Instituto de Pesquisas Tecnológicas do Estado de São Paulo

Edmilson Martins dos Santos

Geração Semiautomática de Casos de Teste: uma Técnica para Apoio à Migração de Sistemas Legados em COBOL

São Paulo
2012

Edmilson Martins dos Santos

Geração Semiautomática de Casos de Teste: uma Técnica para Apoio à Migração de Sistemas Legados em COBOL

Dissertação de Mestrado apresentada ao Instituto de Pesquisas Tecnológicas do Estado de São Paulo – IPT, como parte dos requisitos para obtenção do título de Mestre em Engenharia de Computação.

Data da aprovação: ____/ ____ / ______

____
Prof. Dr. José Eduardo Zindel Deboni (Orientador)
IPT – Instituto de Pesquisas Tecnológicas do Estado de São Paulo

Membros da Banca Examinadora:

Prof. Dr. José Eduardo Zindel Deboni (Orientador)
IPT – Instituto de Pesquisas Tecnológicas do Estado de São Paulo

Profa. Dra. Marcia Ito (Membro)
IBM Research Brazil

Prof. Dr. Marcelo Novaes de Rezende (Membro)
IPT – Instituto de Pesquisas Tecnológicas do Estado de São Paulo

Edmilson Martins dos Santos

Geração Semiautomática de Casos de Teste: uma Técnica para Apoio à Migração de Sistemas Legados em COBOL

Dissertação de Mestrado apresentada ao Instituto de Pesquisas Tecnológicas do Estado de São Paulo – IPT, como parte dos requisitos para obtenção do título de Mestre em Engenharia de Computação.

Área de Concentração: Engenharia de *Software*

Orientador: Prof. Dr. José Eduardo Zindel Deboni

São Paulo
Novembro/2012

Ficha Catalográfica
Elaborada pelo Departamento de Acervo e Informação Tecnológica – DAIT
do Instituto de Pesquisas Tecnológicas do Estado de São Paulo - IPT

S237g Santos, Edmilson Martins dos

Geração semiautomática de casos de teste: uma técnica para apoio à migração de sistemas legados em COBOL. / Edmilson Martins dos Santos. São Paulo, 2012.
113p.

Dissertação (Mestrado em Engenharia de Computação) - Instituto de Pesquisas Tecnológicas do Estado de São Paulo. Área de concentração: Engenharia de Software

Orientador: Prof. Dr. José Eduardo Zindel Deboni

1. Teste de software 2. Migração de programas 3. COBOL (Linguagem de programação) 5. Tese I. Deboni, José Eduardo Zindel, orient. II. IPT. Coordenadoria de Ensino Tecnológico III. Título

12-98 CDU 004.415.53(043)

DEDICATÓRIA

Dedico este trabalho à minha esposa Cátia e às minhas filhas Letícia e Gabriela, pela compreensão e apoio no tempo que deixei de estar ao lado delas.

AGRADECIMENTOS

Ao meu orientador, Prof. Dr. José Eduardo Zindel Deboni, pela dedicação, direcionamento e valiosas dicas no desenvolvimento deste trabalho.

À Profa. Edit G. L. de Campos, pelos ensinamentos e correções nas versões iniciais deste trabalho.

À empresa Tokio Marine Seguradora, pelo subsídio parcial no custeio do curso.

Aos meus amigos de empresa que direta ou indiretamente me ajudaram com a troca de ideias sobre o assunto deste trabalho, e em especial ao meu amigo Flávio Henrique Cerqueira, pelo apoio na parte de programação Java.

RESUMO

Sistemas de software devem evoluir, entre outros fatores, devido à obsolescência técnica ou exigências de manutenção. Para assegurar esta evolução, funcionalmente, um conjunto abrangente de casos de teste deve ser gerado, garantindo que o novo sistema tenha um comportamento similar ao do sistema anterior. Esses casos de teste devem ter uma boa cobertura, percorrendo os diversos caminhos internos do programa. A abordagem de teste direcionado gera casos de teste para atender os critérios de cobertura dos diversos caminhos de um programa. Esta abordagem de teste direcionado combina as abordagens de execução concreta e execução simbólica de um programa. O objetivo deste trabalho é desenvolver uma técnica, apoiada em ferramentas de software, aqui chamada de C4T (COBOL *for Test*), para cobertura de teste utilizando teste direcionado juntamente com uma ferramenta de solucionador de restrições, que calcula os valores das variáveis que atendem às restrições dadas. Essa técnica é utilizada em um programa de exemplo escrito em linguagem COBOL e avalia-se que essa técnica pode ser utilizada para tradução dessa linguagem para outras linguagens mais atuais, com resultados promissores para apoiar os analistas de teste na produção automática de casos de teste.

Palavras-chave: teste automatizado, teste direcionado, COBOL.

ABSTRACT

Semiautomatic Generation of Test Cases: A Technique to Support Migration of Legacy Systems in COBOL

Software systems must evolve, among other factors, due to technical obsolescence or maintenance requirements. To functionally ensure this evolution, a comprehensive set of test cases must be generated, verifying that the new system has the same behavior of the previous one. These test cases should have good coverage, covering the various internal paths of the program. The directional test approach generates test cases to meet the criteria for coverage of the various paths of a program. This targeted testing approach combines the approaches of concrete execution and symbolic execution of a program. The objective of this work is to develop a technique, based on a set of software tools, called here C4T (COBOL for Test) to automatic test generation along with a constraints solver tool, which calculates the values of variables that meet the given constraints. This technique is used in a sample program written in COBOL language and is analysed that this technique can be used for translation of this language into newer ones, with promising results to support testing analysts for automatic production of test cases.

Keywords: automated test, directed test, COBOL.

LISTA DE ILUSTRAÇÕES

Figura 1 - Estratégias para Migração de Programas 21
Figura 2 - Trecho de Programa Cobol com ponto-final 22
Figura 3 - Trecho de Programa Cobol sem ponto-final 22
Figura 4 - Exemplo de Comando GO TO 23
Figura 5 - Evolução da atividade de teste de software 28
Figura 6 - Código-fonte e gráfico de fluxo de controle (CFG) de um programa 29
Figura 7 - Visão geral das técnicas de geração de casos de teste 31
Figura 8 - Gráfico da Subida de Encosta 36
Figura 9 - Geração de dados de teste utilizando algoritmo genético........................38
Figura 10 - Arquitetura de um sistema gerador de casos de teste 47
Figura 11 - Teste de aceitação utilizando FIT 51
Figura 12 - Arquitetura da Técnica C4T 60
Figura 13 - Execução do FitNesse em conjunto com a técnica C4T 61
Figura 14 - Gráfico de fluxo de controle do programa exemplo 67
Figura 15 - Fluxo do programa controlador da técnica C4T 76
Figura 16 - Tabela FitNesse do programa exemplo 84

LISTA DE TABELAS

Tabela 1 - Exemplo de equação com números inteiros 46
Tabela 2 - Exemplo de tabela Fitnesse 50
Tabela 3 - Comparação entre os trabalhos citados...53
Tabela 4 - Código-Fonte original do programa Cobol 64
Tabela 5 - Conversão das variáveis do programa exemplo para o *solver*.................66
Tabela 6 - Código-fonte após aplicação do passo 3 da técnica C4T 70
Tabela 7 - Código-fonte após aplicação do passo 4 da técnica C4T 72
Tabela 8 - Restrições do solver - ciclo 01a..77
Tabela 9 - Restrições do solver - ciclo 01b..77
Tabela 10 - Restrições do solver - ciclo 02a..78
Tabela 11 - Restrições do solver - ciclo 02b..78
Tabela 12 - Restrições do solver - ciclo 02c..78
Tabela 13 - Restrições do solver - ciclo 06 79
Tabela 14 - Restrições do solver - ciclo 07a..79
Tabela 15 - Restrições do solver - ciclo 07b..80
Tabela 16 - Restrições do solver - ciclo 08a..80
Tabela 17 - Restrições do solver - ciclo 08b..80
Tabela 18 Restrições do solver - ciclo 08c 80
Tabela 19 - Restrições do solver - ciclo 09a..81
Tabela 20 - Restrições do solver - ciclo 09b..81
Tabela 21 - Restrições do solver - ciclo 09c..81
Tabela 22 - Restrições do solver - ciclo 12a..82
Tabela 23 - Restrições do solver - ciclo 12b..82
Tabela 24 - Restrições do solver - ciclo 13a..83
Tabela 25 - Restrições do solver - ciclo 13b..83
Tabela 26 - Restrições do solver - ciclo 13c..83
Tabela 27 - Restrições do solver - ciclo 15 84
Tabela 28 - Resultados de entrada e saída após a execução da técnica C4T 84
Tabela 29 - Casos de teste gerados manualmente..86
Tabela 30 - Execução pelo FitNesse dos CT's gerados pela técnica C4T................87
Tabela 31 - Execução pelo FitNesse dos CT's gerados manualmente.....................87
Tabela 32 - Inclusão de mutante de atribuição 88

Tabela 33 - Execução dos CT’s gerados pelo C4T com mutante de atribuição........88
Tabela 34 - Inclusão de mutante de condição..89
Tabela 35 - Execução dos CT’s gerados pelo C4T com mutante de condição.........89
Tabela 36 - Resultados de entrada e saída alterando o valor inicial.........................90
Tabela 37 - Restrições para o solver para A > B...91
Tabela 38 – Saída esperada após inclusão de dados de documentação 91
Tabela 39 – Instrumentações do programa COBOL após os passos 3 e 4 110
Tabela 40 - Pseudocódigo do Algoritmo do Instrumentador 116

LISTA DE ABREVIATURAS E SIGLAS

ACO	*Ant Colony Optimization* (Otimização por Colônia de Formigas)
AG	Algoritmo Genético
C4T	COBOL *for Test*
CFG	*Control Flow Graph (*Gráfico de Fluxo de Controle)
CLP	*Constraint Language Programming* (Linguagem de Programação de Restrições)
COBOL	**CO**mmon **B**usiness **O**riented **L**anguage (Linguagem Comum Orientada aos Negócios)
EATDD	*Executable Acceptance Test-Driven Development* (Desenvolvimento Dirigido por Teste de Aceitação Executável)
FIT	*Framework for Integrated Test*
HTML	*HyperText Markup Language* (Linguagem de Marcação de Hipertexto)
JVM	*Java Virtual Machine* (Máquina Virtual Java)
OCL	*Object Constraint Language*
PSO	*Particle Swarm Optimization* (Otimização por Enxame de Partículas)
SBES	Simpósio Brasileiro de Engenharia de Software
SMT	*Satisfiability Modulo Theories* (Módulo de Teorias de Satisfabilidade)
SUT	*Software Under Test* (*Software* em teste)
UML	*Unified Modeling Language* (Linguagem de Modelagem Unificada)

SUMÁRIO

1 INTRODUÇÃO 15

1.1 Motivação 15

1.2 Objetivo 16

1.3 Método de Trabalho 17

1.4 Organização 17

2 ESTADO DA ARTE 19

2.1 Importância dos Casos de Teste para Migração de Programas 19

2.2 Técnicas de Teste 23

2.2.1 Teste Funcional (caixa-preta) 23

2.2.2 Teste Estrutural (caixa-branca) 24

2.3 Tipos de Testes Funcionais 24

2.4 Tipos de Testes Estruturais 25

2.5 Geração automática de dados de teste 27

2.6 Abordagens para geração de testes 30

2.6.1 Abordagens para testes funcionais 31

2.6.1.1 Especificação Formal 31

2.6.1.2 Teste baseado em modelo 32

2.6.2 Abordagens para testes estruturais 33

2.6.2.1 Aleatório 34

2.6.2.2 Busca Meta-heurística 34

2.6.2.3 Subida de Encosta (*Hill Climbing*) 35

2.6.2.4 Têmpera Simulada (*Simulated Annealing*) 36

2.6.2.5 Algoritmos evolutivos 37

2.6.2.6 Inteligência de Enxames 39

2.6.2.7 Otimização por Colônia de Formigas 42

2.6.2.8 Teste de Pares 42

2.6.2.9 Execução Simbólica 42

2.6.2.10 Execução Dinâmica 43

2.6.2.11 Testes Direcionados 44

2.7 Solucionadores de restrições 46

2.8 Sistema Gerador de Casos de Teste 47

2.9 Ferramentas para Execução dos Testes 48
2.10 Conclusão 53
3 PROPOSTA: A TÉCNICA C4T 57
3.1 Introdução 57
3.2 Escolha da Abordagem 57
3.3 Arquitetura da Técnica C4T 59
3.4 Método para geração de casos de teste 61
3.4.1 Preparação do Ambiente de Execução 61
3.4.2 Etapas da Técnica C4T 61
3.5 Exemplo de Uso 63
4 ANÁLISE DOS RESULTADOS 86
5 CONCLUSÕES 92
5.1 Sugestões para trabalhos futuros 94
REFERÊNCIAS 96
ANEXO 1 – Código-Fonte dos Programas que fazem as Chamadas ao S*olver*. 102
APÊNDICE 1 – Instrumentações no Código-Fonte da Técnica C4T 110
APÊNDICE 2 – Código-Fonte do Programa Controlador 113
APÊNDICE 3 – Código-Fonte do Programa Exemplo Convertido Para Java 114
APÊNDICE 4 – Código-Fonte do Programa F*ixture* do FitNesse 115
APÊNDICE 5 – Pseudocódigo do Algoritmo do Instrumentador 116

1 INTRODUÇÃO

1.1 Motivação

À medida que os sistemas de *software* sofrem alterações, eles se tornam mais complicados. Em determinado momento, as manutenções em um trecho de código provocam problemas em outros trechos, geralmente, que não haviam sido testados suficientemente. Provavelmente, o código precisará ser reescrito, pois o sistema pode estar obsoleto (PRESSMAN, 2006). Muitas empresas adiam ao máximo a decisão de migrar um sistema, pois tem receio de que nem todas as funcionalidades, condições e situações da versão atual estarão presentes no sistema novo.

Quando as empresas finalmente decidem reescrever o sistema, frequentemente, destina-se uma equipe para escrever todas as condições de teste do sistema, e isso pode causar morosidade no processo, além de aumentar o custo do novo sistema. De acordo com BEIZER (1995), o custo de teste representa até 50% do custo total do sistema.

De acordo com Al-Azzoni, Zhang e Down (2011), muitas empresas estão migrando seus sistemas legados, escritos para ambiente *mainframe* em linguagem COBOL. Cerca de 95% dos dados processados atualmente em aplicações comerciais de empresas financeiras e de seguros se dá por programas escritos em COBOL. Qualquer falha nessas aplicações pode provocar sérios impactos legais e financeiros (PU et al, 2007).

Ao reescrever um sistema a empresa deve investir em testes, pois com eles é possível auferir qualidade ao *software,* encontrando falhas (PRESSMAN, 2006).

A geração manual dos casos de teste implica fazer a leitura do código-fonte e identificar as situações de decisão e *loops* do programa, a fim de gerar casos de teste que atendam a cada uma dessas situações.

Um aspecto relevante na atividade de testes é avaliar o quanto um conjunto de dados de teste efetivamente exercitou o *software*. A simples geração aleatória de

entradas muitas vezes não é adequada pois pode não satisfazer alguns requisitos específicos durante a execução do *software* (ABREU, 2006).

Além dos testes funcionais, onde um dado de entrada é aplicado a um programa e a saída obtida é comparada com um resultado esperado, em uma migração de sistema a empresa possui os códigos-fonte do sistema a ser migrado, portanto ela pode fazer uso da técnica de teste estrutural, também conhecida como técnica de teste caixa-branca (SOMMERVILLE, 2003). Com base na análise desses códigos-fonte, pode definir os dados de entrada para efetuar os testes do sistema e complementar os testes funcionais.

Uma vez que a geração manual das diversas combinações dos dados de entrada (ou casos de teste) é uma atividade morosa e cara (MYERS e SANDLER, 2004), a geração automática dos casos de teste é um assunto de grande interesse no campo da engenharia de *software*, pois ela pode acelerar este processo e demandar menos gastos com testes (ABREU, 2006).

A principal motivação para este trabalho é que, em virtude da geração manual de casos de teste, em especial em programas COBOL, ser propensa a erros, a geração automática de casos de teste pode auxiliar no processo de teste de programas escritos nessa linguagem, e de posse de bons casos de teste poder usá-los na migração dos programas para outras linguagens.

1.2 Objetivo

O objetivo principal do trabalho é desenvolver uma técnica, chamada C4T (COBOL *for Test*), para geração semiautomática de casos de teste para programas escritos em COBOL.

Os objetivos secundários deste trabalho são: avaliar a eficiência da geração semiautomática de casos de teste quando comparada com a geração manual; documentar os programas por meio da criação de casos de teste e auxiliar o processo de migração de programas COBOL.

1.3 Método de Trabalho

A fim de atingir o objetivo proposto, analisa-se a literatura para identificar as principais técnicas e ferramentas utilizadas para geração automática de casos de teste. Busca-se verificar qual pode ser a alternativa que melhor se enquadra para geração de testes de programas escritos em COBOL. Como conclusão dessa análise propõe-se o desenvolvimento da técnica C4T para geração automática dos casos de testes de programas escritos em COBOL.

A técnica C4T proposta é desenvolvida e validada por meio de comparação dos casos de teste gerados automaticamente com a geração manual dos casos de teste com relação à facilidade de uso e à quantidade e eficiência dos casos de teste gerados. Identificam-se, então, os benefícios bem como as dificuldades da utilização da técnica caracterizando as possíveis alternativas de solução.

1.4 Organização

A Seção 2, Estado da Arte, apresenta as diversas abordagens para geração automática de casos de teste, englobando testes exaustivos, testes aleatórios, execução simbólica, execução dinâmica, e enfatiza a abordagem de teste direcionado aplicada por Kiezun (2009).

A Seção 3, Técnica C4T, apresenta uma técnica para geração automática dos casos de teste a partir de código-fonte em COBOL.

A Seção 4, Análise dos Resultados, apresenta os resultados da aplicação da técnica C4T em um programa COBOL para verificar se os dados de entrada gerados pela técnica atendem aos critérios de cobertura de testes do programa COBOL escolhido.

A Seção 5, Conclusões, apresenta as conclusões a respeito da análise de viabilidade da substituição da geração de casos de teste efetuada manualmente pela geração automática de casos de teste utilizando a técnica C4T.

O ANEXO 1 – Código-Fonte dos Programas que fazem as Chamadas apresenta os trechos principais do código-fonte da técnica C4T, para que ela possa ser utilizada em outros estudos de migração de sistemas escritos em COBOL.

O APÊNDICE 1 – Instrumentações no Código-Fonte da Técnica C4T apresenta as instrumentações do código-fonte de linguagem COBOL para incluir as instruções enviadas para o *solver* para atender as restrições do programa, bem como as instruções para fazer a execução dinâmica do programa.

O APÊNDICE 2 – Código-Fonte do Programa Controlador apresenta o código-fonte do programa controlador da técnica C4T usado para gerenciar as execuções do programa que está sendo testado.

O APÊNDICE 3 – Código-Fonte do Programa Exemplo Convertido Para Java apresenta o código-fonte do programa exemplo convertido de COBOL para Java.

O APÊNDICE 4 – Código-Fonte do Programa F*ixture* do FitNesse apresenta o código-fonte do programa fixture do FitNesse usado para fazer a ligação entre o servidor FitNesse e o programa que está sendo testado.

O APÊNDICE 5 – Pseudocódigo do Algoritmo do Instrumentador apresenta o pseudocódigo do algoritmo do processo de instrumentação usado na técnica C4T.

2 ESTADO DA ARTE

Apresenta-se o estado da arte da geração automática de casos de teste de software. Inicia-se pela discussão da importância dos casos de teste para migração de programas e por um breve resumo das técnicas de teste para descrever as abordagens propostas para a geração automática de casos de teste.

O teste de *software* é uma atividade que demanda cerca de 50% do tempo total do processo de desenvolvimento (BEIZER, 1995). Pesquisadores relatam várias abordagens para se automatizar essa tarefa, gerando automaticamente os casos de teste. Dentre as abordagens mais citadas (LIU e NAKAJIMA, 2011), (UTTING e LEGEARD, 2006), (XUN et al, 2010), (McMINN, 2004), (WATKINS, 2006), (LI e ZHANG, 2009), (KIEZUN, 2009) encontram-se as seguintes: testes exaustivos, testes aleatórios, especificação formal, baseado em modelo, execução simbólica, execução dinâmica, algoritmos evolutivos, inteligência de enxames (também chamada de inteligência coletiva) e testes direcionados.

São mostradas as características das técnicas de teste funcional e estrutural e seus respectivos métodos para geração automática de casos de teste e, por fim, são apresentadas algumas ferramentas e suas características para automatizar cada tipo de teste.

2.1 Importância dos Casos de Teste para Migração de Programas

Ao fazer a migração de um programa procura-se utilizar alguma estratégia que garanta que o novo programa manterá as mesmas funcionalidades do programa anterior. As principais estratégias para migração de programas são: geração manual do novo código-fonte do programa, geração automática dos casos de testes do programa e conversão automática de código-fonte para a linguagem desejada. Se o novo programa tiver um resultado igual a partir desses casos de teste entende-se que ele tem funcionalidades (ou requisitos) equivalentes ao programa anterior pois, de acordo com Martin e Melnik (2008), os casos de teste podem substituir os requisitos de um programa.

A primeira estratégia, geração manual do novo código-fonte do programa, é a mais utilizada devido às poucas ferramentas de geração automática disponíveis. Também é a mais trabalhosa pois exige um grande esforço de entendimento do programa anterior para possibilitar a reescrita do código-fonte para a nova linguagem.

A segunda estratégia, geração automática dos casos de teste, busca garantir a qualidade do novo programa por meio dos casos de teste gerados para o programa anterior.

A terceira estratégia aplicada é a migração de programa utilizando a conversão automática de código-fonte de uma linguagem para outra. Essa técnica foi relatada por Sneed (2010), que criou uma ferramenta para conversão de programas escritos em linguagem COBOL para a linguagem Java. Apesar de os resultados da migração utilizando a ferramenta proposta por Sneed (2010) mostrarem-se pródigos, o autor relata que para elaborar a ferramenta foi utilizada uma equipe composta por 28 pessoas durante um período de 3 meses. Além disso, essa técnica não poderia ser utilizada caso se desejasse migrar os programas de COBOL para outra linguagem que não seja Java, limitando a aplicação da ferramenta.

A Figura 1 mostra essas estratégias de migração. No primeiro fluxo o programa original é recodificado de forma manual ou automática, e são gerados manualmente os casos de teste após a recodificação. No segundo fluxo os casos de teste são gerados automaticamente a partir do programa original. Havendo geração automática ou manual dos testes são feitos testes de aceitação no programa migrado. Caso os testes de aceitação não sejam validados o fluxo volta para a etapa de recodificação e são feitos novos testes de aceitação até que a aceitação tenha sido feita e gerada a versão final do programa migrado. Isso mostra que, independentemente da estratégia de migração ou da linguagem usada para conversão de programas, o papel dos testes continua sendo imprescindível no processo de desenvolvimento de software (MYERS e SANDLER, 2004).

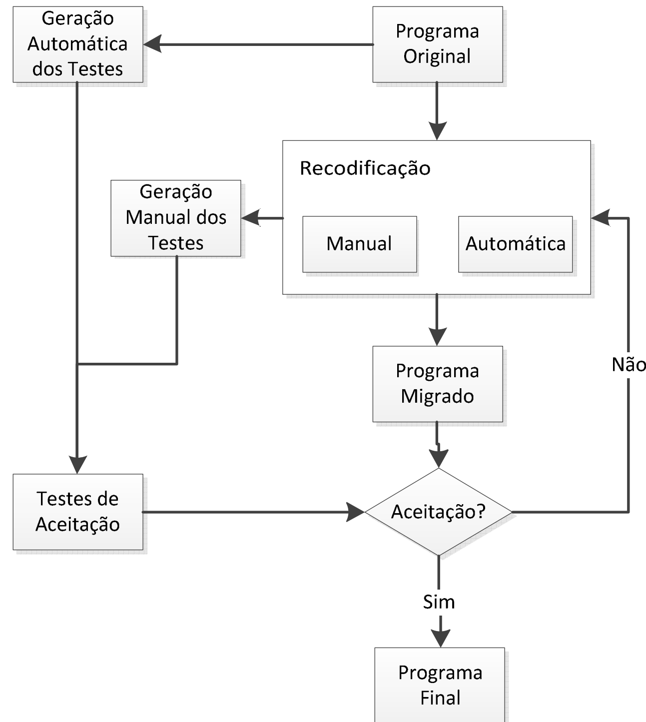


Figura 1 - Estratégias para Migração de Programas
Fonte: elaborado pelo autor

A atividade de geração manual de casos de teste a partir da leitura do código-fonte de um programa é particularmente propensa a erros em programas escritos em COBOL, devido às características intrínsecas dessa linguagem. Nela é possível escrever programas em que um simples ponto final, em uma linha de código, pode significar o término de um bloco inteiro do programa (VEERMAN, 2004). Se a pessoa que está lendo o código-fonte e gerando os casos de teste não percebe algum desses pontos finais pode gerar casos de teste que não refletem as situações reais do programa.

A Figura 2 mostra um trecho de código em que o valor da variável “taxa” é sempre multiplicado por 2, independentemente do programa entrar no trecho de decisão (IF) da linha 2, pois o caracter de ponto final ao término da linha 3 indica o término do trecho de decisão (sentença). Isso faz com que a variável “taxa” tenha o conteúdo “2” ao término desse trecho quando a variável “UF” for diferente de “SP”, ou que a variável “taxa” tenha o conteúdo “10” ao término desse trecho quando a variável “UF” for igual a “SP”.

```
1. MOVE 1 TO TAXA.
2. IF UF = 'SP' THEN
3.    MOVE 5 TO TAXA.
4. COMPUTE TAXA = TAXA * 2.
```

Indicação de término do trecho de decisão. Corresponde ao END-IF.

Figura 2 - Trecho de Programa Cobol com ponto-final
Fonte: Elaborado pelo autor

A Figura 3 mostra um trecho de código bem semelhante ao trecho da Figura 2, com a única diferença de que não há o caracter de ponto final ao término da linha 3. Isso faz com que a variável "taxa" tenha o conteúdo "1" (em vez de "2" do trecho da Figura 2) ao término desse trecho quando a variável "UF" for diferente de "SP.

```
1. MOVE 1 TO TAXA.
2. IF UF = 'SP' THEN
3.    MOVE 5 TO TAXA
4. COMPUTE TAXA = TAXA * 2.
```

Indicação de término do trecho de decisão. Corresponde ao END-IF.

Figura 3 - Trecho de Programa Cobol sem ponto-final
Fonte: Elaborado pelo autor

Veerman e Verhoeven (2006) citam outros aspectos da linguagem COBOL que a fazem propensa a erros, tais como: comando de desvio (declaração "GO TO") e o comando PERFORM, que pode ter resultado diferente dependendo do compilador. Eles criaram uma ferramenta para detecção desses defeitos, que chamaram de "campos-minados" dos programas escritos em COBOL.

Por exemplo, o comando de desvio "GO TO" indica que o controle do fluxo do programa deve ser direcionado para o rótulo (*label*) indicado no comando. A Figura 4 mostra um exemplo de programa que utiliza o comando GO TO. Quando esse comando é executado, o programa executa o seu fluxo de controle para o *label* com o nome LABEL-EXIT.

```
PROCEDURE DIVISION.
MAIN SECTION.
LABEL1.
    PERFORM LABEL2 THRU LABEL-EXIT
    STOP RUN.
LABEL2.
    DISPLAY 'EXECUTING LABEL2'
    IF FLAG
      GO TO LABEL-EXIT
    ELSE
      DISPLAY 'FLAG NOT SET'
    END-IF.
LABEL-EXIT.
    EXIT.
```

Desvia o fluxo de execução para o label indicado

Figura 4 - Exemplo de Comando GO TO
Fonte: adaptado de (VEERMAN e VERHOEVEN, 2006)

Tais práticas de programação podem, eventualmente, causar problemas quando outro programador, que não está familiarizado com tal lógica, precisa modificar o código, ou quando o código é migrado para outro ambiente e o comportamento da aplicação torna-se completamente diferente (VEERMAN e VERHOEVEN, 2006).

2.2 Técnicas de Teste

As técnicas de teste podem ser divididas em dois grandes grupos: testes funcionais e testes estruturais.

2.2.1 Teste Funcional (caixa-preta)

No teste funcional, ou teste de caixa-preta, o código-fonte e as estruturas internas não são visíveis ao engenheiro de testes. Ele é concentrado em encontrar circunstâncias nas quais o programa não se comporta de acordo com suas especificações. O conjunto de dados utilizados como entrada neste tipo de teste é criado a partir de documentos de especificação (MYERS e SANDLER, 2004).

2.2.2 Teste Estrutural (caixa-branca)

No teste estrutural as operações e estruturas internas são visíveis e são o principal foco deste tipo de teste. O conjunto de dados de entrada é criado visando testar estruturas específicas do *software*. A desvantagem da técnica de caixa de caixa branca é que ela não analisa se a especificação está certa pois se concentra apenas no código fonte e não verifica a lógica da especificação (LEWIS, 2008).

Nos testes estruturais os principais caminhos podem ser escolhidos e exercitados através de técnicas de geração de dados de entrada, que forçam o fluxo de controle a percorrer o caminho a ser exercitado. Além da forma tradicional, este pode ser combinado ao teste funcional para prover uma abordagem que verifica a interface e o funcionamento interno.

Por mais minuciosos que sejam, os testes funcionais não são capazes de realizar verificações que evitem alguns tipos de defeitos que são encontrados nos testes estruturais, como por exemplo, o fato de que os caminhos com as menores probabilidades de serem executados possuem as maiores chances de conterem um erro. Processos mais comuns tendem a ser bem entendidos, por isso possuem menor probabilidade de conterem um erro (TAKAKI, 2009),.

2.3 Tipos de Testes Funcionais

Os testes funcionais visam garantir a conformidade do *software* com seus requisitos. Os casos de teste são criados de acordo com as especificações, sem se preocupar com as estruturas ou *design* internos. A base da criação de testes funcionais é particionar os possíveis comportamentos do programa em um número finito de classes, onde se espera que cada classe esteja funcionalmente correta. Um erro ocorre quando um programa não funciona como o esperado, ou seja, um defeito não é apenas causado por erros de programação; a não conformidade com requisitos também é caracterizada como um erro (TAKAKI, 2009). Segundo Myers e Sandler (2004), os seguintes tipos de teste se enquadram nesta abordagem:

Análise de Valores-Limite (*boundary-value analysis*). Exercita os limites do tipo de dados de entrada. Nesse tipo de teste um ou mais valores devem ser

selecionados para representar os extremos da classe e, em vez de se preocupar apenas com valores de entrada, os valores de saída também são verificados. É necessário um bom conhecimento do problema para criar testes de limite. Se o domínio dos valores de entrada de um módulo é [–1.0,+1.0], devem ser criados testes que utilizam valores como –1.0, 1.0, –1.001 e 1.001 (TAKAKI, 2009).

Grafo de Causa e Efeito. Explora uma das deficiências da análise de valores-limite que é a combinação de valores de entrada. Uma situação de erro pode ocorrer após uma sequência de valores de entrada. Esta situação poderia não ser detectada pelas outras técnicas. As possíveis combinações podem tender a infinito e, ao se escolher um subconjunto de condições, pode acarretar em um teste ineficaz. Utiliza uma notação similar à lógica Booleana de circuitos digitais para representar o grafo. Este grafo é utilizado para a escolha da sequência de valores de entrada (TAKAKI, 2009).

Métodos de Estimativa de Erro (*Error-guessing methods*). É um processo *ad hoc* e intuitivo de escrita de casos de teste, onde a experiência prévia do engenheiro de testes é utilizada. A ideia básica é identificar e listar os possíveis erros, ou possíveis situações de erro, e escrever casos de teste baseados nesta lista (TAKAKI, 2009).

2.4 Tipos de Testes Estruturais

Os testes estruturais verificam o funcionamento interno de um *software*. Existem diversas técnicas que seguem a abordagem de teste estrutural. Estas técnicas devem ser utilizadas em conjunto ou em momentos diferentes, pois uma técnica não substitui a outra. Segue abaixo uma lista das principais técnicas de testes estruturais:

- **Teste de Comando de Decisão.** É a forma mais básica e intuitiva de testar o fluxo de controle de um *software*. Cada comando deve ser executado pelo menos uma vez, seguindo a ideia de que um defeito é revelado quando a região de código que contém o defeito é executada. Como visto anteriormente, não é capaz de revelar todos os defeitos sem criar o estado específico em que se manifesta o defeito (PEZZÈ e YOUNG, 2008).

• **Teste de *Branch.*** O teste de *statement*, mesmo com uma cobertura total, pode não cobrir todos os *branches*. Cada *branch* deve ser executado pelo menos uma vez. Os testes cobrem, pelo menos, uma ramificação do grafo de fluxo de controle do software. Em situações em que uma ramificação nunca é executada, por realmente ser inalcançável, o teste não satisfaz o critério de adequação deste teste (PEZZÈ e YOUNG, 2008).

• **Teste de Condição**. Exercita as condições lógicas contidas em um módulo de um *software*. Teste de *branch* enxerga apenas as ramificações, enquanto que teste de condição enxerga as estruturas lógicas que compõem as condições dos *branches*. Por essa característica, alguns defeitos podem escapar, pois o teste de *branch* poderá exercitar apenas as ramificações, sem levar em conta que a condição pode conter um defeito (PEZZÈ e YOUNG, 2008).

• **Teste de Caminho.** Visa cobrir os possíveis caminhos de um *software*. É realizado criando-se entradas que resultem em um determinado caminho. Alguns defeitos podem se manifestar após uma sequência de decisões, ou seja, um caminho específico do software. Programas que contenham *loops* podem gerar um número infinito de caminhos, o que torna este critério difícil de ser totalmente coberto. É importante observar que um aumento na cobertura de caminhos não necessariamente acarreta um aumento na cobertura de *statements* (PEZZÈ e YOUNG, 2008).

• **Teste de *Loop.*** Variação do teste de caminho onde o principal objetivo é testar os *loops*. Para que isto seja realizado é estabelecido um critério de parada para que exista um número finito de caminhos (PEZZÈ e YOUNG, 2008).

• **Teste de Chamada de Procedimento/Método.** Visa cobrir o fluxo de controle entre procedimentos ou métodos, útil em testes de integração ou testes de sistemas. Se o teste unitário foi bem sucedido, os defeitos que restam serão de interface. Esta é a principal razão para se utilizar esta técnica na fase de integração. Nesta fase de teste o principal foco deve ser a interface entre módulos, tornando fundamental o uso de testes de chamada de procedimento (PEZZÈ e YOUNG, 2008).

2.5 Geração automática de dados de teste

O uso de ferramentas para a automação de testes tem grande potencial de aumentar a produtividade, reduzindo tempo e riscos, além de melhorar a qualidade do software e do processo (BURNSTEIN, 2003).

A atividade de teste de software evoluiu muito desde a década de 1950 e atingiu um nível de maturidade considerável nas últimas décadas, sendo que a atividade de geração automática de casos de teste é considerada atualmente como uma das formas mais promissoras para aprimorar o processo de teste (LEWIS, 2008).

A Figura 5 mostra essa evolução nas atividades de testes. Na década de 1950 o teste de *software* era definido como “o que os programadores não fazem para encontrar falhas em seus programas”. Na década de 1960 eram feitos testes exaustivos com a maior quantidade possível de dados de entrada. Em meados da década de 1960 o teste de *software* era definido como “o que é feito para demonstrar a correção do programa”, enquanto que em meados da década de 1970 o objetivo era justamente o contrário: provar que o programa não funcionava, ou seja, encontrar situações que provoquem falha na execução do programa. Na década de 1980 o teste passou a ser visto como forma de prevenção de defeitos e a atividade de teste passou a ser vista como um processo. Nessa década surgiram várias ferramentas para automação de testes. Na década de 1990 o teste passou a ser projetado para ser feito nas fases iniciais do desenvolvimento, e surgiram ferramentas de automação mais avançadas. Em meados da década de 1990 a atividade de teste passou a ser mais aderente à Internet, que requeria processos mais ágeis. Na década de 2000 a atividade de teste passou a integrar a automação da otimização do negócio da empresa.

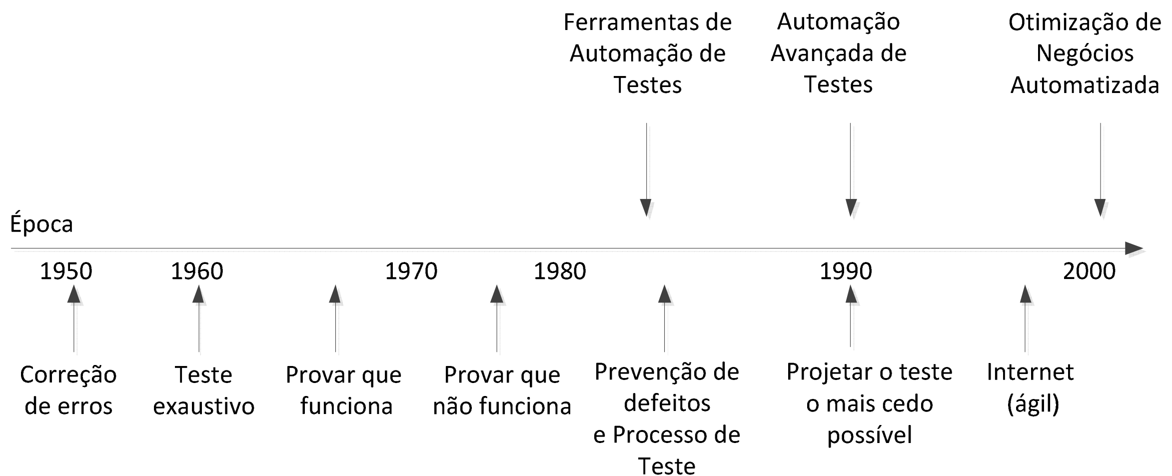


Figura 5 - Evolução da atividade de teste de software
Fonte: adaptado de (LEWIS, 2008)

A geração automática de casos de teste engloba tanto a geração dos dados de entrada quanto dos oráculos de teste. De acordo com Binder (2000), um oráculo é uma fonte confiável de resultados esperados para os testes, podendo ser desde a especificação do programa, uma tabela de exemplos ou simplesmente o conhecimento do programador a respeito de como o software deve funcionar.

Para a automação da geração dos dados de entrada, é importante considerar dois aspectos: o critério de teste e a ferramenta para a geração dos dados. O critério avalia a qualidade dos dados gerados e fornece informações a respeito do final do processo de geração. A avaliação de qualidade está diretamente relacionada à satisfação dos requisitos de teste definidos pelo usuário. Satisfazer um requisito implica a existência de um dado que exercite uma ou mais funções ou instruções do *software* relacionados ao requisito (ABREU, 2006).

Em virtude da dificuldade de se fazer isto manualmente, algoritmos para a geração de dados de testes são utilizados para a automação deste processo. Estes algoritmos são as ferramentas para a geração dos dados, e a geração pode ser guiada utilizando basicamente dois tipos de análise: análise de fluxo de dados e a análise de fluxo de controle (CFG – *Control Flow Graph*) (ABREU, 2006).

A análise de fluxo de dados tem como foco as interações entre as definições de variáveis e suas referências no programa. O propósito desse tipo de análise é determinar as definições de todas as variáveis do programa e os usos que podem ser afetados por estas definições (ABREU, 2006).

A análise de fluxo de controle foca-se na ordem na qual as instruções do programa são executadas. Uma instrução de fluxo controle é aquela em que, ao ser executada, pode provocar uma mudança no fluxo de controle do programa. As instruções de fluxo de controle podem ser categorizadas em: instruções de salto, escolha, repetição, subrotinas e de parada. Nesse tipo de análise os vários fluxos do programa são representados através de caminhos ao longo do CFG. Um caminho é uma sequência finita de nós conectados por arestas, sendo que um caminho completo deve obrigatoriamente começar pelo bloco inicial básico e terminar no bloco final. (ABREU, 2006).

A Figura 6 mostra um exemplo de código-fonte de um algoritmo e seu respectivo CFG. Observa-se que até um programa com poucas linhas de código pode ter um fluxo de controle com vários caminhos possíveis.

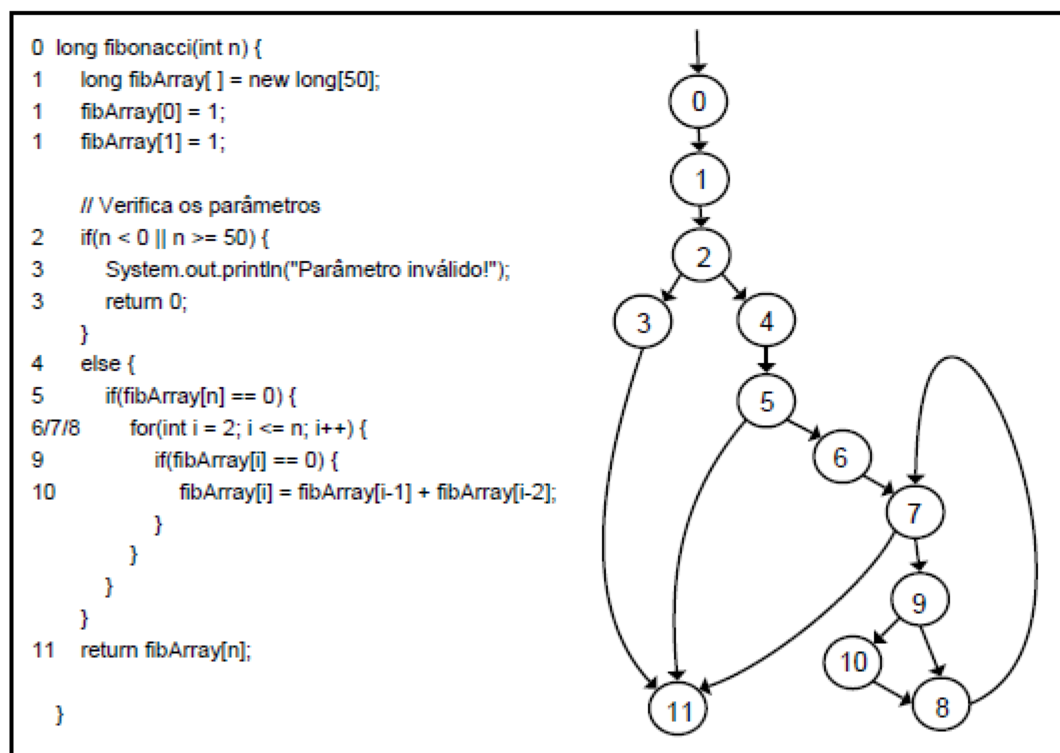


Figura 6 - Código-fonte e gráfico de fluxo de controle (CFG) de um programa
Fonte: ABREU (2006)

2.6 Abordagens para geração de testes

Buscando aumentar a eficiência e eficácia da geração dos casos de teste deve-se seguir uma abordagem automática. Assim como os tipos de teste, as abordagens de teste também podem ser divididas em funcionais e estruturais.

A Figura 7 mostra as técnicas de geração de casos de teste abordadas neste trabalho. As técnicas foram divididas em 2 grupos principais: testes funcionais (no qual não há acesso ao código fonte do programa) e testes estruturais (no qual há acesso ao código-fonte do programa). Dentro do teste funcional temos o teste baseado na especificação formal (que usa um modelo matemático para descrever o comportamento do programa) e o teste de modelo (baseado nos diagramas e modelos do sistema). Na parte de teste estrutural há o teste de execução simbólica (que utiliza as restrições do programa), o teste aleatório (que gera as entradas de teste de forma aleatória), o teste de execução dinâmica (para o qual as entradas são geradas de forma iterativa) e o teste direcionado (que engloba a execução simbólica e a execução dinâmica de forma conjunta). O teste aleatório pode ser dividido em teste de busca meta-heurística (que reduz a busca do dado de teste a um problema de otimização) e a teste de pares (que procura gerar a melhor combinação de entradas do programa). O teste de busca meta-heurística, por sua vez, pode ser dividido de acordo com o algoritmo utilizado, que são: otimização por colônia de formigas (que busca encontrar os caminhos mais comuns do programa), subida de encosta (baseado no critério de melhoria contínua), têmpera simulada (que utiliza a abordagem de subida de encosta com reinicialização) e os algoritmos genéticos e evolutivos (que se baseiam na teoria da evolução).

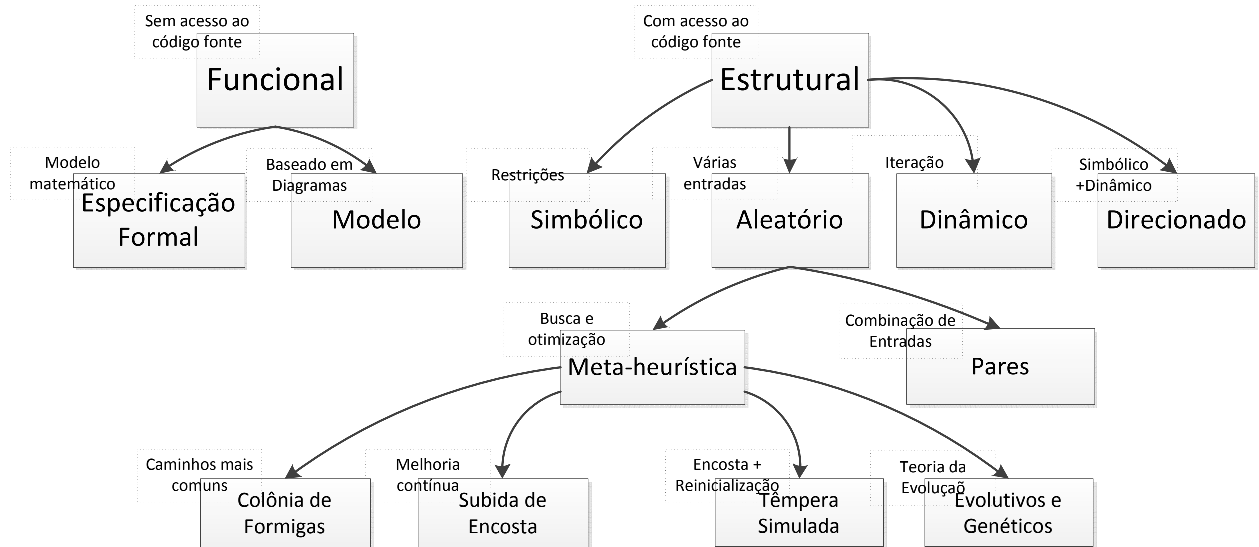


Figura 7 - Visão geral das técnicas de geração de casos de teste
Fonte: elaborado pelo autor

2.6.1 Abordagens para testes funcionais

A seguir são apresentadas as principais abordagens para os testes funcionais.

2.6.1.1 Especificação Formal

De acordo com Pimenta (1998), os métodos formais possibilitam o desenvolvimento e a verificação de software através de uma notação matemática rigorosa. Este rigor matemático permite que aspectos de consistência, completeza e exatidão possam ser avaliados sistematicamente. Para serem consistentes, fatos declarados em uma especificação não devem se contradizer em outro lugar. A consistência é garantida matematicamente, provando que fatos iniciais podem ser formalmente traçados usando-se regras de inferência.

Baseado no conceito de ter-se a semântica definida com rigor matemático, os métodos formais permitem que a modelagem de um sistema, e suas propriedades, sejam definidas precisamente e sem ambigüidades. Os modelos de sistema formal provêem a entrada de ferramentas para vários tipos de análise dinâmica e estática, geração automática de casos de testes, verificação de modelos e verificação formal (SCHAEFER e HÄHNLE, 2011).

Liu e Nakajima (2011) propõem um método chamado “método de vibração” para geração de casos de teste baseado na especificação formal, e utilizam um estudo de caso no qual esse método mostra uma taxa de cobertura de caminhos de 92% em comparação com o método de pares que atingiu uma taxa de 53%.

Nunes, Moreira e Ribeiro (2011) relatam que os artigos sobre métodos formais no Simpósio Brasileiro de Engenharia de Software (SBES) diminuíram ao longo dos anos, e sugerem que esse fato se deve à ideia de que o uso de métodos formais na prática diária da indústria de desenvolvimento de software é inviável devido, primeiro, à curva de aprendizado necessária para a aquisição das habilidades necessárias e, segundo, devido à falta de ferramentas *mainstream* para dar suporte ao trabalho de desenvolvimento rigoroso e fundamentado matematicamente.

Durante as últimas décadas, algumas ferramentas que utilizam métodos formais migraram esse assunto do ambiente acadêmico para o ambiente comercial. Algumas ferramentas que utilizam métodos formais são: Costa, ESC/Java, JPF, KeY, KIV, Krakatoa, PEX, SLAM, Specware, Spec#, Terminator e Vcc, que utilizam semântica formal para linguagens tais como: C, C#, C++, Java e COBOL (SCHAEFER e HÄHNLE, 2011), (LANO, 1995).

2.6.1.2 Teste baseado em modelo

De acordo com Utting e Legeard (2006), é possível automatizar um projeto de testes funcionais com a técnica de Testes Baseados em Modelos, na qual os testes são gerados a partir de um modelo simplificado do sistema, focalizando apenas nos aspectos que precisam ser testados. Alguns modelos são baseados em sistemas de transição, como máquinas de estados finitos e máquinas de estados da *Unified Modeling Language* (UML) e modelos de pré e pós-condições, encontradas, por exemplo, na linguagem *Object Constraint Language* (OCL). A notação UML apresenta um conjunto de diagramas, como o de caso de uso, de classes, de atividades, de interação, de estado, dentre outros (OMG, 2010). A OCL é uma linguagem formal para descrever expressões nos modelos em UML utilizada para

especificar restrições técnicas ou de negócio nas classes definidas em um modelo (OMG, 2006).

Hartmann (2009) criou uma ferramenta que se destaca por possuir uma linguagem de domínio específico com um conjunto próprio de elementos visuais para a definição de um modelo. Ele comenta que uma linguagem de domínio específico é uma linguagem adaptada a um tipo específico de problema, o que a torna simples e pequena, se comparada com uma linguagem genérica que cobre todos os tipos de tarefas.

Fröhlich e Link (2000) apresentaram uma transformação de uma especificação de caso de uso em um modelo baseado em máquina de estados, e um algoritmo para a geração dos testes a partir deste modelo, fazendo a definição manual dos dados de entrada e valores esperados nos casos de testes.

Dallmeir et al (2012) apresentaram uma ferramenta, chamada TAUTOKO, que gera os casos de teste de um programa com base tanto na execução do programa como também na especificação do programa.

Mingsong, Xiaokang e Xuandong (2006) adotaram o diagrama de atividades da UML para a especificação do modelo e criaram um algoritmo que percorre os seus caminhos para a geração dos casos de testes no qual todo o processo é automatizado.

Todavia, para gerar os testes baseados em modelo uma pessoa precisa ter um bom conhecimento para abstrair o domínio e projetar um modelo, conhecimento do negócio e conhecimentos técnicos, como uma linguagem de programação ou uma ferramenta voltada para testes. Isso exige um custo significativo em treinamento e tempo de aprendizado (UTTING e LEGEARD, 2006).

2.6.2 Abordagens para testes estruturais

A seguir apresentam-se as principais abordagens para os testes estruturais:

2.6.2.1 Aleatório

Outra forma utilizada para se testar um módulo é selecionar entradas de maneira aleatória, com a expectativa de que essas entradas revelem falhas críticas no *software*. No entanto, a criação e seleção de testes que tenham potencial para revelar falhas exige um entendimento das funcionalidades do *software*, domínio de entradas e saídas, bem como do ambiente de uso do código em testes (ABREU, 2006). Por essa razão a abordagem aleatória não é muito eficiente para testar sistemas complexos, uma vez que os dados de teste tem poucas chances de exercitar comportamentos muito específicos do software (MICHAEL, MCGRAW e SCHATZ, 2001).

Xuan et al (2010) elaboraram a ferramenta *Walktest*, que é baseada em um algoritmo aleatório para resolver o problema de geração de casos de teste fazendo uso de chamadas randômicas iterativas a fim de buscar a solução ótima. O algoritmo foi comparado com outros algoritmos existentes e esse algoritmo mostrou-se com melhor *performance* em termos de tempo de execução e taxa de cobertura.

2.6.2.2 Busca Meta-heurística

Uma solução promissora para contornar os problemas da abordagem aleatória é o uso de meta-heurísticas de busca, que são o uso de uma combinação de heurísticas de maneira eficiente no intuito de resolver uma classe genérica de problemas computacionais (ABREU, 2006).

Essas meta-heurísticas são problemas de otimização, e o uso destes métodos implica a conversão do problema de geração de dados de teste para um problema de otimização, ou seja, converter os critérios de teste para funções-objetivo, que comparam soluções e as avaliam em relação ao objetivo da busca (por exemplo: gerar um dado de teste que execute a instrução “a = b” de um *software*). Em suma, a função-objetivo conduz a busca pelos dados de teste para as regiões mais promissoras do espaço de busca, aumentando as chances de gerar um dado que atenda ao critério definido (ABREU, 2006).

O uso de meta-heurísticas para geração de casos de teste é útil em função de que a obtenção de conhecimento ocorre durante a sua execução do programa ou previamente, permitindo a adequação dos dados até que seja atingindo o grau de qualidade desejado (PINHEIRO, 2012), ou seja, busca-se uma solução ótima que atenda o caso de teste desejado para o programa que está sendo testado.

A seguir são mostradas algumas meta-heurísticas mais comumente utilizadas.

2.6.2.3 Subida de Encosta (*Hill Climbing*)

A Subida de Encosta (*Hill Climbing*) é uma heurística que busca por uma combinação ótima de atributos, variando um valor de cada atributo por vez, com o intuito de encontrar o melhor resultado para um problema. Enquanto há melhora na solução esse procedimento é repetido. Casa não haja mais uma combinação que melhore a função objetivo do problema a busca termina (McMINN, 2004).

O nome “subida de encosta” é derivado do gráfico da função-objetivo em relação às configurações possíveis do problema de entrada. Este gráfico possui picos, que podem ser ótimos locais ou ótimos globais. Há a possibilidade também da existência de planícies, onde o valor da função-objetivo não varia com a mudança de configuração dos atributos (Figura 8). No algoritmo, inicialmente é dada uma solução inicial e a busca tenta subir o gráfico da função objetivo tentando alcançar um pico. Porém caso este pico seja um ótimo local, ou uma planície, o algoritmo não consegue mais continuar a busca, e não atinge o ótimo do problema (CARVALHO et al, 2011).

Figura 8 - Gráfico da Subida de Encosta
Fonte: (CARVALHO et al, 2011)

O algoritmo de Subida de Encosta basicamente executa uma busca local sem nenhuma técnica que utiliza algum conhecimento sobre a busca. Uma solução inicial é gerada e é definida como a solução corrente. Após isto o algoritmo repete o seguinte procedimento: é gerada uma vizinhança em relação à solução corrente. Em seguida, é escolhida a melhor solução desta vizinhança. Caso esta solução seja melhor que a solução corrente, ela é atribuída como a solução corrente (CARVALHO et al, 2011)

Papadakis e Malevris (2011) propuseram uma abordagem para geração de casos de testes utilizando mutantes juntamente com o algoritmo de subida de encosta, que foi comparada com o teste aleatório e mostrou-se com performance superior. Mutante é um critério de teste que busca inserir um conjunto de dados de entrada que provoque um erro no SUT (DEMILLO, LIPTON e SAYWARD, 1978).

2.6.2.4 Têmpera Simulada (*Simulated Annealing*)

O algoritmo de têmpera simulada (*simulated annealing*), também chamado de recozimento simulado, é similar ao algoritmo de subida de encosta. A diferença reside no fato de que a têmpera simulada permite movimentos menos restritos no espaço de busca (MCMINN, 2004).

O nome “têmpera simulada” é originado a partir da analogia com o processo químico de “têmpera”, que é resfriar ou aquecer um material para moldá-lo (MCMINN, 2004).

A temperatura é resfriada de acordo com o resultado desejado. Inicialmente a temperatura é alta a fim de permitir um movimento livre no espaço de busca, e então a dependência com a solução inicial é perdida. À medida que a busca progride, a temperatura diminui. Entretanto, se o resfriamento é muito rápido não há espaço de busca suficiente para ser explorado, e as chances de a busca se tornar presa a uma solução local ótima é aumentada (McMINN, 2004).

Bin et al. (2007) apresentam um algoritmo chamado têmpera simulada modificado baseado no critério de cobertura de testes orientado a caminhos, e apresenta um estudo de caso de um programa de triângulo no qual é comparada a cobertura do programa utilizando 3 abordagens diferentes (algoritmo genético normal, algoritmo randômico e o próprio algoritmo de têmpera simulada), e o algoritmo de têmpera simulada foi o que apresentou a melhor taxa de cobertura com a menor quantidade de iterações.

2.6.2.5 Algoritmos evolutivos

Dentre os algoritmos para resolver o problema de otimização, os chamados algoritmos evolutivos (no campo da computação evolutiva) são bastante usados (WATKINS, 2006) (MANSOUR; SALAME, 2004) (DIAZ; TUYA; BLANCO, 2003). Eles são métodos de busca estocástica, isto é, possuem em sua solução certo grau de aleatoriedade. São baseados nos princípios e modelos da evolução natural. A ideia por trás desses algoritmos é evoluir uma população de indivíduos (candidatos a solução para o problema) por meio de competição, recombinação e mutação, de forma que a aptidão média da população (qualidade das soluções) seja sistematicamente melhorada dentro do ambiente (problema) em questão (ABREU, 2006).

A utilização de algoritmos evolutivos é crescente pois alguns métodos de otimização clássicos geralmente não apresentam um desempenho satisfatório quando submetidos a problemas reais (ABREU, 2006). Além disso já foi observado

que problemas cujas características envolvam relação não-lineares, caos, variações temporais e função que não seja facilmente descrita são resistentes a abordagens de otimização clássicas (EIBEN; SMITH, 2003).

A Figura 9 mostra um diagrama da geração de casos de teste utilizando algoritmo genético. O primeiro passo é formular o teste de objetivo como problema de busca, depois são definidas as condições dos cromossomos, que para a disciplina de teste são os dados de teste do programa. Defini-se, então, a função-objetivo e em seguida os operadores e configurações do algoritmo genético. Após isso é gerada a população inicial de dados de teste e avaliado o objetivo. Se o dado de teste atende o critério de objetivo definido o programa é interrompido, caso contrário são selecionados “pais” para a nova geração, que são os dados de entrada (“indivíduos”) que mais próximo chegaram da solução. São recombinados os “pais” com a nova geração dos dados de teste e é feita a mutação de alguns dados de teste para gerar uma nova população de dados de teste. A avaliação de atendimento ao critério de objetivo definido é verificada novamente com essa nova população de dados de teste e o ciclo continua até que se gere uma população que atenda o objetivo de teste (ALI et al, 2008 apud PINHEIRO, 2012).

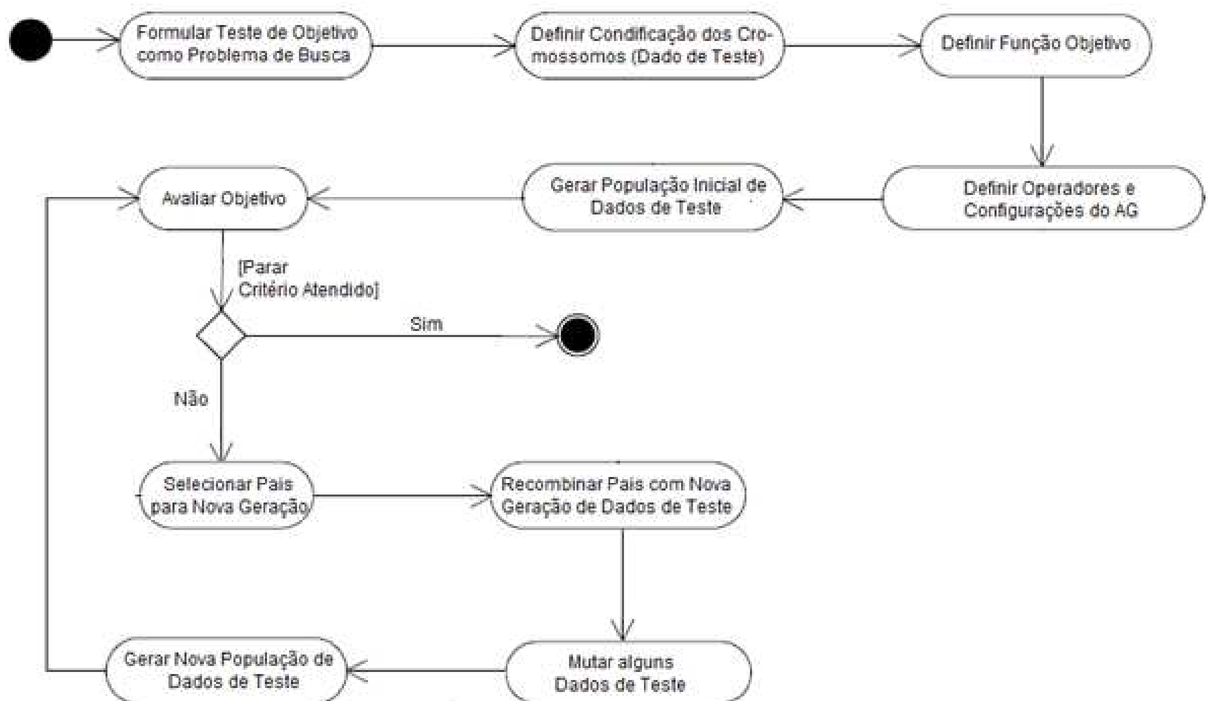


Figura 9 - Geração de dados de teste utilizando algoritmo genético

Fonte: (ALI et al, 2008 adaptado por PINHEIRO, 2012)

2.6.2.6 Inteligência de Enxames

Inspirados em processos naturais, foram desenvolvidos os modelos computacionais baseados no conceito de inteligência de enxames. Neles um agente simples não consegue realizar determinadas tarefas sozinho, mas quando interage com outros indivíduos emerge um sistema auto-organizado capaz de desenvolver tarefas mais complexas (LOPES, 2006). Alguns exemplos desse conceito na natureza são: as abelhas na construção e estruturação de uma colméia; os cupins que constroem complexos sistemas de túneis, as formigas que podem encontrar caminhos diretos quando buscam por alimentos.

Isso levou à criação de uma nova área na inteligência computacional, a inteligência de enxames (*swarm intelligence*). Dentro do contexto da inteligência de enxames, as técnicas que vem sendo aplicadas com bastante sucesso em problemas de otimização são: a Otimização por Colônia de Formigas (*Ant Colony Optimization* - ACO) criada por Colorni, Dorigo e Maniezzo (1992); e a Otimização por Enxame de Partículas (*Particle Swarm Optimization* - PSO), que utilizam o comportamento social de algumas espécies para resolução de problemas complexos (GONÇALVES, 2011).

Os algoritmos de inteligência de enxames (como o algoritmo de otimização por enxame de partículas) compartilham várias similaridades com a computação evolutiva, principalmente na parte de algoritmos genéticos. Em ambos os algoritmos o sistema é inicializado com uma população de soluções randômicas e é feita uma busca por uma solução ótima por meio da atualização das gerações dos indivíduos. Entretanto, ao contrário do algoritmo genético, o algoritmo de otimização por enxame de partículas não tem os operadores de evolução tais como recombinação e mutação. Na otimização por enxame de partículas as soluções em potencial, chamadas partículas, percorrem o espaço do problema buscando a partícula que otimiza o problema, ou seja, a partícula que é a solução do problema (LAZINICA, 2009).

Uma abordagem utilizando algoritmo de inteligência de enxames foi utilizado por Li e Zhang (2009), que usaram um método, chamado pelos autores de “método de todos os caminhos”, e aplicaram o algoritmo de otimização por enxame de partículas para geração de todos os casos de teste de dois programas: o primeiro

era um programa simples para definir o tipo de um triângulo e o segundo era um programa para fazer uma busca binária, que tinha uma estrutura um pouco mais complexa pois envolvia *loops*.

A execução do programa de triângulos foi comparada com outro método proposto pelos mesmos autores que gera um único caminho de teste (LI e ZHANG, 2008), e os resultados mostraram que o método de todos os caminhos foi entre 35% e 70% mais eficiente do que o método de um único caminho. A maior eficiência foi obtida com 50 partículas, ao passo que a pior eficiência foi obtida com 100 partículas. Ao analisar esses dados observa-se que nesse método é preciso definir criteriosamente a quantidade de partículas pois nem sempre uma quantidade maior de partículas implica uma maior eficiência nos testes.

A execução do programa de busca binária também foi comparada com o método anterior dos mesmos autores e os resultados foram melhores do que o programa anterior: o método de todos os caminhos foi entre 54% e 68% mais eficiente do que o método de caminho único. A maior eficiência foi obtida com 20 partículas e a pior eficiência com 140 partículas.

Um fato que se observa na execução desse segundo programa, e que não foi reportado pelos autores, é que a eficiência com 100 partículas (66%) foi muito próxima da melhor eficiência (68%), mas no programa anterior essa quantidade de partículas tinha obtido eficiência muito pior (35%).

Os autores observaram que quando a velocidade da partícula é muito baixa ($\leq$ 16) ou muito alta ($\geq$ 20) a eficiência é pior, e por isso chegaram à conclusão que a velocidade de partícula adequada para o estudo de caso era entre 16 e 20 (LI e ZHANG, 2009).

A técnica denominada Otimização por Enxames de Partículas é uma abordagem estocástica, baseada em população que simula o processo comportamental de interação entre os indivíduos de um grupo (MEDEIROS, 2005). Essa técnica foi desenvolvida a partir da observação do comportamento de grupos de animais como pássaros, peixes, entre outros, que realizam interessantes tarefas de otimização na execução de atividades simples, como a busca por alimentos (KENNEDY e EBERHART, 1995). Com isso o PSO busca a simulação destas atividades naturalmente simples, porém matematicamente complexas.

Outra característica da otimização por enxame de partículas é que ela é um procedimento de otimização baseado no comportamento de grupos de organizações, como por exemplo, uma revoada de pássaros ou um cardume de peixes. Soluções individuais (cada membro em uma população) são consideradas partículas que evoluem e mudam suas posições no espaço de busca conforme sua própria experiência e também na de suas vizinhas, armazenando sempre a sua melhor posição visitada e a melhor posição visitada por suas vizinhas, logo, combinando métodos de busca local e global (KENNEDY e EBERHART, 1995).

Nessa abordagem cada indivíduo é dito ser uma possível solução para o problema investigado, sendo atribuído a cada indivíduo um valor que está relacionado à adequação da partícula com a solução do problema, denominada de *fitness*, que representa a posição do indivíduo no espaço de busca (ambiente); e também uma variável velocidade que representa a direção do movimento do indivíduo (partícula) (GONÇALVES, 2011).

Com o passar do tempo os indivíduos vão ajustando suas velocidades e posições em relação à melhor solução (melhor *fitness*) encontrada por ele próprio e também pela melhor solução do grupo, realizando este processo até que os indivíduos (partículas) encontrem o alimento (melhor solução global ou próxima dela). O valor de *fitness* é calculado com base na função objetivo que se deseja maximizar (ou minimizar) (GONÇALVES, 2011).

Apesar da utilização crescente dos algoritmos evolucionários, Whitley (2001) avalia que os algoritmos evolucionários não são adequados para a maioria dos problemas pois eles são conhecidos como “métodos fracos” (*weak methods*) na comunidade de inteligência artificial, ou seja, eles não exploram domínios de conhecimento específico. Eles são também conhecidos como métodos de busca cega (*blind search method*). De acordo com Whitley (2001), os métodos que exploram domínios de conhecimento específico tem uma performance melhor do que os métodos de busca cega.

2.6.2.7 Otimização por Colônia de Formigas

Otimização por colônia de formigas (ACO – *Ant Colony Optimization*) é o nome que se dá a uma família de algoritmos que seguem um padrão de funcionamento similar, semelhante ao de uma colônia de formigas reais quando estão em busca de comida. Existem diversos algoritmos diferentes que podem ser considerados como otimização por colônia de formigas, sendo que o primeiro deles foi o criado por Dorigo (1992) chamado *Ant System*. Esse algoritmo foi aplicado para geração dos casos de teste por Li, Zhang e Liu (2009).

2.6.2.8 Teste de Pares

Outra ferramenta tem uma abordagem diferente, a qual se baseia em vários conjuntos menores de dados e, a partir deles, são geradas combinações para se tornarem valores de entrada para um teste (UTTING; LEGEARD, 2006). Este abordagem é conhecida como teste de pares (*pairwise testing)*. Uma ferramenta desse tipo se encontra em um estudo realizado por Dalal et al. (1999). Foram definidos quatro casos de estudo para a avaliação da ferramenta. Concluíram que o ponto forte dessa abordagem é a facilidade com que os analistas de teste têm ao escrever o modelo e a rápida geração dos testes por causa de alguma mudança no *software*. Entretanto, existem dois pontos fracos: a necessidade de um oráculo e a exigência de conhecimentos técnicos de desenvolvimento que um analista de teste precisa ter. Essa abordagem é aplicável em sistemas que utilizam um modelo de dados que descrevem o comportamento do sistema.

2.6.2.9 Execução Simbólica

A abordagem de execução simbólica consiste em atribuir valores simbólicos às variáveis do *software* de forma a obter uma caracterização matemática e abstrata do que o *software* faz (ABREU, 2006). Ela consiste nas seguintes etapas: obter uma representação funcional dos caminhos do *software*, atribuir nomes simbólicos às

variáveis de entrada, e avaliar os caminhos a partir da interpretação das instruções e dos caminhos, de acordo com estes nomes simbólicos (STHAMER, 1996).

Meudec (2001) propôs um gerador de dados de testes baseado na execução simbólica de *software* utilizando a linguagem de programação de restrições lógicas (*Constraint Logic Programming*, CLP) como *framework*. Ele apresentou uma ferramenta chamada ATGen, que utiliza essa abordagem para alcançar uma cobertura de decisões de 100% em *software* escrito em SPARK Ada. Um dos problemas dessa abordagem é quando ela é usada para *software* com variáveis que dependem da execução de *loops*, variáveis dependentes de vetores, referências a ponteiros e chamadas de métodos. Tecnicamente, qualquer função computacional pode ser calculada sem o uso de vetores e ponteiros, mas a realidade mostra que os sistemas de *software* construídos raramente deixam de utilizar estes recursos de linguagem de programação. Um *software* que utiliza vetores e ponteiros pode ser, em teoria, executado simbolicamente, mas estas construções inevitavelmente complicam muito a aplicação da abordagem para a geração de dados (MICHAEL, MCGRAW e SCHATZ, 2001).

Outra ferramenta de geração de testes é a SYMEXLAN (KOUTSIKAS; MALEVRIS; SKORDALAKIS, 2000), que é uma linguagem de *script* padrão que pode ser incorporada em sistemas de teste de execução simbólica e pode ser usada independentemente da linguagem no qual foi escrito o programa, pois ela define os possíveis caminhos desse programa.

Além dos problemas com loops e vetores, Pinheiro (2012) cita que outro problema da execução simbólica é o alto consumo computacional necessário para utilização desse método.

2.6.2.10 Execução Dinâmica

Além das abordagens citadas acima, existe também a abordagem dinâmica, que trata as diversas partes do *software* como funções que são avaliadas por meio da execução do *software*. Na medida em que os dados são gerados, eles são avaliados e melhorados utilizando métodos de otimização de funções, que guiam a busca por dados adequados (ABREU, 2006).

O princípio dessa abordagem é que, se um requisito de teste não é satisfeito, os dados coletados durante a execução dos testes são úteis para determinar quais foram os dados que se aproximaram de satisfazer o requisito de teste e, esses dados podem ser alterados até que um deles satisfaça o requisito desejado como, por exemplo, cobrir uma determinada decisão (ABREU, 2006).

Nessa abordagem as informações necessárias para o funcionamento do método de otimização podem ser extraídas do *software* em teste (S*oftware under Test, SUT*) por meio da instrumentação, e com isso é possível saber se durante a execução do SUT foram ou não encontrados os dados de teste que atendam um ou mais requisitos de teste.

Essa abordagem tem sido objeto de vários trabalhos que tratam da geração automática de dados de teste (WATKINS, 2006) (MANSOUR; SALAME, 2004) (DIAZ; TUYA; BLANCO, 2003) pois ela tem a vantagem, em relação às demais abordagens, de reduzir o problema de geração de dados de testes para um problema de otimização, representado por um conjunto de restrições (quando existem) e uma função matemática (também chamada de função-objetivo) que deverá ser minimizada ou maximizada (ABREU, 2006).

2.6.2.11 Testes Direcionados

Os testes direcionados enumeram os caminhos de execução do programa, especificando caminhos alternativos como fórmulas lógicas e usando um solucionador de restrições para encontrar as entradas que correspondem aos caminhos alternativos. O solucionador de restrições é um programa que calcula as soluções para as fórmulas lógicas dadas como entrada (KIEZUN, 2009).

Os testes direcionados usam valores concretos bem como valores simbólicos como entrada e executam um programa de forma concreta e simbólica, por isso é chamado também de execução “concólica” (acrônimo das palavras **conc**reta e simb**ólica**). A parte de execução concreta na execução concólica constitui-se da execução normal do programa. A parte de execução simbólica na execução concólica coleta as restrições simbólicas dos valores de entrada simbólicos de cada ramificação de caminho encontrada ao longo da execução concreta (SEN, 2007).

No final da execução o algoritmo computa a sequência de restrições simbólicas correspondente a cada ramificação. O conjunto dessas restrições é chamado de restrições de caminho (*path constraint*). Todos os valores de entrada que satisfazem uma dada restrição de caminho irão explorar o mesmo caminho de execução (SEN, 2007).

Os testes direcionados primeiramente geram valores randômicos para entradas primitivas e valores nulos para entradas de ponteiros. Após isso o algoritmo faz os seguintes procedimentos em *loop*:

a) Executa o código com as entradas geradas;

b) No fim da execução uma restrição simbólica na restrição de caminho é negada e resolvida usando solucionadores de restrição para gerar uma nova entrada de teste que direciona o programa para um caminho de execução diferente; e

c) O *loop* é repetido com as novas entradas de teste geradas.

O *loop* continua até que o algoritmo explore todos os possíveis caminhos de execução distintos usando uma estratégia de busca de profundidade (SEN, 2007).

Todavia para algumas restrições simbólicas o solucionador de restrições pode não ter poder de computação suficiente para gerar os valores concretos que satisfaçam as restrições. Para direcionar essa dificuldade, tais restrições simbólicas são simplificadas por meio da substituição de valores simbólicos por valores concretos. Por causa disso, o teste direcionado somente é completado se tiver um oráculo que pode resolver todas as restrições de um programa, e além disso o tamanho e a quantidade de caminhos devem ser finitos. Em função de o algoritmo realizar execuções concretas, todos os erros inferidos são reais (SEN, 2007).

Dentre as técnicas de teste direcionado, Emmi, Majumdar e Sen (2007) propuseram uma ferramenta para geração dos casos de teste de uma aplicação com acesso a banco de dados. Essa ferramenta utiliza um solucionador de restrições para resolver restrições simbólicas envolvendo restrições de aritmética linear e restrições de *string.*

2.7 Solucionadores de restrições

Os solucionadores de restrições, também chamados de *solvers*, são programas independentes baseados na resolução de equações lógicas, ou seja, a partir de entradas (restrições/equações) a eles enviadas, retornam os valores de saída correspondentes. Eles também são chamados de SMT (*Satisfiability Modulo Theories*, módulo de teorias de satisfabilidade) (SOK, 2012).

Duas importantes propriedades dos solucionadores de restrições são solidez e plenitude. A propriedade solidez é requerida e garante que o solucionador não prova que “verdadeiro = falso” e ele apenas prova fórmulas verdadeiras. A propriedade plenitude é opcional e afirma que se uma fórmula é verdade então ela pode ser provada (SOK, 2012).

A Tabela 1 mostra um exemplo de um sistema de equações de números inteiros possível de ser resolvido por um *solver*. O sistema de equações “16x + 2y = 18” e “x + y = 2” são convertidos para a linguagem do *solver* (“(= 18 (+ (* x 16) (* y 2)))” e “(= 2 (+ (* x 1) (* y 1)))”, respectivamente), e o *solver* retorna os valores que resolvem esse sistema de equações (x = 1 e y = 1), ou seja, para atendar as restrições dados

Tabela 1 - Exemplo de equação com números inteiros

Sistema de equações:	$\begin{cases} 16x + 2y = 18 \\ x + y = 2 \end{cases}$
Restrições passadas para o *solver*:	(= 18 (+ (* x 16) (* y 2))) (= 2 (+ (* x 1) (* y 1)))
Resultado retornado pelo *solver*:	x = 1 y = 1

Fonte: adaptado de (SOK, 2012)

Há diversos solucionares de restrições. Seguem alguns deles, listados por Sok (2012).

- Barcelogic (BARCELOGIC, 2012): Escrito em C++, esse *solver* implementa um algoritmo usado para resolver fórmulas lógicas expressas com conjunções formais, e é capaz de manipular diferentes teorias, tais como: inteiros, reais, igualdades com funções não interpretadas e uma combinação dessas teorias.

- CVC Lite (CVC, 2012): Também escrito em C++, esta ferramenta é capaz de resolver equações lógicas tais como: igualdades com funções não interpretadas, aritmética linear (inteiros, reais) e vetores.

- Yices (YICES, 2012): Outro *solver* escrito em C++, este *solver* é capaz de lidar com: igualdades com funções não interpretadas, aritmética linear (inteiros, reais), vetores, tuplas, registros e tipos de dados recursivos (ex. listas).

2.8 Sistema Gerador de Casos de Teste

Um Sistema Gerador de Casos de Teste é um sistema para gerar os casos de teste de um programa por meio da análise do código-fonte a fim de encontrar os dados de teste que atendam os critérios de teste do programa que está sendo testado (EDVARDSSON, 1999).

Edvardsson (1999) definiu que um sistema gerador de casos de teste geralmente possui 3 módulos principais: programa analisador, seletor de caminhos e gerador de dados de teste, conforme mostrado na Figura 10.

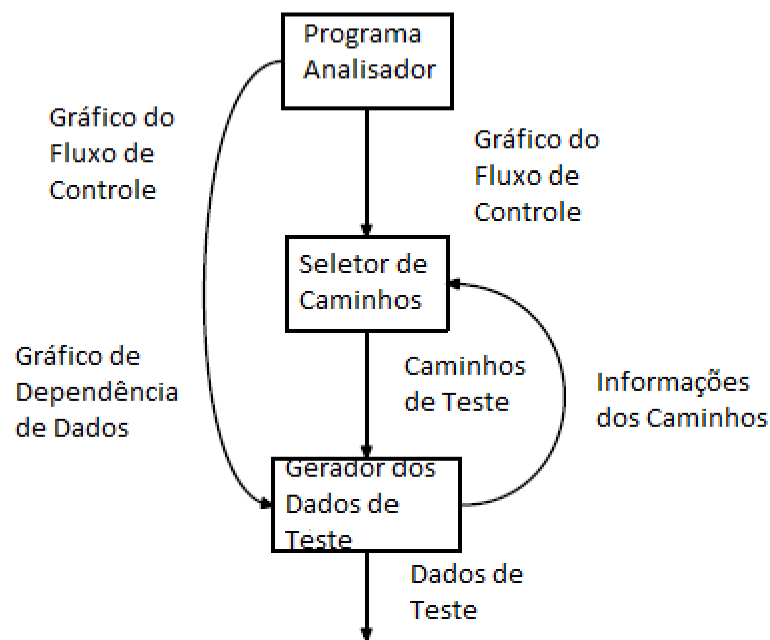


Figura 10 - Arquitetura de um sistema gerador de casos de teste
Fonte: (EDVARDSSON, 1999)

O analisador de programa faz a leitura do código fonte do programa e produz os dados necessários para os módulos seguintes.

O seletor de caminhos verifica os dados de entrada a fim de encontrar os caminhos ajustados para serem percorridos durante a execução do programa. O termo “ajustado” significa que a cobertura de código pode ser ajustada de forma a obter os resultados esperados em cada caso de teste (conhecido como oráculo).

Os caminhos encontrados pelo seletor de caminhos são passados para o gerador de casos de teste, que monta os dados de entrada para percorrer os caminhos definidos. O gerador de casos de teste pode analisar os caminhos de forma a percorrer somente os caminhos plausíveis, ou seja, aqueles que podem ser executados pelo programa (não são código “morto”).

O teste orientado a caminhos prevê a geração de dados para cobrir pelo menos uma vez um subconjunto de caminhos pré-selecionados de um SUT, incluindo caminhos com *loops*. A grande maioria dos programas possui *loops* e estes devem ser exercitados um número mínimo de vezes (BINDER, 2000), (PRESSMAN, 2006), (BEIZER, 1995), sabendo-se que a existência deles dentro do *software* pode levar a um número infinito de caminhos (BINDER, 2000), (PRESSMAN, 2006).

2.9 Ferramentas para Execução dos Testes

Além da utilização de um algoritmo de geração automática de casos de teste é recomendada a utilização de uma ferramenta de automatização de execução dos testes. Ela acelera a reaplicação dos testes à medida que o programa for alterado para aumento de funcionalidades ou correção de falhas encontradas (CRISPIN e GREGORY, 2009).

Crispin e Gregory (2009) recomendam que os testes sejam automatizados devido aos seguintes motivos:

a) Tempo de execução dos testes: à medida em que o tamanho de uma aplicação começa a crescer o tempo para executar todos os testes manuais nessa aplicação também aumenta.

b) Diminuição de erros: é outro motivo para automatização, pois erros podem surgir ao longo dos passos em um teste manual. Caso o procedimento de teste manual fique entediante o testador pode distrair-se e descartar algum passo provando erro nos testes, o que pode se agravar caso os testes sejam efetuados sob pressão de prazos;

c) Respostas freqüentes e rápidas: as respostas da execução dos testes automatizados ajudam o programador a identificar um problema no *software*, porque o código desenvolvido em uma iteração ainda está na mente dele, e isso faz com que ele identifique as mudanças ocorridas na aplicação e a causa do problema de forma mais rápida.

Algumas das ferramentas disponíveis para automatização de execução testes são FitNesse (MARTIN, 2011), Selenium (THOUGHWORKS, 2011) e JUnit (BECK, 2002). A escolha da ferramenta mais apropriada depende do tipo de teste que se deseja realizar. O Selenium é uma ferramenta geralmente utilizada para gravar as ações do usuário no preenchimento de telas; o FitNesse é mais indicado para testes de aceitação com base em dados escritos na forma tabular e o JUnit é largamente utilizado para testes unitários de programas Java.

A ferramenta FitNesse pode ser utilizada tanto por programadores quanto por analistas ou até pelos usuários que definem as regras do sistema devido à sua forma simples (formato HTML) de definição dos casos de teste, ou seja, da combinação dos parâmetros de entrada do programa e as saídas para cada combinação de parâmetros de entrada.

FitNesse é uma ferramenta de teste de *software* que combina uma ferramenta *wiki* e um servidor *web* integrados, e é baseado no FIT (MUGRIDGE; CUNNINGHAM, 2005) que é um framework Java disponível publicamente para a criação de testes de aceitação. No FIT um teste é definido dentro de uma tabela HTML, e podem existir várias dessas tabelas em uma página. A primeira linha de uma tabela especifica o nome da classe (chamada de *fixture*), que interpreta os testes contidos no restante da tabela (MUGRIDGE; TEMPERO, 2003). A Tabela 2

mostra um exemplo de tabela HTML no padrão FitNesse demonstrando dados de teste para um programa de divisão entre dois números. O nome da classe (division) que está sendo testada fica na parte superior da tabela; as colunas “numerador” e “denominador” são os dados de entrada da classe, e a coluna “quociente” é o resultado esperado após a equação da classe. Para identificar que é uma coluna de resultado é colocada a literal “?” após o nome da coluna.

Tabela 2 - Exemplo de tabela Fitnesse

eg.Division		
numerador	denominador	quociente?
10	2	5,0
12,6	3	4,2
22	7	~ = 3,14
9	3	<5
11	2	4 <_ <6
100	4	33

Fonte: adaptado de (FITNESSE, 2012)

A Figura 11 mostra outro exemplo de uma tabela no padrão FIT, no qual há um texto que descreve o requisito e em seguida são colocados alguns exemplos típicos do requisito, sendo que esses exemplos podem ser feitos pela própria pessoa que escreveu o requisito. As tabelas no formato FIT permitem que os exemplos de uso de um requisito se tornem casos de teste do programa que irá atender ao requisito.

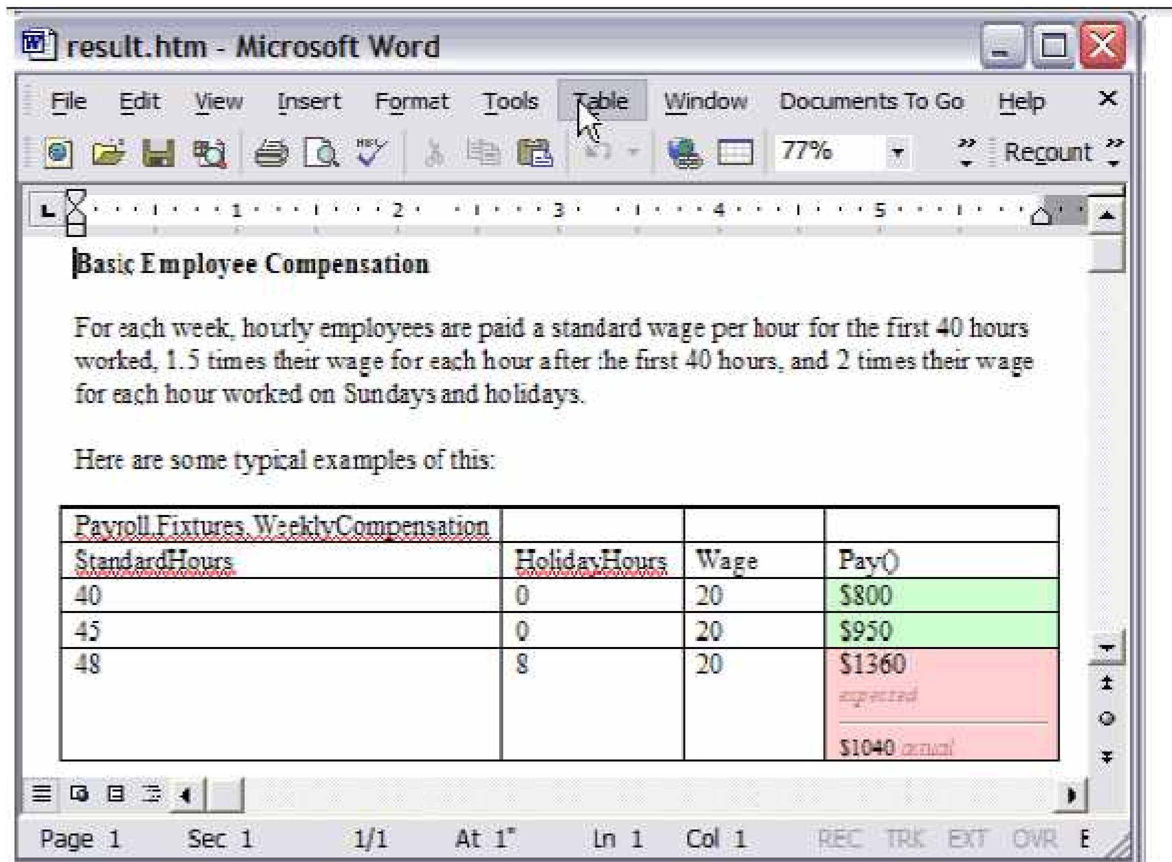


Figura 11 - Teste de aceitação utilizando FIT
Fonte: (CUNNINGHAM, 2002).

De acordo com Mugridge e Cunningham (2005), o padrão FIT se propõe a resolver três problemas principais no desenvolvimento de *software*:

1) **Comunicação**: exigências por escrito podem ser ambíguas, sujeitas a equívocos e concepção errada. Os testes no padrão FIT tendem a melhorar a comunicação do que um produto de *software* deve fazer através de testes concretos que são baseados em exemplos do domínio do negócio definidos pelos usuários;

2) **Agilidade**: Os testes no padrão FIT são destinados a apoiar mudanças frequentes, e os testes garantem que qualquer alteração no *software* não provoca erros nos requisitos anteriormente satisfeitos, permitindo assim agilidade; e

3) **Equilíbrio**: Os testes no padrão FIT são destinadas a ajudar uma equipe de desenvolvimento a gastar menos tempo na obtenção da correção dos problemas, reduzindo o número e a gravidade dos problemas, encontrando-os o mais depressa possível, e isto permite que eles tenham mais tempo para programação.

O FIT já vem com várias classes *fixture* gerais para execução de testes. Em alguns casos, uma classe pode ser usada diretamente; em outros, a classe é uma subclasse de uma *fixture* para fornecer o comportamento desejado ao teste. Por exemplo, ex.Calculadora é uma subclasse de *fit.ColumnFixture*; e esta testa diferentes funções de uma calculadora. O FIT tem as seguintes vantagens: utilização de tabelas; e facilidade com que novos tipos de testes podem ser acrescentados ao conjunto de testes mediante a definição de novas *fixtures* (MUGRIDGE; TEMPERO, 2003).

As tabelas HTML são usadas para definir os testes, assim um editor HTML simples pode ser usado para criar uma página *web* que contém os testes e um navegador pode ser utilizado para vê-los e gerar relatórios a partir deles. (MUGRIDGE; TEMPERO, 2003).

Hanssen e Haugset (2009) relataram uma pesquisa em que a maioria dos estudantes considerou a ferramenta FitNesse de fácil utilização, e que os testes no padrão HTML facilitam a interação entre usuários e desenvolvedores, mas eles relatam que o uso da ferramenta por si só pode não ser suficiente para garantir sistemas de boa qualidade, pois há casos em que a empresa precisa rever seus processos de teste.

Do ponto de vista de testes funcionais, as ferramentas apresentadas possuem algumas deficiências. Andrea (2007) cita que os testes elaborados pelas ferramentas do tipo xUnit também não são fáceis de se lerem e de serem encontrados. A tabela do FitNesse pode não servir para os testes que têm uma estrutura complexa que não se encaixem bem ao formato de tabela (ou uma seqüência de tabelas) (MUGRIDGE; TEMPERO, 2003). As do tipo “grava e executa”, como o Selenium, geram procedimentos de testes que são difíceis de ler e

de manter (ANDREA, 2007). Além disso, podem gerar falhas nos testes, como por exemplo, um botão que tem o seu identificador renomeado continua funcionando, mas os testes que possuem referências a ele deixam de funcionar (HOLMES; KELLOG, 2006). Andrea (2007) comenta que as deficiências das ferramentas a forçam a aprimorá-las, adicionando funcionalidades nelas.

Apesar das diversas ferramentas e recursos, Park e Maurer (2008) declaram em seu estudo que a técnica chamada Desenvolvimento Dirigido por Teste de Aceitação Executável ou *Executable Acceptance Test-Driven Development* (EATDD) não é praticada por dois principais motivos: a falta de uma boa ferramenta e por ser uma técnica relativamente nova. Porém a técnica melhora a comunicação entre o cliente e o desenvolvedor, e isso pode auxiliar a descoberta de problemas antes da construção do sistema. A execução dos testes de aceitação pode ser um indicativo em tempo real do progresso do desenvolvimento com a geração de uma documentação atualizada do *software*, resultando em maior qualidade.

2.10 Conclusão

Nesse capítulo foram mostradas as diversas abordagens para geração automática de casos de teste. Foram mostradas as características de cada abordagem, bem como algumas ferramentas utilizadas.

A Tabela 3 mostra como se resume os trabalhos que foram avaliados ao longo deste capítulo.

Tabela 3 - Comparação entre os trabalhos citados

Autor(es)	**Abordagem / Característica / Ferramenta**	**Relevância para este trabalho**
(ABREU, 2006)	Algoritmo Genético, GEO	Alternativa à técnica C4T
(ANDREA, 2007)	Selenium, xUnit	Comparação das ferramentas de execução de teste
(BECK, 2002)	JUnit	Exemplo de ferramenta para execução de teste
(BIN et al, 2007)	Têmpera Simulada	Alternativa à técnica C4T
(COLORNI, DORIGO e	Colônia de Formigas	Alternativa à técnica C4T

Autor(es)	Abordagem / Característica / Ferramenta	Relevância para este trabalho
MANIEZZO, 1992)		
(CRISPIN e GREGORY, 2009)	Desenvolvimento Dirigido por Testes de Aceitação Executável	Comparação das ferramentas de execução de teste
(DALAL et al., 1999)	Combinações de Entrada (*pairwise testing*)	Alternativa à técnica C4T
(DIAZ; TUYA e BLANCO, 2003	Algoritmo Evolutivo	Alternativa à técnica C4T
(DORIGO, 1992)	Colônia de Formigas	Alternativa à técnica C4T
(EDVARDSSON, 1999)	Sistema Gerador de Caso de Teste	Conceitos de sistema gerador de casos de teste, os quais foram usados na construção da técnica do presente trabalho
(EIBEN e SMITH, 2003)	Algoritmo Evolutivo	Alternativa à técnica C4T
(EMMI, MAJUMDAR e SEN, 2007)	Teste Direcionado	Exemplo de aplicação de teste direcionado para geração de casos de teste
(GONÇALVES, 2011)	Inteligência de Enxames	Alternativa à técnica C4T
(HANSSEN e HAUGSET, 2009)	FIT, FitNesse	Comparação de ferramentas de execução de teste, destacando as vantagens do padrão Fit
(HARTMANN, 2009)	Baseado em Modelo	Alternativa à técnica C4T
(HOLMES e KELLOG, 2006).	Selenium	Exemplo de aplicação da ferramenta de execução de teste Selenium
(KENNEDY e EBERHART, 1995)	Inteligência de Enxames	Alternativa à técnica C4T
(KIEZUN, 2009)	Teste Direcionado	Artigo que inspirou a utilização da abordagem de teste direcionado aplicada no presente trabalho
(KOUTSIKAS; MALEVRIS e SKORDALAKIS, 2000)	Caminhos de Teste, SYMEXLAN	Linguagem de script para execução simbólica
(LAZINICA, 2009)	Inteligência de Enxames - Otimização por Enxame de Partículas	Alternativa à técnica C4T
(LI e ZHANG, 2008)	Inteligência de Enxames – Otimização por Enxame de Partículas, Único caminho	Alternativa à técnica C4T
(LI e ZHANG, 2009)	Inteligência de Enxames -	Alternativa à técnica C4T

Autor(es)	Abordagem / Característica / Ferramenta	Relevância para este trabalho
	Otimização por Enxame de Partículas, Todos os caminhos	
(LI; ZHANG e LIU, 2009)	Inteligência de Enxames – Otimização por Colônia de Formigas	Alternativa à técnica C4T
(LIU e NAKAJIMA, 2011)	Método Formal, Método de Vibração	Alternativa à técnica C4T
(LOPES, 2006)	Inteligência de Enxames	Alternativa à técnica C4T
(MANSOUR e SALAME, 2004)	Algoritmo Evolutivo	Alternativa à técnica C4T
(MARTIN, 2011)	FitNesse	Ferramenta para execução de teste
(MCMINN, 2004)	Geral (*survey*)	Comparação entre algumas abordagens sobre geração de casos de teste
(MEDEIROS, 2005)	Inteligência de Enxames – Otimização por Enxame de Partículas	Alternativa à técnica C4T
(MEUDEC, 2001)	Execução Simbólica, CLP, ATGen	Alternativa à técnica C4T
(MICHAEL, MCGRAW e SCHATZ, 2001)	Aleatório	Alternativa à técnica C4T
(MINGSONG; XIAOKANG e XUANDONG, 2006)	Diagrama de Atividades UML	Alternativa à técnica C4T
(MUGRIDGE e CUNNINGHAM, 2005)	FIT	Conceitos do framework FIT
(MUGRIDGE e TEMPERO, 2003)	FIT	Conceitos do framework FIT
(PAPADAKIS e MALEVRIS, 2011)	Subida de Encosta, Mutantes	Alternativa à técnica C4T
(PARK e MAURER, 2008)	Desenvolvimento Dirigido por Teste de Aceitação Executável	Comparação entre as ferramentas de teste de aceitação
(PIMENTA, 1998)	Método Formal	Alternativa à técnica C4T
(SEN, 2007)	Teste Direcionado	Exemplo de aplicação de teste direcionado para geração de casos de teste
(SNEED, 2010)	Conversão de Código-Fonte	Exemplo de aplicação de migração de código utilizando conversão de código-fonte
(SOK, 2012)	*Solver,* Barcelogic, CVC Lite,	Comparação entre os solucionadores de

Autor(es)	Abordagem / Característica / Ferramenta	Relevância para este trabalho
	Yices	restrições (*solvers*)
(STHAMER, 1996)	Execução Simbólica	Alternativa à técnica C4T
(TAKAKI, 2009)	Meta-heurística	Alternativa à técnica C4T
(THOUGHWORKS, 2011)	Selenium	Ferramenta de execução de teste
(UTTING e LEGEARD, 2006).	Combinações de Entrada	Alternativa à técnica C4T
(WATKINS, 2006)	Algoritmo Evolutivo	Alternativa à técnica C4T
(WHITLEY, 2001)	Algoritmo Evolutivo	Alternativa à técnica C4T
(XUAN et al, 2010)	Aleatório, *Walktest*	Alternativa à técnica C4T

Fonte: Elaborado pelo autor

3 PROPOSTA: A TÉCNICA C4T

3.1 Introdução

Neste capítulo são mostrados os critérios utilizados para a escolha da abordagem da técnica C4T (COBOL *for Test*), os passos desempenhados pela técnica C4T para geração dos casos de teste de programas em linguagem COBOL e também é mostrado um exemplo de aplicação dessa técnica.

3.2 Escolha da Abordagem

Como há diversas abordagens para geração automática de casos de teste, descritas na Figura 7, foram utilizados alguns critérios para escolha do método mais adequado a ser aplicado na técnica C4T.

Não se utilizou o método de especificação formal pois, de acordo com Nunes, Moreira e Ribeiro (2011), esse método não é muito utilizado atualmente devido à elevada curva de aprendizado necessária para a aplicação desse método na indústria de *software*.

O teste baseado em modelo não se mostra também adequado para a linguagem COBOL uma vez que esse teste é mais frequentemente utilizado para modelagem UML, técnica utilizada para linguagens orientadas a objetos e não para linguagens que utilizam paradigma estruturado como é o caso da linguagem COBOL. Também não foi possível utilizar a abordagem de teste baseado na especificação sugerida por Dallmeier et al (2012) em função de que essa abordagem efetua mudanças no programa para explorar novos comportamentos, alterando o fluxo de controle do programa no lugar de, simplesmente, analisá-lo.

Além disso, para utilização dessa técnica é preciso que a documentação do modelo do sistema esteja atualizada para refletir exatamente o comportamento do sistema. A documentação de sistemas legados com frequência não está atualizada, então essa abordagem não se mostra a mais adequada para geração de casos de

teste para sistemas legados em COBOL. Com isso não se utilizam os métodos baseados em testes funcionais para a técnica C4T.

Os métodos de busca meta-heurística foram descartados pois envolvem o critério de otimização, que busca encontrar a melhor solução para um determinado problema. O que se deseja na técnica C4T é encontrar uma solução para o problema, não necessariamente a melhor, ou seja, um dado de entrada qualquer que atenda aos critérios do caminho que se deseja testar já é suficiente para a geração do caso de teste.

Descartaram-se também os métodos evolutivos pois eles são de “busca cega”, que tem uma performance pior do que os os métodos que exploram domínios de conhecimento específico (WHITLEY, 2001).

O método de execução simbólica aplicado isoladamente foi preterido uma vez que esse método exige elevado recurso computacional (PINHEIRO, 2012).

O método de execução dinâmica aplicado isoladamente foi descartado pois, assim como os método de busca meta-heurística, esse método utiliza critérios de otimização na busca da solução “ótima”, sendo que o objetivo da técnica C4T é encontrar uma solução que atenda ao critério de teste.

Com relação ao método de pares de entrada, esse método não foi utilizado em função de que ele apenas gera as entradas de teste, sendo que o resultado esperado de cada conjunto de entradas (oráculo) deve ser gerado manualmente pelo engenheiro de testes, que deve ter um conhecimento prévio das funcionalidades do programa para gerar os oráculos de teste.

Para evitar que haja muitos caminhos de teste a serem alcançados, a técnica C4T usa como critério a execução de pelo menos uma vez em cada *loop*, e os caminhos são ajustados utilizando técnicas de teste estrutural (ex. análise de valor-limite).

Com base na experiência bem sucedida de Kiezun (2009) o presente trabalho procura apresentar uma técnica que utiliza a abordagem de teste direcionado para geração automática de casos de teste para um programa COBOL pois essa técnica possui menos entradas e é mais rápida para geração dos casos de teste.

3.3 Arquitetura da Técnica C4T

A técnica C4T utiliza a estrutura de um sistema gerador de casos de teste, proposta por Edvardsson (1999), mostrado na Figura 10, da seguinte forma: cria-se um *parser* na técnica C4T para conversão de código-fonte em COBOL para a linguagem de restrições da ferramenta Yices (YICES, 2012). Após essa etapa, a técnica C4T usa as entradas de caminho e aplica o teste direcionado para gerar os dados de entrada que atendam cada caminho do programa.

Utiliza-se a ferramenta FitNesse como resultado dos casos de teste gerados em razão dessa ferramenta mostrar o resultado em formato HTML, que é de fácil visualização por parte dos usuários e desenvolvedores, conforme relatado por Hanssen e Haugset (2009)

A Figura 12 mostra a arquitetura da técnica C4T. Dado um programa COBOL, esse programa é processado por um analisador de caminhos (que gera a relação dos caminhos do programa) e um instrumentador (que gera o código instrumentado na linguagem do *solver*). Um usuário da técnica analisa os caminhos e seleciona os caminhos para os quais devem ser gerados casos de teste. Após isso é feita a execução dinâmica do programa instrumentado com as restrições para atender os caminhos selecionados, e nessa execução dinâmica é feita a chamada para um *solver* que devolve as variáveis de entrada que atendem as restrições dadas, gerando os casos de teste do programa. Se o programa for convertido para outra linguagem que suporte a execução utilizando o FitNesse, o FitNesse executa esse programa convertido a partir dos casos de teste gerados anteriormente, e gera um relatório dos casos de teste executados com sucesso e com erro.

A Figura 12 mostra também a associação dos processos da arquitetura da técnica C4T com a estrutura de um sistema gerador de casos de teste sugerida por Edvardsson (1999). O processo “Analisador de Caminhos e Instrumentador”, que faz a leitura de uma programa COBOL e gera o programa instrumentado, corresponde ao processo “Programa Analisador” de Edvardsson. O processo em que um engenheiro de testes seleciona os caminhos a serem testados a partir da lista de caminhos gerados pelo processo anterior corresponde ao processo “Seletor de Caminhos” de Edvardsson. Por fim, o processo de “Execução Dinâmica”, que executa o programa instrumentado buscando os caminhos selecionados pelo

engenheiro de testes e gera os casos de teste utilizando um *solver* corresponde ao processo “Gerador de Casos de Teste” de Edvardsson.

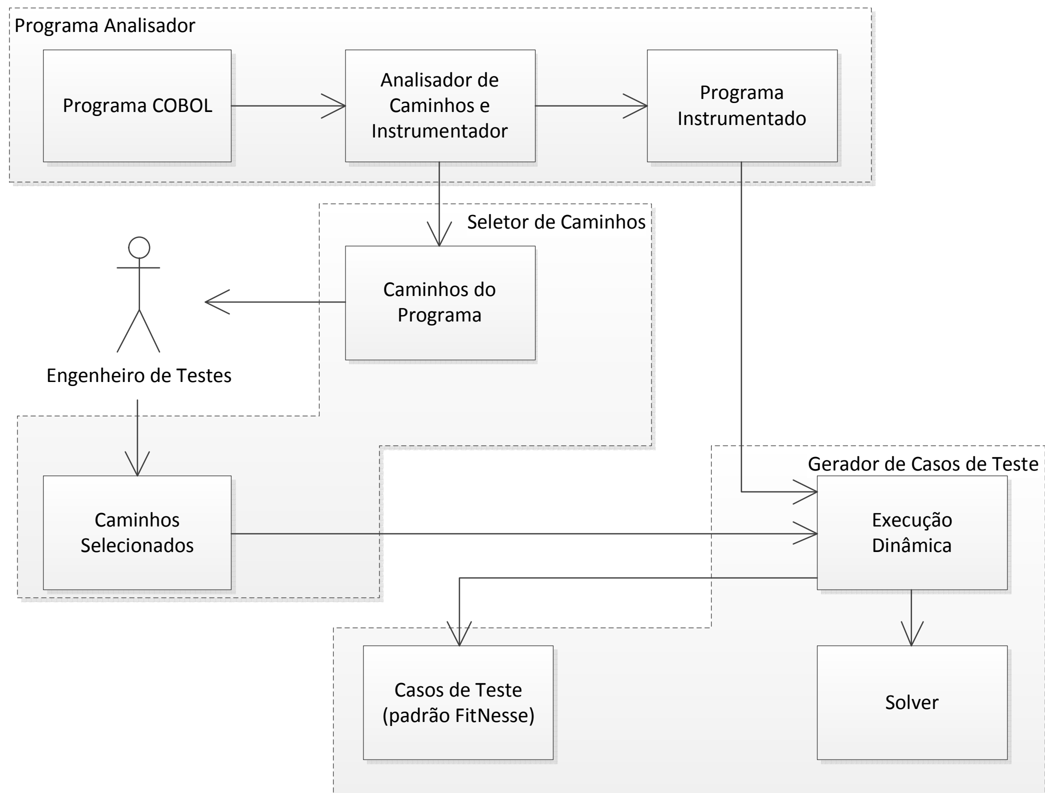


Figura 12 - Arquitetura da Técnica C4T
Fonte: Elaborado pelo autor

O APÊNDICE 5 – Pseudocódigo do Algoritmo do Instrumentador mostra o pseudocódigo do algoritmo do processo “analisador de caminhos e instrumentador” mencionado na Figura 12.

Após os casos de teste serem gerados pela técnica C4T, é possível executá-los utilizando a ferramenta FitNesse. A Figura 13 mostra o processo de execução da ferramenta FitNesse em conjunto com a técnica C4T no caso de migração do programa COBOL para outra linguagem. Após o programa ser convertido para a outra linguagem (desde que a outra linguagem comporte a execução utilizando FitNesse), a ferramenta FitNesse recebe os casos de teste gerados no padrão HTML pela técnica C4T, executa o programa convertido e avalia se os resultados da execução do programa correspondem aos resultados esperados pelo caso de teste, gerando um relatório dos casos de teste cujo resultado foi igual ao esperado e

também um relatório dos casos de teste cujo resultado não foi igual ao esperado, correspondendo às situações de erro na execução do programa.

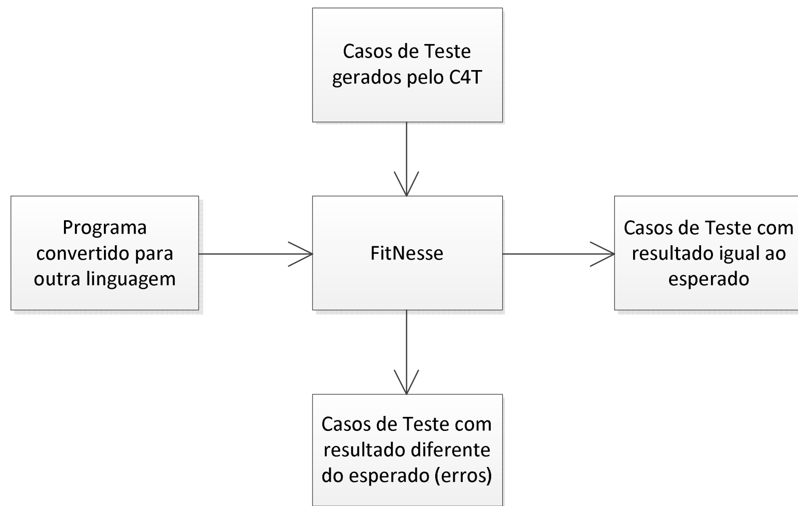


Figura 13 - Execução do FitNesse em conjunto com a técnica C4T
Fonte: Elaborado pelo autor

3.4 Método para geração de casos de teste

Para atingir os objetivos propostos é utilizado um método de solução no qual são seguidas as atividades listadas a seguir:

3.4.1 Preparação do Ambiente de Execução

Para o desenvolvimento da técnica C4T não foi possível utilizar o *mainframe* de uma empresa, portanto para execução dos testes da técnica criada é preparado um ambiente em um microcomputador para simular a execução do programa COBOL. Para isso, é utilizada a ferramenta VisualCobol (MICROFOCUS, 2011), que converte um programa COBOL em linguagem de JVM (*Java Virtual Machine*) e permite com isso usar a linguagem Java para elaborar a parte da técnica que faz a chamada ao *solver.*

3.4.2 Etapas da Técnica C4T

A geração de casos de teste para programas COBOL pela técnica C4T se dá em 6 etapas: identificação de variáveis, identificação de caminhos, instrumentação do código-fonte para instruções de decisão, instrumentação do código-fonte para chamada ao *solver*, geração do programa controlador e geração dos casos de teste. Cada etapa é descrita a seguir em detalhes.

A primeira etapa é a identificação das variáveis do programa. Nessa etapa são identificadas as variáveis de entrada, de saída e de trabalho do programa a ser testado (*software under test* - SUT) e as essas variáveis são definidas no padrão do *solver* (Yices).

A segunda etapa é a identificação dos caminhos do programa. Nessa etapa é feita a identificação dos caminhos do programa e é montada uma lista de combinação de caminhos possíveis para o programa. Nela a técnica C4T lê o código-fonte de um programa COBOL e gera um arquivo contendo a combinação dos caminhos do programa seguindo o critério de teste de cobertura de caminhos.

A terceira etapa é a instrumentação do código com as restrições do programa. O código-fonte do SUT é lido para identificar as instruções de decisão programas (IF, ELSE, WHILE, etc) e atribuição de variáveis. Essa etapa consiste em instrumentar o programa inserindo no código-fonte as instruções de restrição para cada um dos comandos de decisão e atribuição de variáveis do programa para que eles possam ser utilizadas em um *solver*.

A quarta etapa é a instrumentação do código com as chamadas ao *solver*. O objetivo dessa etapa é instrumentar o código-fonte do SUT para que a execução seja interrompida a cada instrução de decisão. Nessa interrupção é acionado um solver (Yices) que retorna qual o dado de entrada que faz com que o SUT entre na condição de decisão desejada.

A quinta etapa é a geração de um programa controlador que aciona o programa a ser testado. O objetivo desse programa controlador é efetuar o gerenciamento das chamadas ao SUT. Ele recebe os casos de teste para os quais devem ser gerados dados de entrada, aciona o SUT com os dados de entrada recebidos e, caso os dados de entrada não atendam ao caso de teste, aciona o SUT novamente com os dados de entrada retornados pelo solver. Esse procedimento se repete até que os dados de entrada atendam ao caso de teste ou o solver retornar

que não há nenhum dado de entrada que atenda ao caso de teste. O programa controlador armazena os dados de entrada e de saída de cada caso de teste em uma base de dados para futura geração dos casos de teste no padrão FitNesse.

A sexta e última etapa é a efetiva geração dos casos de teste no padrão FitNesse. Nessa etapa o programa controlador é acionado com os caminhos escolhidos pelo engenheiro de teste para geração dos dados de entrada. Quando o programa controlador identifica que todos os caminhos do SUT foram atingidos é acionado um programa que lê a base de dados citada na etapa anterior e cria os casos de teste no padrão FitNesse. Cada caso de teste contem os dados de entrada e o resultado esperado a partir desses dados de entrada

3.5 Exemplo de Uso

A seguir é mostrado um exemplo de aplicação da técnica C4T com finalidade didática.

O objetivo do programa de exemplo é identificar quanto cada funcionário pagará de imposto de renda na fonte sobre o salário e conceder 3, 5 ou 10% de aumento de acordo com o salário e o valor de imposto a pagar. Para os casos em que o imposto a pagar é maior ou igual a R$ 125,00 e o salário é menor do que R$ 2.000,00 o aumento é de 5%. Para os casos em que o imposto a pagar é maior ou igual a R$ 125,00 e o salário é maior ou igual a R$ 2.000,00 o aumento é de 10%. Para os casos em que o imposto a pagar é menor do que R$ 125,00 o aumento é de 3%. A lógica do programa consiste basicamente em definir faixas de valores para pagamento de imposto de renda de acordo com o salário. Para cada faixa de valor de salário é definida uma alíquota para pagamento de imposto de renda (quanto maior o valor do salário maior a alíquota de imposto de renda). Além da alíquota de imposto de renda em cada faixa também é definido também um valor de parcela a deduzir, que é um valor atribuído para que a nova alíquota seja aplicada somente aos valores que superarem o limite superior da faixa anterior. Após definidos os valores de alíquota e parcela a deduzir, o valor de imposto de renda a ser pago é calculado multiplicando-se o salário pela alíquota e subtraindo-se o valor da parcela

a deduzir. Para facilitar o entendimento, as faixas de imposto de renda foram reduzidas para apenas duas e os centavos foram desprezados.

Na Tabela 4 é mostrado o código fonte original do programa COBOL do exemplo usado para explicação do funcionamento da técnica C4T.

Tabela 4 - Código-Fonte original do programa Cobol

```
1. IDENTIFICATION DIVISION
2. PROGRAM-ID. IMPOSTO_RENDA
3. DATA DIVISION.
4. LINKAGE SECTION
5. 01 SALARIO          PIC 99999 VALUE ZEROS.
6. 01 AUMENTO          PIC 999 VALUE ZEROS.
7. WORKING-STORAGE SECTION.
8. 01 ALIQUOTA_IR           PIC 99 VALUE ZEROS.
9. 01 PARCELA_A_DEDUZIR     PIC 9999 VALUE ZEROS.
10.  01 IR_A_PAGAR          PIC 9999  VALUE ZEROS.
11.  PROCEDURE DIVISION.
12.  MOVE 0 TO AUMENTO
13.  IF  SALARIO     <=    1499 THEN
14.      MOVE 0 TO ALIQUOTA_IR
15.      MOVE 0 TO PARCELA_A_DEDUZIR
16.      GO TO CALCULA_VALOR_IMPOSTO_RENDA.
17.  IF SALARIO      >     1499
18.      MOVE 27 TO ALIQUOTA_IR
19.      MOVE 306 TO PARCELA_A_DEDUZIR
20.      GO TO CALCULA_VALOR_IMPOSTO_RENDA.
21.  CALCULA_VALOR_IMPOSTO_RENDA.
22.  COMPUTE  IR_A_PAGAR  = (SALARIO * ALIQUOTA_IR / 100)  -
23.                         PARCELA_A_DEDUZIR
24.  IF   IR_A_PAGAR >=    125   THEN
25.      IF SALARIO < 2000 THEN
26.         MOVE 5 TO AUMENTO
27.      ELSE
28.         MOVE 10 TO AUMENTO
29.  ELSE
30.      MOVE 3 TO AUMENTO.
31.  STOP RUN
32.  END.
```

Fonte: elaborado pelo autor

A seguir são mostrados os passos que a técnica executa para instrumentação do código-fonte COBOL.

Passo 1: Identificar variáveis do programa

Nesse passo as variáveis de entrada, de saída e de trabalho do programa são identificadas e definidas no padrão do *solver* (Yices). Em programas COBOL as

variáveis de entrada e saída são definidas na seção "linkage section" (linha 4 da Tabela 4).

Para a variável de entrada "SALARIO" (linha 5 da Tabela 4) é definida a seguinte instrução no formato do *solver* (define SALARIO::real). A instrução "define" do *solver* indica a definição de uma variável de trabalho. Nesse caso, está indicando que a variável "SALARIO" tem um conteúdo com valor "real", ou seja, pode receber números inteiros e decimais.

Além disso, o engenheiro de testes pode definir os limites de cada variável do programa. Nesse caso, o salário deve ser maior do que 0, então é definida a seguinte restrição para o solver: (assert+ (> SALARIO 0)). A instrução "assert" indica para o *solver* que uma afirmação deve ser atendida. O *solver* utiliza a seguinte notação para o comando "assert": primeiramente é definida a operação, que nesse caso é o símbolo ">" (maior que). Em seguida vem o primeiro operando, que nesse exemplo é a própria variável "SALARIO". Depois é definido o segundo operando, que nesse caso é o valor 0.

Para a variável de saída "AUMENTO" (linha 6 da Tabela 4) é definida a seguinte instrução no formato do *solver:* (define AUMENTO::real) pois, da mesma forma que a variável "SALARIO", a variável "AUMENTO" também pode receber valores "reais".

As variáveis de trabalho são definidas na seção "working-storage section" (linha 7 da Tabela 4) dos programas COBOL.

Para a variável de trabalho "ALIQUOTA_IR" (linha 8 da Tabela 4) é definida a seguinte instrução no formato do *solver* : (define ALIQUOTA_IR::real).

Para a variável de trabalho "PARCELA_A_DEDUZIR" (linha 9 da Tabela 4) é definida a seguinte instrução no formato do *solver:* (define PARCELA_A_DEDUZIR::real).

Para a variável de trabalho "IR_A_PAGAR" (linha 10 da Tabela 4) é definida a seguinte instrução no formato do *solver* (define IR_A_PAGAR::real).

A Tabela 5 mostra as variáveis do programa COBOL e a sua respectiva instrução no formato do solver após a execução do passo 1 da técnica C4T.

Tabela 5 - Conversão das variáveis do programa exemplo para o *solver*

Linha do programa	Instrução COBOL	Instrução *solver*
5	SALARIO PIC 99999 VALUE ZEROS	(define SALARIO:real)
6	AUMENTO PIC 999 VALUE ZEROS	(define AUMENTO:real)
8	ALIQUOTA_IR PIC 99 VALUE ZEROS	(define ALIQUOTA_IR:real)
9	PARCELA_A_DEDUZIR PIC 9999 VALUE ZEROS	(define PARCELA_A_DEDUZIR:real)
10	IR_A_PAGAR PIC 9999 VALUE ZEROS	(define IR_A_PAGAR:real)

Passo 2: Identificação dos caminhos do programa

Nesse passo é feita a identificação dos diversos caminhos do programa. Primeiramente são identificadas as instruções que provoquem desvio no fluxo de controle (IF, ELSE, UNTIL, etc). Para cada uma dessas instruções identificam-se o término das instruções de decisão para verificar se uma nova instrução de decisão é dependente de uma instrução de decisão anterior. Também são identificadas as instruções de início e fim de execução do programa como um todo.

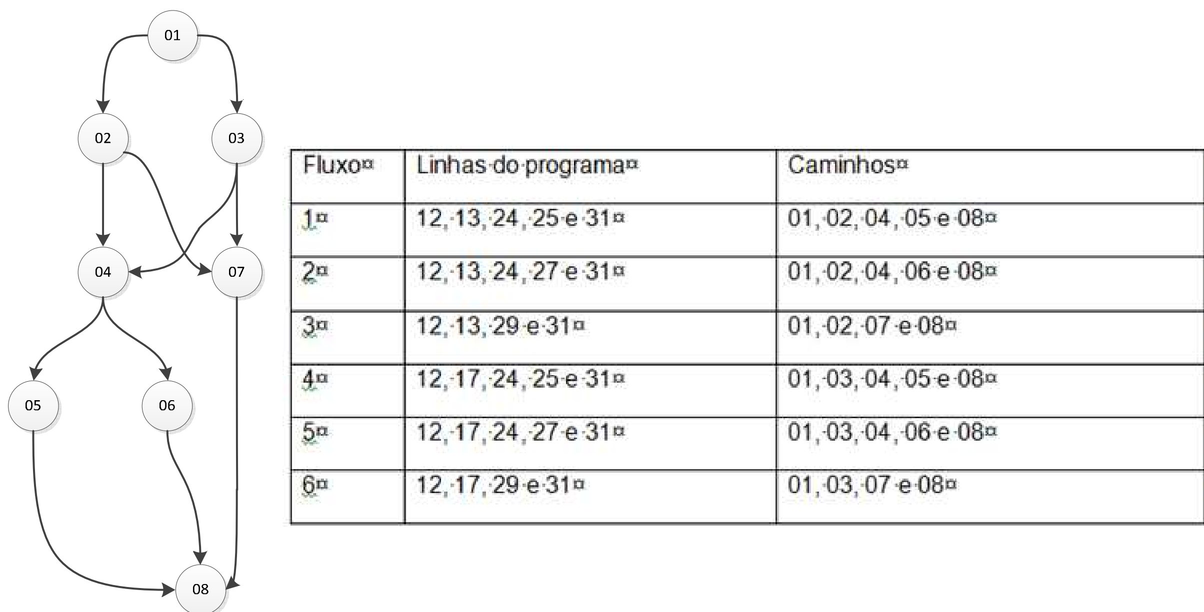
Em programas COBOL, a execução do programa é iniciada na divisão "`procedure division`" (linha 11 da Tabela 4). No programa de exemplo, a linha seguinte à `procedure division` corresponde ao caminho 01 do programa, pois em todas as execuções esse caminho é obrigatório. A instrução "`IF`" da linha 13 da Tabela 4 corresponde ao caminho 02 do programa. A instrução "`IF`" da linha 17 da Tabela 4 do programa corresponde ao caminho 03 do programa, e pelo jeito que o programa foi construído (ao entrar no primeiro caminho é feito o desvio para o label `CALCULA_VALOR_IMPOSTO_RENDA`) eles são caminhos dependentes, ou seja, a execução do caminho 02 implica não serem executadas as instruções do caminho 03.

A instrução `IF` da linha 24 da Tabela 4 do programa corresponde ao caminho 04, e ele pode ser executado tanto vindo do caminho 02 quanto do caminho 03.

A instrução `IF` da linha 25 da Tabela 4 do programa corresponde ao caminho 05, e ele só pode ser executado juntamente com o caminho 04. A instrução `ELSE` da linha 27 da Tabela 4 corresponde ao caminho 06, e ele também só pode ser executado juntamente com o caminho 04. A instrução `ELSE` da linha 29 da Tabela 4 corresponde ao caminho 07, e ele é mutuamente exclusivo com a execução do caminho 04.

E, por último, a instrução "`STOP RUN`" da linha 31 da Tabela 4 é a última instrução a ser executada no programa, então corresponde ao caminho 08, e, assim como o caminho 01, deve sempre ser executado.

A Figura 14 mostra o gráfico de fluxo de controle do programa com os possíveis caminhos a serem executados pelo programa identificados após a execução do passo 2 da técnica C4T.



Fluxo	Linhas do programa	Caminhos
1	12, 13, 24, 25 e 31	01, 02, 04, 05 e 08
2	12, 13, 24, 27 e 31	01, 02, 04, 06 e 08
3	12, 13, 29 e 31	01, 02, 07 e 08
4	12, 17, 24, 25 e 31	01, 03, 04, 05 e 08
5	12, 17, 24, 27 e 31	01, 03, 04, 06 e 08
6	12, 17, 29 e 31	01, 03, 07 e 08

Figura 14 - Gráfico de fluxo de controle do programa exemplo
Fonte: Elaborado pelo autor

Passo 3: Instrumentação do código com as instruções de decisão e atribuição

O objetivo desse passo é instrumentar o código-fonte nas instruções de decisão do programa (IF, ELSE, UNTIL, etc) e atribuição de variáveis, a fim de

conduzir a execução dinâmica do programa pelos caminhos-alvo previamente selecionados.

No programa de exemplo, a primeira instrução de decisão é a instrução de decisão “IF” da linha 13 da Tabela 4, que, como visto no passo 2, corresponde ao caminho 02 do programa. A técnica C4T converte a instrução da linha 13 para o formato do solver, ou seja, a instrução “IF SALARIO <= 1499” é convertida para “(assert+ (<= SALARIO 1499))”, pois na sintaxe do *solver* é colocado primeiro o operador (<=), depois o primeiro operando (SALARIO) e depois o segundo operando (1499). Como essa instrução é válida somente caso o caminho desejado seja o caminho 02, é colocada a instrução “IF CAMINHO_ALVO = 02” antes dessa instrução do *solver*.

A instrução de atribuição “MOVE 0 TO AUMENTO” da linha 12 do Tabela 4 é convertida para “(assert+ (= AUMENTO 0)”.

A instrução de atribuição “MOVE 0 TO ALIQUOTA_IR” da linha 14 do Tabela 4 é convertida para “(assert+ (= ALIQUOTA_IR 0)” para indicar ao solver que deve-se garantir que a variável ALIQUOTA_IR deve assumir o valor 0 quando o programa passar por esse trecho de código.

Seguindo essa linha de raciocínio, a instrução de atribuição “MOVE 0 TO PARCELA_A_DEDUZIR” da linha 15 do Tabela 4 é convertida para “(assert+ (=PARCELA_A_DEDUZIR 0)”.

A instrução “IF SALARIO > 1499” da linha 17 da Tabela 4 (correspondente ao caminho 03 do programa) é convertida pela técnica C4T para “(assert+ (> SALARIO 1499))”, e é colocada a instrução “IF CAMINHO_ALVO = 03” antes dessa instrução do *solver*.

A instrução de atribuição “MOVE 27 TO ALIQUOTA_IR” da linha 18 do Tabela 4 é convertida para “(assert+ (= ALIQUOTA_IR 27)”, e a instrução de atribuição “MOVE 306 TO PARCELA_A_DEDUZIR” da linha 19 do Tabela 4 é convertida para “(assert+ (= PARCELA_A_DEDUZIR 306)”.

A instrução de cálculo “COMPUTE IR_A_PAGAR = (SALARIO * ALIQUOTA_IR) - PARCELA_A_DEDUZIR” da linha 22 do Tabela 4 é convertida

para "(assert+ (= IR_A_PAGAR (+ (/ (* SALARIO ALIQUOTA_IR) 100) (* PARCELA_A_DEDUZIR -1))))". A variável PARCELA_A_DEDUZIR é multiplicada por "-1" em função de uma limitação do *solver* que não consegue trabalhar com valores negativos para as restrições.

A instrução "IF IR_A_PAGAR >= 125" da linha 24 da Tabela 4 (correspondente ao caminho 04 do programa) é convertida pela técnica C4T para "(assert+ (>= IR_A_PAGAR 125))", e é colocada a instrução "IF CAMINHO_ALVO = 04" antes dessa instrução do solver.

A instrução "IF SALARIO < 2000" da linha 25 da Tabela 4 (correspondente ao caminho 05 do programa) é convertida pela técnica C4T para "(assert+ (< SALARIO 2000))", e é colocada a instrução "IF CAMINHO_ALVO = 05" antes dessa instrução do solver.

A instrução de atribuição "MOVE 10 TO AUMENTO" da linha 26 do Tabela 4 é convertida para "(assert+ (= AUMENTO 10)".

A instrução "ELSE" da linha 27 da Tabela 4 (correspondente ao caminho 06 do programa) é convertida pela técnica C4T para "(assert+ (>= SALARIO 2000))", pois essa instrução ELSE corresponde à negação da instrução "IF SALARIO < 2000" da linha 25, e é colocada a instrução "IF CAMINHO_ALVO = 06" antes dessa instrução do solver.

A instrução de atribuição "MOVE 5 TO AUMENTO" da linha 28 do Tabela 4 é convertida para "(assert+ (= AUMENTO 5)".

A instrução "ELSE" da linha 29 da Tabela 4 (correspondente ao caminho 07 do programa) é convertida pela técnica C4T para "(assert+ (< IR_A_PAGAR 125))", pois essa instrução ELSE corresponde à negação da instrução "IF IR_A_PAGAR >= 125" da linha 24, e é colocada a instrução "IF CAMINHO_ALVO = 07" antes dessa instrução do solver.

A Tabela 6 mostra o código-fonte após a aplicação do passo 3 da técnica C4T. Para facilitar o entendimento é mostrada somente a divisão "procedure division", pois é nela que são feitas as principais instrumentações do código.

Tabela 6 - Código-fonte após aplicação do passo 3 da técnica C4T

```
1. PROCEDURE DIVISION.
2. SET STRING_PARA_SOLVER TO STRING_PARA_SOLVER &
3.     "(DEFINE SALARIO:REAL)\N"
4. SET STRING_PARA_SOLVER TO STRING_PARA_SOLVER &
5.     "(DEFINE AUMENTO:REAL)\N"
6. SET STRING_PARA_SOLVER TO STRING_PARA_SOLVER &
7.     "(DEFINE ALIQUOTA_IR:REAL)\N"
8. SET STRING_PARA_SOLVER TO STRING_PARA_SOLVER &
9.     "(DEFINE PARCELA_A_DEDUZIR:REAL)\N"
10.   SET STRING_PARA_SOLVER TO STRING_PARA_SOLVER &
11.       "(DEFINE IR_A_PAGAR:REAL)\N"
12.   SET STRING_PARA_SOLVER TO STRING_PARA_SOLVER &
13.       "(ASSERT+ (> SALARIO 0))\N"
14.   SET STRING_PARA_SOLVER TO STRING_PARA_SOLVER &
15.       "(ASSERT+ (= AUMENTO 0))\N"
16.   MOVE 0 TO AUMENTO
17.   IF CAMINHO_ALVO = 02 THEN
18.      SET STRING_PARA_SOLVER TO STRING_PARA_SOLVER &
19.          "(ASSERT+ (<= SALARIO 1499))\N"
20.   END-IF
21.   IF   SALARIO     <=    1499 THEN
22.      SET STRING_PARA_SOLVER TO STRING_PARA_SOLVER &
23.          "(ASSERT+ (= ALIQUOTA_IR 0))\N"
24.       MOVE 0 TO ALIQUOTA_IR
25.       SET STRING_PARA_SOLVER TO STRING_PARA_SOLVER &
26.          "(ASSERT+ (= PARCELA_A_DEDUZIR 0))\N"
27.       MOVE 0 TO PARCELA_A_DEDUZIR
28.       GO TO CALCULA_VALOR_IMPOSTO_RENDA.
29.   IF CAMINHO_ALVO = 03 THEN
30.      SET STRING_PARA_SOLVER TO STRING_PARA_SOLVER &
31.      "(ASSERT+ (> SALARIO 1499))\N"
32.   END-IF
33.   IF SALARIO      >     1499
34.      SET STRING_PARA_SOLVER TO STRING_PARA_SOLVER &
35.          (ASSERT+ (= ALIQUOTA_IR 27))\N"
36.       MOVE 27 TO ALIQUOTA_IR
37.       SET STRING_PARA_SOLVER TO STRING_PARA_SOLVER &
38.          "(ASSERT+ (= PARCELA_A_DEDUZIR 306))\N"
39.       MOVE 306 TO PARCELA_A_DEDUZIR
40.       GO TO CALCULA_VALOR_IMPOSTO_RENDA.
41.   CALCULA_VALOR_IMPOSTO_RENDA.
42.      SET STRING_PARA_SOLVER TO STRING_PARA_SOLVER &
43.      "(ASSERT+ (= IR_A_PAGAR (+ (/ (* SALARIO ALIQUOTA_IR)
44.       100) (* PARCELA_A_DEDUZIR -1))))"
45.   COMPUTE  IR_A_PAGAR  = (SALARIO * ALIQUOTA_IR / 100)  -
46.                          PARCELA_A_DEDUZIR
47.   IF CAMINHO_ALVO = 04 THEN
48.      SET STRING_PARA_SOLVER TO STRING_PARA_SOLVER &
49.      "(ASSERT+ (>= IR_A_PAGAR 125))\N"
50.   END-IF
51.   IF CAMINHO_ALVO = 07 THEN
52.      SET STRING_PARA_SOLVER TO STRING_PARA_SOLVER &
53.      "(ASSERT+ (< IR_A_PAGAR 125))\N"
54.   END-IF
55.   IF  IR_A_PAGAR  >=   125   THEN
```

```
56.        IF CAMINHO_ALVO = 05 THEN
57.           SET STRING_PARA_SOLVER TO STRING_PARA_SOLVER &
58.           "(ASSERT+ (< SALARIO 2000))\N"
59.        END-IF
60.        IF CAMINHO_ALVO = 06 THEN
61.           SET STRING_PARA_SOLVER TO STRING_PARA_SOLVER &
62.           "(ASSERT+ (>= SALARIO 2000))\N"
63.        END-IF
64.        IF SALARIO < 2000 THEN
65.           SET STRING_PARA_SOLVER TO STRING_PARA_SOLVER &
66.           "(ASSERT+ (= AUMENTO 10))\N"
67.           MOVE 10 TO AUMENTO
68.        ELSE
69.           SET STRING_PARA_SOLVER TO STRING_PARA_SOLVER &
70.           "(ASSERT+ (= AUMENTO 5))\N"
71.           MOVE 5 TO AUMENTO
72.    ELSE
73.        SET STRING_PARA_SOLVER TO STRING_PARA_SOLVER &
74.        "(ASSERT+ (= AUMENTO 3))\N"
75.        MOVE 3 TO AUMENTO.
76.    STOP RUN
77.    END.
```

Fonte: elaborado pelo autor

Passo 4: Interrupção e Chamada do *Solver* nos Pontos de Decisão

Nesse passo o código-fonte é instrumentado a fim de que a execução seja interrompida a cada instrução de decisão. Nessa interrupção é acionado um *solver* (Yices) que retorna o dado de entrada que faz com que o SUT entre no caminho desejado.

No programa de exemplo, a primeira instrução de decisão é a instrução de decisão "`IF`" da linha 13 da Tabela 4. A técnica C4T inclui mais uma restrição para o solver passando o conteúdo corrente da variável a ser comparada. Nesse caso, será colocada a restrição "`(assert (= salario`" concatenado com o conteúdo corrente da variável SALARIO. Isso é necessário para os casos em que a variável de entrada é diferente do que seria retornado pelo *solver* mas o próprio dado de entrada corrente já atenderia as restrições do solver.

Em seguida a técnica C4T insere o comando "`check`", que indica para o solver que ele deve devolver um valor que atenda às restrições dadas, e faz a chamada do solver (comando "`set resultado_solver to yices::solver(`

string_para_solver") passando as restrições inseridas até aquele momento (armazenadas na variável "STRING_PARA_SOLVER", e o *solver* retorna a resposta na variável "RESULTADO_SOLVER". Em seguida é colocada uma condição para avaliar se o dado de entrada corrente é igual ao retornado pelo solver. Caso seja igual, o programa continua normalmente; caso contrário é acionado novamente o solver com a indicação de não usar a ultima restrição (que atribui o conteúdo corrente da variável a ser comparada) e novamente o conteúdo retornado pelo solver é comparado com a variável de entrada. Caso seja igual o programa segue normalmente; caso contrário o programa é interrompido e é devolvido o valor retornado pelo solver para o programa controlador.

Esse trecho de código é colocado na linha imediatamente anterior aos demais pontos de decisão do programa, que para esse programa de exemplo são os comandos "IF" constantes nas linhas 17, 24 e 25 da Tabela 4. Na linha 24 o trecho de código inserido é bem semelhante ao anterior, com a única diferença de que a comparação do resultado do solver é feita com a variável IR_A_PAGAR (em vez da variável SALARIO), pois é essa a variável que é usada para a decisão dessa linha de código.

A Tabela 7 mostra o código-fonte do programa após a aplicação do passo 4 da técnica C4T.

Tabela 7 - Código-fonte após aplicação do passo 4 da técnica C4T

```
1. PROCEDURE DIVISION.
2. SET STRING_PARA_SOLVER TO STRING_PARA_SOLVER &
3.     "(DEFINE SALARIO:REAL)\N"
4. SET STRING_PARA_SOLVER TO STRING_PARA_SOLVER &
5.     "(DEFINE AUMENTO:REAL)\N"
6. SET STRING_PARA_SOLVER TO STRING_PARA_SOLVER &
7.     "(DEFINE ALIQUOTA_IR:REAL)\N"
8. SET STRING_PARA_SOLVER TO STRING_PARA_SOLVER &
9.     "(DEFINE PARCELA_A_DEDUZIR:REAL)\N"
10.   SET STRING_PARA_SOLVER TO STRING_PARA_SOLVER &
11.       "(DEFINE IR_A_PAGAR:REAL)\N"
12.   SET STRING_PARA_SOLVER TO STRING_PARA_SOLVER &
13.       "(ASSERT+ (> SALARIO 0))\N"
14.   SET STRING_PARA_SOLVER TO STRING_PARA_SOLVER &
15.       "(ASSERT+ (= AUMENTO 0))\N"
16.   MOVE 0 TO AUMENTO
17.   IF CAMINHO_ALVO = 02 THEN
18.      SET STRING_PARA_SOLVER TO STRING_PARA_SOLVER &
19.          "(ASSERT+ (<= SALARIO 1499))\N"
20.   END-IF
21.   SET STRING_PARA_SOLVER TO STRING_PARA_SOLVER &
```

```
22.        "(ASSERT+ (<= SALARIO " & SALARIO & ")\N"
23.    SET STRING_PARA_SOLVER TO STRING_PARA_SOLVER &
24.        "(CHECK)\N"
25.    SET RESULTADO_SOLVER TO YICES::SOLVER(STRING_PARA_SOLVER)
26.    IF SALARIO <> RESULTADO_SOLVER THEN
27.       SET RESULTADO_SOLVER TO YICES::SOLVER(STRING_PARA_SOLVER,
28.           "N")
29.       IF SALARIO <> RESULTADO_SOLVER THEN
30.          MOVE "N" TO ATINGIU_CAMINHO
31.          STOP RUN
32.       END-IF
33.    END-IF
34.    IF   SALARIO     <=    1499 THEN
35.       SET STRING_PARA_SOLVER TO STRING_PARA_SOLVER &
36.           "(ASSERT+ (= ALIQUOTA_IR 0))\N"
37.        MOVE 0 TO ALIQUOTA_IR
38.        SET STRING_PARA_SOLVER TO STRING_PARA_SOLVER &
39.           "(ASSERT+ (= PARCELA_A_DEDUZIR 0))\N"
40.        MOVE 0 TO PARCELA_A_DEDUZIR
41.        GO TO CALCULA_VALOR_IMPOSTO_RENDA.
42.    IF CAMINHO_ALVO = 03 THEN
43.       SET STRING_PARA_SOLVER TO STRING_PARA_SOLVER &
44.       "(ASSERT+ (> SALARIO 1499))\N"
45.    END-IF
46.    SET STRING_PARA_SOLVER TO STRING_PARA_SOLVER &
47.        "(ASSERT+ (<= SALARIO " & SALARIO & ")\N"
48.    SET STRING_PARA_SOLVER TO STRING_PARA_SOLVER &
49.        "(CHECK)\N"
50.    SET RESULTADO_SOLVER TO YICES::SOLVER(STRING_PARA_SOLVER)
51.    IF SALARIO <> RESULTADO_SOLVER THEN
52.       SET RESULTADO_SOLVER TO YICES::SOLVER(STRING_PARA_SOLVER,
53.           "N")
54.       IF SALARIO <> RESULTADO_SOLVER THEN
55.          MOVE "N" TO ATINGIU_CAMINHO
56.          STOP RUN
57.       END-IF
58.    END-IF
59.    IF SALARIO       >    1499
60.       SET STRING_PARA_SOLVER TO STRING_PARA_SOLVER &
61.           (ASSERT+ (= ALIQUOTA_IR 27))\N"
62.        MOVE 27 TO ALIQUOTA_IR
63.        SET STRING_PARA_SOLVER TO STRING_PARA_SOLVER &
64.           "(ASSERT+ (= PARCELA_A_DEDUZIR 306))\N"
65.        MOVE 306 TO PARCELA_A_DEDUZIR
66.        GO TO CALCULA_VALOR_IMPOSTO_RENDA.
67.    CALCULA_VALOR_IMPOSTO_RENDA.
68.       SET STRING_PARA_SOLVER TO STRING_PARA_SOLVER &
69.       "(ASSERT+ (= IR_A_PAGAR (+ (/ (* SALARIO ALIQUOTA_IR)
70.        100) (* PARCELA_A_DEDUZIR -1))))"
71.    COMPUTE  IR_A_PAGAR  = (SALARIO * ALIQUOTA_IR / 100)  -
72.                           PARCELA_A_DEDUZIR
73.    IF CAMINHO_ALVO = 04 THEN
74.       SET STRING_PARA_SOLVER TO STRING_PARA_SOLVER &
75.       "(ASSERT+ (>= IR_A_PAGAR 125))\N"
76.    END-IF
77.    IF CAMINHO_ALVO = 07 THEN
```

```
78.       SET STRING_PARA_SOLVER TO STRING_PARA_SOLVER &
79.       "(ASSERT+ (< IR_A_PAGAR 125))\N"
80.    END-IF
81.    SET STRING_PARA_SOLVER TO STRING_PARA_SOLVER &
82.        "(ASSERT+ (<= IR_A_PAGAR " & IR_A_PAGAR & ")\N"
83.    SET STRING_PARA_SOLVER TO STRING_PARA_SOLVER &
84.        "(CHECK)\N"
85.    SET RESULTADO_SOLVER TO YICES::SOLVER(STRING_PARA_SOLVER)
86.    IF IR_A_PAGAR <> RESULTADO_SOLVER THEN
87.       SET RESULTADO_SOLVER TO YICES::SOLVER(STRING_PARA_SOLVER,
88.           "N")
89.       IF IR_A_PAGAR <> RESULTADO_SOLVER THEN
90.          MOVE "N" TO ATINGIU_CAMINHO
91.          STOP RUN
92.       END-IF
93.    END-IF
94.    IF  IR_A_PAGAR  >=   125  THEN
95.        IF CAMINHO_ALVO = 05 THEN
96.           SET STRING_PARA_SOLVER TO STRING_PARA_SOLVER &
97.           "(ASSERT+ (< SALARIO 2000))\N"
98.        END-IF
99.        IF CAMINHO_ALVO = 06 THEN
100.          SET STRING_PARA_SOLVER TO STRING_PARA_SOLVER &
101.          "(ASSERT+ (>= SALARIO 2000))\N"
102.       END-IF
103.       SET STRING_PARA_SOLVER TO STRING_PARA_SOLVER &
104.           "(ASSERT+ (<= SALARIO " & SALARIO & ")\N"
105.       SET STRING_PARA_SOLVER TO STRING_PARA_SOLVER &
106.           "(CHECK)\N"
107.       SET RESULTADO_SOLVER TO YICES::SOLVER(STRING_PARA_SOLVER)
108.       IF SALARIO <> RESULTADO_SOLVER THEN
109.         SET RESULTADO_SOLVER TO YICES::SOLVER(STRING_PARA_SOLVER,
110.             "N")
111.         IF SALARIO <> RESULTADO_SOLVER THEN
112.            MOVE "N" TO ATINGIU_CAMINHO
113.            STOP RUN
114.         END-IF
115.       END-IF
116.       IF SALARIO < 2000 THEN
117.          SET STRING_PARA_SOLVER TO STRING_PARA_SOLVER &
118.          "(ASSERT+ (= AUMENTO 10))\N"
119.          MOVE 10 TO AUMENTO
120.       ELSE
121.          SET STRING_PARA_SOLVER TO STRING_PARA_SOLVER &
122.          "(ASSERT+ (= AUMENTO 5))\N"
123.          MOVE 5 TO AUMENTO
124.   ELSE
125.       SET STRING_PARA_SOLVER TO STRING_PARA_SOLVER &
126.       "(ASSERT+ (= AUMENTO 3))\N"
127.       MOVE 3 TO AUMENTO.
128.   STOP RUN
129.   END.
```

Fonte: elaborado pelo autor

O APÊNDICE 1 – Instrumentações no Código-Fonte da Técnica C4T mostra as instrumentações feitas no programa COBOL após a aplicação dos passos 3 e 4 da técnica C4T.

Passo 5: Programa controlador

Esse passo cria um programa controlador que aciona o programa a ser testado para atingir cada caminho de teste desejado. Esse programa controlador armazena os dados de entrada e os resultados de saída de cada caminho desejado em uma base de dados, para que depois essa tabela possa ser usada na geração dos casos de teste no padrão FitNesse.

A Figura 15 mostra a estrutura do programa controlador. Inicialmente são buscados os caminhos selecionados pelo engenheiro de testes. Caso os caminhos selecionados não tenham terminado, é chamado o SUT passando os dados de entrada (iniciando com zeros ou retornados pela execução anterior) e o caminho desejado. O SUT devolve se com os dados de entrada o caminho desejado foi atingido ou não. Caso o caminho desejado não tenha sido atingido, o SUT devolve o valor retornado pelo solver ao programa controlador, e o programa controlador chama novamente o SUT com o novo dado de entrada retornado pelo solver. Caso o caminho desejado seja atingido, o programa controlador grava os dados de entrada e o resultado obtido pelo caminho desejado. Esse ciclo se repete até que todos os caminhos desejados tenham os casos de teste gerados. Ao término da execução de todos os caminhos desejados o programa controlador gera os casos de teste dos caminhos escolhidos no formato FitNesse.

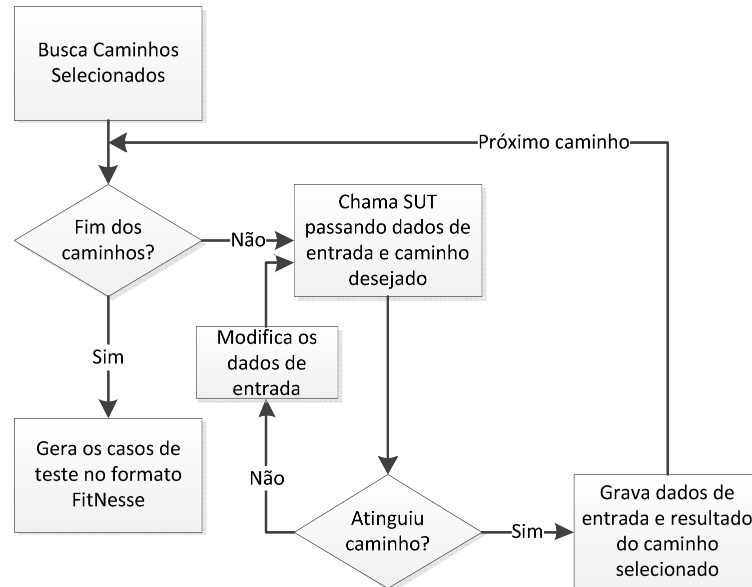


Figura 15 - Fluxo do programa controlador da técnica C4T

Fonte: elaborado pelo autor

O APÊNDICE 2 – Código-Fonte do Programa Controlador apresenta o código-fonte, escrito na linguagem COBOL, do programa controlador aplicado nesse passo da técnica C4T.

Passo 6: Execução do Programa

Nesse passo é feita a execução do programa com as interrupções necessárias. Quando o programa controlador identificar que todos os caminhos do SUT foram atingidos é acionado um programa que lê a base de dados do item 5 e cria os casos de teste no padrão FitNesse. Cada caso de teste contem os dados de entrada e o resultado gerado a partir desses dados de entrada.

A seguir serão mostrados os ciclos da execução dinâmica considerando que o engenheiro de testes selecionou todos os caminhos identificados no passo 2, constantes da Figura 14.

No ciclo 01 os dados de entrada estarão zerados, então `SALARIO = 0`. O programa controlador aciona o programa IMPOSTO_RENDA com `salário = 0` e fluxo 1 (que engloba os caminhos-alvo 01, 02, 04, 05 e 08). O programa IMPOSTO_RENDA executa até a linha 25 da Tabela 7 e chama o solver com as

restrições constantes na Tabela 8. Com essas restrições o solver retorna unsat, que significa que o dado de entrada corrente (SALARIO = 0) não atende as restrições do solver. O programa chama o *solver* novamente (linha 27 da Tabela 7) com as restrições constantes na Tabela 9, cuja única alteração em relação às restrições da Tabela 8 é que não é incluída a última restrição (SALARIO = 0). O solver retorna que o salário que atende essas restrições é 1. Uma vez que o valor retornado pelo *solver* (`SALARIO = 1`) é diferente do dado de entrada (`SALARIO = 0`), o programa é interrompido e reexecutado com o valor de entrada `SALARIO = 1`.

Tabela 8 - Restrições do solver - ciclo 01a

```
(assert+ (> SALARIO 0))
(assert+ (<= SALARIO 1499))
(assert+ (= SALARIO 0))
(check)
Resultado do solver: unsat
```

Tabela 9 - Restrições do solver - ciclo 01b

```
(assert+ (> SALARIO 0))
(assert+ (<= SALARIO 1499))
(check)
Resultado do solver: SALARIO = 1
```

No ciclo 02 o programa controlador aciona o programa IMPOSTO_RENDA com `SALARIO = 1` (retornado pelo solver no ciclo anterior) e os caminhos-alvo iguais ao do ciclo anterior. O programa IMPOSTO_RENDA executa até a linha 25 da Tabela 7 e chama o solver com as restrições constantes na Tabela 10. Dessa vez o valor retornado pelo *solver* (`SALARIO = 1`) é igual ao dado de entrada então o programa continua a execução até a linha 85 da Tabela 7 e chama o solver com as restrições constantes na Tabela 11. O *solver* retorna `unsat`, que significa os dados de entrada correntes não atendem às restrições. O programa chama o *solver* novamente (linha 87 da Tabela 7) com as restrições constantes na Tabela 12. O *solver* retorna novamente `unsat`, que significa que não há nenhum valor que atenda as restrições dadas, pois com essas restrições não há nenhum valor de salário que faça com que o `IR_A_PAGAR` seja maior do que 125. O programa controlador registra que para o fluxo 1 não há uma solução possível, e nesse caso o valor de saída da variável `AUMENTO` é 0.

Tabela 10 - Restrições do solver - ciclo 02a

```
(assert+ (> SALARIO 0))
(assert+ (<= SALARIO 1499))
(assert+ (= SALARIO 1))
(check)
Resultado do Solver: SALARIO = 1
```

Tabela 11 - Restrições do solver - ciclo 02b

```
(assert+ (> SALARIO 0))
(assert+ (<= SALARIO 1499))
(assert+ (= ALIQUOTA_IR 0))
(assert+ (= PARCELA_A_DEDUZIR 0))
(assert+ (= IR_A_PAGAR (+ (/ (* SALARIO 0) 100) (* PARCELA_A_DEDUZIR -1))))
(assert+ (>= IR_A_PAGAR 125))
(assert+ (= IR_A_PAGAR 0))
(check)
Resultado do Solver: unsat
```

Tabela 12 - Restrições do solver - ciclo 02c

```
(assert+ (> SALARIO 0))
(assert+ (<= SALARIO 1499))
(assert+ (= ALIQUOTA_IR 0))
(assert+ (= PARCELA_A_DEDUZIR 0))
(assert+ (= IR_A_PAGAR (+ (/ (* SALARIO 0) 100) (* PARCELA_A_DEDUZIR -1))))
(assert+ (>= IR_A_PAGAR 125))
(check)
Resultado do Solver: unsat
```

O ciclo 03 é uma repetição do ciclo 01 pois os dados de chamada do *solver* e o resultado são iguais, e novamente o programa é interrompido e reexecutado com o valor de entrada `SALARIO = 1`.

No ciclo 04 o programa controlador aciona o programa IMPOSTO_RENDA com `SALARIO = 1` (retornado pelo *solver* no ciclo anterior) e os caminhos-alvo iguais ao do ciclo anterior. O programa IMPOSTO_RENDA executa até a linha 25 da Tabela 7 e chama o solver com as restrições constantes na Tabela 10. Dessa vez o valor retornado pelo *solver* (`SALARIO = 1`) é igual ao dado de entrada então o programa continua a execução até a linha 85 da Tabela 7 e chama o solver com as restrições constantes na Tabela 11. O *solver* retorna `unsat`, que significa que os dados de entrada correntes não atendem às restrições. O programa chama o *solver* novamente (linha 87 da Tabela 7) com as restrições constantes na Tabela 12. O *solver* retorna `unsat`, que significa que não há nenhum valor que atenda as restrições dadas, pois com essas restrições não há nenhum valor de salário que faça com que o `IR_A_PAGAR` seja maior do que 125. O programa controlador registra que para o fluxo 2 não há uma solução possível e nesse caso o valor de saída da variável `AUMENTO` é 0.

O ciclo 05 é uma repetição dos ciclos 01 e 03 pois os dados de chamada do *solver* e o resultado são iguais, e novamente o programa é interrompido e reexecutado com o valor de entrada `SALARIO = 1`.

No ciclo 06 o programa controlador aciona o programa IMPOSTO_RENDA com `salário = 1` (retornado pelo *solver* no ciclo anterior) e os caminhos-alvo iguais ao do ciclo anterior. O programa IMPOSTO_RENDA executa até a linha 25 da Tabela 7 e chama o *solver* com as restrições constantes na Tabela 8. Dessa vez o valor retornado pelo *solver* (`SALARIO = 1`) é igual ao dado de entrada então o programa continua a execução até a linha 85 da Tabela 7 e chama o solver com as restrições constantes na Tabela 13. O *solver* retorna `SALARIO = 1` (valor igual ao do dado de entrada), então o programa executa até o final e atinge os caminhos-alvo. O programa controlador registra que para o fluxo 3 o dado de entrada é `SALARIO = 1` e o dado de saída é `AUMENTO = 3`.

Tabela 13 - Restrições do solver - ciclo 06

```
(assert+ (> SALARIO 0))
(assert+ (<= SALARIO 1499))
(assert+ (= ALIQUOTA_IR 0))
(assert+ (= PARCELA_A_DEDUZIR 0))
(assert+ (= IR_A_PAGAR (+ (/ (* SALARIO 0) 100) (* PARCELA_A_DEDUZIR -1))))
(assert+ (< IR_A_PAGAR 125))
(assert+ (= IR_A_PAGAR 0))
(check)
Resultado do Solver: SALARIO = 1
```

No ciclo 07 o programa controlador aciona o programa IMPOSTO_RENDA com `salário = 0` e fluxo 4 (que engloba os caminhos-alvo 01, 03, 04, 05 e 08). O programa IMPOSTO_RENDA executa até a linha 25 da Tabela 7 e chama o *solver* com as restrições constantes na Tabela 14. O *solver* retorna `unsat`, que significa que os dados de entrada correntes não atendem às restrições. O programa chama o *solver* novamente (linha 87 da Tabela 7) com as restrições constantes na Tabela 15. Com essas restrições o solver retorna que o salário que atende essas restrições é 1500. Uma vez que o valor retornado pelo *solver* (`SALARIO = 1500`) é diferente do dado de entrada (`SALARIO = 0`), o programa é interrompido e reexecutado com o valor de entrada `SALARIO = 1500`.

Tabela 14 - Restrições do solver - ciclo 07a

```
(assert+ (> SALARIO 0))
(assert+ (> SALARIO 1499))
(assert+ (= SALARIO 0))
(check)
Resultado do Solver: unsat
```

Tabela 15 - Restrições do solver - ciclo 07b

```
(assert+ (> SALARIO 0))
(assert+ (> SALARIO 1499))
(check)
Resultado do Solver: SALARIO = 1500
```

No ciclo 08 o programa controlador aciona o programa IMPOSTO_RENDA com `salário = 1500` (retornado pelo *solver* no ciclo anterior) e os caminhos-alvo iguais ao do ciclo anterior. O programa IMPOSTO_RENDA executa até a linha 25 da Tabela 7 e chama o *solver* com as restrições constantes na Tabela 16. Dessa vez o valor retornado pelo *solver* (`SALARIO = 1500`) é igual ao dado de entrada então o programa continua a execução até a linha 85 da Tabela 7 e chama o *solver* com as restrições constantes na Tabela 17. O *solver* retorna `unsat`, que significa que os dados de entrada correntes não atendem às restrições. O programa chama o *solver* novamente (linha 87 da Tabela 7) com as restrições constantes na Tabela 15. Com essas restrições o solver retorna que o salário que atende essas restrições é 1500. Uma vez que o valor retornado pelo *solver* (`SALARIO = 1600`) é diferente do dado de entrada (`SALARIO = 1500`), o programa é interrompido e reexecutado com o valor de entrada `SALARIO = 1600.`

Tabela 16 - Restrições do solver - ciclo 08a

```
(assert+ (> SALARIO 0))
(assert+ (> SALARIO 1499))
(assert+ (= SALARIO 1500))
(check)
Resultado do Solver: SALARIO = 1500
```

Tabela 17 - Restrições do solver - ciclo 08b

```
(assert+ (> SALARIO 0))
(assert+ (> SALARIO 1499))
(assert+ (= ALIQUOTA_IR 27))
(assert+ (= PARCELA_A_DEDUZIR 306))
(assert+ (= IR_A_PAGAR (+ (/ (* SALARIO 27) 100) (* PARCELA_A_DEDUZIR -1))))
(assert+ (> IR_A_PAGAR 125))
(assert+ (= IR_A_PAGAR 99))
(check)
Resultado do Solver: unsat
```

Tabela 18 Restrições do solver - ciclo 08c

```
(assert+ (> SALARIO 0))
(assert+ (> SALARIO 1499))
(assert+ (= ALIQUOTA_IR 27))
(assert+ (= PARCELA_A_DEDUZIR 306))
(assert+ (= IR_A_PAGAR (+ (/ (* SALARIO 27) 100) (* PARCELA_A_DEDUZIR -1))))
(assert+ (> IR_A_PAGAR 125))
(check)
Resultado do Solver: SALARIO = 1600
```

No ciclo 09 o programa controlador aciona o programa IMPOSTO_RENDA com `salário = 1600` (retornado pelo *solver* no ciclo anterior) e os caminhos-alvo

iguais ao do ciclo anterior. O programa IMPOSTO_RENDA executa até a linha 25 da Tabela 7 e chama o *solver* com as restrições constantes na Tabela 19. Dessa vez o valor retornado pelo *solver* (`SALARIO = 1600`) é igual ao dado de entrada então o programa continua a execução até a linha 85 da Tabela 7 e chama o *solver* com as restrições constantes na Tabela 17. O *solver* retorna `SALARIO = 1600` (valor igual ao do dado de entrada), então o programa continua a execução até a linha 107 da Tabela 7 e chama o *solver* com as restrições constantes na Tabela 21. O *solver* retorna `SALARIO = 1600` (valor igual ao do dado de entrada), então o programa executa até o final e atinge os caminhos-alvo. O programa controlador registra que para o fluxo 4 o dado de entrada é `SALARIO = 1600` e o dado de saída é `AUMENTO = 10`

Tabela 19 - Restrições do solver - ciclo 09a

```
(assert+ (> SALARIO 0))
(assert+ (> SALARIO 1499))
(assert+ (= SALARIO 1600))
(check)
Resultado do Solver: SALARIO = 1600
```

Tabela 20 - Restrições do solver - ciclo 09b

```
(assert+ (> SALARIO 0))
(assert+ (> SALARIO 1499))
(assert+ (= ALIQUOTA_IR 27))
(assert+ (= PARCELA_A_DEDUZIR 306))
(assert+ (= IR_A_PAGAR (+ (/ (* SALARIO 27) 100) (* PARCELA_A_DEDUZIR -1))))
(assert+ (> IR_A_PAGAR 125))
(assert+ (= IR_A_PAGAR 126))
(check)
Resultado do Solver: SALARIO = 1600
```

Tabela 21 - Restrições do solver - ciclo 09c

```
(assert+ (> SALARIO 0))
(assert+ (> SALARIO 1499))
(assert+ (= ALIQUOTA_IR 27))
(assert+ (= PARCELA_A_DEDUZIR 306))
(assert+ (= IR_A_PAGAR (+ (/ (* SALARIO 27) 100) (* PARCELA_A_DEDUZIR -1))))
(assert+ (> IR_A_PAGAR 125))
(assert+ (< SALARIO 2000))
(assert+ (= SALARIO 1600))
(check)
Resultado do Solver: SALARIO = 1600
```

O ciclo 10 é uma repetição do ciclo 07, mas buscando o fluxo 5 (que engloba os caminhos-alvo 01, 03, 04, 06 e 08). Os dados de chamada do *solver* e o resultado são iguais, e novamente o programa é interrompido e reexecutado com o valor de entrada `SALARIO = 1500`.

O ciclo 11 é uma repetição do ciclo 08. Os dados de chamada do *solver* e o resultado são iguais, e novamente o programa é interrompido e reexecutado com o valor de entrada `SALARIO = 1600`.

O ciclo 12 é uma repetição do ciclo 09 até a execução da linha 107 da Tabela 7, quando chama o *solver* com as restrições constantes na Tabela 22. O *solver* retorna `unsat`, que significa que os dados de entrada correntes não atendem às restrições. O programa chama o *solver* novamente (linha 109 da Tabela 7) com as restrições constantes na Tabela 23. Com essas restrições o solver retorna que o salário que atende essas restrições é 2000. Uma vez que o valor retornado pelo *solver* (`SALARIO = 2000`) é diferente do dado de entrada (`SALARIO = 1600`), o programa é interrompido e reexecutado com o valor de entrada `SALARIO = 2000.`

Tabela 22 - Restrições do solver - ciclo 12a

```
(assert+ (> SALARIO 0))
(assert+ (> SALARIO 1499))
(assert+ (= ALIQUOTA_IR 27))
(assert+ (= PARCELA_A_DEDUZIR 306))
(assert+ (= IR_A_PAGAR (+ (/ (* SALARIO 27) 100) (* PARCELA_A_DEDUZIR -1))))
(assert+ (> IR_A_PAGAR 125))
(assert+ (>= SALARIO 2000))
(assert+ (= SALARIO 1600))
(check)
Resultado do Solver: unsat
```

Tabela 23 - Restrições do solver - ciclo 12b

```
(assert+ (> SALARIO 0))
(assert+ (> SALARIO 1499))
(assert+ (= ALIQUOTA_IR 27))
(assert+ (= PARCELA_A_DEDUZIR 306))
(assert+ (= IR_A_PAGAR (+ (/ (* SALARIO 27) 100) (* PARCELA_A_DEDUZIR -1))))
(assert+ (> IR_A_PAGAR 125))
(assert+ (>= SALARIO 2000))
(check)
Resultado do Solver: SALARIO = 2000
```

No ciclo 13 o programa controlador aciona o programa IMPOSTO_RENDA com `SALARIO = 2000` (retornado pelo *solver* no ciclo anterior) e os caminhos-alvo iguais ao do ciclo anterior. O programa IMPOSTO_RENDA executa até a linha 25 da Tabela 7 e chama o *solver* com as restrições constantes na Tabela 24. Dessa vez o valor retornado pelo *solver* (`SALARIO = 2000`) é igual ao dado de entrada então o programa continua a execução até a linha 85 da Tabela 7 e chama o *solver* com as restrições constantes na Tabela 25. O *solver* retorna `SALARIO = 2000` (valor igual ao do dado de entrada), então o programa continua a execução até a linha 107 da Tabela 7 e chama o *solver* com as restrições constantes na Tabela 26. O *solver* retorna `SALARIO = 2000` (valor igual ao do dado de entrada), então o programa

executa até o final e atinge os caminhos-alvo. O programa controlador registra que para o fluxo 5 o dado de entrada é `SALARIO = 2000` e o dado de saída é `AUMENTO = 5`.

Tabela 24 - Restrições do solver - ciclo 13a

```
(assert+ (> SALARIO 0))
(assert+ (> SALARIO 1499))
(assert+ (= SALARIO 2000))
(check)
Resultado do Solver: SALARIO = 2000
```

Tabela 25 - Restrições do solver - ciclo 13b

```
(assert+ (> SALARIO 0))
(assert+ (> SALARIO 1499))
(assert+ (= ALIQUOTA_IR 27))
(assert+ (= PARCELA_A_DEDUZIR 306))
(assert+ (= IR_A_PAGAR (+ (/ (* SALARIO 27) 100) (* PARCELA_A_DEDUZIR -1))))
(assert+ (> IR_A_PAGAR 125))
(assert+ (= IR_A_PAGAR 234))
(check)
Resultado do Solver: SALARIO = 2000
```

Tabela 26 - Restrições do solver - ciclo 13c

```
(assert+ (> SALARIO 0))
(assert+ (> SALARIO 1499))
(assert+ (= ALIQUOTA_IR 27))
(assert+ (= PARCELA_A_DEDUZIR 306))
(assert+ (= IR_A_PAGAR (+ (/ (* SALARIO 27) 100) (* PARCELA_A_DEDUZIR -1))))
(assert+ (> IR_A_PAGAR 125))
(assert+ (>= SALARIO 2000))
(assert+ (= SALARIO 2000))
(check)
Resultado do Solver: SALARIO = 2000
```

O ciclo 14 é uma repetição do ciclo 07, mas buscando o fluxo 6 (que engloba os caminhos-alvo 01, 03, 07 e 08). Os dados de chamada do *solver* e o resultado são iguais, e novamente o programa é interrompido e reexecutado com o valor de entrada `SALARIO = 1500`.

No ciclo 15 o programa controlador aciona o programa IMPOSTO_RENDA com `salário = 1500` (retornado pelo *solver* no ciclo anterior) e os caminhos-alvo iguais ao do ciclo anterior. O programa IMPOSTO_RENDA executa até a linha 25 da Tabela 7 e chama o *solver* com as restrições constantes na Tabela 16. Dessa vez o valor retornado pelo *solver* (`SALARIO = 1500`) é igual ao dado de entrada então o programa continua a execução até a linha 85 da Tabela 7 e chama o *solver* com as restrições constantes na Tabela 27. O *solver* retorna SALARIO = 1500 (valor igual ao do dado de entrada corrente), então o programa continua a execução até o final e atinge os caminhos-alvo. O programa controlador registra que para o fluxo 6 o dado de entrada é `SALARIO = 1500` e o dado de saída é `AUMENTO = 3`.

Tabela 27 - Restrições do solver - ciclo 15

```
(assert+ (> SALARIO 0))
(assert+ (> SALARIO 1499))
(assert+ (= ALIQUOTA_IR 27))
(assert+ (= PARCELA_A_DEDUZIR 306))
(assert+ (= IR_A_PAGAR (+ (/ (* SALARIO 27) 100) (* PARCELA_A_DEDUZIR -1))))
(assert+ (<= IR_A_PAGAR 125))
(assert+ (= SALARIO 1500))
(check)
Resultado do Solver: SALARIO = 1500
```

Os resultados de todos os ciclos são gravados em uma base de dados conforme mostrado na Tabela 28.

Tabela 28 - Resultados de entrada e saída após a execução da técnica C4T

Fluxo	Caminhos	Salário	Aliquota	Parcela a deduzir	IR a pagar	Aumento	Observação
01	01, 02, 04, 05 e 08	1	0	0	0	0	Caminho não possível
02	01, 02, 04, 06 e 08	1	0	0	0	0	Caminho não possível
03	01, 02, 07 e 08	1	0	0	0	3	
04	01, 03, 04,05 e 08	1600	27	306	126	10	
05	01, 03, 04, 06 e 08	2000	27	306	234	5	
06	01, 03, 07 e 08	1500	27	306	99	3	

Fonte: Elaborado pelo autor

Para esse exemplo os casos de teste gerados no padrão FitNesse correspondem ao mostrado na Figura 16. Nessa figura somente são mostrados os dados de entrada e saída pois para o FitNesse não importam às variáveis internas do programa que está sendo testado. Também observa-se nessa figura que só aparecem os dados de entrada para os fluxos 3, 4, 5 e 6, pois para os fluxos 1 e 2 não há dado de entrada que atenda às restrições desses fluxos.

classpath: C:\workspace\ImpostoRendaJava.jar

br.ipt.imposto.renda.FixtureImposto	
salario	aumento?
1	3
1600	10
2000	5
1500	3

Figura 16 - Tabela FitNesse do programa exemplo
Fonte: elaborado pelo autor

Uma limitação da técnica de utilização do *solver* yices para resolução de restrições é que esse solver não trabalha com variáveis do tipo *string*. Para tentar resolver esse problema uma solução poderia ser a conversão das variáveis *string* em variáveis numéricas. Exemplo: converter a variável “A” em “01”, a variável “B” em “02”, e assim por diante.

Nota-se pela Tabela 28 que para vários fluxos do programa de exemplo (fluxos 01 e 02) não é possível encontrar dados que façam com que o fluxo seja executado. Essa indicação permite que o engenheiro de testes concentre a análise nos demais fluxos do programa, evitando desperdício de tempo.

4 ANÁLISE DOS RESULTADOS

Para efeito de avaliação da eficiência da técnica foram criados os casos de teste do programa de exemplo de forma manual, aplicando principalmente a técnica de valor-limite. Nela, o código fonte do programa é analisado para identificar variáveis que são utilizadas nas condições de decisão do programa. Um exemplo da utilização da técnica de valor-limite seria em uma instrução de decisão (IF) que possua uma instrução do tipo "`IF NUMERO <= 5`". Nesse caso, a técnica de valor-limite usaria dois valores para a variável "`NUMERO`": valor igual a 5 e algum valor menor do que 5 (poderia ser o valor 4).

Uma das condições de decisão do programa de exemplo usado nesse trabalho está nas linhas 13 e 17 da Tabela 4, nas quais é verificado se a variável `SALARIO` é maior, menor ou igual ao valor 1499, então essas são duas condições do programa. Nas linhas 24 e 28 da Tabela 4 é verificado se a variável `IR_A_PAGAR` é maior, menor ou igual ao valor 125, caracterizando mais duas condições do programa. Nas linhas 25 e 27 da Tabela 4 é verificado se a variável `SALARIO` é maior, menor ou igual ao valor 2000, caracterizando mais duas condições do programa. Para a geração dos casos de teste do programa é feita uma combinação de todas as condições de decisão citadas acima. Os casos de teste gerados manualmente são mostrados na Tabela 29.

Tabela 29 - Casos de teste gerados manualmente

Fluxo	Salário (condição 1)	IR a Pagar	Salário (condição 2)	Salário (entrada)	Aumento
1	<= 1499	>= 125	< 2000	Não tem	0
2	<= 1499	>= 125	>= 2000	Não tem	0
3a	<= 1499	< 125	n/a	1498	3
3b	<= 1499	< 125	n/a	1499	3
4a	> 1499	>= 125	< 2000	1600	10
4b	> 1499	>= 125	< 2000	1999	10
5a	> 1499	>= 125	>= 2000	2000	5
5b	> 1499	>= 125	>= 2000	2001	5
6a	> 1499	< 125	n/a	1500	3
6b	> 1499	< 125	n/a	1586	3

Observa-se que para cada fluxo que possui dados de entrada válidos foram gerados 2 casos de teste, que procuram valores próximos aos valores-limite de cada condição. Exemplo: para a condição "`SALARIO <= 1499`" foram gerados 2 casos

de teste: um caso de teste "`SALARIO = 1498`" para testar a condição "menor que" e outro caso de teste "`SALARIO = 1499`" para testar a condição "igual a".

Comparando-se os 10 casos de teste gerados manualmente (Tabela 29) com os 6 casos de teste gerados automaticamente (Tabela 28), eles mostraram a mesma eficácia, e uma vez que a quantidade de casos de teste gerados automaticamente foi menor, pode-se considerar que os os casos de teste gerados automaticamente foram mais eficientes que os casos de teste gerados manualmente para o programa de exemplo.

Com os casos de teste do programa é possível reescrevê-lo em qualquer outra linguagem pois pode-se retestar o programa para garantir que os dados de entrada gerem os mesmos dados de saída do programa original. Para simular essa situação o programa de exemplo escrito originalmente em COBOL foi reescrito em linguagem Java. O código-fonte do programa Java está no APÊNDICE 3 – Código-Fonte do Programa Exemplo Convertido Para Java.

Os casos de teste no padrão FitNesse da Figura 16, gerados automaticamente pela técnica C4T, foram executados pelo servidor FitNesse e o resultado consta na Tabela 30. Para permitir a execução desses casos de teste foi criado um programa *fixture* para fazer a ligação entre o servidor FitNesse e o programa Java. O código-fonte do programa fixture consta no APÊNDICE 4 – Código-Fonte do Programa Fixture do FitNesse.

Tabela 30 - Execução pelo FitNesse dos CT's gerados pela técnica C4T

salario	aumento?
1	3
1600	10
2000	5
1500	3

Os casos de teste da Tabela 29, gerados manualmente, foram convertidos para o padrão FitNesse, executados pelo servidor FitNesse e o resultado consta na Tabela 31.

Tabela 31 - Execução pelo FitNesse dos CT's gerados manualmente

salario	aumento?
1498	3
1499	3
1600	10
1999	10
2000	5

2001	5
1500	3
1586	3

Todos os casos de teste gerados, tanto os gerados automaticamente pela técnica C4T quanto os gerados manualmente, foram executados sem erro, ou seja, o resultado da execução do programa correspondeu ao resultado esperado.

Outra forma de avaliar a eficiência dos casos de testes gerados é a utilização da técnica de análise de mutantes, que nada mais é do que inserir propositalmente erros no programa a ser testado e avaliar se os casos de teste conseguem identificar esses erros (SIMÃO, 2004).

Existem diversas formas de inserir mutantes no programa. Nesse estudo foram inseridos os mutantes de atribuição e de condição. Para o mutante de atribuição foi inserido um erro na linha 16 do código-fonte do programa Java conforme Tabela 32. O erro inserido consiste em alterar o valor atribuído à variável `ALIQUOTA` de 27 (programa original) para 29 (programa mutante).

Tabela 32 - Inclusão de mutante de atribuição

Código-fonte original	Código-fonte com mutante
alíquota = 27	alíquota = 29

A Tabela 33 mostra a execução pelo FitNesse dos casos de teste gerados automaticamente pela técnica C4T utilizando a versão do programa Java com o mutante especificado na Tabela 32. Nota-se que o último caso de teste (salario = 1500) apresentou erro pois o resultado esperado era 3 e o resultado obtido pela execução do programa foi 10.

Tabela 33 - Execução dos CT's gerados pelo C4T com mutante de atribuição

salario	aumento?
1	3
1600	10
2000	5
1500	3 (esperado) 10 (obtido)

Com base na Tabela 33 pode-se concluir que os casos de teste gerados pela técnica C4T são eficientes ao validar casos de mutantes nos quais foram inseridos erros de atribuição.

Para o mutante de condição foi inserido um erro na linha 23 do código-fonte do programa Java conforme Tabela 34. O erro inserido consiste em alterar a condição de decisão de “`SALARIO < 2000`” (programa original) para “`SALARIO <= 2000`” (programa mutante).

Tabela 34 - Inclusão de mutante de condição

Código-fonte original	Código-fonte com mutante
`IF (salario < 2000)`	`IF (salario <= 2000)`

A Tabela 35 mostra a execução pelo FitNesse dos casos de teste gerados automaticamente pela técnica C4T utilizando a versão do programa Java com o mutante especificado na Tabela 34. Nota-se que o terceiro caso de teste (salario = 2000) apresentou erro pois o resultado esperado era 5 e o resultado obtido pela execução do programa foi 10.

Tabela 35 - Execução dos CT’s gerados pelo C4T com mutante de condição

salario	aumento?
1	3
1600	10
2000	5 (esperado)
	10 (obtido)
1500	3

Com base na Tabela 35 pode-se concluir que os casos de teste gerados pela técnica C4T também são eficientes ao validar casos de mutantes nos quais foram inseridos erros de condição.

Uma característica observada na técnica C4T é que o código-fonte do programa COBOL que está sendo testado aumentou consideravelmente após a instrumentação do código-fonte. Nota-se pela Tabela 39 que em muitos casos uma única linha do programa COBOL original gerou até 14 linhas a mais no programa instrumentado. O tamanho original do programa era de 32 linhas, e após a instrumentação o tamanho do programa passou a ser de 129 linhas (aumento de mais de 300%) mas essa característica não mostrou ser preocupante pois a geração automática dos casos de teste pela técnica C4T (utilizando o programa instrumentado) demorou cerca de 2 segundos.

Para a geração dos casos de teste do programa de exemplo a técnica C4T iniciou cada fluxo atribuindo o valor 0 para a variável `SALARIO`. Esse valor foi definido a partir do conhecimento do engenheiro de testes de que esse é um valor que não é válido como entrada para o programa, e por isso ele iria ser desprezado pela técnica C4T, e a técnica buscaria valores permitidos para cada fluxo.

Para avaliar se esse valor inicial de entrada poderia influenciar os casos de teste gerados foi feita uma alteração na técnica C4T para iniciar cada fluxo com o valor de 10.000 (dez mil) para a variável `SALARIO`. Os casos de teste gerados com essa alteração são mostrados na Tabela 36. As mudanças da Tabela 36 (com valor inicial de 10.000) em relação à Tabela 28 (em que o valor inicial é 0) é o valor de salário do fluxo 05, que era 2.000 na Tabela 28 e passou a ser 10.000 na Tabela 36, e consequentemente o IR a pagar passou de 234 para 2.394. Isso mostra que o valor inicial assumido pela técnica C4T pode influenciar os casos de teste gerados pela técnica. Apesar disso, a qualidade dos casos de teste não sofreu alteração pois a execução desses casos de teste com o programa original convertido para Java ou com os mutantes mostraram resultados esperados equivalentes.

Tabela 36 - Resultados de entrada e saída alterando o valor inicial

Fluxo	Caminhos	Salário	Aliquota	Parcela a deduzir	IR a pagar	Aumento	Observação
01	01, 02, 04, 05 e 08	1	0	0	0	0	Caminho não possível
02	01, 02, 04, 06 e 08	1	0	0	0	0	Caminho não possível
03	01, 02, 07 e 08	1	0	0	0	3	
04	01, 03, 04,05 e 08	1600	27	306	126	10	
05	01, 03, 04, 06 e 08	10000	27	306	2394	5	
06	01, 03, 07 e 08	1500	27	306	99	3	

Outra análise feita foi verificar se o solver conseguiria resolver uma restrição do tipo “`IF A > B`” sendo que A e B são dados de entrada. As restrições passadas para o *solver* e o resultado retornado pelo *solver* para essa situação são mostrados na Tabela 37. Observa-se que para essa situação o *solver* retornou `A = 1 e B = 0`, que é uma solução possível para as restrições dadas. Isso mostra que o *solver* consegue resolver restrições do tipo `A > B` sendo A e B dados de entrada.

Tabela 37 - Restrições para o solver para A > B

```
(define a::real)
(define b::real)
(assert+ (> a b))
(check)
Resultado do Solver:
a = 1
b = 0
```

Com relação à documentação, o teste direcionado poderia ser utilizado para gerar as diversas condições de um programa. Para isso a técnica precisaria ser adaptada para gerar como resultado, além dos dados de entrada e o resultado esperado, também a condição que está sendo atendida pelo requisito. Um exemplo de como seria esse resultado é mostrado na Tabela 38. Essa tabela é semelhante à Tabela 28, com a diferença de que são incluídas as colunas "Condição 1", "Condição 2" e "Condição 3" que poderiam servir para documentação do programa.

Tabela 38 – Saída esperada após inclusão de dados de documentação

Fluxo	Condição 1	Condição 2	Condição 3	Salário	Aliquota	Parcela a deduzir	IR a pagar	Aumento
01	Salario <= 1499	IR_a_pagar >= 125	Salario < 2000	1	0	0	0	0
02	Salario <= 1499	IR_a_pagar >= 125	Salario >= 2000	1	0	0	0	0
03	Salario <= 1499	IR_a_pagar < 125		1	0	0	0	3
04	Salario > 1499	IR_a_pagar >= 125	Salario < 2000	1600	27	306	126	10
05	Salario > 1499	IR_a_pagar >= 125	Salario >= 2000	2000	27	306	234	5
06	Salario > 1499	IR_a_pagar < 125		1500	27	306	99	3

O esforço para geração dos casos de teste utilizando a técnica C4T foi de cerca de 20 segundos, enquanto que o esforço para geração dos casos de testes manualmente foi de cerca de 10 minutos, mostrando uma considerável economia de tempo com a utilização da técnica C4T. Nessa análise foi considerado que o programa já estava instrumentado, uma vez que a parte de instrumentação automática do programa não foi desenvolvida completamente neste estudo.

Com relação à cobertura dos testes gerados, a técnica C4T e a geração manual dos casos de teste mostraram o mesmo resultado, pois cobriram todos os trechos do programa (nível de cobertura de 100%).

5 CONCLUSÕES

Os casos de teste gerados pela técnica C4T para o programa exemplo mostraram-se com eficácia equivalente (pois conseguiu descobrir as falhas incluídas propositalmente no programas com os mutantes) e melhor eficiência (pois fez isso com um menor número de casos de teste) comparados aos casos de teste gerados manualmente a partir da aplicação da técnica em um programa COBOL que utiliza uma lógica simples que possui comandos de desvio (`GO TO`) e comandos de decisão (`IF`).

Apesar do programa de exemplo ser simples, pode-se supor que a técnica C4T também atenda satisfatoriamente programas mais complexos, mas para isso algumas adaptações na técnica tornam-se necessárias, principalmente para atender outras instruções COBOL (por exemplo: comando `PERFORM`, que tem comportamento similar às instruções "`FOR / WHILE`" de outras linguagens de programação) não encontrados no programa de exemplo utilizado nesse estudo.

A aplicação da análise de mutantes nos casos de teste gerados pela técnica C4T mostrou resultados satisfatórios para o programa de exemplo uma vez que os casos de teste gerados conseguiram identificar os erros inseridos propositalmente no programa reescrito em linguagem Java.

A técnica C4T mostrou ter um de tempo de geração de casos de teste menor do que a geração manual para o programa de exemplo. Para programas mais complexos a tendência é a economia de tempo na geração dos casos de teste ser ainda maior, uma vez que o engenheiro de teste que fará a geração manual dos casos de teste provavelmente demorará mais tempo para entender o programa e gerar os casos de teste.

Cabe ressaltar que a técnica deve ser completamente automatizada com o desenvolvimento do gerador dos caminhos e do instrumentador, sendo que os esforços para isso não estão sendo considerados nessa análise, o que faz com que a comparação se dê no campo hipotético e que corre-se o risco desta etapa não ser exitosa.

Para grandes projetos o custo para desenvolvimento desses itens faltantes na técnica C4T passa a ser marginal, fazendo com que o esforço para aplicação da técnica C4T tenda a valer a pena em comparação com a geração manual para grandes projetos.

A cobertura de casos de teste foi igual tanto usando a técnica C4T quanto fazendo a geração manual dos casos de teste, em ambos os casos mostrando cobertura de 100% dos trechos de código do programa de exemplo.

A quantidade de casos de teste gerados automaticamente pela técnica C4T foi menor que a quantidade de casos de teste gerados manualmente, o que mostra que a técnica é mais direcionada, não usando várias entradas para atender uma mesma condição do programa, Mas isso não significa que o fato da quantidade de casos de teste gerados manualmente ser maior signifique que o teste seja pior. Em muitas situações é justamente o contrário, ou seja, quando mais casos de teste, melhor para testar um sistema. Para equalizar a quantidade de casos de teste, a técnica C4T pode ser adaptada para gerar duas restrições para cada condição do programa aplicando o mesmo critério de valor-limite utilizado neste trabalho para a geração manual de casos de teste.

A técnica C4T mostrou ser sensível ao valor inicial das variáveis de entrada, portanto o engenheiro de testes precisa atentar-se a esse fato no momento de definir o valor inicial das variáveis de entrada no início de cada fluxo utilizado para cada caso de teste, mostrando ser um grande problema da técnica de teste direcionado.

A simulação a utilização de restrições do tipo A > B sendo A e B dados de entrada mostrou que o *solver* yices consegue resolver esse tipo de situação. Mas para atender esse tipo de situação a técnica C4T precisaria ser adaptada para fazer a instrumentação de forma a comparar ambas as variáveis no retorno do *solver* pois atualmente a instrumentação da técnica somente faz a comparação com uma variável de retorno do *solver*.

A técnica também pode ser utilizada para documentação de programas incluindo-se nos casos de teste as condições (requisitos) atendidos pelos dados de entrada.

Com base nas análises acima pode-se concluir que a abordagem utilizada pela técnica C4T, para o programa de exemplo, atende aos objetivos propostos para o presente trabalho.

A criação de casos de teste passíveis de automação não é uma atividade considerada trivial, mas sua aplicação pode ser responsável pela redução de uma parte considerável do tempo despendido com a atividade de teste.

Cada vez mais as tecnologias estão sendo aprimoradas para facilitar e aumentar a produtividade do trabalho de desenvolvimento de software, e nesse sentido a geração automática de casos de teste pode ser uma boa alternativa. Assim como estão em constante evolução as atividades de geração automática de código-fonte e geração automática de documentação, a geração automática de casos de teste é uma atividade que também será aprimorada para facilitar o desenvolvimento de software.

Por outro lado, em função de basear-se no código-fonte, a geração automática de casos de teste não atende os casos em que houve erro de codificação, ou seja, o programa não foi escrito de acordo com o que estava previsto no requisito. Por esse motivo, por mais que se desenvolva essa técnica deve-se ter em mente que ela não substitui totalmente o trabalho de um engenheiro de testes.

Como a técnica C4T é baseada no código-fonte do programa como referência para geração dos casos de teste, a aplicação dessa técnica pressupõe que o código-fonte esteja correto. Portanto, sua aplicação está limitada aos casos em que o programa não esteja em desenvolvimento e há risco de gerar casos de teste errados caso o código-fonte original esteja com algum falha.

5.1 Sugestões para trabalhos futuros

A seguir são mostradas algumas sugestões de melhorias, extensões e possíveis novas pesquisas relacionadas a este trabalho:

Poucos programas deixam de fazer uso de acesso a banco de dados em sua lógica interna. A técnica C4T poderia ser adaptada para ser utilizada em programas COBOL que acessem algum banco de dados (por exemplo: DB2). A abordagem de

usar um solucionador de restrições em um programa com acesso a banco de dados foi proposta por Emmi, Majumdar e Sen (2007) mas utilizava um programa Java e banco de dados SQL.

A técnica C4T ora apresentada faz a geração semiautomática dos casos de teste uma vez que a escolha dos caminhos depende de um engenheiro de testes. Uma possível melhoria na técnica C4T seria torná-la totalmente automática, e para isso é preciso implementar na técnica uma forma para gerar automaticamente todos os caminhos do programa e gerar os casos de entrada que atendam aos diversos caminhos do programa.

Outro ponto que poderia ser objeto de trabalhos futuros seria adaptar a técnica C4T para atender outras linguagens de programação que não sejam COBOL, tais como linguagens que utilizem paradigma orientado a objetos.

Além disso, a técnica C4T pode ser estendida para gerar os casos de teste para outros frameworks de execução de testes, como , por exemplo, o padrão XUnit.

REFERÊNCIAS

ABREU, B. T. **Uma Abordagem Evolutiva para a Geração Automática de Dados de Teste**. Campinas, 2006, 117 f. Dissertação (Mestrado) – Instituto de Computação, Universidade Estadual de Campinas, Campinas, 2006

ALI, S.; BRIAND, L. C.; HEMMATI, H.; PANESAR-WALAWEGE, R. K. **A systematic review of the application and empirical investigation of search-based test-case generation**. IEEE Transactions on Software Engineering. p. 1-22, 2010

AL-AZZONI, I.; ZHANG, L.; e DOWN, D. G. 2011. **Performance evaluation for software migration**. In *Proceedings of the second joint WOSP/SIPEW international conference on Performance engineering* (ICPE '11). ACM, New York, NY, USA, 323-328.

ANDREA, J. **Envisioning the Next Generation of Functional Testing Tools**. IEEE Software, Los Alamitos, v.24, n.3, p.58-66, 2007.

BARCELOGIC. **Barcelogic**. Disponível em http://www.lsi.upc.es/ oliveras/bclt-main.html, Acessado em janeiro de 2012

BECK, K. **Test-Driven Development: By Example**. Boston: Addison Wesley, 2002. 240p.

BEIZER, B. **Black-Box Testing**. John Wiley & Sons, 1995.

BIN, L. et al. **Automatic Test Data Generation Tool Based on Genetic Simulated Annealing Algorithm**. International Conference on Computational Intelligence and Security Workshops, 2007. CISW 2007. p. 183-186

BINDER, R. V. **Testing Object-Oriented Systems: Models, Patterns, and Tools**. Addison-Wesley, 2000. 1150p.

BURNSTEIN, I. **Practical Software Testing: A Process-oriented Approach**. Springer, 1st edition, 2003.

CARVALHO, A. B. de, et al. **Relatório Técnico: Busca Tabu Aplicada ao Problema do Caixeiro Viajante**. Disponível em http://www.inf.ufpr.br/aurora/disciplinas/ topicosia2/ downloads/trabalhos/BuscaTabuTSP.pdf. Acessado em novembro de 2011

COLORNI, A.; DORIGO, M.; MANIEZZO, V. **Distributed optimization by ant colonies**. In: Proceedings European Conference on Artificial Life. Paris, França: Elsevier, Amsterdam, 1992. p. 134-142. Disponível em: http://iridia.ulb.ac.be/ ~mdorigo/ACO% -/publications.html

CRISPIN, L.; GREGORY, J. **Agile Testing: A Practical Guide for Testers and Agile Teams**. Boston: Addison Wesley, 2009. 536p.

CUNNINGHAM, W. **Introduction To Fit**. Disponível em http://fit.c2.com/wiki.cgi?IntroductionToFit. Acesso em janeiro de 2012.

CVC. **CVC Lite**. Disponível em http://www.cs.nyu.edu/acsys/cvcl. Acessado em janeiro de 2012.

DALAL, S. R. et al. **Model-Based Testing in Practice**. In: 21st International Conference on Software Engineering, 1999, Los Angeles, USA. New York: ACM, 1999. p.285-294.

DALLMEIER, V. et al. **Automatically Generating Test Cases for Specification Mining**. 2012. *IEEE Trans. Softw. Eng.* 38, 2 (March 2012), 243-257

DEMILLO, R. A.; LIPTON, R. J. e SAYWARD, F. G. 1978. **Hints on Test Data Selection: Help for the Practicing Programmer**. *Computer* 11, 4 (April 1978), 34-41

DIAZ, E., TUYA, J.; BLANCO, R. **Automated software testing using a metaheuristic technique based on tabu search**. In ASE, pages 310–313, 2003.

DORIGO, M. **Optimization, Learning and Natural Algorithms**, PhD thesis, Politecnico di Milano, Italy, 1992

EDVARDSSON, J. **A survey on automatic test data generation**. In Proceedings of the Second Conference on Computer Science and Engineering in Linköping (October 1999), p. 21-28

EIBEN, A. E.; SMITH, J. E. **Introduction to Evolutionary Computing**. Springer, 2003.

EMMI, M.; MAJUMDAR, R.; SEN. K. **Dynamic test input generation for database applications**, Proceedings of the 2007 international symposium on Software testing and analysis, London, United Kingdom, 2007

FITNESSE. **TwoMinuteExample**. Disponível em http://fitnesse.org/FitNesse.UserGuide.TwoMinuteExample. Acessado em janeiro de 2012

FRÖHLICH, P.; LINK, J. **Automated Test Case Generation from Dynamic Models**. 2000. In *Proceedings of the 14th European Conference on Object-Oriented Programming* (ECOOP '00), Elisa Bertino (Ed.). Springer-Verlag, London, UK, 472-492

GONÇALVES, A. R. **Inteligência de Enxames**. Disponível em http://www.dca.fee.unicamp.br/~andreric/arquivos/pdfs/enxames.pdf. Acessado em 01/03/2011

GOOGLE CODE. Disponível em http://code.google.com/p/infrastructure-ipam-hkust/source/browse/trunk/Prevision/src/hk/ust/cse/Prevision/Solver/Yices/YicesLoader.java?r=98. Acessado em 10 jan 2012.

HANSSEN , G. K.; HAUGSET, B. **Automated Acceptance Testing Using Fit**, 42nd Hawaii International Conference on System Sciences, pp. 1-8, 2009.

HARTMANN, T. Model **Based Testing of End-to-End Chains using Domain Specific Languages**. In: Testing: Academic and Industrial Conference - Practice and Research Techniques, 2009, Windsor, UK. New York: IEEE, 2009. p.82-91.

HOLMES, A.; KELLOGG, M. **Automating Functional Tests Using Selenium**. In: AGILE Conference, 2006, Minneapolis, USA. New York: IEEE, 2006. 6p.

KENNEDY, J.; EBERHART, R. **Particle swarm optimization**. In: Proceedings IEEE International Conference on Neural Networks. [s.n.], 1995. v. 4, p. 1942-1948.

KIEZUN, A. **Effective Software Testing with a String-Constraint Solver**. Tese (Doutorado). Massachusetts Institute of Technology, 2009

KOUTSIKAS, C. C.; MALEVRIS, N.; SKORDALAKIS, E. **SYMEXLAN – A Symbolic Execution Script Language**, Proceedings of the fourth International Conference on Software Engineering (SEA 2000), Las Vegas, November 2000, pp. 29-34.

LANO, K. **Specifying static analysis tools using formal methods**. 1995. In *Proceedings of the 1st International Conference on Engineering of Complex Computer Systems* (ICECCS '95). IEEE Computer Society, Washington, DC, USA, 254

LAZINICA, A. **Particle Swarm Otimization**. 2009. Disponível em: http://www.intechopen.com/books/show/title/particle_swarm_optimization. Acessado em 15/05/2011.

LEWIS, W. E. 2008. **Software Testing and Continuous Quality Improvement.** Third Edition*(2nd ed.). Auerbach Publications, Boston, MA, USA*

LI, A.; ZHANG, Y. **Generation Method of Test Data for Software Structure Based on PSO**. Computer Engineering, 2008, 34(6): 93-94.

LI, A.; ZHANG, Y. **Automatic Generating All-Path Test Data of a Program Based on PSO**. World Congress on Software Engineering, 2009.

LI, K.; ZHANG, Z. ; LIU, W. **Automatic Test Data Generation Based on Ant Colony Optimization**. Fifth International Conference on Natural Computation, 2009. ICNC '09. p. 216-220

LIU, S.; e NAKAJIMA, S. **A "Vibration" Method for Automatically Generating Test Cases Based on Formal Specificacions**. 18th Asia Pacific Software Engineering Conference (APSEC), pp. 73-80. 2011

LOPES, H. S. **Fundamentos da Computação Evolucionária e Aplicações**. Bandeirantes, Paraná, 2006. 52-106 p.

MANSOUR, N.; SALAME, M. **Data Generation for Path Testing**. Software Quality Journal, 12(2):121–136, 2004.

MARTIN, R. C. et al. **FitNesse Acceptance Tests Framework**. Disponível em: <http://www.fitnesse.org>. Acesso em: 22 fev. 2011.

MARTIN, R., MELNIK, G., **Tests and Requirements, Requirements and Tests: A Mobius Strip**, IEEE Software, 25(1), pp. 54-59. 2008.

MCMINN, P. **Search-based software teste data generation: A survey**. Software Testing, Verification and Reliability, v. 14, n. 2, p. 105-156, p. 105-156, jun. 2004.

MEDEIROS, J. A. C. C. **Enxame de Partículas como ferramenta de Otimização em Problemas Complexos da Engenharia Nuclear**. Tese (Doutorado). Universidade Federal do Rio de Janeiro, 2005

MEUDEC, C. **ATGen: automatic test data generation using constraint logic programming and symbolic execution**. Software Testing, Verification & Reliability, 11(2):81–96, 2001.

MICHAEL, C. C., MCGRAW, G. e SCHATZ, M. **Generating Software Test Data by Evolution**. *IEEE Trans. on Software Engineering*, 27(12):1085–1110, 2001.

MICROFOCUS. **Visual Cobol R4**. Disponível em http://www.microfocus.com/plus/visualcobol/index.aspx. Acesso em: 05 nov. 2011.

MINGSONG, C.; XIAOKANG, Q.; XUANDONG, L. **Automatic Test Case Generation for UML Activity Diagrams** In: International Workshop on Automation of Software Test, 2006, Shanghai, China. New York: ACM, 2006. p.2-8.

MUGRIDGE, R.; CUNNINGHAM, W. **Fit for Developing Software: Framework for Integrated Tests**. Upper Saddle River: Prentice Hall PTR, 2005. 384p

MUGRIDGE, R.; TEMPERO, E. **Retrofitting an Acceptance Test Framework for Clarity**. In: Agile Development Conference, 2003, Salt Lake City, USA. Proceedings... New York: IEEE, 2003. p.92-99.

MYERS, G. J.; SANDLER, C. **The Art of Software Testing**. 2. ed. Hoboken:John Wiley & Sons, Inc., 2004.

NUNES, D. J.; MOREIRA, A.; e RIBEIRO, L. 2011. 2011. **Formal Methods at SBES: Evolution and Perspectives**. In *Proceedings of the 2011 25th Brazilian Symposium on Software Engineering* (SBES '11). IEEE Computer Society, Washington, DC, USA, 1-3

OMG. ***Object Constraint Language***. Needham: Object Management Group. 2006. Disponível em http://www.omg.org/spec/OCL/2.0/. 232p.

OMG. ***Unified Modeling Language***. Needham: Object Management Group, 2010. Disponível em http://www.omg.org/spec/UML/2.3/>. 984p.

PAPADAKIS, M; MALEVRIS, N. **Automatic mutation based test data generation**. Proceedings of the 13th annual conference companion on Genetic and evolutionary computation. New York, NY, USA. 2011

PARK, S. S.; MAURER, F. **The Benefits and Challenges of Executable Acceptance Testing.** In: 2008 international workshop on Scrutinizing agile practices or shoot-out at the agile corral, 2008, Leipzig, Germany. New York: ACM, 2008. p. 19-22.

PEZZÈ, M.; YOUNG, M. **Teste e Análise de Software: processos, princípios e técnicas**. Porto Alegre: Bookman, 2008, 512p

PIMENTA, A. **Especificação formal de uma ferramenta de reutilização de especificações de requisitos**. Dissertação (Mestrado) em Ciências da Computação – Instituto de Informática, Universidade Federal do Rio Grande do Sul, Rio Grande do Sul, 1998. 121 f.

PINHEIRO, A. C. **Geração Automática de Dados de Teste – Visão Geral**. Disponível em moodle.stoa.usp.br/mod/resource/view.php?id=13681. Acessado em maio de 2012.

PRESSMAN, R. S. **Engenharia de Software**. 6a. ed. McGraw-Hill. 2006.

PU, J. et al. **Using Aspect Orientation in Understanding Legacy COBOL Code**, compsac, vol. 2, pp.385-390, 2007 31st Annual International Computer Software and Applications Conference, 2007

SCHAEFER, I.; e HÄHNLE, R. 2011. **Formal Methods in Software Product Line Engineering**. *Computer* 44, 2 (February 2011), 82-85

SEN, K. **Concolic Testing**. In: ASE’07 - Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, New York, USA. ACM: 2007

SIMÃO, A. S. **Aplicação da análise de mutantes no contexto do teste e validação de redes de Petri coloridas**. 2004. Tese (Doutorado em Ciências de Computação e Matemática Computacional) - Instituto de Ciências Matemáticas e de Computação, University of São Paulo, São Carlos, 2004. Disponível em: <http://www.teses.usp.br/teses/disponiveis/55/55134/tde-13042005-152434/>. Acesso em: 2012-06-10.

SNEED, H. M. **Migrating from COBOL to Java: A Report from the Field**. IEEE International Conference on Software Maintenance (ICSM), 2010

SOK, S. **Java and Web Extensions of the Yices Little Engine of Proof**. Disponível em http://atlantis.seidenberg.pace.edu/wiki/lep/Home?action=AttachFile&do=get &target=sokreport.pdf. Acessado em janeiro de 2012.

SOMMERVILLE, I.. **Engenharia de Software**. Prentice-Hall, 6th edition, 2003.

STHAMER, H. **The Automatic Generation of Software Test Data Using Genetic Agorithms**. PhD thesis, University of Glamorgan, Pontyprid, Wales, Great ritain, 1996.

TAKAKI, M. **Busca meta-heurística para resolução de CSP em teste de software**. Dissertação (Mestrado em Ciências da Computação) – Centro de Informática,Universidade Federal de Pernambuco, Recife, 2009. 118 f.

THOUGHTWORKS. **SeleniumHQ**. Disponível em: <http://seleniumhq.org>. Acesso em: 10 mar. 2011.

UTTING, M.; LEGEARD, B. **Practical Model-Based Testing: A Tools Approach**. San Francisco: Elsevier, 2006. 455p.

VEERMAN, N. **Revitalizing modifiability of legacy assets**. Software Maintenance and Evolution: Research and Practice, Special issue on CSMR 2003 2004; 16(4–5):219–254.

VEERMAN, N.; VERHOEVEN, E. J. 2006. **Cobol minefield detection**. *Softw. Pract. Exper.*36, 14 (November 2006), 1605-1642.

WATKINS, A. e HUFNAGEL, E. M. **Evolutionary test data generation: a comparison of fitness functions**. Software Practice and Experience, 36(1):95–116, 2006.

WHITLEY, D. **An Overview of Evolutionary Algorithms: Practical Issues and Common Pitfalls**. Information and Software Technology, Volume: 43, Issue: 14, Publisher: Elsevier, Pages: 817-831, 2001.

XUAN, J. et al. **A Random Walk Based Algorithm for Structural Test Case Generation**. 2nd International Conference on Software Engineering and Data Mining (SEDM), pp. 583-588, 2010.

YICES. **Yices**. Disponível em http://fm.csl.sri.com/yices. Acessado em janeiro de 2012.

ANEXO 1 – Código-Fonte dos Programas que fazem as Chamadas ao S*olver*

A seguir é mostrado o código-fonte de um programa escrito em Java que faz a chamada ao *solver* Yices.


```
The MIT License (MIT)

Copyright (c) <year> <copyright holders>

Permission is hereby granted, free of charge, to any person obtaining a
copy of this software and associated documentation files (the "Software"),
to deal in the Software without restriction, including without limitation
the rights to use, copy, modify, merge, publish, distribute, sublicense,
and/or sell copies of the Software, and to permit persons to whom the
Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN THE SOFTWARE.
```


```java
package hk.ust.cse.YicesWrapper;

public class YicesLoader {
  public enum SOLVER_COMP_PROCESS {SAT, UNSAT, ERROR, TIMEOUT}

  public SOLVER_COMP_PROCESS check(String input) {
    // call YicesWrapper directly
    boolean result = YicesWrapper.check(input);
    String output  = YicesWrapper.getLastOutput();
    String errMsg  = YicesWrapper.getLastErrorMsg();

    // create a solver result
    if (m_lastResult == null) {
      m_lastResult = new YicesResult();
    }
```

```
    if (output.length() > 0) {       // SMT Check finished
      // parse and save result
      m_lastResult.parseOutput(output);

      // return satisfactory or not
      return (result) ? SOLVER_COMP_PROCESS.SAT :
SOLVER_COMP_PROCESS.UNSAT;
    }
    else if (errMsg.length() > 0) {       // SMT Check throws error
      System.out.println("SMT Check error: " + errMsg);
      return SOLVER_COMP_PROCESS.ERROR;
    }
    else {       // SMT Check timeout
      System.out.println("SMT Check timeout!");
      return SOLVER_COMP_PROCESS.TIMEOUT;
    }
  }

  public String getLastOutput() {
    return YicesWrapper.getLastOutput();
  }

  public String getLastInput() {
    return YicesWrapper.getLastInput();
  }

  public YicesResult getLastResult() {
    return m_lastResult;
  }

  private YicesResult m_lastResult;
}

package hk.ust.cse.YicesWrapper;

import java.util.ArrayList;
import java.util.List;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class YicesResult {

  private static final Pattern s_pattern1 = Pattern.compile("^\\(=
([\\S]+) ([\\S]+)\\)$");

  public void parseOutput(String output) {
    // save output first
    m_output = output;

    // split outputs
    String[] outLines = output.split(LINE_SEPARATOR);

    // analyze
    m_satModel = new ArrayList<String[]>();
    if (outLines.length == 0) {
      m_bSatisfactory = false;
```

```
    }
    else if (!outLines[0].startsWith("sat")) {
      m_bSatisfactory = false;
    }
    else {
      m_bSatisfactory = true;

      // analyze each model line
      for (int i = 1; i < outLines.length; i++) {
        if (outLines[i].length() > 0) {
          String[] term = toSMTTerm(outLines[i]);
          if (term != null) {
            m_satModel.add(term);
          }
          else {
            System.err.println("Unable to analyze model line: " +
outLines[i]);
          }
        }
      }
    }
  }

  public boolean isSatisfactory() {
    return m_bSatisfactory;
  }

  public String getOutputStr() {
    return m_output;
  }

  public List<String[]> getSatModel() {
    return m_satModel;
  }

  private String[] toSMTTerm(String str) {
    String[] smtTerm = null;

    Matcher matcher = null;
    if ((matcher = s_pattern1.matcher(str)).find()) {
      smtTerm = new String[] {matcher.group(1), matcher.group(2)};
    }
    return smtTerm;
  }

  private String          m_output;
  private boolean         m_bSatisfactory;
  private List<String[]> m_satModel;
  private static final String LINE_SEPARATOR =
System.getProperty("line.separator");
}

package hk.ust.cse.YicesWrapper;

import java.io.BufferedReader;
import java.io.File;
```

```
import java.io.FileReader;
import java.io.IOException;
import java.net.URL;

public class YicesWrapper {
  // Yices Lite API Definitions
  public static native void   yicesl_set_verbosity(short l);
  public static native String yicesl_version();
  public static native void   yicesl_enable_type_checker(short
flag);
  public static native void   yicesl_enable_log_file(String
filename);
  public static native int    yicesl_mk_context();
  public static native void   yicesl_del_context(int ctx);
  public static native int    yicesl_read(int ctx, String cmd);
  public static native int    yicesl_inconsistent(int ctx);
  public static native String yicesl_get_last_error_message();
  public static native void   yicesl_set_output_file(String
filename);

  static {
    // use an absolute path
    //String currDir = System.getProperty("user.dir");

    // load libraries according to os type and JVM architecture
    String osName  = System.getProperty("os.name");
    String jvmArch = System.getProperty("os.arch");

    if (osName.startsWith("Windows")) {
      String yicesLibName     = "libyices.dll";
      String yicesWrapperName =
"hk_ust_cse_YicesWrapper_YicesWrapper.dll";
      if (jvmArch.contains("64")) {
        yicesLibName     = "libyices_64.dll";
        yicesWrapperName =
"hk_ust_cse_YicesWrapper_YicesWrapper_64.dll";
      }
      // do not use loadLibrary(), since it only accepts library
name
      // use load() with an absolute path to the library instead!
      URL yicesLib     =
YicesWrapper.class.getClassLoader().getResource(yicesLibName);
      URL yicesWrapper =
YicesWrapper.class.getClassLoader().getResource(yicesWrapperName);
      if (yicesLib != null && yicesWrapper != null) {
        System.load(yicesLib.getPath());
        System.load(yicesWrapper.getPath());
      }
      else {
        System.err.println("No Yices module available for platform:
" + osName + " (" + jvmArch + ")");
      }
    }
    else if (osName.startsWith("Mac")) {
      String yicesWrapperName =
"libhk_ust_cse_YicesWrapper_YicesWrapper.jnilib";
```

```
      if (jvmArch.contains("64")) {
        yicesWrapperName =
"libhk_ust_cse_YicesWrapper_YicesWrapper_64.jnilib";
      }
      URL yicesWrapper =
YicesWrapper.class.getClassLoader().getResource(yicesWrapperName);
      if (yicesWrapper != null) {
        System.load(yicesWrapper.getPath());
      }
      else {
        System.err.println("No Yices module available for platform:
" + osName + " (" + jvmArch + ")");
      }
    }
    else {
      // we are using a hk_ust_cse_YicesWrapper_YicesWrapper.so that
is
      // statically linked against libyices.a. That's because linux
might
      // not have the right libgmp.so v4.1.2. Thus, we use a static
libyices.a
      String yicesWrapperName =
"hk_ust_cse_YicesWrapper_YicesWrapper.so";
      if (jvmArch.contains("64")) {
        yicesWrapperName =
"hk_ust_cse_YicesWrapper_YicesWrapper_64.so";
      }
      URL yicesWrapper =
YicesWrapper.class.getClassLoader().getResource(yicesWrapperName);
      if (yicesWrapper != null) {
        System.load(yicesWrapper.getPath());
      }
      else {
        System.err.println("No Yices module available for platform:
" + osName + " (" + jvmArch + ")");
      }
    }

    // create temporary output file
    try {
      s_tempOutput = File.createTempFile("tmp", ".tmp");
    } catch (IOException e) {}
  }

  public static boolean check(String input) {
    // create a context
    int ctx = yicesl_mk_context();

    // redirect output to a file
    yicesl_set_output_file(s_tempOutput.getAbsolutePath());

    // input commands
    boolean error = false;
    String[] lines = input.split("\n");
    for (int i = 0; i < lines.length; i++) {
      if (yicesl_read(ctx, lines[i]) == 0) {
```

```
        error = true;
        break;
      }
    }

    // save input
    s_lastInput = input;

    boolean result;
    if (!error) {
      result      = yicesl_inconsistent(ctx) == 0;
      s_lastError = "";

      // read output from temporary output file
      int nRead = 0;
      char[] buff = new char[51200];
      try {
        BufferedReader reader = new BufferedReader(new
FileReader(s_tempOutput));
        nRead = reader.read(buff, 0, buff.length);
        reader.close();
      } catch (IOException e) { /* should not throw exception */ }
      s_lastOutput = String.valueOf(buff, 0, nRead);
      int posicao  = s_lastOutput.indexOf("salario");
      String string_a_partir_posicao =
s_lastOutput.substring(posicao,posicao+15);

      int posicao_parentese  = string_a_partir_posicao.indexOf(")");
      String valor_str =
string_a_partir_posicao.substring(8,posicao_parentese);
      s_tempValor = Integer.parseInt(valor_str);
      System.out.println("s_valor: " + s_tempValor);
      // type error message is passed through the normal output
      if (s_lastOutput.startsWith("type error: ")){
        result = false;
        s_lastError  = s_lastOutput; // move error message to error
output
        s_lastOutput = "";
      }
    }
    else {
      result       = false;
      s_lastOutput = "";
      s_lastError  = yicesl_get_last_error_message();
    }

    // clean context
    yicesl_del_context(ctx);

    return result;
  }
  public static int getValor(){
       return s_tempValor;
  }
  public static String getLastInput() {
    return s_lastInput;
```

```
  }

  public static String getLastOutput() {
    return s_lastOutput;
  }

  public static String getLastErrorMsg() {
    return s_lastError;
  }
  public  int main(String entrada_solver) {
            String[] linhas = entrada_solver.split("EOF");

            StringBuilder input = new StringBuilder();

            for (int i = 0; i < linhas.length; i++)
            {
                 input.append(linhas[i]+"\n");
            }

    input.append("(set-evidence! true)\n");
    input.append("(define a::int)\n");
    input.append("(define b::int)\n");
    input.append("(assert (= (+ a b) 2))\n");
    input.append("(assert (> b 0))\n");
    input.append("(check)\n");
     */
    System.out.println("Result: " +
YicesWrapper.check(input.toString()));

    System.out.println("-YicesWrapper.getValor: " +
YicesWrapper.getValor());
    return YicesWrapper.getValor();
    /*
    input = new StringBuilder();
    input.append("(reset)\n");
    input.append("(set-evidence! true)\n");
    input.append("(define a::int)\n");
    input.append("(define b::int)\n");
    input.append("(assert (= 2 1))\n");
    input.append("(assert (= 1 1))\n");
    input.append("(check)\n");
    System.out.println("Result: " +
YicesWrapper.check(input.toString()));
    System.out.println("Input: " + YicesWrapper.getLastInput());
    System.out.println("Output: " + YicesWrapper.getLastOutput());
    System.out.println("Error: " + YicesWrapper.getLastErrorMsg());
    System.out.println("--------------------------------");

    input = new StringBuilder();
    input.append("(reset)\n");
    input.append("(set-evidence! true)\n");
    input.append("(define-type reference (scalar null notnull))\n");
    input.append("(define-type
Lorg/apache/commons/math/MathRuntimeException reference)\n");
    input.append("(define-type [Ljava/lang/Object reference)\n");
    input.append("(define-type Ljava/lang/Throwable reference)\n");
```

```
    input.append("(define rootCause::Ljava/lang/Throwable)\n");
    input.append("(define arguments::[Ljava/lang/Object)\n");
    input.append("(define
this::Lorg/apache/commons/math/MathRuntimeException)\n");
    input.append("(assert (/= this null))\n");
    input.append("(assert (= arguments null))\n");
    input.append("(check)\n");
    System.out.println("Result: " +
YicesWrapper.check(input.toString()));
    System.out.println("Input: " + YicesWrapper.getLastInput());
    System.out.println("Output: " + YicesWrapper.getLastOutput());
    System.out.println("Error: " + YicesWrapper.getLastErrorMsg());
    */
  }

  private static String s_lastInput;
  private static String s_lastOutput;
  private static String s_lastError;
  private static File   s_tempOutput;
  private static int    s_tempValor;
}
```

Fonte: adaptado de GOOGLE CODE (2012)

APÊNDICE 1 – Instrumentações no Código-Fonte da Técnica C4T

A Tabela 39 mostra as instrumentações feitas no programa COBOL pela técnica C4T para cada linha do programa COBOL. A coluna “Linha do programa” refere-se ao programa original mostrado na Tabela 4.

Esta tabela pode servir como uma especificação para o conversor e instrumentador de código da técnica C4T.

Tabela 39 – Instrumentações do programa COBOL após os passos 3 e 4

Linha do programa	Instrução COBOL	Instrumentações
13	IF SALARIO <= 1499 THEN	IF CAMINHO_ALVO = 02 THEN SET STRING_PARA_SOLVER TO STRING_PARA_SOLVER & "(ASSERT+ (<= SALARIO 1499))\N" END-IF SET STRING_PARA_SOLVER TO STRING_PARA_SOLVER & "(ASSERT+ (<= SALARIO " & SALARIO & ")\N" SET STRING_PARA_SOLVER TO STRING_PARA_SOLVER & "(CHECK)\N" SET RESULTADO_SOLVER TO YICES::SOLVER(STRING_PARA_SOLVER) IF SALARIO <> RESULTADO_SOLVER THEN SET RESULTADO_SOLVER TO YICES::SOLVER(STRING_PARA_SOLVER, "N") IF SALARIO <> RESULTADO_SOLVER THEN MOVE "N" TO ATINGIU_CAMINHO STOP RUN END-IF END-IF
14	MOVE 0 TO ALIQUOTA_IR	SET STRING_PARA_SOLVER TO STRING_PARA_SOLVER & "(ASSERT+ (= ALIQUOTA_IR 0))\N"
15	MOVE 0 TO PARCELA_A_DEDUZIR	SET STRING_PARA_SOLVER TO STRING_PARA_SOLVER & "(ASSERT+ (= PARCELA_A_DEDUZIR 0))\N
17	IF SALARIO > 1499 THEN	IF CAMINHO_ALVO = 03 THEN SET STRING_PARA_SOLVER TO STRING_PARA_SOLVER & "(ASSERT+ (> SALARIO 1499))\N" END-IF

		SET STRING_PARA_SOLVER TO STRING_PARA_SOLVER & "(ASSERT+ (<= SALARIO " & SALARIO & ")\N" SET STRING_PARA_SOLVER TO STRING_PARA_SOLVER & "(CHECK)\N" SET RESULTADO_SOLVER TO YICES::SOLVER(STRING_PARA_SOLVER) IF SALARIO <> RESULTADO_SOLVER THEN SET RESULTADO_SOLVER TO YICES::SOLVER(STRING_PARA_SOLVER, "N") IF SALARIO <> RESULTADO_SOLVER THEN MOVE "N" TO ATINGIU_CAMINHO STOP RUN END-IF END-IF
18	MOVE 27 TO ALIQUOTA_IR	SET STRING_PARA_SOLVER TO STRING_PARA_SOLVER & (ASSERT+ (= ALIQUOTA_IR 27))\N"
19	MOVE 306 TO PARCELA_A_DEDUZIR	SET STRING_PARA_SOLVER TO STRING_PARA_SOLVER & "(ASSERT+ (= PARCELA_A_DEDUZIR 306))\N"
22/23	COMPUTE IR_A_PAGAR = (SALARIO * ALIQUOTA_IR / 100) - PARCELA_A_DEDUZIR	SET STRING_PARA_SOLVER TO STRING_PARA_SOLVER & "(ASSERT+ (= IR_A_PAGAR (+ (/ (* SALARIO ALIQUOTA_IR) 100) (* PARCELA_A_DEDUZIR -1))))"

24	IF IR_A_PAGAR >= 125 THEN	IF CAMINHO_ALVO = 04 THEN
		SET STRING_PARA_SOLVER TO STRING_PARA_SOLVER &
		"(ASSERT+ (>= IR_A_PAGAR 125))\N"
		END-IF
		SET STRING_PARA_SOLVER TO STRING_PARA_SOLVER &
		"(ASSERT+ (<= IR_A_PAGAR " & IR_A_PAGAR & ")\N"
		SET STRING_PARA_SOLVER TO STRING_PARA_SOLVER &
		"(CHECK)\N"
		SET RESULTADO_SOLVER TO YICES::SOLVER(STRING_PARA_SOLVER)
		IF IR_A_PAGAR <> RESULTADO_SOLVER THEN
		SET RESULTADO_SOLVER TO YICES::SOLVER(STRING_PARA_SOLVER, "N")
		IF IR_A_PAGAR <>

		RESULTADO_SOLVER THEN MOVE "N" TO ATINGIU_CAMINHO STOP RUN END-IF END-IF
25	IF SALARIO < 2000 THEN	IF CAMINHO_ALVO = 05 THEN SET STRING_PARA_SOLVER TO STRING_PARA_SOLVER & "(ASSERT+ (< SALARIO 2000))\N" END-IF SET STRING_PARA_SOLVER TO STRING_PARA_SOLVER & "(ASSERT+ (<= SALARIO " & SALARIO & ")\N" SET STRING_PARA_SOLVER TO STRING_PARA_SOLVER & "(CHECK)\N" SET RESULTADO_SOLVER TO YICES::SOLVER(STRING_PARA_SOLVER) IF SALARIO <> RESULTADO_SOLVER THEN SET RESULTADO_SOLVER TO YICES::SOLVER(STRING_PARA_SOLVER, "N") IF SALARIO <> RESULTADO_SOLVER THEN MOVE "N" TO ATINGIU_CAMINHO STOP RUN END-IF END-IF
26	MOVE 10 TO AUMENTO	SET STRING_PARA_SOLVER TO STRING_PARA_SOLVER & "(ASSERT+ (= AUMENTO 10))\N"
27	ELSE	IF CAMINHO_ALVO = 06 THEN SET STRING_PARA_SOLVER TO STRING_PARA_SOLVER & "(ASSERT+ (>= SALARIO 2000))\N" END-IF
28	MOVE 5 TO AUMENTO	SET STRING_PARA_SOLVER TO STRING_PARA_SOLVER & "(ASSERT+ (= AUMENTO 5))\N"
29	ELSE	IF CAMINHO_ALVO = 07 THEN SET STRING_PARA_SOLVER TO STRING_PARA_SOLVER & "(ASSERT+ (< IR_A_PAGAR 125))\N" END-IF
30	MOVE 3 TO AUMENTO	SET STRING_PARA_SOLVER TO STRING_PARA_SOLVER & "(ASSERT+ (= AUMENTO 3))\N"

APÊNDICE 2 – Código-Fonte do Programa Controlador

A seguir é mostrado o código fonte do programa controlador escrito em linguagem COBOL.

Este programa é usado para fazer a execução dinâmica do programa COBOL, fazendo as diversas chamadas ao programa que se deseja testar para cada um dos caminhos selecionados pelo engenheiro de testes.

```
program-id. ExecDinamica as "br.ipt.execucao.dinamica.ExecDinamica".
data division.
working-storage section.
 01 areaentrada.
   02 caminho pic 9(5).
   02 salario pic 9(5).
   02 aumento pic 9(5).
   01 atingiu_caminho pic x(1).
  01 salario_ant pic 9(9).
  01 vez  pic 9(9) value 0.
  01 vez_controla pic 9(9) value 0.
  01 acabou pic x(1) value "N".
  01 tb_caminho occurs 2 times pic 9(9).
  01 tb_saida  occurs 2 times pic 9(9).
  01 i_caminho pic 9(9) value 1.
  01 caminho_maximo pic 9(9) value 0.
procedure division.
  perform  controla thru fim-controla
  until acabou = "S".
 controla section.
  if acabou = "N" or vez_controla = 0 then
    add 1 to vez_controla
    move tb_caminho(i_caminho) to caminho
    move "N" to atingiu_caminho
    perform Calcula  thru fim-calcula
            until atingiu_caminho = "S"
    if i_caminho = caminho_maximo then
       move "S" to acabou
    else
       add 1 to i_caminho.
 fim-controla section.
 calcula section.
if atingiu_caminho = "N"  or vez = 0 then
  add 1 to vez
  move salario to salario_ant
  call "ImpostoRenda" using areaentrada, atingiu_caminho
  if atingiu_caminho = "S" then
          & " caminho: " & caminho
          & " salario: " & salario
          & " aumento: " & aumento
else
      move 0 to vez.
  .
fim-calcula section.
end program ExecDinamica.
```

APÊNDICE 3 – Código-Fonte do Programa Exemplo Convertido Para Java

A seguir é mostrado o código fonte do programa COBOL usado como exemplo convertido para a linguagem Java.

```
1  package br.ipt.imposto.renda;
2
3  public class CalculaImposto {
4
5          public static int Imposto(int salario) {
6                  int aliquota;
7                  int     parcela_a_deduzir;
8                  int     aumento;
9                  int     ir_a_pagar;
10
11                 if (salario     <=      1499) {
12                         aliquota                =       0;
13                         parcela_a_deduzir       =       0;
14                 }
15                 else {
16                         aliquota                =       27;
17                         parcela_a_deduzir       =       306;
18                 }
19
20                 ir_a_pagar      =       (salario        *       aliquota        /       100)    -
21                                         parcela_a_deduzir;
22
23                 if      (ir_a_pagar     >=      125)    {
24                         if      (salario        <       2000)   {
25                                 aumento =       10;
26                         }
27                         else {
28                                 aumento =       5;
29                         }
30                 }
31                 else {
32                         aumento =       3;
33                 }
34
35                 return  aumento;
36         }
37 }
38
```

APÊNDICE 4 – Código-Fonte do Programa F*ixture* do FitNesse

A seguir é mostrado o código fonte do programa *fixture* usado para ligação entre o servidor FitNesse e o programa Java que deseja-se testar.

```
package br.ipt.imposto.renda;
import fit.ColumnFixture;
public class FixtureImposto extends ColumnFixture {
 public int salario;
 public int aumento(){
     int vl_aumento = CalculaImposto.Imposto(salario);

  return vl_aumento;
 }
 }
```

APÊNDICE 5 – Pseudocódigo do Algoritmo do Instrumentador

A Tabela 40 mostra o pseudocódigo do algoritmo do processo “analisador de caminhos e instrumentador” mencionado na Figura 12.

Tabela 40 - Pseudocódigo do Algoritmo do Instrumentador

```
Para cada linha do programa
   Buscar palavras-chave (IF, ELSE, MOVE, COMPUTE)
   Se a linha contem as palavras-chave IF ou ELSE
      Somar 1 ao contador de caminhos
      Separar os operandos e o operador da instrução IF
      Incluir comando “IF CAMINHO_ALVO = ” concatenado com o contador de caminhos
      Concatenar os seguintes comandos para o solver:
      “Assert” acrescido do operador “=”, do primeiro operando da instrução IF e do conteúdo
      corrente da variável do operando
      “Assert” acrescido do operador, do primeiro e do segundo      operandos da instrução IF
      Chamar o solver passando os comandos do solver
      Incluir comando para verificar se o retorno do solver é igual ao conteúdo corrente da variável
      do primeiro operando
      Caso o conteúdo não seja igual, chamar novamente o solver
      Com os comandos, mas retirar o comando do solver citado no item 1
      Chamar o solver passando os comandos do solver
      Incluir comando para verificar se o retorno do solver é “unsat”
      Caso o retorno seja “unsat”, registrar que o caminho não tem solução possivel
      Caso o retorno seja diferente de “unsat”, incluir instrução para interromper o programa e devolver
      o valor retornado pelo solver     para o programa chamador
   Se a linha contem a palavra-chave MOVE
      Separar os valores de atribuição da instrução MOVE
      Concatenar os seguintes comandos para o solver:
      “Assert” acrescido do operador “=”, do primeiro e do segundo atributos da instrução MOVE
   Se a linha contem a palavra-chave COMPUTE
      Separar os operadores e operandos da instrução COMPUTE
      Concatenar os seguintes comandos para o solver:
      “Assert” acrescido do operador da instrução COMPUTE, do primeiro e do segundo atributos da
      instrução COMPUTE
      Se houver mais de um operador na instrução COMPUTE, fazer o procedimento para todos os
      operadores da instrução.
```

Fonte: Elaborado pelo autor