Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Coordenação de Pós-Graduação em Informática

Análise de Mutação Aplicada à Verificação Funcional de IP *Core*

Henrique do Nascimento Cunha

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Informática da Universidade Federal de Campina Grande - Campus I como parte dos requisitos necessários para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: Ciência da Computação
Linha de Pesquisa: Redes de Computadores e Sistemas Distribuídos

Joseana Macêdo Fechine
(Orientadora)

Campina Grande, Paraíba, Brasil
©Henrique do Nascimento Cunha, 29/08/2008

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA CENTRAL DA UFCG

C972a

2008 Cunha, Henrique do Nascimento.
Análise de mutação aplicada à verificação funcional de IP core / Henrique do Nascimento Cunha. - Campina Grande, 2008.
88 f.: il.

Dissertação (Mestrado em Ciências da Computação) - Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática.

Referências.
Orientadora: Dr[a]. Joseane Macêdo Fechine.

1. Verificação funcional. 2. Teste de Mutação. 3. IP Core. I. Titulo

CDU 004.415.5(043)

Resumo

Existe uma necessidade crescente de aumento da confiabilidade dos IP *cores* produzidos atualmente. Para tanto, faz-se necessário o uso de uma metodologia de verificação funcional rigorosa para este tipo de produto. Como a verificação consome em média 70% dos recursos de um projeto de um IP *core*, torna-se necessário o uso das técnicas de verificação funcional a fim de reduzir os custos dos projetos. Entretanto, essas técnicas ainda não conseguem detectar todos os possíveis problemas de um projeto. Surge, então, a necessidade de construção e/ou aperfeiçoamento das metodologias de verificação funcional. Uma metodologia de verificação funcional, denominada VeriSC tem por objetivo eliminar algumas lacunas existentes em outras metodologias. Porém, há alguns passos da metodologia que ainda necessitam de refinamento. Um deles consiste em determinar como medir a qualidade da cobertura. Existem algumas técnicas de teste de software que visam a obtenção de parâmetros de qualidade relacionados à cobertura de um conjunto de casos de teste. Dentre essas técnicas, destaca-se a análise de mutação, que possibilita a geração de uma métrica relativa à qualidade de um conjunto de casos de teste de um dado IP *core*, a partir da análise da execução de mutantes. Estes mutantes são gerados automaticamente com base nos operadores de mutação escolhidos cuidadosamente. Este trabalho tem como meta a aplicação de testes de mutação na verificação funcional de sistemas digitais, para a avaliação da contribuição da técnica na melhoria da qualidade da cobertura da verificação funcional. Várias melhorias puderam ser observadas durante os experimentos, dentre estas destacam-se a possibilidade de encontrar defeitos no modelo de referência. Pôde-se também, observar um módulo IDCT, onde se considera que foi realizada uma verificação funcional de qualidade, ainda pôde ser melhorada em 11% de acordo com o parâmetro de qualidade conhecido como "escore de mutação".

Abstract

There is a growing need to make IP *core*s more reliable. Hence, it is necessary to use a rigorous functional verification methodology to this kind of product. This process is responsible for 70% of a project's resources, so it becomes important to enhance the functional verification techniques in order to reduce the cost of a project. A functional verification methodology, named VeriSC, is targeted at eliminating some flaws in other methodologies. Though there is an aspect in this methodology that needs refinement. One of them consists in determining the quality of the coverage achieved. There are some software techniques that try to extract quality parameters from test case sets of a given system. Among them, there is the mutation analysis technique, which proposes the generation of a quality metric by the analysis of mutants generated from the original program. These mutants are automatically created by applying carefully chosen mutation operators to the source code. Therefore, these operators must be chosen very carefully. The objective of this work is the application of mutation analysis within a digital system functional verification methodology, to check the contribution of this tecnique in evaluating the quality of the functional verification coverage. Various contributions could be made while apllying mutation analysis over a functional verification methodology, among them were the possibility to find defects on the reference model. It was possible to find out that a IDCT module, that was submitted to a high quality verification process, this process can still be enhanced another 11% according to the test quality parameter called mutation score.

Agradecimentos

Agradeço a Deus pela vida e pela oportunidade de conhecer pessoas tão maravilhosas, que sempre me apoiaram nesta caminhada.

Agradeço aos meus pais, Ivonete e Nascimento, pelos ensinamentos de vida, coragem e honestidade e por me apoiarem sempre.

Agradeço à minha esposa Karina por ser companheira, prestativa e atenciosa até nas horas mais difíceis.

Agradeço aos meus irmãos Aida, Ronnie e Neto por serem verdadeiramente irmãos.

Agradeço aos amigos cafas Wil Wil, Mago, Zambo, Coquim, Silveira e Marginal por toda sorte de situações que tivemos que lidar juntos, perdermos juntos e ganharmos juntos.

Agradeço à professora Joseana, minha orientadora e amiga nesta Odisséia científica.

Agradeço ao professor Elmar por tantos anos de ensinamentos e amizade desde a época da graduação.

Agradeço aos demais membros da banca por aceitarem avaliar e contribuir com este trabalho.

E a tantos outros que encontrei, que contribuíram direta ou indiretamente para a realização deste trabalho. (Na verdade, a lista é tão grande que fico com medo de colocar aqui e esquecer alguém)

Conteúdo

1 Introdução **1**

1.1 Objetivos . . . 2

1.1.1 Objetivos Específicos . . . 2

1.2 Organização da Dissertação . . . 3

2 Fundamentação Teórica **4**

2.1 A Metodologia de Verificação Funcional VeriSC . . . 9

2.2 Análise de mutação (*Mutation Analysis*) . . . 13

2.2.1 Hipótese do Programador Competente . . . 13

2.2.2 Efeito de acoplamento (*Coupling effect*) . . . 14

2.2.3 Análise e Teste de Mutação . . . 14

2.2.4 Operadores de Mutação . . . 16

2.2.5 Aplicação da Análise de Mutação . . . 17

2.3 Trabalhos Relacionados . . . 18

2.4 Discussão . . . 19

3 Descrição da Análise de Mutação Aplicada à Metodologia VeriSC **20**

3.1 Verificação das pré-condições . . . 21

3.2 Aplicação de mutação sobre o modelo de referência . . . 21

3.3 Verificação do modelo de referência mutado em relação ao original . . . 25

3.4 Discussão . . . 25

4 Apresentação e Análise de Resultados **27**

4.1 Caso de estudo: DPCM - *Differential Pulse Code Modulator* . . . 27

4.1.1 Modelo de Referência . . . 28

4.1.2 Ambiente de Verificação 28
4.1.3 Preparação do Ambiente para Aplicação das Mutações 29
4.1.4 Mutações 30
4.2 Caso de estudo: IDCT - *Inverse Discrete Cosine Transform* 34
4.2.1 Modelo de Referência 34
4.2.2 Ambiente de Verificação 35
4.2.3 Mutações 35
4.3 Discussão 43

5 Considerações Finais e Sugestões para Trabalhos Futuros 45
5.1 Considerações Finais 45
5.2 Sugestões para Trabalhos Futuros 46

A Ambiente e Ferramentas 50
A.1 Ambiente de Desenvolvimento 50
A.2 Sistema Operacional 50
A.3 Linguagem de Programação 51
A.4 Ferramenta de Mutação 51
A.5 Ferramenta para geração semi-automática de *Testbenches* 51
A.6 Controle de versões 51

B Código Fonte do *Testbench* Original do módulo DPCM 53

C Código Fonte do *Testbench* alterado do módulo DPCM. 64

D Código Fonte do Modelo de Referência do módulo IDCT 71

Lista de Símbolos

AVC - *Advanced Video Coding*

CETENE - Centro de Tecnologias Estratégicas do Nordeste

CI - *Circuito Integrado*

DPCM - *Differential Pulse Code Modulation*

DUV - *Design Under Verification*

FIFO - *First-in First-out*

H.261 - Padrão de codificação de vídeo do VCEG

H.263 - Padrão de codificação de vídeo do VCEG desenvolvido para vídeo conferência

H.263+ - Padrão de codificação de vídeo do VCEG versão 2

H.264 - Padrão de codificação de vídeo também chamado de MPEG-4 *Part 10* ou AVC

HDL - *Hardware Description Language*

IP-core - *Intellectual Property of Hardware Project*

IDCT - *Inverse Discrete Cosine Transform*

JPEG - *Joint Photographic Experts Group*

JBIG - *Joint Bi-level Image Experts Group*

LINCS - Laboratório para Integração de Circuitos e Sistemas

MPEG - *Moving Picture Experts Group*

MPEG-l *Moving Picture Experts Group Layer 1* MPEG-2 *Moving Picture Experts Group Layer 2* MPEG-4 *Moving Picture Experts Group Layer 4* OSCI - *Open SystemC Initiative*

LINCS-CETENTE - Laboratório para Integração de Circuitos e Sistemas - Centro de Tecnologias Estratégicas do Nordeste

RTL - *Register Transfer Level*

SCV - *SystemC Verification Library*

SoC - *System on Chip*

TL - *Transaction Level*

TLDS - *Transaction Level Data Structure*

VCEG - *Video Coding Experts Group*

VHDL - *VHSIC Hardware Description Language*

Lista de Figuras

2.1 Visões de um projeto de hardware (IP *core*). 5

2.2 Fases de um projeto de hardware (IP *core*). 5

2.3 Exemplo hipotético de um DUV. 6

2.4 Estrutura geral de um *testbench* segundo a metodologia VeriSC. 10

2.5 Primeiro passo para construção de um *testbench*. 11

2.6 Segundo passo para a construção do *testbench*. 12

2.7 Terceiro passo para a construção do *testbench*. 13

3.1 Aplicação da análise de mutação na metodologia VeriSC. 20

3.2 Aplicação das mutações sobre o modelo de referência no passo 2 da metodologia VeriSC. 22

3.3 Aplicação das mutações sobre o modelo de referência no passo 3 da metodologia VeriSC. 23

3.4 Aplicação das mutações sobre o modelo de referência após a inserção do DUV. 24

4.1 Aplicação das mutações sobre o modelo de referência do DPCM. 29

4.2 Aplicação das mutações sobre o modelo de referência do módulo IDCT. . . 35

4.3 Gráfico da relação entre quantidade de mutantes e tempo de trabalho manual para classificação. 43

Lista de Tabelas

4.1 Resultados da análise de mutação sobre a unidade 1. 31
4.2 Resultados da análise de mutação sobre a unidade 2. 33
4.3 Resultados da análise de mutação sobre a IDCT na primeira rodada de análise de mutação. 40
4.4 Resultados da análise de mutação sobre a IDCT na segunda rodada de análise de mutação. 42

Lista de Códigos Fonte

4.1 Código fonte original da subrotina de subtração do modelo de referência do DPCM 30
4.2 Código fonte da subrotina de subtração do modelo de referência do DPCM com mutação gerada pela ferramenta Proteum, operador SRSR 30
4.3 Trecho do código fonte do modelo de referência do módulo DPCM que continha uma falta 31
4.4 Trecho do código fonte do modelo de referência do módulo DPCM onde a falta foi corrigida 32
4.5 Trecho de código que representa a adição dos parâmetros de cobertura que identificariam a falta encontrada com a análise de mutação. 33
4.6 Trecho do código fonte do modelo de referência do IDCT sem aplicação de mutação 37
4.7 Trecho do código fonte do modelo de referência do IDCT com mutante na linha 4 37
4.8 Código fonte do exemplo de deslocamento de 8 bits para a direita. 38
4.9 Código fonte do exemplo de deslocamento de 2408 bits para a direita. . . . 38
4.10 Código fonte na linguagem Assembly do exemplo de deslocamento de 8 bits para a direita. 38
4.11 Código fonte na linguagem Assembly do exemplo de deslocamento de 2408 bits para a direita. 39
4.12 Trecho do Código Fonte de um mutante vivo não-equivalente ao programa original. 41
4.13 Trecho do Código Fonte do programa original onde foi aplicada à mutação. 41
B.1 Código fonte do modelo gerador automático de estímulos do módulo DPCM 53

B.2 Código fonte do modelo de referência do módulo DPCM 55
B.3 Código fonte das subrotinas que executam as funcionalidades do DPCM . . 58
B.4 Código fonte do *Checker* do módulo DPCM 58
B.5 Código fonte das estruturas de dados das transações do módulo DPCM . . . 59
B.6 Código fonte do módulo que conecta todos os elementos do *testbench* . . . 62
C.1 Código fonte do modelo de referência do módulo DPCM modificado para a mutação. 64
C.2 Código fonte das subrotinas que executam as funcionalidades do DPCM modificado para a mutação. 67
C.3 Código fonte do *checker* do módulo DPCM 67
C.4 Código fonte do módulo que liga todos os elementos do *testbench*. 69
D.1 Código fonte da subrotina que implementa a IDCT 71

Capítulo 1

Introdução

O desenvolvimento de circuitos integrados (CI) é levado a efeito por meio de um processo que engloba várias fases. Cada fase consome recursos do projeto (mão-de-obra, tempo, dinheiro, etc.). O objetivo do projeto desses circuitos é a obtenção de um produto com qualidade aceitável pelo mercado, mas que respeite os recursos alocados para a sua execução.

Em geral, o mercado de circuitos integrados não aceita que artefatos de *hardware* apresentem defeitos que possam ser detectados pelos usuários finais. Por isso, torna-se necessário o uso de um procedimento que forneça um indicativo da qualidade do artefato que está sendo desenvolvido. Por este motivo, é utilizada uma metodologia de verificação funcional. A verificação é realizada por meio da simulação de IP *core* em um ambiente de verificação, denominado *testbench*.

Estima-se que a verificação funcional consome a maior parte dos recursos de um projeto de um CI. Cerca de 70% dos recursos do projeto são gastos nessa fase [3][25]. Por isso, é necessário realizá-la com o máximo de qualidade, pois é a mais importante, a mais difícil e a mais onerosa, em termos econômicos, de todo projeto.

Dentre as metodologias de verificação funcional de IP *core* existentes, a metodologia VeriSC [10] tem por objetivo suprir as necessidades de um projeto, eliminando algumas lacunas identificadas em outras metodologias. Esta metodologia será utilizada neste trabalho e será descrita mais detalhadamente no Capítulo 2.

Independentemente da metodologia a ser utilizada, torna-se necessário saber quando a verificação funcional deve parar, visto os recursos para cada projeto são limitados. Para tanto, será utilizada, neste trabalho, a cobertura funcional. Entende-se por cobertura fun-

cional, a maneira pela qual o engenheiro de verificação funcional se posiciona em relação ao término da verificação. É uma medida de progresso deste processo [3]. Portanto, uma das questões envolvidas no processo de verificação funcional é: como definir um “bom” conjunto de parâmetros de cobertura? O termo “bom” é bastante subjetivo. Por isso, é importante estabelecer métricas objetivas que dêem um indicativo de que estes parâmetros de cobertura - a meta a ser atingida pela verificação - são satisfatórios.

No contexto de software, existe uma técnica de teste capaz de fornecer o tipo de métrica anteriormente citada. Essa técnica é conhecida como teste de mutação [12], a ser detalhada no Capítulo 2, Seção 2.2, deste trabalho.

Diante do exposto, pode-se estabelecer os objetivos deste trabalho conforme descrição a seguir.

1.1 Objetivos

O principal objetivo do trabalho é avaliar o uso da análise de mutação na metodologia de verificação funcional VeriSC [10]. Para a aplicação da técnica devem ser escolhidos *testbenches* nos quais o engenheiro de verificação tenha conseguido uma cobertura que ele considera adequada, mas que não haja parâmetro objetivo para determinar a qualidade da cobertura dos parâmetros de cobertura escolhidos.

1.1.1 Objetivos Específicos

Dentre os objetivos específicos, pode-se destacar:

- Entender e aplicar, de forma prática, os conceitos de teste de mutação no contexto de hardware;
- Adaptar a técnica de análise de mutação à metodologia de verificação funcional VeriSC;
- Avaliar a qualidade da verificação funcional realizada com base nos resultados obtidos a partir da análise de mutação.

Para averiguação da técnica proposta, foram analisados dois casos de estudo, que constituem módulos importantes no contexto do desenvolvimento de IP *core*, com diferentes níveis de complexidade, sendo estes:

- O módulo DPCM (*Differential Pulse Code Modulation*), exemplo de aplicação da técnica em um módulo cuja principal função é converter um sinal analógico em um sinal digital;
- O módulo IDCT (*Inverse Discrete Cosine Transform*), parte importante de um IP-*core* decodificador de vídeo em formato MPEG4, cujo objetivo é calcular a transformada inversa do cosseno de um sinal.

1.2 Organização da Dissertação

Os demais capítulos desta dissertação estão organizados conforme descrição a seguir.

- **Capítulo 2**: Concentra-se a fundamentação teórica necessária à aplicação da técnica de teste de mutação à metodologia VeriSC;
- **Capítulo 3**: Está contida a descrição das mutações a serem realizadas sobre o ambiente de verificação;
- **Capítulo 4**: São apresentados e analisados os resultados obtidos com a aplicação da análise de mutação sobre os dois casos de estudo escolhidos.
- **Capítulo 5**: São apresentadas as considerações finais e as sugestões para trabalhos futuros.

Capítulo 2

Fundamentação Teórica

Um IP *core* é uma unidade de propriedade intelectual (IP - *Intellectual Property*). Quando se trata de um IP *core* digital, este é passível de implementação em HDL (*Hardware Description Language*).

Dentro do contexto do projeto de um IP *core*, pode-se trabalhar com três visões (Figura 2.1). A visão que corresponde à intenção do projeto, que é anterior à especificação. Portanto, só existe no pensamento do projetista. A visão que corresponde à especificação é aquela que o projetista - consciente da intenção do projeto - consegue documentar, de maneira a elaborar uma especificação funcional do sistema a ser implementado. A última visão é a implementação, que corresponde ao que foi implementado daquilo que foi especificado. O ideal seria que os três conjuntos fossem iguais, de tal forma que a intenção do projeto fosse preservada em sua implementação [25].

No modelo apresentado na Figura 2.1, existem alguns subconjuntos importantes que devem ser observados:

- *Subconjunto E*: É a parte da intenção e da especificação do projeto que não foi implementada;

- *Subconjunto F*: É tudo aquilo que foi especificado e implementado, mas não era a intenção do projeto;

- *Subconjunto G*: Representa a parte que é intenção do projeto e que foi implementada, porém não fazia parte da especificação;

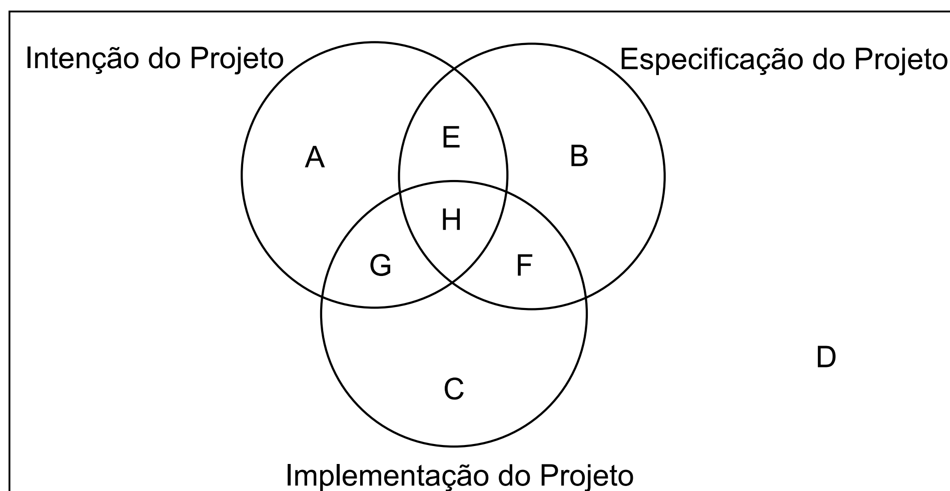

Figura 2.1: Visões de um projeto de hardware (IP *core*).

- *Subconjunto H*: É tudo aquilo que foi implementado e que faz parte da especificação e da intenção do projeto.

Existe um esforço para maximizar o tamanho do subconjunto H, de forma que a diferença entre os três conjuntos seja vazia.

Um projeto de IP *core* consiste de várias fases dispostas em um processo de desenvolvimento, destacando-se especificação, implementação e verificação. As fases mais comumente realizadas em um projeto de IP *core* podem ser visualizadas na Figura 2.2 [10].

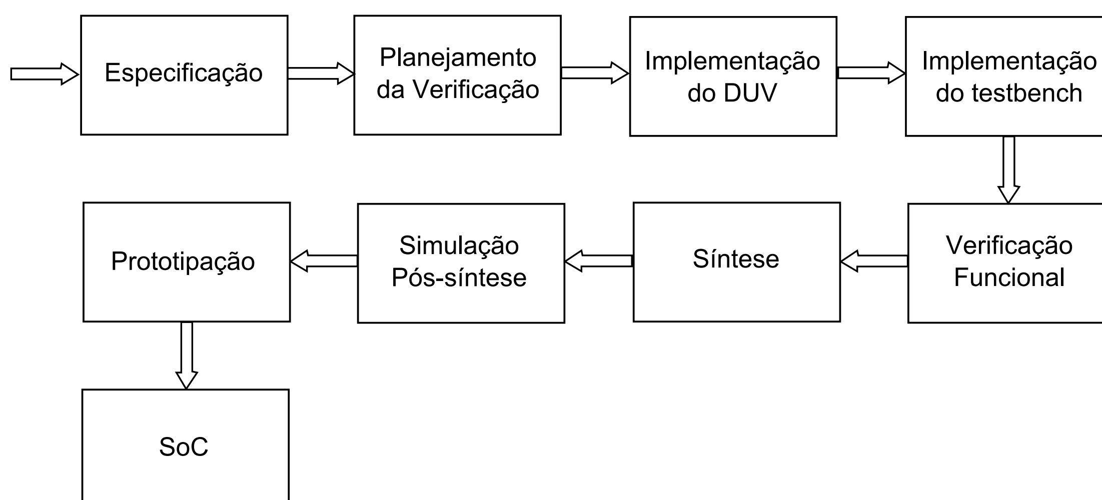

Figura 2.2: Fases de um projeto de hardware (IP *core*).

As fases acima representadas podem ser resumidas da seguinte forma:

- *Especificação*: Nesta fase, é elaborada a documentação referente às funcionalidades do projeto. O projetista tenta fazer a intenção do projeto tomar a forma de um documento (ou uma série de documentos) que possa ser compartilhado por toda a equipe de desenvolvimento.

- *Planejamento da Verificação*: Nesta fase, a equipe de verificação planeja como será desenvolvido o ambiente de verificação, define os vetores de teste que serão usados, determina os objetivos a serem alcançados pela verificação e estabelece um cronograma para o cumprimento de tais objetivos.

- *Implementação do DUV*: Nesta fase, o DUV (*Design Under Verification*) é implementado. Um IP *core* pode ser um DUV, ou uma combinação de vários DUV, cada um exercendo uma função específica dentro do sistema, como mostrado na Figura 2.3. Um DUV é uma unidade que implementa uma funcionalidade descrita na especificação. Portanto, deve ser verificada com respeito a esta especificação. Este processo de verificação é executado por meio da simulação do DUV.

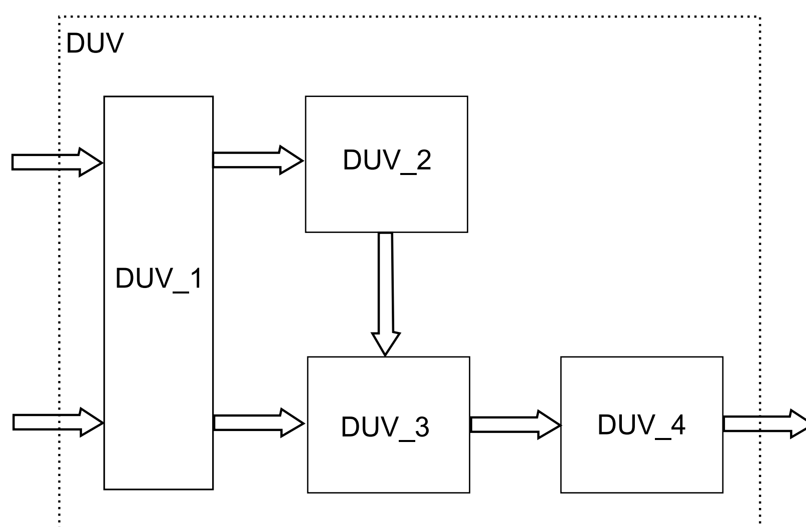

Figura 2.3: Exemplo hipotético de um DUV.

Um DUV é descrito, geralmente, no nível chamado de RTL [16], que é uma forma de descrever o IP-*core* por meio do uso de registradores. Ou seja, o IP-*core* é visto a partir da transferência de dados entre os seus registradores. Para que um IP-*core* se transforme em um dispositivo real de hardware se faz necessária a sua implementação no nível RTL por meio de alguma linguagem de descrição de hardware (HDL).

- *Implementação do testbench*: Nesta fase, implementa-se o ambiente de verificação, chamado de *testbench*, que tem por objetivo a geração de estímulos e observação das respostas. Um *testbench* deve conter as seguintes características [3]:

  - Ser auto-verificável: Não deve haver intervenção humana na comparação de estímulos esperados e estímulos recebidos;
  - Especificado no nível de transação (*Transaction Level*, ou TL): Interfaces descritas em termos de fios e sinais devem ser utilizadas apenas para conectar o DUV;
  - Randômico (Aleatório): Os estímulos devem ser gerados de forma aleatória dentro de um intervalo bem definido;
  - Dirigido à cobertura: A geração de estímulos e a quantidade de estímulos gerados devem depender apenas dos parâmetros de cobertura funcional medidos durante a simulação.

  Para este trabalho, não são considerados *testbenches* que não possuam estas características.

- *Verificação*: A verificação trata de determinar se se o IP *core* em questão está de acordo com os requisitos do projeto [3][25]. Os tipos de verificação podem ser divididos da forma [3]:

  - Verificação formal ou estática: Está relacionada com a prova de teoremas ou verificação de equivalência. Pode envolver a análise do espaço de estados do problema o que, em muitos casos, leva a uma complexidade inviável;
  - Verificação funcional ou dinâmica: A partir de simulações de modelos em diferentes níveis de abstração, busca-se mostrar que o IP *core* está de acordo com a especificação;
  - Verificação Híbrida: Utilização das duas técnicas anteriores sobre os submódulos do IP *core* onde cada uma for melhor aplicável;

  O trabalho ora descrito, concentra-se na verificação funcional.

- *Síntese*: Nesta fase, após ser verificado, o DUV passa por uma tradução do modelo descrito em uma linguagem de descrição de *hardware* para uma *netlist* que contém,

além da própria descrição, informações sobre os atrasos de portas lógicas necessárias para o funcionamento do DUV em um dispositivo de prototipação pré-determinado.

- *Simulação Pós-síntese*: Nesta fase, o DUV passa novamente pela mesma simulação pela qual já passou na fase de Verificação, permitindo determinar se a introdução dos atrasos de portas lógicas altera o funcionamento de tal forma que algum erro seja produzido.

- *Prototipação*: O DUV é, de fato, prototipado em um dispositivo a partir do qual o seu comportamento possa ser concretamente (e não mais em simulação) analisado.

- *SoC*: Após ser prototipado, o DUV pode ser enviado para a fase de desenho do *layout* do circuito que se quer colocar em um CI (Circuito Integrado).

Considerando o tipo de verificação (funcional) utilizado para este trabalho, a simulação apenas não é suficiente para assegurar que todas as funcionalidades previstas na especificação foram implementadas. Por isso, torna-se necessário o uso de mecanismos, como a cobertura funcional, para medir o estado e o progresso da simulação [10].

Estima-se que o processo de verificação funcional requer cerca de 70% dos recursos (tempo, dinheiro, mão de obra) de um projeto de hardware [3][25], pois visa detectar erros em fases iniciais do projeto, minimizando a probabilidade de encontrar erros em fases nas quais estes erros causem maior prejuízo. Isto é, na fase de produção do CI.

Existe, portanto, a necessidade de tornar o processo de verificação funcional tão rápido quanto possível sem comprometer a sua eficiência, para que os custos com esta fase sejam minimizados.

Para este trabalho, será utilizada a metodologia de verificação funcional VeriSC, pois os *testbenches* propostos na metodologia atendem aos requisitos anteriormente mencionados. Essa metodologia também dispõe de um suporte ferramental que automatiza muitas das tarefas de construção dos *testbenches*, proporcionando a redução do tempo de verificação. Esta metodologia foi desenvolvida no contexto do projeto Brazil-IP [5]. Este projeto visa a formação de recursos humanos no Brasil para a indústria de microeletrônica. Outro objetivo do projeto é o desenvolvimento de IP *cores* com qualidade de acordo com os padrões internacionais. Para atingir esta meta, foram desenvolvidos três IP cores: Um decodificador de

MP3, um microcontrolador 8051 e um decodificador de MPEG4 [29]. Para os três IP *cores* foi elaborado o *layout* para fabricação em silício. Vale destacar que estes chips funcionaram corretamente.

Em conferência realizada na França, o IP-SoC 2006 [15], o trabalho apresentado pelos integrantes do Brazil-IP, referente aos resultados do projeto, foi premiado como o melhor artigo do evento. Outro resultado importante obtido pelo Brazil-IP foi a criação do LINCS-CETENE [8], uma das seis *Design Houses* do Brasil, que liderará os esforços do - agora programa do Governo Federal - Brazil-IP.

2.1 A Metodologia de Verificação Funcional VeriSC

As metodologias de verificação funcional existentes apresentam alguns problemas em comum que devem ser observados [10]:

- Consomem uma quantidade substancial dos recursos de um projeto;
- O código gerado para o *testbench* muitas vezes não pode ser reusado;
- A decomposição hierárquica não é considerada no processo de verificação funcional, causando o aparecimento de trabalho extra para a construção de mais *testbenches*;
- A verificação funcional só começa quando todo o projeto é modelado no nível RTL;
- *Testbenches* são depurados junto com o DUV, de tal forma que quando uma falha é encontrada, não se sabe se ela está no DUV ou no *testbench*;

Muitos destes problemas surgem pelo fato das metodologias tradicionais de verificação funcional proporem que o DUV seja implementado antes do *testbench*.

A metodologia de verificação funcional VeriSC propõe que o *testbench* seja feito antes do DUV, de tal forma que ele possa ser testado independentemente, eliminando vários dos problemas anteriormente expostos.

A seguir, uma breve descrição da metodologia, com destaque até o passo a partir do qual será feita uma proposta de enriquecimento.

Para a utilização da metodologia, é necessário um RM (*Reference Model*), um modelo que implementa todas as funcionalidades previstas na especificação. Não está no escopo da metodologia definir a forma de obtenção do modelo de referência.

Na Figura 2.4 é apresentada a estrutura geral de um *testbench* [10].

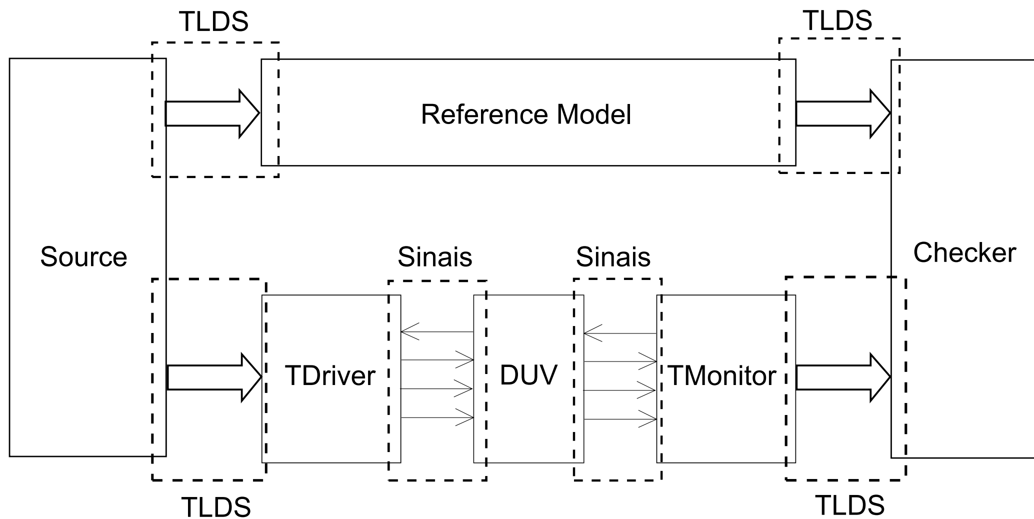

Figura 2.4: Estrutura geral de um *testbench* segundo a metodologia VeriSC.

A função do *testbench* é fornecer estímulos ao RM e ao DUV e capturar suas saídas verificando se estas são equivalentes. A sincronização é feita por meio de estruturas de dados que carregam transações (pode ser, por exemplo do tipo FIFO, ou *First-In-First-Out*, ou seja, o primeiro elemento a ser inserido é o primeiro a ser retirado). Neste trabalho, estas estruturas serão chamadas TLDS (*Transaction Level Data Structure*). A metodologia foi elaborada para verificação de sistemas digitais síncronos, que são aqueles sistemas digitais regidos (sincronizados) por um sinal de relógio (ou *clock*).

A seguir, uma breve descrição dos componentes do *testbench* e suas respectivas funções.

- *Source*: Responsável por fornecer dados em TL para o RM e para o DUV através do TDriver. O *Source* se conecta a estes elementos por meio de TLDS. A mesma quantidade de conexões que há com TDrivers há, também, com o RM e há uma conexão para cada interface;

- *TDriver*: Tem por objetivo receber os dados em TL do *Source*, gerar sinais referentes ao protocolo utilizado pelo DUV e enviar estes sinais juntamente com os dados recebidos, também no nível de sinais. Existe um TDriver para cada interface do DUV.

A vantagem é a modularização, de modo que, se for necessário mudar uma interface, apenas um TDriver é alterado;

- *TMonitor*: É o análogo do TDriver para as interfaces de saída do DUV. Recebe os sinais da interface correspondente, os transforma em dados em TL e os envia por meio de uma TLDS para o *Checker*;

- *Reference Model*: O modelo de referência recebe dados em TL do *Source* através de uma TLDS, realiza as operações que estão na especificação e envia dados em TL para o *Checker*, também por meio de uma TLDS.

De acordo com a metodologia VeriSC, é preciso então construir estes elementos independentemente do DUV e depurá-los de maneira simples. A primeira fase da construção de um *testbench* é a geração de seus elementos para o DUV como um todo [10]. Esta fase pode ser concretizada em três passos:

1. O RM (Reference Model) deve ser testado de acordo com sua capacidade de interagir com o *testbench* (receber e produzir dados em TL). Para isso, cria-se um *Pre-Source* que gera apenas entradas para o RM e um *Sink* que também recebe saídas apenas do RM. A realização deste passo, tem por objetivo a verificação das ligações entre os elementos *Pre-Source*, RM e *Sink*. Pode-se, por exemplo, ter no *Sink* uma instrução que imprime na saída padrão o resultado enviado pelo RM. Para que o engenheiro de verificação visualize a saída, e verifique mediante análise manual se as ligações estão corretas. Esta construção pode ser visualizada na Figura 2.5.

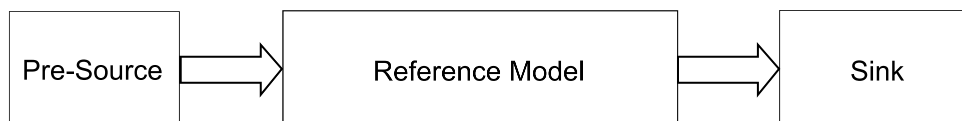

Figura 2.5: Primeiro passo para construção de um *testbench*.

2. O *Source* e o *Checker* também devem ser testados de acordo com sua capacidade de gerar estímulos válidos e de verificar a equivalência da resposta do RM a estes estímulos. Isto pode ser verificado utilizando duas instâncias do modelo de referência. O *Source* estimula estas duas instâncias e o *Checker* verifica se suas saídas são equivalentes. Ao utilizar esta abordagem com todos os estímulos especificados e verificar

que o *Checker* não acusa erros, erros devem ser introduzidos para avaliar se o *Checker* é capaz de acusá-los. Esta estrutura é mostrada na Figura 2.6.

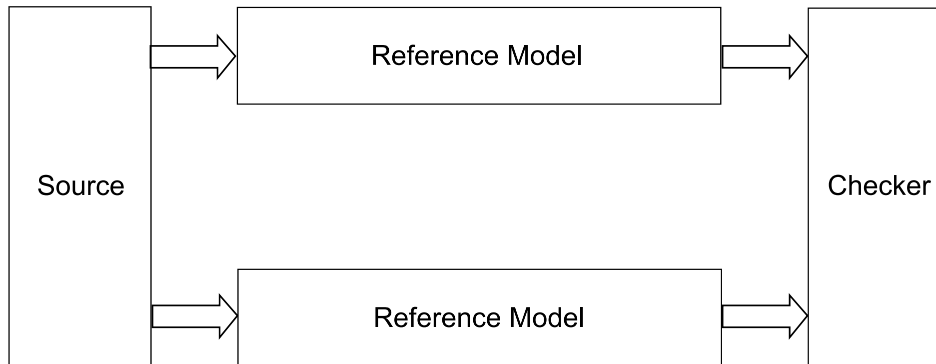

Figura 2.6: Segundo passo para a construção do *testbench*.

3. Neste passo, serão testados o(s) *TDriver*(s) e o(s) TMonitor(s). Para efeito de simplificação, serão considerados apenas um *TDriver* e um *TMonitor*, que serão chamados TDriverA e TMonitorA. Este é um dos passos mais importantes, pois é nele que o *testbench* pode ser simulado sem a presença do DUV. Analogamente ao passo anterior, o modelo de referência fará o papel do DUV. O problema é que o modelo de referência se comunica em TL, e o DUV no nível de sinais. Portanto, torna-se necessário incluir elementos adicionais para interconectar o RM e o TDriverA e o TMonitorA. Desta forma, são criados um *TMonitor* chamado TMonitor0 que tem a mesma interface do TDriverA e recebe os sinais do TDriverA e repassa os dados recebidos para o RM em TL, e um *TDriver* chamado TDriver0 que tem a mesma interface do TMonitorA e é responsável por receber as respostas do RM em TL e enviá-las para o TMonitorA no nível de sinais. Em momento posterior, a tripla (TMonitor0, RM, Tdriver0) será substituída pelo DUV para que seja realizada a sua verificação funcional. Na Figura 2.7, é mostrada a estrutura proposta.

   Neste ponto, foi identificada uma lacuna na metodologia. Fica claro que devem ser introduzidos erros no modelo de referência para testar a funcionalidade do *Checker*, mas a metodologia não especifica uma forma sistemática para realizar esta atividade. Também foge do escopo da metodologia determinar critérios objetivos para analisar se a cobertura está satisfatória. Propõe-se, então, que seja introduzida na metodologia a técnica de análise de mutação (apresentada no Capítulo 2, Seção 2.2), para preencher

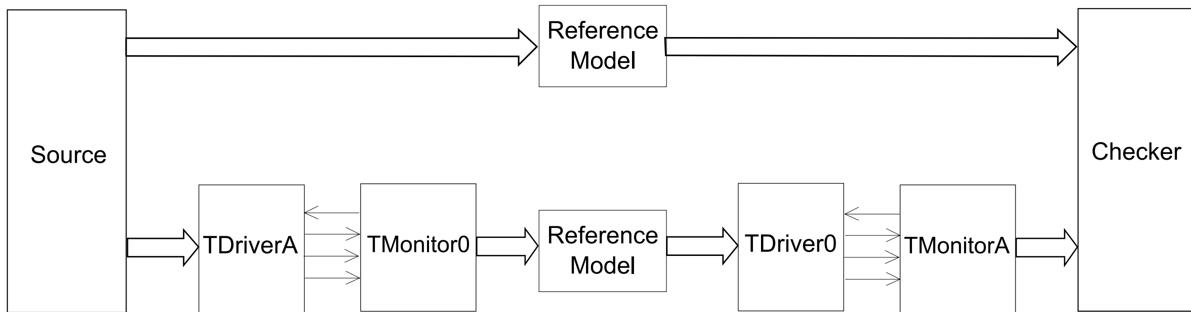

Figura 2.7: Terceiro passo para a construção do *testbench*.

estas lacunas.

A metodologia VeriSC não prevê em que passo devem ser acrescentados os parâmetros de cobertura. Para que seja possível a inclusão da técnica, deve-se acrescentá-los no *testbench* a partir do passo 2, pois para identificar os mutantes, torna-se necessário ter as duas instâncias do modelo de referência, uma que recebe as mutações e outra que é, de fato, o modelo de referência. Mais detalhes sobre a aplicação das mutações podem ser encontrados na Seção 2.2.

2.2 Análise de mutação (*Mutation Analysis*)

No contexto dos projetos de software, existe uma busca crescente pela qualidade dos produtos finais. Muitas técnicas de teste visam assegurar esta qualidade para que os sistemas tenham aceitação do mercado sempre mais exigente. Uma dessas técnicas que merece destaque é a análise de mutação. Para a descrição da análise de mutação deve-se considerar a Hipótese do Programador Competente e o Efeito de Acoplamento.

2.2.1 Hipótese do Programador Competente

A hipótese do programador competente afirma que programadores têm a tendência de escrever programas que são próximos de estarem corretos [12]. Eles não escrevem programas ao acaso, mas estão sempre diminuindo a distância entre o que é feito e o programa esperado. Programadores também têm uma idéia dos erros mais comuns de acontecer e a habilidade e oportunidade de analisar seus programas. Em outras palavras, um programador pode até escrever um programa incorreto, porém este programa difere de um programa correto por

faltas relativamente simples. Faltas simples e complexas são definidas da seguinte forma [19]:

- Falta simples: É uma falta que pode ser corrigida fazendo-se uma única alteração na expressão de origem;
- Falta complexa: É uma falta que não pode ser corrigida fazendo-se uma única alteração na expressão de origem.

2.2.2 Efeito de acoplamento (*Coupling effect*)

Serão denominados de mutantes aqueles programas nos quais foi introduzida uma falta simples, e de mutantes de alta ordem aqueles que são gerados ao introduzir múltiplas faltas simples ao programa original. Existem, porém, faltas complexas que não podem ser geradas por mutação de alta ordem. Desta forma, pode-se formular a hipótese do efeito de acoplamento da seguinte forma [19]: "Faltas complexas estão acopladas a faltas simples de uma forma tal que um conjunto de casos de teste que detecta todas as faltas simples de um programa, vai detectar um alto percentual das faltas complexas."

O efeito de acoplamento foi proposto em 1978 [12], depois foi mostrado empiricamente em 1992 [19] e demonstrado teoricamente em 1995 [34][35].

Restringindo, então, o efeito de acoplamento para o domínio das mutações, tem-se que [19]: "Mutantes de alta ordem estão acoplados a mutantes simples de uma forma tal que um conjunto de casos de teste que detecta todos os mutantes simples de um programa, vai detectar um alto percentual dos mutantes de alta ordem."

2.2.3 Análise e Teste de Mutação

Para tornar claro o objetivo deste trabalho, faz-se necessário a introdução dos conceitos de Análise de Mutação (*Mutation Analysis*) e Teste de Mutação (*Mutation Testing*) [23].

Análise de Mutação

A análise de mutação é a introdução de faltas em programas, criando várias versões deste mesmo programa, cada um contendo uma única falta.

Casos de teste são então executados sobre estes programas que contêm faltas, com o objetivo de determinar se o conjunto de casos de teste consegue detectar tais faltas. Desta forma, pode-se dizer que estes programas que contêm faltas são chamados de *mutantes*, e um mutante é *morto* quando consegue-se distinguir, a partir dos testes, a resposta de um mutante da resposta do programa original.

Apenas mutantes simples são gerados , visto que o efeito de acoplamento garante a detecção de um alto percentual dos mutantes de alta ordem analisando apenas os simples.

A medida de qualidade do conjunto de casos de teste, gerada pela análise de mutação é chamada escore de mutação e é obtida segundo a relação representada na Equação 2.1.

$$EM = \frac{M}{E} \tag{2.1}$$

em que EM é o escore de mutação, M é a quantidade de mutantes mortos e E é a quantidade de mutantes gerados não equivalentes ao programa original. Pode-se, então, dispor de duas situações:

- $EM = 1$, ou seja, nenhum mutante ficou vivo, ou ficaram vivos apenas os mutantes equivalentes ao programa original.
- $EM < 1$, ou seja, há mutantes vivos que não são equivalentes ao programa original e, portanto, a cobertura especificada tem um grau de qualidade proporcional ao escore de mutação.

Assim sendo, quanto mais próximo de 1 (um), melhor o escore de mutação, e conseqüentemente a cobertura. Quanto mais próximo de 0 (zero), pior o escore de mutação.

Teste de Mutação

O teste de mutação refere-se à geração de casos de teste visando melhorar o escore obtido pela análise de mutação. Caso o escore de mutação seja menor do que 1 (um), mais casos de teste devem ser gerados para aproximar este escore de 1 (um). Desta forma, pode-se afirmar que o teste de mutação é uma técnica mais abrangente do que a análise de mutação. Este trabalho, porém, concentra-se na análise de mutação, pois o objetivo é a introdução de faltas nos casos de teste dos programas, criando várias versões destes programas, cada um

contendo uma única falta. Foge do escopo do trabalho, a construção de novos casos de teste visando melhorar o escore obtido pela análise de mutação.

2.2.4 Operadores de Mutação

As modificações sintáticas que resultam em mutantes são determinadas por um conjunto de operadores de mutação. Este conjunto é determinado pela linguagem de programação utilizada e pelo sistema de mutação adotado [20]. Operadores são criados por dois motivos: Induzir mudanças sintáticas baseadas em erros que programadores cometem tipicamente, ou forçar objetivos de teste comumente requeridos. A seguir, são apresentados alguns operadores de mutação para a linguagem ANSI C, pois os modelos de referência utilizados neste trabalho são escritos nesta linguagem. Estes operadores podem ser classificados em 4 tipos, de acordo com o elemento da linguagem sobre o qual o operador atua. Assim sendo, os operadores de mutação definidos para a linguagem C atuam sobre instruções, operadores da linguagem, variáveis e constantes. Todos os operadores desenvolvidos para esta linguagem podem ser consultados no trabalho desenvolvido por Mathur [1].

- STRP (*Trap on Statement Execution*): Este operador de instrução tem por objetivo revelar código inalcançável no programa. Para isso, ele substitui cada instrução do programa original pela instrução *trap_on_statement()* que termina a execução do mutante. Caso esta instrução seja atingida, o mutante é tratado como morto.

- VTWD (*Twiddle Mutations*): Valores de variáveis e expressões podem comumente diferir do valor desejado por +1 ou -1. Esse operador representa esse erro. Cada variável x é substituída por $pred(x)$ e $succ(x)$, em que cada uma retorna, respectivamente, o predecessor imediato de x e o sucessor imediato de x.

- CRCR (*Required Constant Replacement*): Sejam I e R, os conjuntos $\{0, 1, -1, u_i\}$ e $\{0,0, 1,0, -1,0, u_r\}$, respectivamente. u_i e u_r denotam valores inteiros e reais utilizados pelo usuário do programa, respectivamente. Este operador modela o uso de uma variável onde deveria ter sido utilizado um elemento de I ou de R. Como exemplo, pode-se considerar a seguinte expressão *k = j + *p*, onde *k* e *j* são variáveis do tipo inteiro e *p* é um ponteiro para um inteiro. Quando aplicado à essa expressão, CRCR gerará os seguintes mutantes:

- *k = 0 + *p*
- *k = 1 + *p*
- *k = 1 + *p*
- *k = ui + *p*
- *k = j + null*

2.2.5 Aplicação da Análise de Mutação

Dadas as hipóteses do programador competente e o efeito de acoplamento, é possível definir um sistema para determinar a qualidade dos casos de teste de um programa [12]. O objetivo é determinar se um conjunto de casos de teste T é adequado para testar um programa P. O sistema proposto é executado da seguinte forma:

1. É executado o conjunto de casos de teste T sobre o programa P. Caso P não passe no conjunto de casos de teste, então P certamente é errôneo. Caso contrário, o programa ainda pode conter erro e o conjunto de casos de teste não é suficientemente sensível para encontrar o erro, ou seja, não é adequado;

2. O sistema de mutação, cria então um conjunto k de mutantes de P que diferem do programa original apenas por uma falta simples. Esses mutantes serão denominados $P_1, P_2, ..., P_k$;

3. Para o conjunto de casos testes T anteriormente definido, tem-se que:

   - A saída de P_i mutantes difere da saída de P para algum teste do conjunto de casos de teste. Neste caso, diz-se que o mutante está “morto”. Ou seja, o erro introduzido pela mutação foi, de fato, detectado pelo conjunto de casos de teste;
   - A saída de P_j mutantes não difere da saída de P para qualquer dos testes do conjunto de casos de teste. Neste caso, diz-se que o mutante está vivo. Este fato pode acontecer por dois motivos:
     - O conjunto de casos de teste não é suficientemente sensível para verificar o erro introduzido pela mutação em P_j;

  – P_j e P são, de fato, equivalentes e nenhum conjunto de casos de teste pode distingui-los.

Para saber qual dos dois motivos de "sobrevivência" de um mutante é o correto, o método empregado mais comumente é a análise manual por parte do engenheiro de testes.

Pode-se afirmar, portanto, que um conjunto de casos de teste que "mata" todos os mutantes, ou deixa vivos apenas os mutantes equivalentes ao programa original é dito adequado no seguinte sentido: ou o programa P está correto em relação aos testes realizados, ou existe um erro inesperado em P.

Um dos grandes problemas da análise de mutação é o consumo de recursos computacionais. Existem, porém, algumas variantes da técnica de análise de mutação que a tornam menos onerosa. Pode-se, dentre elas, destacar a mutação seletiva [23][20]. Esta técnica visa reduzir a quantidade de mutantes gerados por meio de uma abordagem de aproximação sugerida por Mathur [17]. Outra técnica é a mutação fraca [21][22], que tem por objetivo criar um conjunto de casos de teste mais limitado (mais "fraco"), que exige menos poder computacional, e é quase tão completo quanto a análise de mutação em sua forma original.

2.3 Trabalhos Relacionados

No contexto de hardware, existem outros trabalhos relacionados à mutação na verificação funcional de IP *core*. O primeiro é o trabalho de Vado [33], cuja proposta é a melhoria sistemática dos vetores de teste utilizados na validação de circuitos digitais. Em sua abordagem, as mutações são aplicadas sobre os modelos de circuitos digitais, escritos em linguagens como Verilog ou VHDL. O autor faz uma comparação entre os métodos utilizados anteriormente com o que ele desenvolveu, obtendo resultados significativos.

Outro trabalho relacionado é o desenvolvido por Serrestou [30], no qual são utilizadas métricas de teste de mutação na verificação de componentes descritos na linguagem VHDL. O autor propõe que as mutações sejam realizadas sobre o DUV e, baseado no escore de mutação, um algoritmo genético é utilizado para melhorar automaticamente os vetores de teste do IP *core*.

As abordagens dos autores desses dois trabalhos diferem do método descrito nesta dissertação por inserir as mutações no DUV. Para desenvolver *testbenches* segundo a metodologia VeriSC, não é necessária a presença do DUV. Portanto, as mutações utilizadas neste trabalho são sobre o modelo de referência. Outro ponto de diferença reside no fato de que os autores propõem o melhoramento dos vetores de teste baseando-se no resultado da mutação realizando, portanto, teste de mutação. Neste trabalho, o objetivo é utilizar a análise de mutação como medida de qualidade dos parâmetros de cobertura funcional adotados.

2.4 Discussão

Um IP *core* é uma unidade de propriedade intelectual que, ao ser implementada em uma linguagem de descrição de hardware, precisa ser verificada em relação à sua especificação. Metodologias de verificação podem ser utilizadas para esse propósito. A cobertura funcional pode ser definida como um mecanismo para medir o estado e o progresso da verificação do IP *core*. A cobertura é um parâmetro que permite ao engenheiro de verificação se posicionar em relação ao término da verificação. Uma questão que deve ser levada em consideração é: “Os parâmetros de cobertura são adequados?”. O termo “adequado” é subjetivo. Portanto, precisa-se de uma métrica objetiva que forneça uma indicação de quão adequada é a cobertura funcional. Para esse propósito, será utilizada neste trabalho a análise de mutação. No contexto de software, a aplicação dessa técnica é capaz de produzir um escore de mutação (relação entre mutantes vivos e mutantes mortos) que pode ser utilizado como medida de qualidade de um dado conjunto de casos de teste. Foram encontrados outros trabalhos que utilizam a análise e o teste de mutação no contexto de *hardware*, o que demonstra que há interesse em aplicar mutação para auxílio na verificação de sistemas digitais.

Capítulo 3

Descrição da Análise de Mutação Aplicada à Metodologia VeriSC

Considerando os objetivos apresentados no Capítulo 1 (Introdução) e os conceitos abordados no Capítulo 2 (Fundamentação Teórica), a aplicação da análise de mutação na metodologia VeriSC se dá de acordo com o modelo apresentado na Figura 3.1.

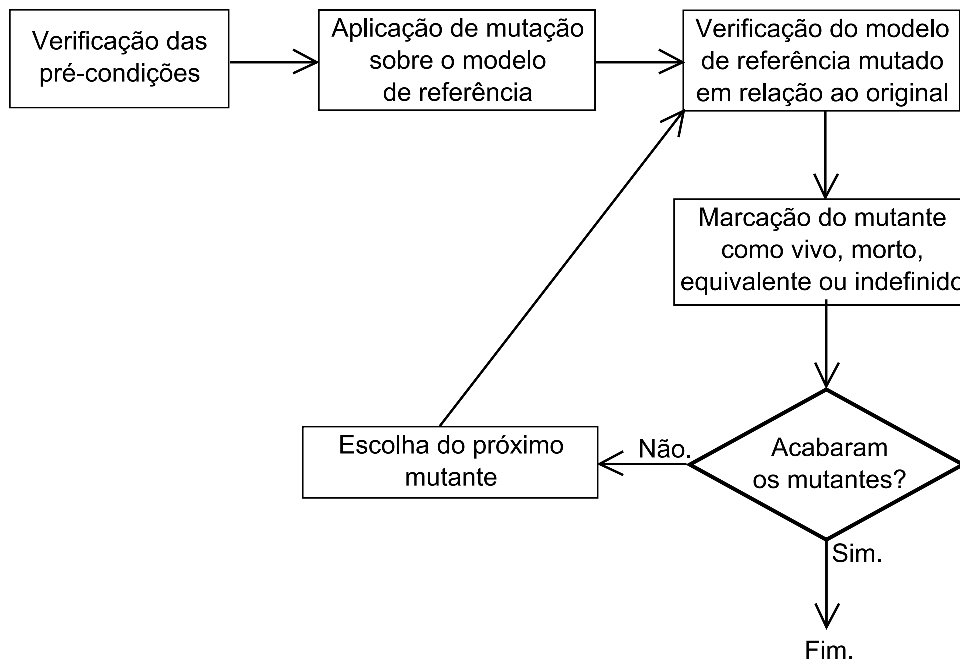

Figura 3.1: Aplicação da análise de mutação na metodologia VeriSC.

Nas seções a seguir, será definida cada etapa da aplicação da análise de mutação.

3.1 Verificação das pré-condições

Para a aplicação da análise de mutação, deve-se ter um ambiente de verificação que atenda as seguintes pré-condições:

- *Cobertura*: Como a intenção é obter uma medida de qualidade dos parâmetros de cobertura da verificação funcional, é necessário que estes parâmetros estejam estabelecidos antes da aplicação da técnica. Em VeriSC, estes parâmetros são adicionados nos passos 2 ou 3 descritos no Capítulo 2 deste trabalho.

- *Simulação*: A simulação deve atingir 100% de cobertura e não acusar erros ou "travar" a execução. Um erro ou travamento encontrado antes da aplicação da análise de mutação caracteriza um programa original já morto, o que não faz sentido para esta análise.

- *Programador Competente*: Quem realiza a análise de mutação é um membro da equipe de verificação. De preferência um engenheiro de verificação que conheça a linguagem de programação do modelo de referência, mas que não tenha participado do seu desenvolvimento.

3.2 Aplicação de mutação sobre o modelo de referência

Na metodologia VeriSC, não é necessária a presença do DUV para que se possa construir o ambiente de verificação. Até mesmo os parâmetros de cobertura já têm que estar definidos antes de introduzir o DUV no *testbench*. Assim sendo, pretende-se obter a medida de qualidade da verificação funcional da mesma forma: sem a necessidade da presença do DUV. Portanto, as mutações são aplicadas sobre uma das instâncias do modelo de referência como mostrado na Figura 3.2.

Para a geração dos mutantes, foi utilizada a ferramenta Proteum. Esta foi a única ferramenta encontrada que implementa todos os operadores disponíveis para a linguagem C. A Proteum está disponível mediante solicitação aos seus idealizadores [11]. Mais informações sobre a Proteum e sobre as outras ferramentas utilizadas (eTBc e Subversion) podem ser encontradas no Anexo A.

Na Figura 3.2, tem-se um *testbench* que possui *Source* e *Checker* completos. Os parâmetros de cobertura foram adicionados, e a simulação neste passo atinge as pré-condições estabelecidas na Seção 3.1 do Capítulo 3.

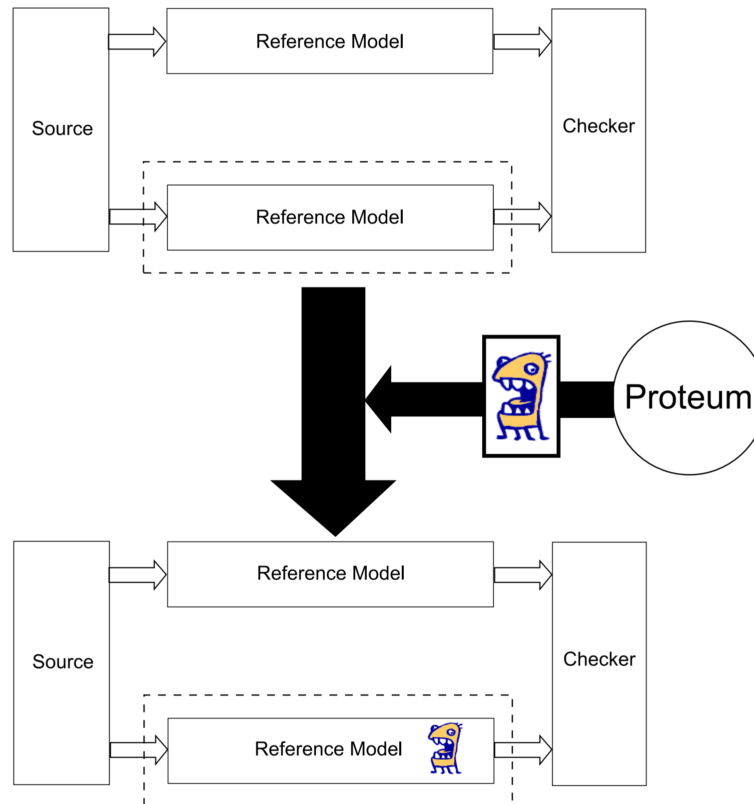

Figura 3.2: Aplicação das mutações sobre o modelo de referência no passo 2 da metodologia VeriSC.

Na Figura 3.2, tem-se um *testbench* que possui *Source*, *TDriver*, *Tmonitor*, e *Checker* completos. Os parâmetros de cobertura foram adicionados, e a simulação neste passo atinge as pré-condições estabelecidas na Seção 3.1 deste capítulo.

Apesar de não ser necessária a presença do DUV para a realização de análise de mutação em *testbenches*, está prática também é possível. As mutações ainda são aplicadas sobre o modelo de referência como mostrado na figura 3.4.

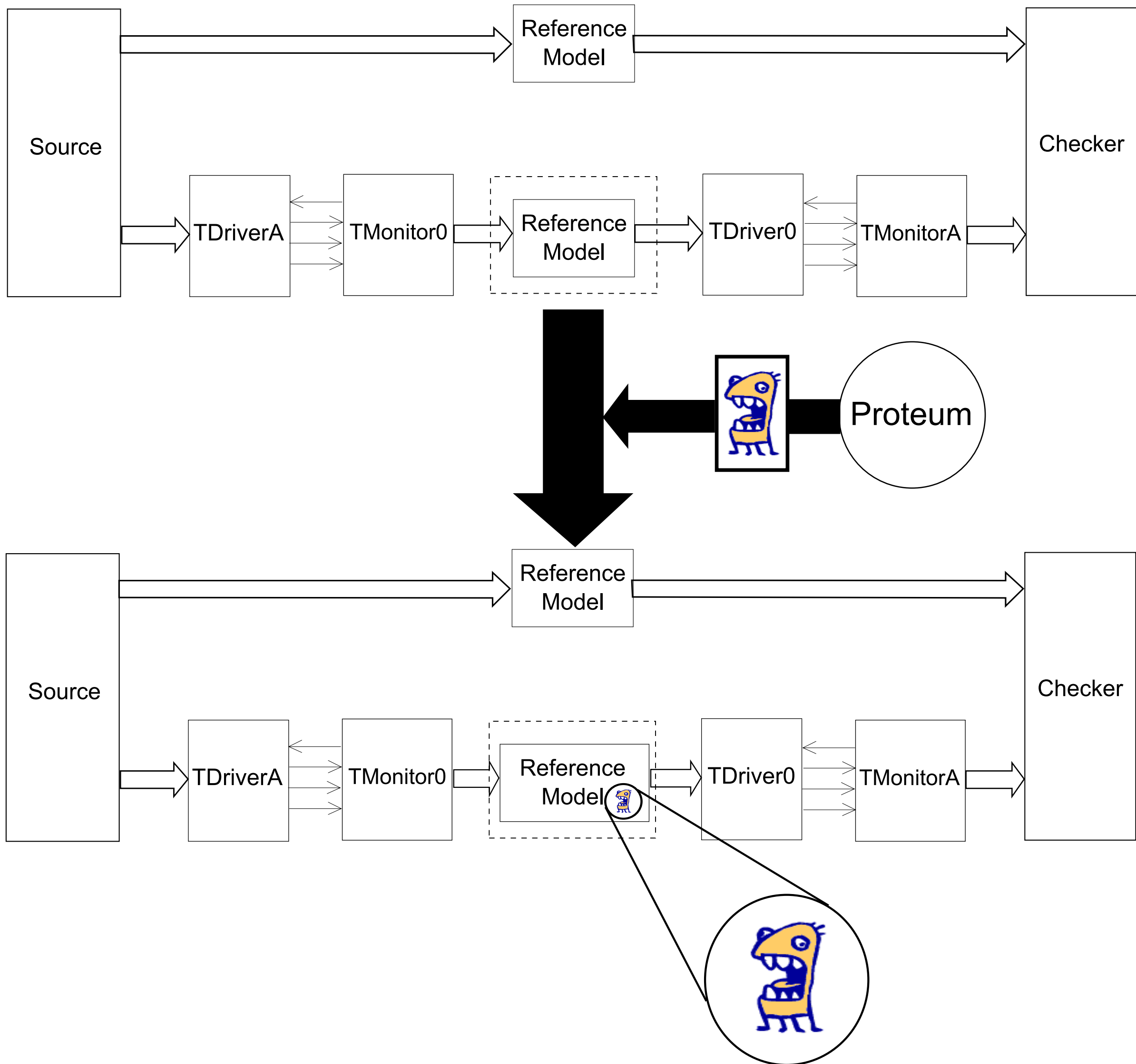

Figura 3.3: Aplicação das mutações sobre o modelo de referência no passo 3 da metodologia VeriSC.

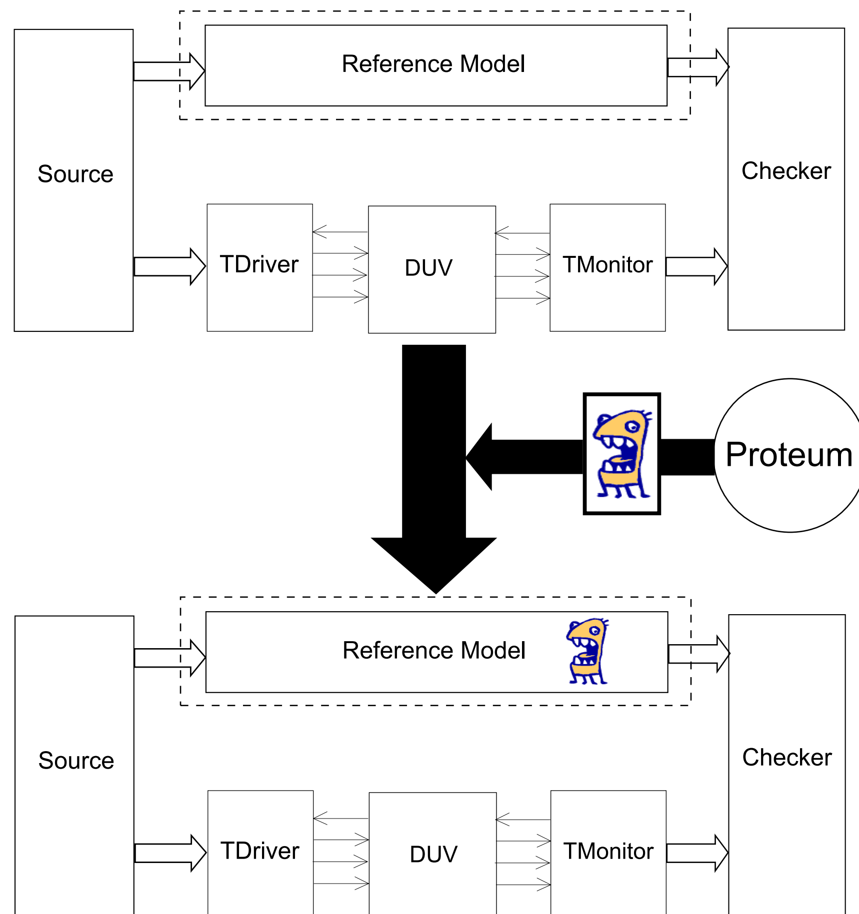

Figura 3.4: Aplicação das mutações sobre o modelo de referência após a inserção do DUV.

3.3 Verificação do modelo de referência mutado em relação ao original

Em qualquer um dos casos mostrados nas Figuras 3.2 ou 3.3, a cada mutante gerado é executada a simulação até que aconteça uma das situações de final de simulação, que podem ser quatro. As situações são as seguintes:

1. O *Checker* acusa um erro, o que quer dizer que o conjunto de casos de teste consegue diferir o mutante do programa original. Este mutante é, então, marcado como morto.

2. A simulação executa até o fim com 100% de cobertura e sem nenhum erro. Neste caso, o mutante é marcado como vivo e deve ser avaliado manualmente para determinar se ele é equivalente ao programa original. Caso a resposta seja positiva, ele é um mutante irrelevante e não entra na equação da métrica de qualidade descrita no Capítulo 2.

3. A simulação trava, a cobertura nunca é atingida e, portanto, isto configura um comportamento inesperado. Neste caso, o mutante é marcado como morto.

4. A simulação é abortada antes de atingir 100% pelo próprio mutante. Neste caso, ele também é marcado como morto.

Um mutante marcado como vivo pode representar um problema na cobertura. Pois, como aquele trecho de código não foi exercitado o suficiente para detectar o mutante, consiste em uma funcionalidade que pode conter uma falha.

Ao terminar a execução de um mutante, outro é escolhido até que não restem mais mutantes.

3.4 Discussão

Para a aplicação da análise de mutação na metodologia VeriSC, foram estabelecidas as pré-condições e os estágios da metodologia nos quais é possível aplicar a técnica. Para cada mutante gerado, é também estabelecida uma série de situações que a simulação de cada mutante pode atingir. A grande vantagem desta abordagem é a avaliação da cobertura do *testbench* mesmo antes da introdução do DUV. Característica herdada das próprias práticas

existentes na metodologia VeriSC. Uma desvantagem é que a simulação de mutantes que "travam" não é detectada automaticamente, forçando o engenheiro de verificação a monitorar manualmente os mutantes que tem esse comportamento.

Capítulo 4

Apresentação e Análise de Resultados

Neste capítulo, são apresentados os casos de estudo da aplicação da análise de mutação em verificação funcional. Foram escolhidos dois casos, implementados em ordem de complexidade. O primeiro deles é o DPCM (*Differential Pulse Code Modulator*). O segundo é um módulo que realiza a função de uma IDCT (*Inverse Discrete Cosine Transform*). A seguir são apresentados os detalhes dos experimentos.

4.1 Caso de estudo: DPCM - *Differential Pulse Code Modulator*

Uma das técnicas bastante utilizadas na área de compressão de imagem é a codificação preditiva, visto que visa eliminar a redundância interpixels presente na informação original, codificando somente a diferença ou resíduo entre o valor do pixel original e o valor predito para este pixel.

Como em uma imagem há redundância de informação entre os pixels vizinhos e, conseqüentemente, o valor de cada pixel pode ser predito por sua vizinhança. Nesse contexto, a Modulação por Código de Pulso Diferencial (Differential Pulse Code Modulation - DPCM) é o método que utiliza a soma do valor do pixel predito com o valor do resíduo para obter o valor do pixel original.

Quando o valor predito se aproxima do valor do pixel original, obtém-se um resíduo pequeno permitindo uma codificação, por meio de um código de comprimento variável, com

menos bits para o resíduo do que se a codificação fosse feita diretamente sobre o valor do pixel original (normalmente 8 bits), possibilitando, assim, o processo de compressão .

A DPCM pode ser modelada de forma bastante simples por apresentar apenas as seguintes funcionalidades [26]:

- Diferença: Realiza a diferença entre duas amostras de valor inteiro, a recebida no instante de tempo anterior e a recebida no instante de tempo atual. Para isso, é necessário armazenar a amostra anterior.
- Saturação: A amostra resultante da diferença deve estar dentro de uma faixa de valores pré-estabelecida. Caso o resultado da diferença seja maior do que o limite superior, a saída será o próprio limite superior. Caso o resultado da diferença seja menor do que o limite inferior, a saída será o próprio limite inferior.

A codificação preditiva, mais especificamente a DPCM, tem fundamental importância nos seguintes padrões de compressão para imagens: JPEG, JBIG e MPEG [4].

Apesar da simplicidade na forma de obtenção, observa-se, portanto, que a DPCM se constitui uma técnica bastante relevante para o contexto do Processamento Digital de Imagens.

4.1.1 Modelo de Referência

As subrotinas que implementam as funcionalidades do DPCM foram escritas na linguagem C e estão representadas no código fonte B.3, constituindo o modelo de referência. Para realizar a comunicação com o *testbench*, foi utilizado um módulo em SystemC que contém as estruturas de dados que se conectam com o *Source* e o *Checker*. Este módulo é representado no código fonte B.2.

4.1.2 Ambiente de Verificação

O ambiente de verificação foi escrito em SystemC e de acordo com a metodologia VeriSC até o passo chamado de *Testbench Conception* [10]. Neste passo, foram adicionados os parâmetros de cobertura (ver código fonte B.2, linhas 49 até 80), tornando este estágio da verificação suficiente para a aplicação da análise de mutação de acordo com os pré-requisitos

estabelecidos no Capítulo 3. Na Figura 4.1, vê-se a estrutura do *testbench* do DPCM com a indicação de onde foram aplicadas as mutações.

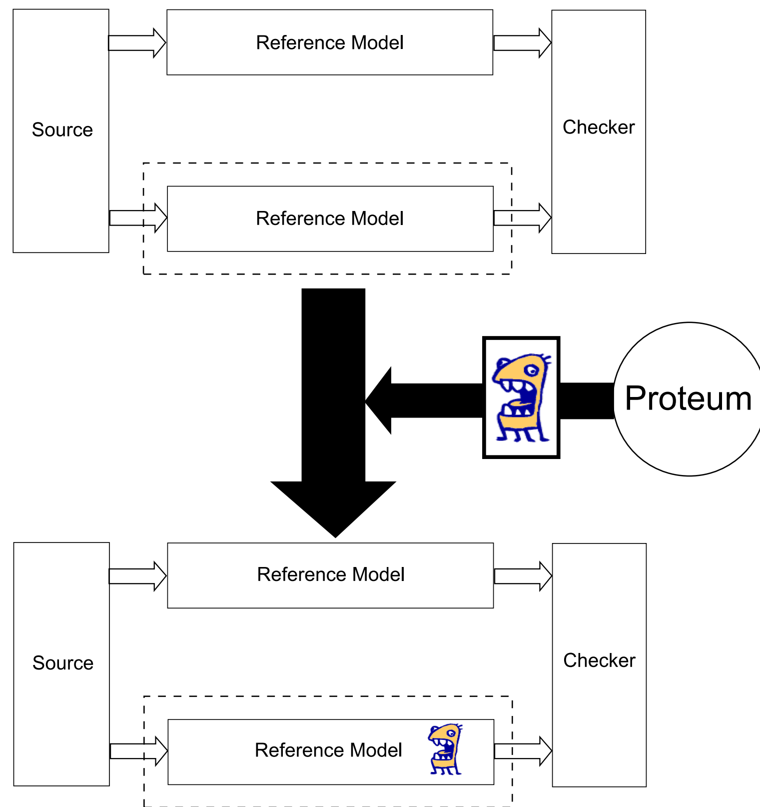

Figura 4.1: Aplicação das mutações sobre o modelo de referência do DPCM.

4.1.3 Preparação do Ambiente para Aplicação das Mutações

Como os modelos de referência utilizados são instâncias do mesmo código, para a aplicação das mutações utilizando a ferramenta Proteum torna-se necessário fazer algumas modificações no código fonte original do *testbench*. O código fonte completo do *testbench* original encontra-se no Apêndice B. O código fonte, com as devidas alterações encontra-se no Apêndice C, dentre as quais destacam-se:

- A criação de um arquivo dpcm_api_mut.c que é o arquivo que receberá as mutações;
- O *Checker* recebe apenas a instrução $sc_stop()$ na linha 28 para interromper a simulação assim que um erro for encontrado;

- O novo arquivo de ligações representado no Código Fonte C.4 agora precisa incluir o modelo de referência que foi alterado (linha 6), instanciá-lo (linha 20) e realizar as devidas ligações no *Source* e no *Checker* do *testbench* (linhas 41 e 48).

4.1.4 Mutações

Neste caso de estudo, existem duas unidades de trabalho, as subrotinas *int subtract(int a, int b)* e *int saturation(int a)* do modelo de referência, apresentadas no código fonte C.2, no Apêndice C. A seguir, os mutantes gerados para cada subrotina serão analisados separadamente.

Unidade 1 - subrotina *int subtract(int a, int b)*

Na Tabela 4.1, são apresentados os resultados da análise de mutação utilizando todos os operadores possíveis para esta unidade. Com a cobertura estabelecida, apenas o mutante número 27 ficou vivo. Os trechos de código 4.1 e 4.2 representam, respectivamente, a subrotina original e a subrotina com o mutante que ficou vivo. Verifica-se, portanto, que este mutante é equivalente ao programa original. Porque isso acontece? O operador SRSR (*Return Replacement*) verifica quantas instruções de *return* existem na unidade e gera mutantes trocando cada linha de código por uma operação de *return* existente no código [1]. Como nesta subrotina existe apenas uma linha de código, o único mutante gerado é o próprio programa original. Descobrir que este programa é equivalente ao original não demorou mais do que alguns poucos minutos, portanto teve impacto pequeno no tempo total da análise de mutação.

Código Fonte 4.1: Código fonte original da subrotina de subtração do modelo de referência do DPCM

```
1 int subtract_mut( int a, int b ) {
2         return (a - b);
3 }
```

Código Fonte 4.2: Código fonte da subrotina de subtração do modelo de referência do DPCM com mutação gerada pela ferramenta Proteum, operador SRSR

```
1 int subtract_mut( int a, int b ) {return (a - b) ;}
```

Tabela 4.1: Resultados da análise de mutação sobre a unidade 1.

Operadores	Total de Mutantes	Mutantes Vivos	Mutantes equivalentes
CRCR	10	0	0
OAAN	4	0	0
OABN	3	0	0
OALN	2	0	0
OARN	6	0	0
OASN	2	0	0
SRDL	2	0	0
SRSR	1	1	1
STRP	2	0	0
VDTR	6	0	0
VGSR	2	0	0
VTWD	4	0	0
TOTAL	44	1	1

Desta forma, o escore de mutação EM apresentado para esta unidade é 1 (um), o melhor escore possível para uma análise de mutação.

Unidade 2 - subrotina *int saturation(int a)*

Em uma primeira análise de mutação sobre esta unidade, verificou-se que *todos* os mutantes estavam vivos. Este fato pareceu inesperado e não tinha motivo aparente. Porém, verificou-se que a análise de mutação encontrou um problema que foge do escopo da metodologia VeriSC: a presença de faltas no modelo de referência. A seguir, uma descrição do problema encontrado.

No código fonte 4.3, tem-se o trecho de código antigo que contém a falta. Nas linhas 12 a 14, é possível observar que o valor inserido na transação é o valor da variável $diff$, quando o valor correto a ser enviado deveria ser o valor da variável sat, que contém o resultado da saturação.

Código Fonte 4.3: Trecho do código fonte do modelo de referência do módulo DPCM que continha uma falta

```
1   // ####### Reference Model Main Functions #######
```

```
2
3   int diff = subtract( audio_entrada_ptr->amostra, buff );
4   int sat = saturation( diff );
5
6   // ####### Reference Model Main Functions #######
7
8   buff = audio_entrada_ptr->amostra;
9
10  audio_saida_ptr = new sat_audio();
11
12  audio_saida_ptr->amostra = diff;
13
14  audio_saida_stim.write(audio_saida_ptr);
```

O código fonte 4.4 contém o trecho de código onde a falta foi corrigida.

Código Fonte 4.4: Trecho do código fonte do modelo de referência do módulo DPCM onde a falta foi corrigida

```
1   // ####### Reference Model Main Functions #######
2
3   int diff = subtract( audio_entrada_ptr->amostra, buff );
4   int sat = saturation( diff );
5
6   // ####### Reference Model Main Functions #######
7
8   buff = audio_entrada_ptr->amostra;
9
10  audio_saida_ptr = new sat_audio();
11
12  audio_saida_ptr->amostra = sat;
13
14  audio_saida_stim.write(audio_saida_ptr);
```

Neste caso, o que ocorreu foi que o engenheiro de verificação não testou uma situação: a de verificar se a saída em algum tempo foi maior do que o limite superior ou menor do que o limite inferior. Caso esta situação tivesse sido contemplada como um parâmetro de cobertura a ser atingido, este erro teria sido descoberto antes da análise de mutação. O trecho de código

4.5 é uma representação da cobertura que descobriria a falta identificada com a análise de mutação.

Código Fonte 4.5: Trecho de código que representa a adição dos parâmetros de cobertura que identificariam a falta encontrada com a análise de mutação.

```
1 Cv_bucket_illegal_refmod_audio.begin();
2         BVE_COVER_ILLEGAL(Cv_bucket_illegal_refmod_audio, diff < -4);
3         BVE_COVER_ILLEGAL(Cv_bucket_illegal_refmod_audio, diff >  3);
4 Cv_bucket_illegal_refmod_audio.end();
```

Na Tabela 4.2, é apresentado um resumo dos resultados da análise de mutação sobre esta unidade após a descoberta da falta. Esta nova situação contém apenas quatro mutantes vivos (2,64% do total de mutantes), todos eles equivalentes ao programa original. Esta análise dos mutantes e a descoberta da equivalência entre eles e o programa original também levou alguns poucos minutos e teve pequeno impacto no tempo total da análise de mutação. O escore de mutação para este caso é 1 (um).

Tabela 4.2: Resultados da análise de mutação sobre a unidade 2.

Operadores	Total de Mutantes	Mutantes Vivos	Mutantes equivalentes
CCCR	4	1	1
CCSR	6	0	0
CRCR	15	0	0
OCNG	2	0	0
ORAN	10	0	0
ORBN	6	0	0
ORLN	4	0	0
ORRN	10	2	2
ORSN	4	0	0
SRDL	6	0	0
SRSR	15	0	0
STRI	4	0	0
STRP	6	0	0
VDTR	9	0	0
VTWD	6	1	1
TOTAL	107	4	4

4.2 Caso de estudo: IDCT - *Inverse Discrete Cosine Transform*

Codificação por transformada é um dos principais módulos da maioria dos padrões de compactação de vídeo. Este caso de estudo concentra-se na transformação conhecida como DCT (*Discrete Cosine Transform*), mais especificamente em sua inversa (IDCT) utilizada na decodificação de vídeo. A transformada direta é utilizada na codificação. No padrão MPEG4, são utilizados blocos 8 x 8 na amostragem da imagem para realizar a codificação em coeficientes DCT. Na decodificação, este conjunto de coeficientes reconstruídos é utilizado para reconstruir a imagem [28].

A equação de reconstrução de amostras de imagem a partir de coeficientes DCT é dada pela equação 4.1 [28]:

$$f_{i,j} = \sum_{x=0}^{7} \sum_{y=0}^{7} \frac{C(x)C(y)}{4} F_{x,y} \cos\left(\frac{(2i+1)\,x\pi}{16}\right) \cos\left(\frac{(2j+1)\,x\pi}{16}\right) \tag{4.1}$$

A DCT é a transformada mais utilizada nos padrões de codificação de imagem, dentre os quais: JPEG, H.261, H.263, H.263+, H.264, MPEG-l, MPEG-2 e MPEG-4 [28].

A implementação de um artefato de *hardware*, que faz a função da IDCT, foi realizada durante o desenvolvimento do IP *core* MPEG4 citado no Capítulo 2 deste trabalho. Esse caso de estudo utilizará o ambiente de verificação, bem como o modelo de referência e o DUV deste módulo do decodificador de vídeo MPEG4 para a análise de mutação. A idéia é descobrir se a cobertura projetada para este bloco é satisfatória, de acordo com o critério estabelecido no Capítulo 2, Seção 2.2 deste trabalho.

4.2.1 Modelo de Referência

O modelo de referência faz uso do decodificador de vídeo xvid versão 0.9 patch-20 [36]. O modelo é uma implementação na linguagem C do padrão de decodificação de vídeo MPEG4 que tem os computadores pessoais como plataforma alvo. Não foi realizada nenhuma modificação no modelo de referência para a aplicação das mutações com a ferramenta Proteum. O código da subrotina que implementa a funcionalidade da IDCT pode ser visualizada no Anexo D.

4.2.2 Ambiente de Verificação

O ambiente de verficação está completo, contendo todos os elementos implementados (*Source*, *TDrivers*, *TMonitors* e *Checker*). Na Figura 4.2, é apresentado o *testbench* do módulo IDCT com uma indicação de onde foram inseridas as mutações.

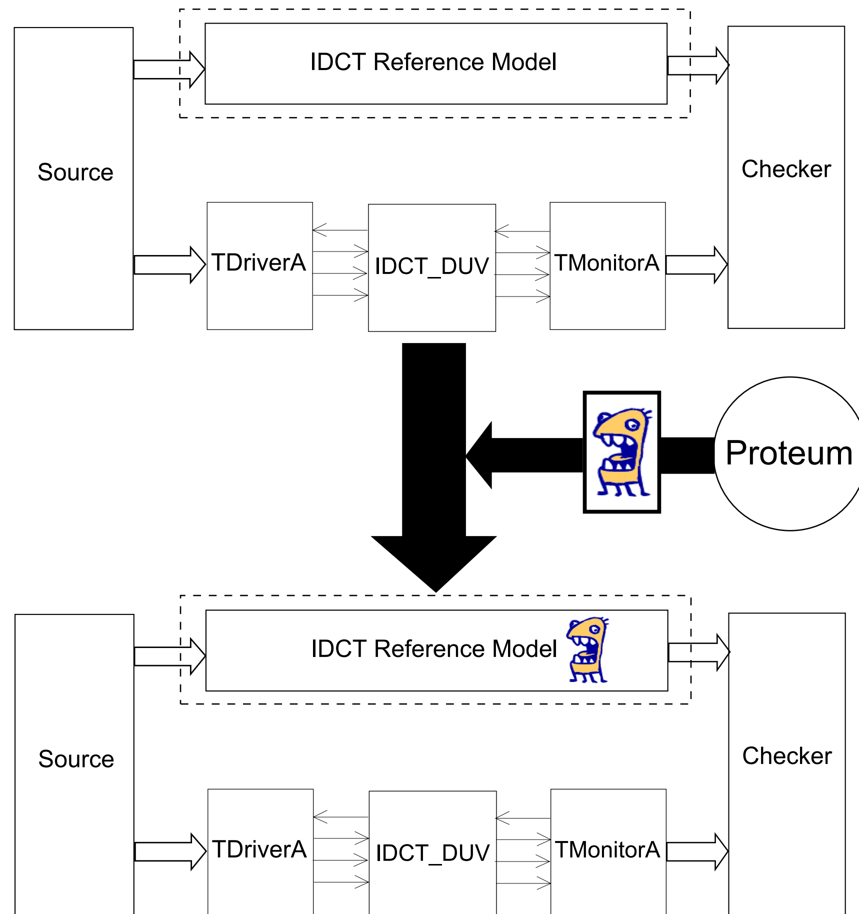

Figura 4.2: Aplicação das mutações sobre o modelo de referência do módulo IDCT.

4.2.3 Mutações

Na simulação das primeiras mutações, foi identificado que a análise de mutação iria consumir um tempo demasiado. Isto porque a simulação do *testbench* do módulo IDCT consome cerca de 32,48 min no computador descrito no Anexo A. Foram gerados ao todo 13.683 mutantes. Obviamente, existia a possibilidade de muitos deles serem “mortos” e não necessitarem dos 32,48 min de simulação para a “morte”. Por outro lado, tomando como referência o caso de estudo anterior no qual 2,64% dos mutantes ficaram vivos, poder-se-ia esperar cerca de 361

mutantes vivos (2,64% de 13.683). Assim, 361 multiplicado por 32,48 min tem como resultado 11725,28 min, ou 195,42 horas, ou ainda 8,14 dias. Aliado a este fato, há mutantes que "travam" a simulação, forçando o engenheiro de verificação a "ficar de olho" nas simulações para prevenir que um mutante desta natureza atrase ainda mais o trabalho.

Diante do exposto, foi necessário buscar uma forma de reduzir o custo computacional dessas simulações. Verificou-se, que dois dos parâmetros de cobertura demoravam cerca de 30,98 min para serem atingidos, enquanto o restante dos parâmetros era atingido em 1,5 min. Estes parâmetros excessivamente onerosos podem ser visualizados nas linhas 24 e 49 do código fonte **??**. Assim, em uma primeira rodada de simulação dos mutantes, esta cobertura não foi utilizada, o que reduziu para 10,25 horas o tempo estimado das simulações.

Foram observados quatro motivos para a sobrevivência dos mutantes ao realizar a análise de mutação:

1. *Código não exercitado*: O mutante está vivo porque a mutação inserida está em um trecho de código que nunca é exercitado pelo ambiente de verificação. Pode-se dizer, então, que a cobertura de código estabelecida poderia ser melhorada para contemplar o exercício dos trechos de código que nunca são exercitados.

2. *Equivalência*: O mutante é equivalente ao programa original e não há verificação funcional que consiga distingui-los.

3. *Indefinido*: O *testbench* não exercita o modelo de referência mutado de forma que ele produza uma saída diferente da saída do modelo de referência original. Também não foi encontrada uma maneira de mostrar se este conjunto de mutantes é equivalente ao programa original.

4. *Arquitetura*: A arquitetura do computador onde está sendo realizada a simulação interfere na morte/vida do mutante. Este fato foi confirmado por uma série de mutantes que realizam operações de deslocamento para a direita de um número de bits maior do que 31. A seguir, um exemplo que descreve melhor o problema encontrado.

   O código fonte 4.6 apresenta a versão original do trecho no qual será aplicada a mutação. O código fonte 4.7 apresenta o trecho de código onde foi realizada uma mutação

de acordo com a regra estabelecida pelo operador Cccr (*Constant for constant replacement*).

Código Fonte 4.6: Trecho do código fonte do modelo de referência do IDCT sem aplicação de mutação

```
1  .
2  .
3  .
4  X2 = (181 * (X4 + X5) + 128) >> 8;
5  .
6  .
7  .
```

Código Fonte 4.7: Trecho do código fonte do modelo de referência do IDCT com mutante na linha 4

```
1  .
2  .
3  .
4  X2 = (181 * (X4 + X5) + 128) >> 2408;
5  .
6  .
7  .
```

O *testbench* não é capaz de diferenciar os códigos 4.6 e 4.7. Assim, foi levada a efeito a depuração deste mutante para identificar os motivos que fazem com que ele fique vivo. Depois de analisar as mensagens exibidas durante a compilação, descobriu-se que o compilador emitiu a seguinte mensagem de aviso:

teste_2408.c: In function ‘main’:

teste_2408.c:4: warning: right shift count >= width of type

Ou seja, o compilador percebeu que o número de bits a serem deslocados para a direita é maior do que o tamanho do tipo de dado que está sendo deslocado. Buscando entender melhor o que acontece nesses casos, foram criados dois programas simples que realizam operações de deslocamento semelhantes às encontradas nos códigos fonte 4.6 e 4.7. São eles, os programas descritos nos códigos fonte 4.8 e 4.9.

Código Fonte 4.8: Código fonte do exemplo de deslocamento de 8 bits para a direita.

```
1 int main (int argc, char *argv[])
2 {
3         int a;
4         a = a >> 8;
5         return 0;
6 }
```

Código Fonte 4.9: Código fonte do exemplo de deslocamento de 2408 bits para a direita.

```
1 int main (int argc, char *argv[])
2 {
3         int a;
4         a = a >> 2408;
5         return 0;
6 }
```

Verificou-se que a compilação do código fonte 4.9 apresentou a mesma mensagem de aviso anteriormente citada, que mostra que o número de bits a serem deslocados para a direita é maior do que o tamanho do tipo de dado que está sendo deslocado. Analisou-se, então, o código na linguagem Assembly gerada pelo compilador gcc versão 3.3.2 [13]. Os códigos fonte em assembly estão representados nos códigos fonte 4.10 e 4.11, respectivamente.

Código Fonte 4.10: Código fonte na linguagem Assembly do exemplo de deslocamento de 8 bits para a direita.

```
1         .file   "teste_8.c"
2         .text
3 .globl main
4         .type   main, @function
5 main:
6         pushl   %ebp
7         movl    %esp, %ebp
8         subl    $8, %esp
9         andl    $-16, %esp
10        movl    $0, %eax
11        subl    %eax, %esp
```

```
12          leal    -4(%ebp), %eax
13          sarl    $8, (%eax)
14          movl    $0, %eax
15          leave
16          ret
17          .size   main, .-main
18          .ident  "GCC: (GNU) 3.3.2"
```

Código Fonte 4.11: Código fonte na linguagem Assembly do exemplo de deslocamento de 2408 bits para a direita.

```
1           .file   "teste_2408.c"
2           .text
3   .globl main
4           .type   main, @function
5   main:
6           pushl   %ebp
7           movl    %esp, %ebp
8           subl    $8, %esp
9           andl    $-16, %esp
10          movl    $0, %eax
11          subl    %eax, %esp
12          leal    -4(%ebp), %eax
13          movb    $104, %cl
14          sarl    %cl, (%eax)
15          movl    $0, %eax
16          leave
17          ret
18          .size   main, .-main
19          .ident  "GCC: (GNU) 3.3.2"
```

Observando a linha 13, vê-se que a operação de deslocamento de 8 bits para a direta na linguagem C é traduzida para a instrução *sarl*, que tem 2 operandos: uma constante e um registrador. A constante é o número 8 que é o número de bits a ser deslocado. O problema começa a aparecer quando são observadas as linhas 13 e 14 do código fonte 4.11. Na linha 14, a instrução *sarl* tem dois operandos, ambos registradores, sendo o primeiro, o registrador %cl, o que determina a quantidade de bits a serem deslocados.

Na linha 13, este registrador é carregado com o valor 104. Este valor deveria ser 2408, mas o compilador só considera para esta operação os 9 bits menos significativos do operando, e é por isso que ele emite o aviso anteriormente mencionado no momento da compilação. Mesmo com esta descoberta, não se pode justificar a sobrevivência do mutante, pois 8 é diferente de 104 e esse deslocamento deveria ter resultado diferente.

Foi então que partiu-se para uma análise da instrução *sarl* da arquitetura do Pentium®[14] e verificou-se que em uma operação de deslocamento para a direita de um número de bits maior do que 31, apenas os 5 bits menos significativos são utilizados para realizar de fato o deslocamento. Portanto, neste caso os 5 bits menos significativos do número introduzido pelo mutante tem o mesmo valor do programa original. Estes mutantes são considerados equivalentes ao programa original apenas para esta arquitetura. Não foi encontrado nenhum registro do impacto da arquitetura de um processador na análise de mutação. Portanto, é importante que o engenheiro de verificação que conduz a análise de mutação conheça não somente a linguagem de desenvolvimento do modelo de referência, mas também a arquitetura do processador onde a simulação ocorre.

A análise deste único mutante durou 12 horas distribuídas em 3 dias. Verificou-se, também, que outros 12 mutantes são equivalentes ao programa original pelo mesmo motivo.

Os resultados referentes aos mutantes vivos na primeira rodada estão dispostos na Tabela 4.3.

Tabela 4.3: Resultados da análise de mutação sobre a IDCT na primeira rodada de análise de mutação.

Classificação de Mutantes Vivos	Quantidade	Tempo de trabalho manual
Código não exercitado	1165	< 9,7 horas (0,5 min por mutante)
Arquitetura	13	12 horas
Equivalência	56	0,93 hora (1 min por mutante)
Indefinido	420	5,33 horas (1 min por mutante)
Total	1654	29,63

Dos 69 mutantes equivalentes ao programa original, 13 são devido a arquitetura do

Pentium®, como citado anteriormente. Os outros 56 mutantes foram identificados em poucos minutos e tiveram impacto mínimo no tempo da análise de mutação.

Foi necessário, também, avaliar os 420 que não foram inicialmente identificados como equivalentes ao programa original. Um destes mutantes é o representado no trecho de código no Código Fonte 4.12. O trecho do programa original associado a este mutante está representado no Código Fonte 4.13.

Código Fonte 4.12: Trecho do Código Fonte de um mutante vivo não-equivalente ao programa original.

```
1  .
2  .
3  .
4    X0 = (blk[8 * 0] << 8) + 8192;
5
6
7    (X8 = ((565 * (X4 + X5)) + 3)) ;
8
9    X4 = (X8 + (2841 - 565) * X4) >> 3;
10   X5 = (X8 - (2841 + 565) * X5) >> 3;
11 .
12 .
13 .
```

Código Fonte 4.13: Trecho do Código Fonte do programa original onde foi aplicada à mutação.

```
1  .
2  .
3  .
4    X0 = (blk[8 * 0] << 8) + 8192;
5
6
7    X8 = ((565 * (X4 + X5)) + 4) ;
8
9    X4 = (X8 + (2841 - 565) * X4) >> 3;
10   X5 = (X8 - (2841 + 565) * X5) >> 3;
11 .
```

```
12 .
13 .
```

Inicialmente, os códigos parecem iguais, mas é justamente esse o problema. A diferença entre eles (localizada na linha 7) é tão sutil que, com os parâmetros de cobertura escolhidos, a diferença se perde em operações de deslocamento localizadas mais ao final da subrotina. Estas operações de deslocamento podem ser visualizadas nas linhas 136 a 146 no Código Fonte D.1, do Anexo D. Os outros 419 mutantes vivos não-equivalentes ao programa original apresentam o mesmo comportamento.

De posse dessas informações, pode-se, então, calcular o escore de mutação (EM) da seguinte forma: $EM = \frac{12029}{13614} = 0,883$. Vale lembrar, que os mutantes equivalentes ao programa original não são considerados para o cálculo do escore de mutação.

Após essa análise preliminar, foi então adicionado o parâmetro de cobertura que tinha sido retirado anteriormente e aplicado sobre os mutantes que ficaram vivos. O resultado é apresentado na Tabela 4.4.

Tabela 4.4: Resultados da análise de mutação sobre a IDCT na segunda rodada de análise de mutação.

Classificação de Mutantes Vivos	Quantidade	Tempo de trabalho manual (horas)
Código não exercitado	1165	9,7 (0,5 min por mutante)
Arquitetura	13	12
Equivalência	56	0,93 (1 min por mutante)
Indefinido	320	5,33 (1 min por mutante)
Total	1554	27,96

Foram simulados novamente os 1485 mutantes vivos não-equivalentes ao programa original, cada um com duração de 32,48 min. Assim, a simulação de todos os mutantes demorou 48232,8 min, ou 803,88 horas, ou 13,39 dias. Como as simulações são independentes, poder-se-ia utilizar várias máquinas para rodar várias simulações em paralelo, inclusive com o uso de *grids* computacionais.

O novo escore de mutação é, então, recalculado considerando a diminuição dos mutantes que foram mortos com a cobertura nova: $EM = \frac{12129}{13614} = 0,890$. Observa-se, então, que quando o parâmetro de cobertura foi adicionado, mais mutantes foram mortos aumentando o escore de mutação. Há espaço para melhorar ainda mais a cobertura, já que restaram ainda

1485 mutantes vivos não equivalentes ao programa original. Esta tarefa de criar parâmetros de cobertura e estímulos que matem mais mutantes é chamada de teste de mutação e não está no escopo deste trabalho.

É importante observar, que a análise de mutação pode apresentar mutantes que são "difíceis" de matar, ou de se determinar sua equivalência em relação ao programa original. O gráfico da Figura 4.3 representa o tempo de trabalho manual necessário para classificar um conjunto de mutantes, possibilitando uma outra visão da Tabela 4.4.

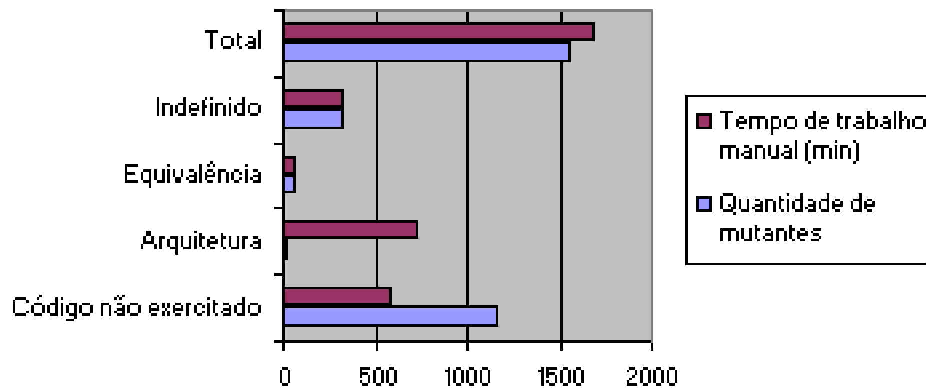

Figura 4.3: Gráfico da relação entre quantidade de mutantes e tempo de trabalho manual para classificação.

É possível observar na figura 4.3 e na tabela 4.4, que apenas 13 dos 1554 mutante levaram 12 horas para serem classificados, o que representa 42% do tempo de trabalho manual para classificação dos mutantes vivos. Portanto, ao planejar a análise mutação dentro do contexto de um projeto de IP *core*, este tipo de dado deve ser levado em consideração. Podem surgir mutantes "difíceis" de serem analisados, que consomem muito tempo e têm potencial de atrasar o andamento da verificação funcional. Portanto, ao planejar o cronograma, o gerente de projeto deve prever que esta situação é possível de acontecer e tomar a decisão de utilizar, ou não, a análise de mutação, respeitando os prazos estabelecidos para o término do projeto.

4.3 Discussão

Dois casos de estudo foram adotados para a validação do trabalho. No primeiro caso (mais simples), o ambiente de verificação foi construído a partir do início. Apesar da baixa complexidade do módulo verificado, foi possível encontrar problemas com a ajuda da análise de

mutação. No final, o escore de mutação foi igual a 1 (um), o que quer dizer que a cobertura estabelecida teve grau máximo de qualidade, de acordo com a métrica estabelecida. O segundo (e mais complexo) caso de estudo, foi a IDCT. Em uma primeira rodada de simulação, verificou-se que o tempo de simulação de cada mutante era demasiado alto. Portanto, foram feitas alterações na cobertura já estabelecida para diminuir este tempo. Com esta cobertura limitada, foram mortos 12029 (doze mil e vinte e nove) do total de mutantes não equivalentes ao programa original. Ao reintroduzir os parâmetros que foram retirados, foram mortos mais 100 (cem) mutantes, mostrando que uma cobertura mais adequada, mata mais mutantes. A partir dos dados avaliados durante a análise de mutação, observou-se que o tempo de trabalho manual necessário para a realização da tarefa de classificação é um fator de risco no planejamento do cronograma do projeto.

Capítulo 5

Considerações Finais e Sugestões para Trabalhos Futuros

5.1 Considerações Finais

Este trabalho visa fornecer um parâmetro objetivo ao engenheiro de verificação que auxilie na análise da qualidade da sua verificação funcional. A idéia principal consiste em utilizar a análise de mutação na metodologia de verificação funcional VeriSC.

A análise de mutação realizada sobre os casos de estudo apresentados no Capítulo 4 mostram que o escore de mutação é uma boa medida da qualidade dos parâmetros de cobertura estabelecidos para a verificação funcional. No exemplo da DPCM, é possível ver que mesmo para um projeto pouco complexo, a análise de mutação foi capaz de revelar problemas que poderiam afetar o funcionamento do sistema em um ambiente real. Já no exemplo da IDCT, um módulo de um sistema em que se considera que foi feita uma boa verificação funcional, já que este projeto é reconhecido inclusive internacionalmente, 11% dos mutantes (1485 mutantes) são vivos e não equivalentes ao programa original. Cada um deles está vivo porque alguma funcionalidade não foi exercitada o suficiente para matá-lo. Cada funcionalidade mal exercitada pode conter um erro de programação que afeta a qualidade final do produto, no caso o módulo IDCT.

Apesar dos benefícios provenientes do uso da análise de mutação na verificação funcional, o tempo de simulação dos mutantes e o trabalho manual envolvido são fatores de grande impacto para o projeto, e devem ser considerados ao ser aplicada a análise de mu-

tação. A partir do uso da análise de mutação, foi possível constatar a importância do conhecimento da arquitetura do processador em uso, por parte do engenheiro de verificação, para que a técnica seja aplicada com sucesso. No contexto desse trabalho, a análise de mutação poderia ter sido realizada em outros tipos de arquitetura, o que permitiria avaliar de forma ainda mais efetiva o impacto de diferentes arquiteturas de hardware.

5.2 Sugestões para Trabalhos Futuros

Os resultados obtidos no trabalho indicam a eficácia do uso de operadores de mutação como técnica auxiliar para melhoria da verificação funcional. Entretanto, foram identificadas algumas limitações dos resultados que vislumbram possibilidades para trabalhos futuros, destacando-se:

- Desenvolvimento e/ou adaptação de técnicas de teste de mutação para utilização na metodologia VeriSC buscando melhorar o escore de mutação dos ambientes de verificação.
- Aplicação de técnicas como a mutação seletiva ou a mutação fraca para a diminuição do tempo da análise de mutação.
- Aplicação de análise de mutação em computadores com arquitetura diferente da utilizada neste trabalho, com o objetivo de avaliar o impacto deste parâmetro sobre a análise.

Bibliografia

[1] Hiralal Agrawal, Richard A. DeMillo, Bob Hathaway, William Hsu, Wynne Hsu, E.W. Krauser, R. J. Martin, Aditya P. Mathur, and Eugene Spafford. Design of mutant operators for the c programming language. Technical report, Department of Computer Science - Purdue University, 2006.

[2] C. Michael Pilato Ben Collins-Sussman, Brian W. Fitzpatrick. *Version Control with Subversion For Subversion 1.4*. TBA, 2007.

[3] J. Bergeron. *Functional verification of hdl models*. Kluwer Academic Publisher, 2003.

[4] V. Bhaskaran and K. Konstantinides. *Image and Video Compression Standards Algorithmos and Architectures*. Kluwer Academic Pub., 1995.

[5] 2008. http://www.brazilip.org.br/ - Acessado em Novembro de 2008.

[6] 2008. http://www.cadence.com - Acessado em Novembro de 2008.

[7] 2008. http://www.centos.org - Acessado em Novembro de 2008.

[8] http://www.bergbrandt.com.br/cetene/asp/sobre_ocetene.asp - Acessado em Novembro de 2008.

[9] 2007. http://savannah.nongnu.org/projects/cvs/ - Acessado em Novembro de 2008.

[10] K. R. G. da Silva, E. U. K. Melcher, I. Maia, and H. do N. Cunha. A methodology aimed at better integration of functional verification and rtl design. *Design Automation for Embedded Systems*, 2006.

[11] Márcio Eduardo Delamaro and José Carlos Maldonado. Proteum - a tool for the assessment of test adequacy for c programs - user's guide. Technical report, University of São Paulo (USP), 1996.

[12] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 1978.

[13] 2008. http://gcc.gnu.org/ - Acessado em Novembro de 2008.

[14] Intel. *Pentium Processor Family Developer's Manual - Volume 3: Architecture and Programming Manual*, 1995.

[15] 2007. http://www.us.design-reuse.com/ipsoc2006/ - Acessado em Novembro de 2008.

[16] Luciano Lavagno, Grant Martin, and Louis Scheffer. *Electronic Design Automation for Integrated Circuits Handbook - 2 Volume Set*. CRC Press, Inc., Boca Raton, FL, USA, 2006.

[17] A.P. Mathur. Reducing the cost of mutation testing: An empirical study. *Journal of Systems and Software archive*, 1993.

[18] 2008. http://www.mentor.com - Acessado em Novembro de 2008.

[19] A. J. Offutt. Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering Methodology*, 1992.

[20] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. An experimental determination of sufficient mutant operators. *ACM Trans. Softw. Eng. Methodol.*, 1996.

[21] A. J. Offutt and S. D. Lee. How strong is weak mutation? In *In Proceedings of the symposium on Testing, analysis, and verification*, 1991.

[22] A. J. Offutt and S. D. Lee. An empirical evaluation of weak mutation. *IEEE Trans. Softw. Eng.*, 1994.

[23] A. J. Offutt and R. H. Untch. Mutation 2000: Uniting the orthogonal. In *Mutation Testing in the Twentieth and the Twenty First Centuries*, 2000.

[24] Isaac Maia Pessoa. Dissertação de mestrado: Geração semi-automática de testbenches para circuitos integrados digitais, 2007.

[25] A. Piziali. *Functional verification Coverage Measurement and Analysis*. Kluwer Academic Publisher, 2004.

[26] Ken C. Pohlmann. *Principles of Digital Audio, 2nd ed.* Sams/Prentice-Hall Computer Publishing, Carmel, Indiana., 1985.

[27] 2008. http://www.redhat.com - Acessado em Novembro de 2008.

[28] Iain E. G. Richardson. *Video Codec Design - Devloping Image and Video Compression Systems*. JOHN WILEY & SONS, LTD, 2002.

[29] A. K. Rocha, P. Lira, Y. Y. Ju, E. Barros, and E. Melcher. Silicon validated ip cores designed by the brazil-ip network. In *Proceedings of IP/SOC 2006*, 2006.

[30] Youssef Serrestou and Vincent Beroulle Chantal Robach. Functional verification of rtl designs driven by mutation testing metrics. In *10th IEEE Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD 2007)*, 2007.

[31] 2008. http://www.synopsys.com - Acessado em Novembro de 2008.

[32] 2008. http://www.systemc.org - Acessado em Novembro de 2008.

[33] P. Vado, Yvon Savaria, Yannick Zoccarato, and Chantal Robach. A methodology for validating digital circuits with mutation testing. In *IEEE International Symposium on Circuits and Systems*, 2000.

[34] K. S. H. T. Wah. Fault coupling in finite bijective functions. *The Journal of Software Testing, Verification and Reliability, vol 5, pp 3-47*, 1995.

[35] K. S. H. T. Wah. A theoretical study of fault coupling. *The Journal of Software Testing, Verification and Reliability, vol 5, pp 3-47*, 2000.

[36] 2008. http://www.xvid.org - Acessado em Novembro de 2008.

Apêndice A

Ambiente e Ferramentas

Para o desenvolvimento dos experimentos relacionados a este trabalho, foram necessárias várias ferramentas computacionais, descritas a seguir.

A.1 Ambiente de Desenvolvimento

O computador onde foram realizadas as simulações tem as seguintes características:

- Processador: AMD-Opteron 1.6 GHz
- Memória: 1.5GB DDR
- Dsico: Serial ATA 140GB

A.2 Sistema Operacional

O Sistema Operacional (S.O.) utilizado para os experimentos foi o CentOS 4 [7]. Este S.O. fornece uma plataforma da classe *enterprise* compatível com o RHEL 4 (*Red Hat Enterprise Linux* 4) [27]. O RHEL 4 é o S.O. adotado pelas maiores empresas que desenvolvem ferramentas para a indústria de microeletrônica. São elas, a CADENCE ®[6], a Synopsys ®[31] e a Mentor Graphics ®[18]. Desta forma, o procedimento de mutação apresentado neste trabalho deverá funcionar nesta plataforma para que os problemas de portabilidade sejam mínimos.

A.3 Linguagem de Programação

SystemC é uma biblioteca da linguagem C++ destinada à elaboração de ambientes de verificação e projeto de circuitos digitais. Esta linguagem é particularmente direcionada à implementação de blocos de *hardware* de alto desempenho com vários níveis de abstração. Pode-se dizer, portanto, que SystemC é uma linguagem unificada para projeto e verificação na forma de uma biblioteca *open-source* de classes C++ [32].

A.4 Ferramenta de Mutação

A Proteum[11] é uma ferramenta de suporte à teste de mutação. Essa ferramenta foi utilizada para a geração dos mutantes aplicados ao ambiente de verificação funcional. A Proteum foi escolhida por ter sido a única ferramenta encontrada que contém todos os operadores de mutação disponíveis para a linguagem C. Na ferramenta, um teste é guiado por seções de teste. Em cada seção, o testador pode criar um teste, interrompê-lo e retomá-lo mais tarde. As funcionalidades são separadas em diversas sub-ferramentas, que podem ser usadas separadamente para proporcionar máxima flexibilidade. A sub-ferramenta utilizada é chamada "exemuta", que gera os mutantes de acordo com os parâmetros fornecidos.

A.5 Ferramenta para geração semi-automática de *Testbenches*

A geração do *testbench* do módulo DPCM foi realizada utilizando a ferramenta eTBc, desenvolvida por Isaac Maia Pessoa [24], que gera automaticamente uma grande parte do ambiente de verificação funcional baseado na especificação do usuário e nos *templates* da metodologia VeriSC.

A.6 Controle de versões

Para o desenvolvimento do trabalho e desta dissertação, foi utilizada uma ferramenta de controle de versões conhecida como Subversion [2]. Essa ferramenta tem o objetivo de substituir

outro sistema de controle de versões, chamado CVS (*Concurrent Versioning System*) [9], que os desenvolvedores do SVN consideram ter muitas limitações. Dentre as funcionalidades mais importantes do SVN estão:

- Contém Todas as funcionalidades disponíveis para o CVS;
- A operação de *commit* é realmente atômica. Nenhuma parte da operação é executada sem que toda ela tenha sido. Os números de revisão são por operação de *commit* e não por arquivo. Além disso, a mensagem associada ao *commit* é anexada à revisão e não a cada arquivo relacionado com aquela revisão;
- Copiar, renomear e apagar são operações versionadas;
- As permissões de execução de cada arquivo são preservadas;
- Arquivos binários são tratados eficientemente, tanto quanto os arquivos de texto. Isso porque o SVN tem um algoritmo que resolve a diferença entre arquivos binários para transmitir e armazenar revisões;
- A resolução de conflitos é realizada de forma interativa.

Apêndice B

Código Fonte do *Testbench* Original do módulo DPCM

A seguir, o Código Fonte do *testbench* do DPCM divido em módulos de acordo com a divisão do *testbench*.

Código Fonte B.1: Código fonte do modelo gerador automático de estímulos do módulo DPCM

```
1
2
3
4 class audio_entrada_constraint_class: public scv_constraint_base {
5
6     scv_bag<pair<int,int> > amostra_distrib;
7
8  public:
9   scv_smart_ptr<audio> audio_sptr;
10   SCV_CONSTRAINT_CTOR(audio_entrada_constraint_class) {
11      amostra_distrib.push(pair <int,int>(-10, 10),1);
12      audio_sptr->amostra.set_mode(amostra_distrib);
13
14
15   // amostra_distrib.push(pair<int,int>(?,?), ?);
16   // audio_sptr->amostra.set_mode(amostra_distrib);
17
```

```
18    //    SCV_CONSTRAINT(audio_sptr->?() > ?);
19      }
20  };
21
22
23
24  SC_MODULE(source) {
25
26
27
28      sc_fifo_out<audio_ptr> audio_entrada_to_refmod;
29      sc_fifo_out<audio_ptr> audio_entrada_to_driver;
30      audio_entrada_constraint_class audio_entrada_constraint;
31
32
33    void audio_entrada_p(){
34       string type;
35       ifstream ifs_audio_entrada("audio_entrada.stim");
36       audio audio_entrada_stim;
37       //Static stimuli generation
38       while( !ifs_audio_entrada.fail() && !ifs_audio_entrada.eof() ) {
39         ifs_audio_entrada >> type;
40         if ( type == "audio" ) {
41           ifs_audio_entrada >> audio_entrada_stim;
42           audio_entrada_to_refmod.write(new audio(audio_entrada_stim ) );
43         }
44
45         ifs_audio_entrada.ignore(225, '\n');
46       }
47
48       //Random stimuli generantion
49       while(1) {
50         audio_entrada_constraint.next();
51         audio_entrada_stim =  audio_entrada_constraint.audio_sptr.read();
52         audio_entrada_to_refmod.write(new  audio(audio_entrada_stim ) );
53         audio_entrada_to_driver.write(new  audio(audio_entrada_stim ) );
54
```

```
55        }
56
57  }
58
59
60      SC_CTOR( source ):
61
62         audio_entrada_constraint("audio_entrada_constraint")
63
64      {
65           SC_THREAD(audio_entrada_p);
66
67      }
68  };
```

Código Fonte B.2: Código fonte do modelo de referência do módulo DPCM

```
 1  #include "dpcm_api.c"
 2
 3  SC_MODULE(refmod_dpcm) {
 4
 5
 6    sc_fifo_in <audio_ptr  > audio_entrada_stim;
 7
 8
 9    sc_fifo_out <sat_audio_ptr > audio_saida_stim;
10
11
12
13    audio_ptr audio_entrada_ptr;
14
15
16    sat_audio_ptr audio_saida_ptr;
17
18    int buff;
19
20  bve_cover_bucket Cv_bucket_refmod_audio;
21  bve_cover_bucket Cv_bucket_refmod_diff;
```

```
22 bve_cover_bucket Cv_bucket_refmod_sat;
23
24 void p() {
25
26  while (1) {
27
28   audio_entrada_ptr = audio_entrada_stim.read();
29
30   // ####### Reference Model Main Functions #######
31
32   int diff = subtract( audio_entrada_ptr->amostra, buff );
33   int sat = saturation( diff );
34
35   // ####### Reference Model Main Functions #######
36
37   buff = audio_entrada_ptr->amostra;
38
39   audio_saida_ptr = new sat_audio();
40
41   audio_saida_ptr->amostra = diff;
42
43   audio_saida_stim.write(audio_saida_ptr);
44
45   delete( audio_entrada_ptr );
46
47   // Coverage
48
49   Cv_bucket_refmod_audio.begin();
50   BVE_COVER_BUCKET(Cv_bucket_refmod_audio, buff == 0, 1000);
51   BVE_COVER_BUCKET(Cv_bucket_refmod_audio, buff == LIM_SUP, 1000);
52   BVE_COVER_BUCKET(Cv_bucket_refmod_audio, buff == LIM_INF, 1000);
53   BVE_COVER_BUCKET(Cv_bucket_refmod_audio, buff > LIM_INF, 1000);
54   BVE_COVER_BUCKET(Cv_bucket_refmod_audio, buff < LIM_SUP, 1000);
55   BVE_COVER_BUCKET(Cv_bucket_refmod_audio, buff > LIM_SUP, 1000);
56   BVE_COVER_BUCKET(Cv_bucket_refmod_audio, buff < LIM_INF, 1000);
57
58   Cv_bucket_refmod_audio.end();
```

```
59
60    Cv_bucket_refmod_diff.begin();
61
62    BVE_COVER_BUCKET(Cv_bucket_refmod_diff, diff > LIM_SUP, 1000);
63    BVE_COVER_BUCKET(Cv_bucket_refmod_diff, diff < LIM_INF, 1000);
64    BVE_COVER_BUCKET(Cv_bucket_refmod_diff, diff == LIM_SUP, 1000);
65    BVE_COVER_BUCKET(Cv_bucket_refmod_diff, diff == LIM_INF, 1000);
66    BVE_COVER_BUCKET(Cv_bucket_refmod_diff, diff > LIM_INF, 1000);
67    BVE_COVER_BUCKET(Cv_bucket_refmod_diff, diff < LIM_SUP, 1000);
68    BVE_COVER_BUCKET(Cv_bucket_refmod_diff, diff == 0, 1000);
69
70    Cv_bucket_refmod_diff.end();
71
72    Cv_bucket_refmod_sat.begin();
73
74    BVE_COVER_BUCKET(Cv_bucket_refmod_sat, sat == LIM_SUP, 1000);
75    BVE_COVER_BUCKET(Cv_bucket_refmod_sat, sat == LIM_INF, 1000);
76    BVE_COVER_BUCKET(Cv_bucket_refmod_sat, sat > LIM_INF, 1000);
77    BVE_COVER_BUCKET(Cv_bucket_refmod_sat, sat < LIM_SUP, 1000);
78    BVE_COVER_BUCKET(Cv_bucket_refmod_sat, sat == 0, 1000);
79
80    Cv_bucket_refmod_sat.end();
81
82
83   }
84 }
85
86
87    SC_CTOR(refmod_dpcm):
88
89       Cv_bucket_refmod_audio("Cv_bucket_refmod_audio"),
90       Cv_bucket_refmod_diff("Cv_bucket_refmod_diff"),
91       Cv_bucket_refmod_sat("Cv_bucket_refmod_sat")
92
93       {
94          buff = 0;
95          SC_THREAD(p);
```

```
96      }
97
98  };
```

Código Fonte B.3: Código fonte das subrotinas que executam as funcionalidades do DPCM

```
 1  #define LIM_SUP 3
 2  #define LIM_INF -4
 3
 4  int subtract( int a, int b ) {
 5    return a - b;
 6  }
 7
 8  int saturation( int unsat_data ) {
 9    if( unsat_data < LIM_INF ) return LIM_INF;
10    if( unsat_data > LIM_SUP ) return LIM_SUP;
11    return unsat_data;
12  }
13
14  #ifdef GEN_MUT
15  int main (int argc, char *argv[])
16  {
17    return 0;
18  }
19  #endif
```

Código Fonte B.4: Código fonte do *Checker* do módulo DPCM

```
 1
 2
 3  //Pre-Checker Template
 4
 5  SC_MODULE( checker )
 6  {
 7
 8
 9    sc_fifo_in<sat_audio_ptr> audio_saida_from_refmod;
10    sc_fifo_in<sat_audio_ptr> audio_saida_from_duv;
11
```

```
12
13
14
15  sc_signal<unsigned int> error_count_audio_saida;
16    void audio_saida_p() {
17
18      while(1) {
19        sat_audio_ptr  trans_refmod = audio_saida_from_refmod.read();
20        sat_audio_ptr  trans_duv = audio_saida_from_duv.read();
21
22        if ( !(*trans_refmod == *trans_duv) ) {
23          ostringstream ms;
24          ms << "expected: " << *trans_refmod << endl
25             << "received: " << *trans_duv << ends;
26          SCV_REPORT_ERROR("audio_saida_access",ms.str().c_str());
27          error_count_audio_saida= error_count_audio_saida.read()+1;
28          sc_stop();
29        }
30        delete ( trans_refmod );
31        delete ( trans_duv );
32      }
33    }
34
35
36
37 SC_CTOR( checker )
38 {
39       SC_THREAD(audio_saida_p);
40
41  }
42
43 };
```

Código Fonte B.5: Código fonte das estruturas de dados das transações do módulo DPCM

```
1 #ifndef STRUCTS_H
2 #define STRUCTS_H
3
```

```
4  #include <iomanip.h>
5  #include <iostream.h>
6  using namespace std;
7
8  // struct for audio
9  struct  audio {
10
11   int amostra;
12
13   inline bool operator== (const audio& arg) const {
14       bool result = true;
15       result &= amostra == arg.amostra;
16       return result;
17   }
18
19 };
20
21 typedef audio *audio_ptr;
22
23 // struct for sat_audio
24 struct  sat_audio {
25
26   int amostra;
27
28   inline bool operator== (const sat_audio& arg) const {
29       bool result = true;
30       result &= amostra == arg.amostra;
31       return result;
32    }
33
34 };
35
36 typedef sat_audio *sat_audio_ptr;
37
38 // struct for diff_audio
39 struct  diff_audio {
40
```

```
41   int diff_amostra;
42
43   inline bool operator== (const diff_audio& arg) const {
44       bool result = true;
45       result &= diff_amostra == arg.diff_amostra;
46       return result;
47   }
48
49 };
50
51 typedef diff_audio *diff_audio_ptr;
52
53 //********************* operators *********************
54
55 istream &operator >> (istream &is , audio &arg) {
56    is >> arg.amostra;
57    return is;
58 }
59
60 istream &operator >> (istream &is , sat_audio &arg) {
61    is >> arg.amostra;
62    return is;
63 }
64
65 istream &operator >> (istream &is , diff_audio &arg) {
66    is >> arg.diff_amostra;
67    return is;
68 }
69
70 inline ostream& operator << (ostream& os, const audio& arg){
71    os << "audio= (" ;
72    os << arg.amostra << " " ;
73    os << ")";
74    return os;
75 }
76
77 inline ostream& operator << (ostream& os, const sat_audio& arg){
```

```
78     os << "sat_audio= (" ;
79     os << arg.amostra << " "  ;
80     os << ")";
81     return os;
82  }
83
84  inline ostream& operator << (ostream& os, const diff_audio& arg){
85     os << "diff_audio= (" ;
86     os << arg.diff_amostra << " "  ;
87     os << ")";
88     return os;
89  }
90
91  #endif
```

Código Fonte B.6: Código fonte do módulo que conecta todos os elementos do *testbench*

```
 1  #include "structs_ext.h"
 2  #include "bve.h"
 3  #include "source.h"
 4  #include "checker.h"
 5  #include "refmod_dpcm.h"
 6  #include "refmod_dpcm_mut.h"
 7
 8  ofstream *logfile;
 9
10  int sc_main (int argc, char *argv[])
11  {
12     sc_set_time_resolution(1, SC_PS);
13     scv_tr_text_init();
14     scv_tr_db db("txdb.txt");
15     sc_trace_file *tf = sc_create_vcd_trace_file ("wave");
16     ((vcd_trace_file*)tf)->sc_set_vcd_time_unit(-12);
17     logfile = new ofstream("fifo.log");
18
19     refmod_dpcm refmod_1("refmod_1");
20     refmod_dpcm_mut refmod_mut("refmod_mut");
21     source  source_i("source_i");
```

```
22     checker checker_i("checker_i");
23
24     //Input Fifos
25
26     bve_fifo<audio_ptr > audio_entrada_refmod("audio_entrada_refmod",
           logfile);
27     bve_fifo<audio_ptr > audio_entrada_to_driver("audio_entrada_to_driver
           ", logfile);
28
29     //Output Fifos
30
31     bve_fifo<sat_audio_ptr > audio_saida_refmod("audio_saida_refmod" ,
           logfile );
32     bve_fifo<sat_audio_ptr > audio_saida_from_driver("
           audio_saida_from_driver" , logfile );
33
34
35
36     //FIFOs links of input interfaces
37
38     source_i.audio_entrada_to_refmod (audio_entrada_refmod);
39     refmod_1.audio_entrada_stim (audio_entrada_refmod);
40     source_i.audio_entrada_to_driver (audio_entrada_to_driver);
41     refmod_mut.audio_entrada_stim (audio_entrada_to_driver);
42
43
44     //FIFOs links of output interfaces
45
46     refmod_1.audio_saida_stim ( audio_saida_refmod);
47     checker_i.audio_saida_from_refmod (audio_saida_refmod);
48     refmod_mut.audio_saida_stim ( audio_saida_from_driver);
49     checker_i.audio_saida_from_duv (audio_saida_from_driver);
50
51
52     sc_start();
53     return 0;
54 };
```

Apêndice C

Código Fonte do *Testbench* alterado do módulo DPCM.

A seguir, o Código Fonte do *testbench* do DPCM alterado para que pudessem ser aplicadas as mutações.

O Código Fonte B.1 referente ao *Source* não muda, já que os estímulos utilizados são iguais aos originais.

São acrescentados dois novos códigos fonte para diferenciar o modelo de referência, que permanece original, daquele sobre o qual serão aplicadas as mutações. Os códigos fonte C.1 e C.2 representam estes novos módulos. Os originais permanecem inalterados.

Código Fonte C.1: Código fonte do modelo de referência do módulo DPCM modificado para a mutação.

```
1  #include "dpcm_api_mut.c"
2
3  SC_MODULE(refmod_dpcm_mut) {
4
5
6    sc_fifo_in <audio_ptr  > audio_entrada_stim;
7
8
9    sc_fifo_out <sat_audio_ptr > audio_saida_stim;
10
11
```

```
12
13     audio_ptr  audio_entrada_ptr;
14
15
16     sat_audio_ptr  audio_saida_ptr;
17
18     int  buff;
19
20  bve_cover_bucket  Cv_bucket_refmod_audio;
21  bve_cover_bucket  Cv_bucket_refmod_diff;
22  bve_cover_bucket  Cv_bucket_refmod_sat;
23
24  void  p()  {
25
26   while  (1)  {
27
28     audio_entrada_ptr  =  audio_entrada_stim.read();
29
30     //  #######  Reference  Model  Main  Functions  #######
31
32     int  diff  =  subtract_mut(  audio_entrada_ptr->amostra,  buff  );
33     int  sat  =  saturation_mut(  diff  );
34
35     //  #######  Reference  Model  Main  Functions  #######
36
37     buff  =  audio_entrada_ptr->amostra;
38
39     audio_saida_ptr  =  new  sat_audio();
40
41     audio_saida_ptr->amostra  =  sat;
42
43     audio_saida_stim.write(audio_saida_ptr);
44
45     delete(  audio_entrada_ptr  );
46
47     //  Coverage
48
```

```
49      Cv_bucket_refmod_audio.begin();
50      BVE_COVER_BUCKET(Cv_bucket_refmod_audio, buff == 0, 1000);
51      BVE_COVER_BUCKET(Cv_bucket_refmod_audio, buff == LIM_SUP, 1000);
52      BVE_COVER_BUCKET(Cv_bucket_refmod_audio, buff == LIM_INF, 1000);
53      BVE_COVER_BUCKET(Cv_bucket_refmod_audio, buff > LIM_INF, 1000);
54      BVE_COVER_BUCKET(Cv_bucket_refmod_audio, buff < LIM_SUP, 1000);
55      BVE_COVER_BUCKET(Cv_bucket_refmod_audio, buff > LIM_SUP, 1000);
56      BVE_COVER_BUCKET(Cv_bucket_refmod_audio, buff < LIM_INF, 1000);
57
58      Cv_bucket_refmod_audio.end();
59
60      Cv_bucket_refmod_diff.begin();
61
62      BVE_COVER_BUCKET(Cv_bucket_refmod_diff, diff > LIM_SUP, 1000);
63      BVE_COVER_BUCKET(Cv_bucket_refmod_diff, diff < LIM_INF, 1000);
64      BVE_COVER_BUCKET(Cv_bucket_refmod_diff, diff == LIM_SUP, 1000);
65      BVE_COVER_BUCKET(Cv_bucket_refmod_diff, diff == LIM_INF, 1000);
66      BVE_COVER_BUCKET(Cv_bucket_refmod_diff, diff > LIM_INF, 1000);
67      BVE_COVER_BUCKET(Cv_bucket_refmod_diff, diff < LIM_SUP, 1000);
68      BVE_COVER_BUCKET(Cv_bucket_refmod_diff, diff == 0, 1000);
69
70      Cv_bucket_refmod_diff.end();
71
72      Cv_bucket_refmod_sat.begin();
73
74      BVE_COVER_BUCKET(Cv_bucket_refmod_sat, sat == LIM_SUP, 1000);
75      BVE_COVER_BUCKET(Cv_bucket_refmod_sat, sat == LIM_INF, 1000);
76      BVE_COVER_BUCKET(Cv_bucket_refmod_sat, sat > LIM_INF, 1000);
77      BVE_COVER_BUCKET(Cv_bucket_refmod_sat, sat < LIM_SUP, 1000);
78      BVE_COVER_BUCKET(Cv_bucket_refmod_sat, sat == 0, 1000);
79
80      Cv_bucket_refmod_sat.end();
81
82
83    }
84  }
85
```

```
86
87   SC_CTOR(refmod_dpcm_mut):
88
89      Cv_bucket_refmod_audio("Cv_bucket_refmod_audio"),
90      Cv_bucket_refmod_diff("Cv_bucket_refmod_diff"),
91      Cv_bucket_refmod_sat("Cv_bucket_refmod_sat")
92
93      {
94         buff = 0;
95         SC_THREAD(p);
96      }
97
98 };
```

O arquivo dpcm_api_mut.c precisa da função $main()$ da linguagem C, apenas para que a ferramenta de mutação possa gerar os mutantes a partir dele. Sem o $main()$, a ferramenta não consegue gerar os mutantes.

Código Fonte C.2: Código fonte das subrotinas que executam as funcionalidades do DPCM modificado para a mutação.

```
 1
 2 #include     "/opt/soft/proteumIM2.0/LINUX/bin/proteum.h"
 3 int subtract_mut( int a, int b ) {
 4    return (0 - b) ;
 5 }
 6
 7 int saturation_mut( int unsat_data ) {
 8   if( unsat_data < -4 ) return -4;
 9   if( unsat_data > 3 ) return 3;
10   return unsat_data;
11 }
```

O *Checker* recebe apenas a instrução $sc_stop()$ na linha 28 para interromper a simulação assim que um erro for encontrado.

Código Fonte C.3: Código fonte do *checker* do módulo DPCM

```
1
2
```

```
3  //Pre-Checker Template
4
5  SC_MODULE( checker )
6  {
7
8
9     sc_fifo_in<sat_audio_ptr> audio_saida_from_refmod;
10    sc_fifo_in<sat_audio_ptr> audio_saida_from_duv;
11
12
13
14
15   sc_signal<unsigned int> error_count_audio_saida;
16      void audio_saida_p() {
17
18         while(1) {
19            sat_audio_ptr  trans_refmod = audio_saida_from_refmod.read();
20            sat_audio_ptr  trans_duv = audio_saida_from_duv.read();
21
22            if ( !(*trans_refmod == *trans_duv) ) {
23              ostringstream ms;
24              ms << "expected: " << *trans_refmod << endl
25                 << "received: " << *trans_duv << ends;
26              SCV_REPORT_ERROR("audio_saida_access",ms.str().c_str());
27              error_count_audio_saida= error_count_audio_saida.read()+1;
28              sc_stop();
29            }
30            delete ( trans_refmod );
31            delete ( trans_duv );
32         }
33      }
34
35
36
37  SC_CTOR( checker )
38  {
39        SC_THREAD(audio_saida_p);
```

```
40
41   }
42
43 };
```

O novo arquivo de ligações representado no Código Fonte C.4 agora precisa incluir o modelo de referência que foi alterado (linha 6), instanciá-lo (linha 20) e realizar as devidas ligações no *Source* e no *Checker* do *testbench* (linhas 41 e 48).

Código Fonte C.4: Código fonte do módulo que liga todos os elementos do *testbench*.

```
1  #include "structs_ext.h"
2  #include "bve.h"
3  #include "source.h"
4  #include "checker.h"
5  #include "refmod_dpcm.h"
6  #include "refmod_dpcm_mut.h"
7
8  ofstream *logfile;
9
10 int sc_main (int argc, char *argv[])
11 {
12     sc_set_time_resolution(1, SC_PS);
13     scv_tr_text_init();
14     scv_tr_db db("txdb.txt");
15     sc_trace_file *tf = sc_create_vcd_trace_file ("wave");
16     ((vcd_trace_file*)tf)->sc_set_vcd_time_unit(-12);
17     logfile = new ofstream("fifo.log");
18
19     refmod_dpcm refmod_1("refmod_1");
20     refmod_dpcm_mut refmod_mut("refmod_mut");
21     source  source_i("source_i");
22     checker checker_i("checker_i");
23
24     //Input Fifos
25
26     bve_fifo<audio_ptr > audio_entrada_refmod("audio_entrada_refmod",
           logfile);
27     bve_fifo<audio_ptr > audio_entrada_to_driver("audio_entrada_to_driver
```

```
        ", logfile);
28
29    //Output Fifos
30
31    bve_fifo<sat_audio_ptr > audio_saida_refmod("audio_saida_refmod" ,
        logfile );
32    bve_fifo<sat_audio_ptr > audio_saida_from_driver("
        audio_saida_from_driver" , logfile );
33
34
35
36    //FIFOs links of input interfaces
37
38    source_i.audio_entrada_to_refmod (audio_entrada_refmod);
39    refmod_1.audio_entrada_stim (audio_entrada_refmod);
40    source_i.audio_entrada_to_driver (audio_entrada_to_driver);
41    refmod_mut.audio_entrada_stim (audio_entrada_to_driver);
42
43
44    //FIFOs links of output interfaces
45
46    refmod_1.audio_saida_stim ( audio_saida_refmod);
47    checker_i.audio_saida_from_refmod (audio_saida_refmod);
48    refmod_mut.audio_saida_stim ( audio_saida_from_driver);
49    checker_i.audio_saida_from_duv (audio_saida_from_driver);
50
51
52    sc_start();
53    return 0;
54 };
```

Apêndice D

Código Fonte do Modelo de Referência do módulo IDCT

A seguir, o código fonte da subrotina que implementa a IDCT do decodificador de vídeo xvid versão 0.9 patch-20 utilizada como modelo de referência para um dos casos de estudo deste trabalho.

Código Fonte D.1: Código fonte da subrotina que implementa a IDCT

```
1
2  void idct_int32_init(void);
3  void idct_ia64_init(void);
4
5  typedef void (idctFunc) (short *const block);
6  typedef idctFunc *idctFuncPtr;
7
8  extern idctFuncPtr idct;
9
10 idctFunc idct_int32;
11
12 idctFunc idct_mmx;
13 idctFunc idct_xmm;
14 idctFunc idct_sse2;
15
16 idctFunc idct_altivec;
17 idctFunc idct_ia64;
```

```
18 static short iclip[1024];
19 static short *iclp;
20 idctFuncPtr idct;
21
22
23
24
25 void
26 idct_int32(short *const block)
27 {
28  static short *blk;
29  static long i;
30  static long X0, X1, X2, X3, X4, X5, X6, X7, X8;
31
32
33  for (  (i = 3) ;  i < 8; i++)
34  {
35   blk = block + (i << 3);
36   if (!
37    ((X1 = blk[4] << 11) | (X2 = blk[6]) | (X3 = blk[2]) | (X4 =
38                 blk[1]) |
39     (X5 = blk[7]) | (X6 = blk[5]) | (X7 = blk[3]))) {
40    blk[0] = blk[1] = blk[2] = blk[3] = blk[4] = blk[5] = blk[6] =
41     blk[7] = blk[0] << 3;
42    continue;
43   }
44
45   X0 = (blk[0] << 11) + 128;
46
47
48   X8 = 565 * (X4 + X5);
49   X4 = X8 + (2841 - 565) * X4;
50   X5 = X8 - (2841 + 565) * X5;
51   X8 = 2408 * (X6 + X7);
52   X6 = X8 - (2408 - 1609) * X6;
53   X7 = X8 - (2408 + 1609) * X7;
54
```

```
55
56    X8 = X0 + X1;
57    X0 -= X1;
58    X1 = 1108 * (X3 + X2);
59    X2 = X1 - (2676 + 1108) * X2;
60    X3 = X1 + (2676 - 1108) * X3;
61    X1 = X4 + X6;
62    X4 -= X6;
63    X6 = X5 + X7;
64    X5 -= X7;
65
66
67    X7 = X8 + X3;
68    X8 -= X3;
69    X3 = X0 + X2;
70    X0 -= X2;
71    X2 = (181 * (X4 + X5) + 128) >> 8;
72    X4 = (181 * (X4 - X5) + 128) >> 8;
73
74
75
76    blk[0] = (short) ((X7 + X1) >> 8);
77    blk[1] = (short) ((X3 + X2) >> 8);
78    blk[2] = (short) ((X0 + X4) >> 8);
79    blk[3] = (short) ((X8 + X6) >> 8);
80    blk[4] = (short) ((X8 - X6) >> 8);
81    blk[5] = (short) ((X0 - X4) >> 8);
82    blk[6] = (short) ((X3 - X2) >> 8);
83    blk[7] = (short) ((X7 - X1) >> 8);
84
85   }
86
87
88
89   for (i = 0; i < 8; i++)
90   {
91    blk = block + i;
```

```
92
93    if (!
94     ((X1 = (blk[8 * 4] << 8)) | (X2 = blk[8 * 6]) | (X3 =
95                  blk[8 *
96                   2]) | (X4 =
97                    blk[8 *
98                     1])
99      | (X5 = blk[8 * 7]) | (X6 = blk[8 * 5]) | (X7 = blk[8 * 3]))) {
100    blk[8 * 0] = blk[8 * 1] = blk[8 * 2] = blk[8 * 3] = blk[8 * 4] =
101     blk[8 * 5] = blk[8 * 6] = blk[8 * 7] =
102     iclp[(blk[8 * 0] + 32) >> 6];
103    continue;
104   }
105
106   X0 = (blk[8 * 0] << 8) + 8192;
107
108
109   X8 = 565 * (X4 + X5) + 4;
110   X4 = (X8 + (2841 - 565) * X4) >> 3;
111   X5 = (X8 - (2841 + 565) * X5) >> 3;
112   X8 = 2408 * (X6 + X7) + 4;
113   X6 = (X8 - (2408 - 1609) * X6) >> 3;
114   X7 = (X8 - (2408 + 1609) * X7) >> 3;
115
116
117   X8 = X0 + X1;
118   X0 -= X1;
119   X1 = 1108 * (X3 + X2) + 4;
120   X2 = (X1 - (2676 + 1108) * X2) >> 3;
121   X3 = (X1 + (2676 - 1108) * X3) >> 3;
122   X1 = X4 + X6;
123   X4 -= X6;
124   X6 = X5 + X7;
125   X5 -= X7;
126
127
128   X7 = X8 + X3;
```

```
129     X8 -= X3;
130     X3 = X0 + X2;
131     X0 -= X2;
132     X2 = (181 * (X4 + X5) + 128) >> 8;
133     X4 = (181 * (X4 - X5) + 128) >> 8;
134
135
136     blk[8 * 0] = iclp[(X7 + X1) >> 14];
137     blk[8 * 1] = iclp[(X3 + X2) >> 14];
138     blk[8 * 2] = iclp[(X0 + X4) >> 14];
139     blk[8 * 3] = iclp[(X8 + X6) >> 14];
140     blk[8 * 4] = iclp[(X8 - X6) >> 14];
141     blk[8 * 5] = iclp[(X0 - X4) >> 14];
142     blk[8 * 6] = iclp[(X3 - X2) >> 14];
143     blk[8 * 7] = iclp[(X7 - X1) >> 14];
144   }
145
146 }
147
148
149
150
151 void
152 idct_int32_init(void)
153 {
154   int i;
155
156   iclp = iclip + 512;
157   for (i = -512; i < 512; i++)
158     iclp[i] = (i < -256) ? -256 : ((i > 255) ? 255 : i);
159 }
```