The Fifteenth IEEE International Conference on* (2000), IEEE, pp. 219–227.

[23] HAYHURST, K. J., VEERHUSEN, D. S., CHILENSKI, J. J., AND RIERSON, L. K. A practical tutorial on modified condition/decision coverage.

[24] HOARE, C. A. R. An axiomatic basis for computer programming. *Communications of the ACM 12*, 10 (1969), 576–580.

[25] HOARE, C. A. R. Proof of correctness of data representations. In *Software pioneers*. Springer, 2002, pp. 385–396.

[26] JÚNIOR, M., ET AL. Aplicação do método b ao projeto formal de software embarcado.

[27] KANDL, S., KIRNER, R., AND PUSCHNER, P. Development of a framework for automated systematic testing of safety-critical embedded systems. In *Intelligent Solutions in Embedded Systems, 2006 International Workshop on* (2006), IEEE, pp. 1–13.

[28] KAPOOR, K., AND BOWEN, J. Experimental evaluation of the variation in effectiveness for dc, fpc and mc/dc test criteria. In *Empirical Software Engineering, 2003. ISESE 2003. Proceedings. 2003 International Symposium on* (2003), IEEE, pp. 185–194.

[29] KHAN, S. U. R., PECK LEE, S., PARIZI, R. M., AND ELAHI, M. An analysis of the code coverage-based greedy algorithms for test suite reduction. In *The Second International Conference on Informatics Engineering & Information Science (ICIEIS2013)* (2013), The Society of Digital Information and Wireless Communication, pp. 370–377.

[30] KORNECKI, A., AND ZALEWSKI, J. Certification of software for real-time safety-critical systems: state of the art. *Innovations in Systems and Software Engineering 5*, 2 (2009), 149–161.

[31] LEROY, X. The compcert verified compiler, software and commented proof, 2008.

[32] LEROY, X. Formal verification of a realistic compiler. *Communications of the ACM 52*, 7 (2009), 107–115.

[33] LEUSCHEL, M. User manual - prob documentation, 2017.

[34] LIM, J. An engineering disaster: Therac-25. *Joanne Lim* (1998).

[35] LIONS, J.-L. Flight 501 failure. *Report by the Inquiry Board* (1996).

[36] MARINESCU, R., SECELEANU, C., LE GUEN, H., AND PETTERSSON, P. A research overview of tool-supported model-based testing of requirements-based designs. Advances in Computers. Elsevier, 2015.

[37] MATOS, E. C. B. *BETA: a B based testing approach*. PhD thesis, Federal University of Rio Grande do Norte, Natal, 2016.

[38] MEDEIROS JR., V. *Método B e a síntese verificada para código de montagem*. PhD thesis, Federal University of Rio Grande do Norte, Natal, 2016.

[39] MINJ, J. Feasible test case generation using search based technique. *International Journal of Computer Applications 70*, 28 (2013).

[40] PELESKA, J. A unified approach to abstract interpretation, formal verification and testing of c/c++ modules. In *International Colloquium on Theoretical Aspects of Computing* (2008), Springer, pp. 3–22.

[41] PRETSCHNER, A., SLOTOSCH, O., AIGLSTORFER, E., AND KRIEBEL, S. Model-based testing for real. *International Journal on Software Tools for Technology Transfer 5*, 2-3 (2004), 140–157.

[42] RODRIGUEZ, V., DOMINGUEZ, J., AND SÁNCHEZ, G. Yaact: A genetic algorithm tool for code coverage analysis.

[43] SATPATHY, M., LEUSCHEL, M., AND BUTLER, M. ProTest: An Automatic Test Environment for B Specifications. *ENTCS 111* (2005), 113–136.

[44] STUERMER, I., CONRAD, M., DOERR, H., AND PEPPER, P. Systematic testing of model-based code generators. *IEEE Transactions on Software Engineering 33*, 9 (2007), 622.

[45] TIWARI, S. *AUTOMATIC GENERATION OF TESTCASES FOR HIGH MCDC COVERAGE*. PhD thesis, INDIAN INSTITUTE OF TECHNOLOGY KANPUR, 2014.

[46] VERIFIED SYSTEMS INTERNATIONAL GMBH. *RT-Tester 6.2 - User Manual*, 2007.

[47] VILKOMIR, S. A., AND BOWEN, J. P. From mc/dc to rc/dc: formalization and analysis of control-flow testing criteria. *Formal Aspects of Computing 18*, 1 (2006), 42–62.

[48] WEISER, M., GANNON, J. D., AND MCMULLIN, P. R. Comparison of structural test coverage metrics. *IEEE Software 2*, 2 (1985), 80.

[49] WHALEN, M. W., PERSON, S., RUNGTA, N., STAATS, M., AND GRIJINCU, D. A flexible and non-intrusive approach for computing complex structural coverage metrics. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1* (2015), IEEE Press, pp. 506–516.

[50] YANG, X., CHEN, Y., EIDE, E., AND REGEHR, J. Finding and understanding bugs in c compilers. In *ACM SIGPLAN Notices* (2011), vol. 46, ACM, pp. 283–294.