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

DigiSeal
Um Estudo de Caso para Modelagem de Transações Temporais Assíncronas na Metodologia VeriSC

Ana Karina de Oliveira Rocha

Campina Grande
Maio – 2008

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

DigiSeal
Um Estudo de Caso para Modelagem de Transações Temporais Assíncronas na Metodologia VeriSC

Ana Karina De Oliveira Rocha

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

Elmar Uwe Kurt Melcher
Orientador

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

Campina Grande
Maio – 2008

X001x
2008 Rocha, Ana Karina de Oliveira
DigiSeal – Um estudo de caso para modelagem de transações temporais assíncronas na metodologia VeriSC/Ana Karina de Oliveira Rocha. ─ Campina Grande: 2008.
001x.: il

Dissertação (Mestrado em Ciência da Computação) – Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática.
Referências.
Orientador: Elmar Uwe Kurt Melcher.

1. Verificação Funcional. 2. Testbench. 3. SystemC 4. VLSI. I. Título.

CDU 010.00(000)

"DigiSeal - Um Estudo de Caso para Modelagem de Transações Temporais Assíncronas na Metodologia VeriSC"

ANA KARINA DE OLIVEIRA ROCHA

DISSERTAÇÃO APROVADA EM 16.05.2008

PROF. ELMAR UWE KURT MELCHER, Dr.
Orientador

PROFa. JOSEANA MACÊDO FECHINE , D.Sc
Examinadora

PROF. JOSÉ EUSTÁQUIO RANGEL DE QUEIROZ, D.Sc
Examinador

PROF. JOÃO MARQUES DE CARVALHO, Ph.D
Examinador

CAMPINA GRANDE – PB

DigiSeal
Um Estudo de Caso para Modelagem de Transações Temporais Assíncronas na Metodologia VeriSC

Ana Karina De Oliveira Rocha

Elmar Uwe Kurt Melcher
Orientador

João Marques de Carvalho
Componente da Banca

José Eustáquio Rangel de Queiroz
Componente da Banca

Joseana Macêdo Fechine
Componente da Banca

Campina Grande
Maio – 2008

“O segredo da felicidade não é sempre fazer o que se quer, mas querer sempre o que se faz. “ (Léon Tolstói)

Dedico este trabalho à minha mãe Wagna, ao meu pai Vital e a Daniel Ricarte, por terem me ajudado a superar todos os obstáculos que surgiram na minha vida até hoje.

AGRADECIMENTOS

A **Deus**, pela sublime proteção e inspiração em todos os momentos da minha vida.

À minha mãe **Wagna**, por ter me dado muitos exemplos de força, determinação e competência. E por sempre ter tido como prioridade o investimento na educação dos seus filhos, que foi fundamental para que eu pudesse chegar até aqui.

Ao meu pai **Vital**, por ter evoluído como pai e ser humano durante todo este tempo em que estive longe de casa para estudar.

A **Daniel Ricarte,** pelo amor, amizade, dedicação, por sempre estar ao meu lado nas horas difíceis e por me ensinar a ser um ser humano melhor.

Aos meus irmãos **Anne Caroline** e **Vital Filho**, pela paciência que resultou no fortalecimento dos nossos laços fraternos.

Aos professores **Elmar Melcher** e **Joseana Fechine,** pela amizade, pelo exemplo de competência e pelas preciosas contribuições às minhas conquistas acadêmicas.

Aos demais **professores do DSC**, pelos conhecimentos transmitidos ao longo do curso, com destaque especial aos professores **Eustáquio e Bernardo Lula,** com quem tive o prazer de compartilhar experiências de trabalho fora da sala de aula.

A todos os **companheiros de trabalho do LAD**, pelas experiências compartilhadas.

A todos os **amigos do mestrado e dos cursos de graduação** que fiz ao longo dos últimos anos.

Por fim, a todos os **funcionários da COPIN**, pela atenção, carinho e assistência prestada.

SUMÁRIO

LISTA DE ABREVIAÇÕES E SIGLAS

ABV - *Assertion-Based Verification*

ACID - *Atomicity, Consistency, Isolation, Durability*

AVM - *Advanced Verification Methodology*

Brazil-IP - *Brazil-Intellectual Property*

CETENE - *Centro de Tecnologias Estratégicas do Nordeste*

CLM - *Cycle-Level Modeling*

DUT - *Design Under Test*

DUV - *Design Under Verification*

eRM - *e Reuse Methodology*

eVC - *e Verification Component*

FPGA - *Field-Programmable Gate Array*

FIFO - *First-in First-out*

HDL - *Hardware Description Language*

HVL - *High-level Verification Language*

IC - *Integrated Circuit*

IP - *Intellectual Property*

IPCM - *Incisive Plan-to-Closure Methodology*

IP Cores - *Intellectual Property Cores*

MCT - *Ministério da Ciência e Tecnologia*

MRM - *Mixed-Language Reuse Methodology*

NoC - *Network-on-chip*

OSCI - *Open SystemC Initiative*

OVM - *Open Verification Methodology*

PDA - *Personal Digital Assistants*

PLD - *Programmable Logic Device*

PV - *Programmer View*

PVT - *Programmer View Timing*

RTL - *Register Transfer Level*

SAM - *System Architectural Model*

SCV - *SystemC Verification Library*

SoC - *System on Chip*

SVM - *System Verification Methodology*

TBA - *Transaction-based Acceleration*

TBV - *Transaction-based Verification*

TLM - *Transaction Level Modeling*

TTLM - *Timed Transaction Level Modeling*

UNISIM **-** *UNIted SIMulation environment*

URM - *Universal Reuse Methodology*

UTLM - *Untimed Transaction Level Modeling*

UVC - *Unified Verification Component*

VMM - *Verification Methodology Manual*

LISTA DE FIGURAS

RESUMO

A necessidade de sistemas cada vez mais complexos é uma realidade em quase todas as áreas de aplicação da eletrônica. Os avanços recentes da microeletrônica possibilitam o surgimento de soluções inovadoras para diversos problemas do mundo moderno, devido à criação, em ritmo cada vez mais acelerado, de sistemas digitais de qualidade, sendo possível integrar dezenas de milhões de transistores em um único chip, com baixo custo operacional. Esses sistemas estão em constante evolução, impulsionada pelo desenvolvimento da indústria de semicondutores. Assim, há fortes pressões de mercado para a disponibilização de novos produtos com um número cada vez maior de funcionalidades. As implementações dos circuitos eletrônicos complexos necessitam da utilização de metodologias eficientes e automatizadas, que auxiliem na diminuição das falhas de projeto, a exemplo da metodologia de verificação funcional denominada VeriSC, que fornece *testbenches* e utiliza a biblioteca SCV (*SystemC Verification Library*), mas se restringe à verificação de circuitos digitais que processam transações temporais síncronas. O trabalho desenvolvido consiste na criação de um mecanismo de implementação de transações temporais, aplicada à metodologia de verificação funcional VeriSC, tornando-a uma metodologia de verificação eficiente também para circuitos digitais capazes de processar transações temporais assíncronas.

ABSTRACT

The necessity for more complex systems is a reality in almost all electronic application areas. Recent advances in microelectronics make possible the appearance of innovative solutions for several problems of the modern world, due to the creation in accelerated rhythm of quality digital systems, allowing the integration of tens of millions of transistors in a single chip with low operational cost. Those systems are in constant evolution promoted by the development of the semiconductors industry. Thus, there are strong pressures from the market to make new products available with an increasing number of functionalities. Implementations of complex electronic circuits must use of efficient and automated verification methodologies, which help in reducing design failures. In this context VeriSC, a functional verification methodology which provides testbenches and uses the SCV Library (SystemC Verification Library), but it is restricted to the digital circuit verification that has only synchronous time transactions. This work consists in creating a mechanism for the implementation of time transactions, applied to the VeriSC functional verification methodology, and in making it an efficient methodology for digital circuits capable of processing asynchronous time transactions.

Capítulo 1

Introdução

Neste capítulo, apresenta-se um breve resumo do estado da arte no tocante ao uso de metodologias de verificação funcional na construção de circuitos digitais. Em seguida, define-se o problema que este trabalho de mestrado propõe resolver, apresentando os seus objetivos, a relevância do trabalho e a metodologia adotada.

1.1. Cenário técnico-científico

Um circuito integrado, também conhecido por *chip*, é um dispositivo eletrônico de dimensões reduzidas, que consiste de muitos transistores e de outros componentes que juntos são capazes de desempenhar diversas funções.

A importância da sua integração está no baixo custo e alto desempenho, aliados à alta confiabilidade e estabilidade do seu funcionamento. Os seus componentes possuem uma resistência mecânica que permite montagens robustas contra choques e impactos mecânicos. Além disso, os *chips* possuem a concepção de portabilidade dos dispositivos eletrônicos.

A arquitetura de *hardware* de um circuito integrado pode conter processadores, memórias, interfaces para periféricos e blocos dedicados analógicos e digitais. Tais blocos são chamados de IP *Cores (Intellectual Property Cores)* que executam tarefas específicas e são definidos de modo a permitir o seu reuso em diferentes sistemas [Moraes 2004]. Os diversos IP *Cores* de um determinado SoC (*System-on-Chip*) são interligados por uma estrutura de comunicação, que pode variar de um barramento a uma rede complexa (NoC – *Network-on-Chip*) [Micheli 2002].

O projeto de um SoC é bastante complexo por envolver por exemplo, questões de mobilidade, limite de consumo de potência, baixa disponibilidade de memória, necessidade de segurança, entre outras coisas, o que torna o seu desenvolvimento uma área de pesquisa bastante abrangente [Wolf 2001] [Carro 2003].

Nestes projetos, as linguagens de descrição de *hardware* (HDL – *Hardware Description Language*) podem ser utilizadas para realizar a descrição dos circuitos eletrônicos. Essas linguagens permitem descrever a forma como os circuitos operam, possibilitando a simulação

desses circuitos antes mesmo da sua fabricação. Uma das grandes dificuldades enfrentadas no projeto de um SoC ou de um IP *Core* é o processo de verificação funcional, que consiste em certificar que o SoC ou IP *Core* está obedecendo aos requisitos do projeto [Bergeron 2003] [Piziali 2004]. Tais requisitos podem ser estabelecidos com relação às funcionalidades que devem ser cobertas de acordo com as necessidades da aplicação.

A verificação funcional é a fase mais importante em projetos de *hardware*, pois se algum erro não for detectado nesta fase, ele será repassado para o processo de criação do IC (*Integrated Circuit*) e, posteriormente, para a sua fabricação, de forma a tornar a utilização do IC inviável.

Em projetos de circuitos digitais, estima-se que cerca de 70% dos recursos de projeto de *hardware* são gastos na verificação funcional, sendo a fase mais importante em termos de custos financeiros, de recursos humanos e de tempo [Bergeron 2003] [Piziali 2004]. Como o projeto de um circuito digital deve ser concluído em pouco tempo, existe a necessidade de uma metodologia de verificação funcional que consiga otimizar o seu tempo de desenvolvimento.

1.2. Definição do problema

O problema consiste na criação de um mecanismo de implementação de transações temporais assíncronas para a metodologia de verificação funcional VeriSC (ver seção 3.2). Este problema surgiu da necessidade de fazer verificação funcional de circuitos digitais que possuem transações síncronas e assíncronas, utilizando esta metodologia de verificação funcional, que é restrita à verificação de circuitos síncronos.

1.3. Objetivos do trabalho

1.3.1. Objetivo geral

O principal objetivo deste trabalho é criar um mecanismo para resolver transações temporais assíncronas para a metodologia de verificação funcional VeriSC, a fim de torná-la abrangente aos circuitos digitais assíncronos.

1.3.2. Objetivos específicos

- Estudar a importância das transações temporais síncronas e assíncronas nos circuitos digitais.
- Pesquisar a existência de outras abordagens para o tratamento de transações temporais assíncronas nas metodologias de verificação funcional da literatura [Bergeron 2005] [Mentor 2008] [Cadence 2006] [August 2007].
- Estudar os mecanismos de controle de eventos na linguagem de programação SystemC.
- Criar um mecanismo de implementação de transações temporais assíncronas para a metodologia de verificação VeriSC.
- Implementar um projeto de circuito digital capaz de processar transações temporais assíncronas com VeriSC.
- Verificar o funcionamento das transações temporais assíncronas neste projeto de circuito digital.
- Comparar abordagens de transações temporais assíncronas existentes em outras metodologias com a solução proposta neste trabalho.

1.4. Relevância

A utilização de transações não temporais (UTLM - *Untimed Transaction Level Modeling)* (ver seção 2.6.1), também chamadas de transações síncronas para a sincronização de um sistema não captura os detalhes da microarquitetura do SoC [Ghenassia 2005]. Isto é um problema para metodologias de verificação funcional como VeriSC, que só utilizam transações não temporais na sua modelagem.

Com a incorporação do comportamento temporal ao modelo TLM (ver seção 2.6), tem-se então um modelo que inclui mais características de um SoC, resultando em uma especificação mais completa da sua implementação [Ghenassia 2005]. A incorporação do elemento temporal, que utiliza transações assíncronas, é essencial para a verificação funcional de sistemas embarcados de tempo real.

Diante desta lacuna existente na metodologia VeriSC, é proposta a criação de um mecanismo para o tratamento de transações temporais assíncronas para VeriSC, utilizando o estudo de caso DigiSeal. Isto tornará a metodologia VeriSC mais robusta, visto que ela já possui alguns diferenciais em relação a outras metodologias tradicionais de verificação [Brahme

2000][Randjic 2002][Wagner 2005]: permite o acompanhamento do fluxo de projeto, de forma que o *testbench* (ambiente de simulação – ver seção 2.8) seja gerado antes da implementação do dispositivo que será verificado, chamado de DUV (*Design Under Verification*) (ver seção 2.7), tornando o processo de verificação funcional mais rápido e o *testbench* mais confiável por ser verificado antes do início da verificação funcional do DUV [Silva 2007].

Na literatura, existem duas orientações gerais para se realizar um modelo de transação mista, que utiliza transações síncronas e assíncronas, para um projeto de um IP *Core*. Primeiro, desenvolve-se um modelo puramente UTLM, descrevendo o comportamento funcional do IP *Core*, independentemente das suas características temporais. Segundo, desenvolve-se um modelo temporal TTLM (ver seção 2.6.2), que concentre todas as transações temporais relacionadas à micro-arquitetura do IP *Core*, sem a duplicação dos códigos funcionais criados pelo modelo UTLM [Ghenassia 2005].

Desta forma, a metodologia de verificação funcional VeriSC se tornará ainda mais robusta, podendo ser utilizada em um número maior de projetos digitais com a inserção da solução temporal proposta.

1.5. Metodologia de trabalho

As atividades desenvolvidas ao longo do trabalho proposto consistiram do estudo, criação, desenvolvimento e implementação de um modelo de transações temporais assíncronas, que possibilita à metodologia de verificação VeriSC testar sistemas embarcados com aplicações de tempo real. Em seguida, foi desenvolvido um sistema digital, que utilizou o modelo de transações temporais assíncronas que foi criado com o objetivo de validar e avaliar a eficiência do modelo. Para tanto, tem-se as seguintes etapas:

(1) Estudo da importância das transações temporais nos circuitos digitais;

Nesta etapa, foi realizado o estudo da importância das transações temporais, a fim de identificar os pontos relevantes para a implementação deste trabalho.

(2) Pesquisa da existência de outras abordagens de tratamento de transações temporais assíncronas nas metodologias VMM [Bergeron 2005], AVM [Mentor 2008], IPCM [Cadence 2006] e UNISIM [August 2007];

Nesta fase, foi feita uma pesquisa detalhada da existência de outras abordagens de utilização de transações temporais assíncronas nas metodologias de verificação funcional: VMM [Bergeron 2005], AVM [Mentor 2008], IPCM [Cadence 2006] e UNISIM [August 2007].

(3) Estudo dos mecanismos de controle de eventos na linguagem de Programação SystemC;

Nesta etapa, foi realizado o estudo de controle de eventos e das bibliotecas que serão utilizadas para a criação de um modelo de transações temporais assíncronas utilizando SystemC.

(4) Criação de um mecanismo de implementação de transações temporais assíncronas para a metodologia de verificação VeriSC;

Nesta etapa, foram utilizados alguns recursos encontrados na linguagem SystemC para a elaboração do modelo de transações temporais assíncronas, aplicado à metodologia de verificação funcional VeriSC.

(5) Implementação de um projeto digital, sem transações temporais assíncronas, utilizando VeriSC;

Nesta etapa, foi implementado um projeto digital chamado DigiSeal, que tem por objetivo fazer o monitoramento de caixas de medição de energia de baixa tensão. Ele provê a comunicação de um DigiSeal com um PDA (*Personal Digital Assistants*) em tempo real, sempre que solicitado.

(6) Inserção do modelo de transações temporais assíncronas no projeto DigiSeal;

Nesta etapa, foram tratados os casos que necessitam de transações temporais assíncronas no projeto DigiSeal.

(7) Teste do funcionamento das transações temporais assíncronas no projeto DigiSeal;

Nesta fase, foi feita a verificação funcional do sistema utilizando a metodologia de verificação VeriSC, que tem por objetivo verificar todas as funcionalidades do projeto, a fim de assegurar que tudo ocorre da maneira especificada. Esta fase teve grande importância para o trabalho, pois nela foi testada a eficiência do novo modelo de transações temporais assíncronas.

(8) Comparação das abordagens existentes em outras metodologias com a solução proposta para transações temporais assíncronas;

Nesta fase, foi feita uma análise comparativa dos resultados obtidos na investigação das outras abordagens de transações temporais assíncronas nas metodologias: VMM, AVM, IPCM e UNISIM, com o objetivo de identificar os pontos negativos e positivos, que poderão ajudar no aprimoramento da solução proposta.

1.6. Estrutura da dissertação

A dissertação está organizada em 7 capítulos. No **Capítulo 1** estão as informações necessárias ao entendimento introdutório da pesquisa, tais como: cenário técnico-científico, definição do problema, objetivos do trabalho, sua relevância e sua metodologia de realização.

O **Capítulo 2** destina-se a apresentar a fundamentação teórica, contendo os principais conceitos técnicos utilizados na pesquisa.

O **Capítulo 3** trata das principais metodologias de verificação funcional cujo conhecimento é relevante a este trabalho.

No **Capítulo 4**, apresenta-se o estudo de caso do DigiSeal que é um circuito de um lacre digital destinado à detecção da violação de dispositivos instalados em redes aéreas de distribuição de energia elétrica.

O **Capítulo 5** traz uma descrição geral da proposta deste trabalho, mostrando como o modelo de transações temporais é inserido na metodologia VeriSC.

No **Capítulo 6**, mostra-se trabalhos relacionados com transações temporais nas principais metodologias de verificação funcional.

Por fim, no **Capítulo 7,** apresenta-se as considerações finais da pesquisa e sugestões para trabalhos futuros.

Capítulo 2

Fundamentação Teórica

Este capítulo destina-se à apresentação de alguns conceitos básicos relacionados a esta pesquisa que são fundamentais ao seu entendimento.

2.1. Verificação

Verificação é o processo usado para mostrar que um dado sistema obedece a sua especificação. Existem alguns tipos de verificação [Bergeron 2005]:

- Estática ou formal
- Dinâmica ou funcional
- Híbrida

Estes três tipos de verificação propõem "verificar" se um dado modelo ou sistema está equivalente a outro. Tipicamente, os modelos formais e funcionais possuem níveis de abstração distintos. A verificação formal [Bergeron 2005] no contexto de hardware está relacionada à prova ou refutação da corretude de um determinado sistema. A fundamentação do sistema é feita com respeito a uma especificação formal ou por meio de propriedades que podem ser definidas por meio de métodos formais ou matemáticos. A verificação dinâmica ou funcional também é usada para provar a corretude de sistemas com relação a sua especificação, mas ela não precisa ser formal. Além disso, este tipo de abordagem faz uso de simulações para mostrar que o modelo está de acordo com as especificações. A implementação em alguma linguagem de descrição de *hardware* de uma dada funcionalidade específica do *IP Core* é chamada de DUV.

O processo de verificação funcional consiste em verificar por meio de simulação, que um dado DUV está de acordo com a especificação. Esse método é vantajoso porque não sofre limitações em termos de complexidade do *IP Core* a ser verificado. Por outro lado, apenas a simulação não é o bastante para mostrar que um *IP Core* está livre de erros. Por isso, faz-se necessário o uso de cobertura funcional que é uma técnica usada para medir o progresso da simulação e reportar quais as funcionalidades deixaram de ser testadas.

Para a verificação formal e funcional, é possível fazer a prova da presença de erros, porém não é possível provar a ausência de erros.

A verificação híbrida é a união da verificação formal com a verificação funcional. Isso quer dizer que, em alguns casos, faz-se o uso da verificação funcional e, em outros casos, faz-se o uso da verificação formal. Neste trabalho, será usado apenas o conceito de verificação funcional [Lavagno 2006].

2.2. Verificação Funcional

Em projetos de *IP Core* ou SoC, um processo bastante utilizado no seu desenvolvimento é a verificação funcional, que consiste em certificar se os requisitos estão sendo obedecidos [Bergeron 2003] [Piziali 2004]. Tais requisitos podem ser estabelecidos com relação às funcionalidades que devem ser cobertas de acordo com as necessidades da aplicação [Lavagno 2006].

A verificação funcional é demonstrada quando a intenção de um projeto é preservada na sua implementação [Piziali 2004]. Existem três aspectos importantes a serem observados para a realização da verificação funcional, que são: especificação, intenção do projeto e implementação [Piziali 2004]. Esses aspectos são apresentados na Figura 01.

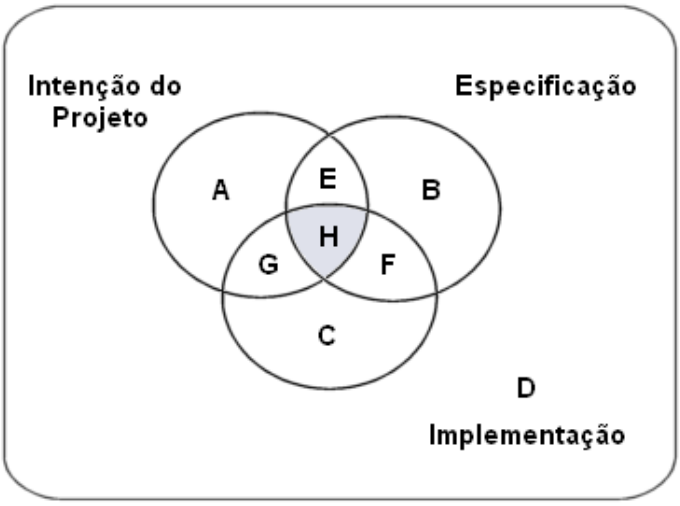

Figura 01: Características do Projeto.

A visão da especificação (B ∪ E ∪ F ∪ H) é aquela em que o projetista estando consciente da intenção do projeto (A ∪ E ∪ G ∪ H) [2], consegue elaborar toda a análise e documentação de maneira a dispor de uma especificação funcional do sistema a ser implementado (C ∪ F ∪ G ∪ H) . Já a visão que corresponde à intenção do projeto é tudo

aquilo que o cliente imagina antes mesmo de sua especificação. A última visão é a da implementação, que corresponde ao que foi implementado em relação ao que foi especificado. O conjunto D representa o comportamento não pretendido, não especificado e não implementado. Uma situação ideal ocorreria se os conjuntos da visão de especificação, intenção do projeto e implementação fossem iguais. Desta forma, são definidos os seguintes subconjuntos [Piziali 2004]:

- **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;
- **Subconjunto H:** É tudo aquilo que foi implementado e que faz parte da especificação e da intenção do projeto;
- **Subconjunto EH (E ∪ H):** Representa a intenção do projeto capturada a partir da especificação parcialmente implementada;
- **Subconjunto BF (B ∪ F):** É a parte não intencional, contudo especifica o comportamento;
- **Subconjunto FH (F ∪ H):** Representa o comportamento da especificação capturado na implementação;
- **Subconjunto CG (C ∪ G):** É a parte não especificada com implementação comportamental;
- **Subconjunto GH (G ∪ H):** Representa o que se deseja e o comportamento da implementação;
- **Subconjunto AE (A ∪ E):** É a parte não implementada com intenção comportamental;
- **Subconjunto AG (A ∪ G):** É a parte não especificada com intenção comportamental;
- **Subconjunto BE (B ∪ E):** É a parte especificada com comportamento não implementado;
- **Subconjunto CF (C ∪ F):** É a parte não intencional com comportamento implementado;

Existe um esforço para maximizar os subconjuntos H e F e minimizar os subconjuntos B e C. A idéia da verificação funcional é fazer com que os casos das áreas A e B sejam apontados provocando a redução das mesmas e como conseqüência realizando benefício para as áreas H e F. A verificação funcional é a fase mais importante de projetos de hardware, pois se algum erro não for detectado nesta fase, o mesmo será repassado no processo de criação do circuito integrado e, posteriormente, na sua fabricação [Piziali 2004].

O processo de verificação funcional começa com a escrita de um plano de verificação que implementa o ambiente de verificação. O plano de verificação define o que deve ser verificado e como será verificado. Neste contexto, a necessidade de uma metodologia de verificação funcional que consiga otimizar o tempo de desenvolvimento de um projeto de *hardware* torna-se essencial [Silva 2007]. Outro ponto importante a se observar são os possíveis cenários para um dado projeto, tais como, o mapeamento de todas as possíveis combinações de entradas para um dado IP *Core*.

Porém, gerar todos os possíveis cenários tem um custo inviável para módulos maiores, assim como é inviável testar muitas unidades menores, visto que pode levar um tempo muito grande.

Assim, é necessário fazer algumas escolhas [Pessoa 2007]:

- Verificar situações mencionadas na especificação;
- Verificar situações extremas;
- Utilizar estímulos reais;
- Criar situações aleatórias.

Nesse contexto, situações aleatórias são especialmente importantes porque são capazes de gerar cenários que seriam esquecidos.

2.3. Linguagens de Descrição de Hardware

As linguagens de descrição de *hardware* (HDL) são uma alternativa à entrada esquemática de um circuito digital por utilizarem a técnica de projeto de dispositivos lógicos programáveis (PLDs - *Programmable Logic Devices*), em que o projetista cria um arquivo de texto, seguindo certo conjunto de regras, conhecido como a sintaxe da linguagem. Em seguida, usa um compilador para criar dados de programação do dispositivo lógico programável (PLD). Esta descrição de *hardware* pode ser usada para gerar projetos hierárquicos, ou seja, um componente definido em uma descrição pode ser usado para gerar um *hardware* específico ou ser usado como parte de outro projeto [Midorikawa 2007].

As HDLs têm uma grande semelhança com as linguagens de programação, mas são especificamente orientadas à descrição das estruturas e do comportamento do *hardware*. Uma grande vantagem das HDLs em relação à entrada esquemática é que elas podem representar diretamente equações booleanas, tabelas verdade e operações complexas (por exemplo: operações aritméticas) [Midorikawa 2007].

Uma descrição estrutural descreve a interconexão entre os componentes que fazem parte do circuito. Esta descrição é usada como entrada para uma simulação lógica da mesma forma que uma entrada esquemática. Uma descrição comportamental descreve o funcionamento de cada um dos componentes do circuito. Uma HDL pode ser usada para descrever vários níveis do circuito em desenvolvimento, partindo de uma descrição de alto nível, podendo ser usada para refinar e particionar esta descrição em outras, que possuem níveis mais baixos, durante o processo de desenvolvimento. A descrição final deve conter componentes primitivos e blocos funcionais. Uma forte razão para o uso de HDLs é a síntese lógica. Uma descrição em HDL em conjunto com uma biblioteca de componentes é usada por uma ferramenta de síntese para a geração automática de um circuito digital. Além disto, o uso destas ferramentas inclui uma etapa de otimização da lógica interna do circuito gerado, antes da geração das estruturas internas de armazenamento, da lógica combinatória e da estrutura de conexão dos componentes (*Netlist*) [Midorikawa 2007].

Atualmente, as HDLs mais utilizadas para síntese são o Verilog [IEEE 1996], SystemVerilog [Bergeron 2005], VHDL [IEEE 1993] e o SystemC [Bhasker 2002], sendo que as linguagens SystemVerilog e SystemC são mais usadas para verificação. Estas linguagens estão disponíveis em diversas ferramentas comerciais. Além dessas linguagens, existem muitas outras com diferentes sintaxes, porém com a mesma idéia fundamental de tentar modelar um sistema de *hardware*.

2.4. SystemC

O SystemC possui uma biblioteca de classes C++ (SCV) com código aberto que foi disponibilizada gratuitamente na internet pelo OSCI (*Open SystemC Initiative)*. Tal biblioteca fornece ao C++ padrão recursos de verificação, dando suporte para: criação de APIs para a verificação baseada em transações, randomização direcionada, manipulação de exceções, temporização de *hardware*, atividade comportamental, concorrência e mecanismos de simulação, podendo ser utilizada como uma linguagem de verificação (*HVL – High-level Verification Language*) [Bhasker 2002]. A SCV permite programação no nível de transações,

possibilitando abstração de alto nível, reutilização de código e alta velocidade de simulação [Silva 2007] .

O SystemC cria uma "especificação executável" de uma micro-arquitetura. Essa especificação provê muitos benefícios, tais como, evitar inconsistências e erros para garantir a completude do sistema; assegurar uma interpretação não ambígua para a especificação do sistema; verificar a funcionalidade do sistema antes do início da sua implementação; criar modelos de desempenho do sistema e validar a sua performance.

2.5. Register Transfer Level (RTL)

A mais discutida e praticada forma de modelagem HDL é o RTL. O RTL é utilizado para projetar chips. O seu objetivo é descrever a intenção do projeto em um nível menos detalhado do que o nível de portas lógicas, mas detalhado o suficiente para ser entendido por um sintetizador. O sintetizador decompõe a descrição RTL em uma descrição no nível de portas lógicas que pode ser usada para criar um layout ASIC ou para gerar um programa para uma FPGA.

O projetista de *hardware* ao escrever a descrição RTL está preocupado principalmente com a função interna do circuito. Para o projetista, o chip é o projeto em alto nível. Quando o RTL é utilizado na simulação, ele normalmente é chamado de DUV. Os modelos RTL podem e devem ser simulados, isto verifica a funcionalidade do código. No entanto, o código RTL da forma como é escrito para a síntese não inclui qualquer atraso ou restrição de tempo. Evidentemente, atrasos poderiam ser incluídos, mas até que uma implementação física seja determinada estes valores não são precisos [Munden 2005].

Na verificação funcional, usam-se os dois níveis de descrição: RTL e TLM (*Transaction Level Modelling*). O TLM [Cai 2003] pode ser representado por um *testbench*, pois esse não possui descrições no nível de registradores, apenas no nível de transações [Lavagno 2006].

2.6. Transaction Level Modeling (TLM)

A complexidade dos sistemas que vêm sendo integrados em um único SoC e a necessidade de reduzir seu tempo de projeto requer não apenas avanços tecnológicos como também métodos de projeto mais eficientes. Um dos recursos adotados para este fim foi incluir níveis mais abstratos do que o RTL para modelar o sistema em alto nível de abstração. Assim surgiu o nível de abstração denominado TLM onde a comunicação entre os diferentes componentes do sistema é separada da descrição do comportamento de cada um destes

blocos [Caldari 2003]. O uso do TLM reduz o esforço na modelagem do sistema e aumenta a velocidade de sua simulação. Ele se propõe a ser uma ponte diante das lacunas existentes nos atuais métodos de integração entre o *hardware* e o *software* [Donlin 2004].

O modelo TLM se caracteriza por uma abstração intermediária entre o nível SAM (*System Architectural Model*) [Black 2004] e o nível RTL. O nível SAM modela a funcionalidade do sistema sem considerar os detalhes da arquitetura ou da implementação. Nele o sistema é descrito como um conjunto de eventos que incluem informações temporais. O nível RTL modela com precisão a implementação e a arquitetura do sistema com as linguagens de descrição de *hardware.*

A modelagem de comunicação do nível TLM é uma abstração da comunicação em termos de interfaces que implementam um conjunto de métodos [Moussa 2003]. O modelo TLM define uma transação como sendo a transferencia de dados (isto é, comunicação) ou sincronização entre dois módulos em um instante (isto é, um evento de SoC) determinado pela especificação do hardware ou do software [Ghenassia 2005]. Uma transação é a unidade fundamental da descrição do sistema TLM [Pasricha 2002], pois ela envolve a transferência de informação através de uma plataforma de comunicação.

Existe duas classes fundamentais do modelo TLM que são bastante utilizadas no desenvolvimento de sistemas: *Untimed* TLM (PV - *Programmer View*) e *Timed* TLM (PVT - *Programmer View Timing*). Eles são modelos adaptados para fins distintos. O objetivo do *Timed* TLM é criar uma plataforma única que simula dois modelos diferentes (temporais e não temproais) de acordo com a necessidade dos usuários. O *Untimed* TLM é um modelo arquitetônico destinado especificamente à verificação funcional no qual condições de tempo não são obrigatórias, a alta velocidade de simulação é o objetivo deste modelo. Por outro lado, o modelo *Timed* TLM é um modelo micro-arquitetônico no qual as condições de tempo são indispensáveis para o comportamento especificado na sua comunicação. Este modelo é relativamente menos abstrato por estar localizado em um nível mais baixo no fluxo do projeto de um SoC. O foco do *Timed* TLM é a simulação precisa que é exigida no desenvolvimento de sistemas embarcados com aplicações em tempo real [Ghenassia 2005].

2.6.1. Untimed Transaction Level Modeling (UTLM)

É um modelo arquitetural que visa especificamente o desenvolvimento funcional rápido do software e da verificação funcional onde condições temporais não são obrigatórias. A alta

velocidade de simulação é o objetivo deste modelo. Então, o modelo UTLM também é chamado de PV (Programmer´s view) [Ghenassia 2005].

O modelo *Untimed* TLM é um nível especialmente concebido para servir os programadores de software e os engenheiros de verificação no início do desenvolvimento do software e da verificação funcional. Neste nível não existe a preocupação com o tempo, assim nenhuma das informações relacionadas à micro-arquitetura do IP *Cores* deve ser incluída. Pela mesma razão, todas as informações relacionadas com a interconexão da topologia não serão capturadas no modelo *Untimed* TLM. Neste modelo, o estado interno de um componente é modelado usando variáveis internas [Ghenassia 2005].

O modelo *Untimed* TLM não tem informação relacionada com o tempo na micro-arquitetura, ou seja, não existem transações temporais assíncronas em um sistema *Untimed* TLM. Assim, todos os processos são executados de forma concorrente para acessar qualquer recurso do sistema instantaneamente. No entanto, o sistema deve demonstrar um comportamento correto durante a execução paralela de processos concorrentes. Isto implica que sistemas *Untimed* TLM devem respeitar a ordem na execução dos processos, a fim de garantir um bom desempenho funcional do sistema [Ghenassia 2005] .

Para preencher estes requisitos, o *Untimed* TLM emprega um modelo específico da computação que possui como principais características: Executar processos concorrentes de forma independente; Respeitar as causas de dependências entre os processos usando um sistema de sincronização.

2.6.2. Timed Transaction Level Modeling (TTLM)

É um modelo microarquitetural que contém condições temporais essenciais para o comportamento e a comunicação do sistema. O foco do modelo TTLM é a precisão de simulação requerida no desenvolvimento de sistemas embarcados em tempo real e na análise de arquitetura. TTLM também é conhecido como PVT (Programmer's View Timing) [Ghenassia 2005].

O modelo TLM faz referência a um nível de abstração formado por um conjunto de sub-níveis que variam o grau de detalhe das descrições funcional e temporal do sistema [Donlin 2004]. De acordo com a abordagem temporal, a modelagem de comunicação do sistema pode ser dividida em: não temporal, temporal aproximado e temporal.

A modelagem temporal aproximada modela o processamento do sistema como um conjunto de componentes que se comunicam de forma sincronizada, já na modelagem temporal os componentes do sistema se comunicam com base no tempo [Flóres 2006].

Para desenvolver um modelo temporal no nível de transação (ver definição de transação na seção 2.6), algumas considerações devem ser feitas em relação ao tempo de consumo, principalmente em dois aspectos: cálculo e comunicação. O atraso computacional é o tempo necessário para executar cálculos específicos na caracterização de um determinado sistema comportamental ou funcional, em que o atraso na comunicação é o tempo total consumido no acesso e na transferência dos dados ou das informações. As várias restrições físicas que poderiam trazer um impacto significativo sobre o comportamento de um sistema temporal, como o tamanho do barramento ou o tamanho da memória também devem ser tidas em conta durante o desenvolvimento de sistemas *Timed* TLM [Ghenassia 2005].

Os principais objetivos para o desenvolvimento das TTLM são [Ghenassia 2005]:

- Estabelecimento de metas de desempenho de uma determinada microarquitetura;
- Execução dos últimos ajustes da microarquitetura;
- Otimização do software para uma microarquitetura, a fim de que ela responda a requisitos em tempo real.
- Modelagem flexível e refinamento do *timing* de acordo com a necessidade dos usuários;
- Reutilização de modelos UTLM para reduzir o tempo de criação do SoC;
- Capacidade de integração de diferentes modelos temporais com um mesmo modelo UTLM;
- Possibilidade de acionamento (ou não) uma transação temporal em um determinado modelo;
- Independência entre as equipes de verificação de transações temporais e não temporais, mesmo que o desenvolvimento seja integrado.

2.7. Design Under Verification (DUV)

O DUV é o centro do ambiente de verificação, sendo também conhecido como DUT (Device Under Test). A maioria dos componentes de verificação interage com o DUV. Se houver *bugs* no DUV, a equipe de verificação deve encontrá-los. O código fonte do DUV é escrito em linguagem de descrição de *hardware*, de forma que sendo executada uma simulação ou uma

verificação formal, este código será interpretado ou compilado no modelo do DUV, que será utilizado pela equipe de verificação na sua simulação [Wile 2005].

O DUV pode possuir qualquer nível da hierarquia, podendo representar um único projeto, uma unidade lógica, um chip, etc. Independentemente do nível, o engenheiro de verificação deve personalizar os estímulos e os componentes de verificação para exercitar e validar o DUV [Wile 2005].

A profundidade em que o DUV é descrito também pode variar. O código fonte HDL poderá descrever funções no nível RTL, no nível de portas lógicas, no nível de transistor ou mesmo no mesmo nível comportamental (não sintetizável). O DUV interage diretamente com os estímulos e com o Checker. Os estímulos manipulam a entrada do DUV, enquanto o Monitor e o Checker observam as suas saídas. Em algumas situações, Monitores e Checkers podem estar no interior do DUV, que poderiam ser atribuídos de acordo com a necessidade de observação do DUV [Wile 2005].

O DUV é o projeto que está sendo verificado, ele não faz parte do *testbench,* no entanto, eles estão diretamente ligados um ao outro. O DUV é implementado no nível de sinais, possuindo uma interface de comunicação que só permite a recepção e envio de dados no nível de sinais. Devido a isso, é necessário algum mecanismo para que ele se comunique com o *testbench* que trabalha no nível de transações (TLM). Essa comunicação se faz com a ajuda dos blocos TDriver(s) e TMonitor(es) que traduzem sinais em transações e vice-versa. O engenheiro de projeto precisa implementar as funcionalidades que irão compor o DUV [Silva 2007].

2.8. Testbench

As simulações em projetos de *hardware* são concretizadas ou implementadas por meio de elementos chamados *testbenches*. Um *testbench* é um artefato escrito em linguagem formal, usado para criar simulações para o modelo do DUV, que é representado em alguma linguagem de descrição de *hardware*. O *testbench* deve criar estímulos a fim de conseguir ativar as funcionalidades desejadas no DUV. Por exemplo, se um DUV tem uma funcionalidade de fazer a soma de números inteiros, um estímulo para ele pode ser representado por dois números inteiros. O *testbench* é a “montagem” em volta do DUV que confrontará a sua funcionalidade com o Modelo de Referência que pode ser até mesmo um software escrito em alguma linguagem de alto nível como C, C++, Java, Python, etc.

O Modelo de Referência representa as funcionalidades que foram especificadas no projeto. Esse modelo pode ser visto como um projeto de software já testado funcionalmente que está sendo usando neste caso para ser confrontado com o modelo do *hardware* (DUV).

Algumas características desejáveis de um *testbench* são [Bergeron 2003] :

- Ser escrito em alguma linguagem de verificação;
- Não possuir entradas e saídas;
- Poder cria estímulos e verificar as respostas;
- Imprimir mensagens e criar um histórico quando o DUV apresentar um comportamento inesperado;
- Ser baseado em Transações, modelado a partir do conceito de TLM [Brahme 2000].
- Ser dirigido por cobertura funcional;
- Utilizar estímulos aleatórios.

Neste capítulo, apresentou-se os principais conceitos necessários ao entendimento desta pesquisa, por exemplo: verificação funcional, HDL, RTL, TLM, UTLM, TTLM, DUV, SystemC e testbench . No próximo capítulo, será feito um resumo a respeito das metodologias de verificação funcional (VeriSC, VMM, AVM, IPCM e UNISIM) estudadas neste trabalho.

Capítulo 3

Metodologias de Verificação Funcional

Neste capítulo, será feita uma breve explanação a respeito das metodologias de Verificação estudadas neste trabalho, tentando definir o que vem a ser uma metodologia de verificação e qual o seu papel na construção de circuitos digitais de forma eficiente.

3.1. Metodologia de Verificação Funcional

Uma metodologia de verificação funcional é o processo que rege as regras de todas as etapas do processo de verificação funcional. Ela busca certificar-se de que o projeto atende aos seus requisitos. Para isso, sistemas no nível de *testbenches* são criados baseados nas especificações do DUV. Em seguida, os diversos aspectos do fluxo de dados e de controle devem ser validados. Por fim, os *testbenches* criados durante a concepção funcional são usados para verificar o desempenho de circuitos complexos que integram vários IP *Cores*.

3.2. Metodologia VeriSC

As metodologias tradicionais propõem a implementação do DUV seguido da implementação do *testbench* [Bergeron 2003] [Silva 2004] [Randjic 2002]. Usando estas abordagens de verificação, o *testbench* tende a ser mais complexo, em comparação com uma metodologia que considere em primeiro lugar a verificação funcional (*testbench*), antes de considerar a sua implementação (DUV).

A metodologia VeriSC é composta de um novo fluxo de verificação, que não se inicia pela implementação do DUV, pois a implementação do *testbench* e do Modelo de Referência antecedem a sua implementação. Para permitir que o *testbench* seja implementado antes do DUV, esta metodologia implementa um mecanismo para simular a presença do DUV com os próprios elementos do *testbench*, sem a necessidade da geração de código adicional que não será reutilizado depois. Com esse fluxo, todas as partes do *testbench* podem estar prontas antes do início da implementação do DUV [Silva 2007].

O *testbench* da metodologia VeriSC é implementado no nível de transações. Ele é a parte da Figura 02 em forma de U invertido. Este *testbench* é composto pelos seguintes blocos:

Source, *TDriver(s), TMonitor(es)*, Modelo de Referência e *Checker*, sendo que cada bloco do *testbench* é ligado ao outro por FIFO(s), representadas na Figura 02 como setas largas. O *testbench* é ligado ao DUV por interfaces formadas por sinais, representados na Figura 02 por setas finas [Silva 2007].

Figura 02: *Testbench* da metodologia VeriSC.

Cada bloco do *testbench* pode ser descrito sucintamente da seguinte forma [Silva 2007]:

- O *testbench* é ligado por FIFOs, representadas na Figura 02 por setas largas. As FIFOs exercem uma tarefa muito importante no *testbench*, pois são responsáveis por controlar o seqüenciamento e o sincronismo dos dados que entram e saem das mesmas. Os dados das FIFOs devem ser retirados de forma seqüencial e em seguida comparados.
- O DUV é o projeto que está sendo verificado. Este deve ser implementado no nível de sinais. Por isso, precisa de algum mecanismo para se comunicar com o *testbench* que trabalha no nível TLM. Essa comunicação se faz com a ajuda dos blocos *TDriver*(s) e *TMonitor(es)* que traduzem sinais em transações e vice-versa.
- O S*ource* é responsável por criar estímulos para a simulação. Esses estímulos devem ser cuidadosamente escolhidos para satisfazer aos critérios de cobertura especificados. Todos os estímulos criados pelo *Source* são transações enviadas diretamente para os blocos Modelo de Referência e *TDriver*. Os estímulos são enviados através de FIFOs, como é mostrado na Figura 02, devendo estimular todas as funcionalidades especificadas para saber se as respostas do DUV estão corretas. Os estímulos inseridos no *testbench* podem ser ajustados de acordo com a cobertura medida durante a simulação.
- O *testbench* possui um *TDriver* para cada interface de entrada do DUV (as interfaces de entrada são as comunicações por onde o DUV recebe dados de outros blocos). O *TDriver* é responsável por converter as transações, recebidas pelo *Source*, em sinais e submetê-los para o DUV. Ele também executa o protocolo de comunicação

(*handshake*) com o DUV, através de sinais.

- O *testbench* possui um *TMonitor* para cada interface de saída do DUV (cada interface de saída representa um bloco que recebe dados do DUV). Ele executa o papel inverso do *TDriver,* convertendo todos os sinais de saída do DUV em transações para repassá-las ao C*hecker* via FIFO. Além disso, o *TMonitor* executa um protocolo de comunicação com o DUV, através de sinais.
- O módulo C*hecker* é o responsável por comparar as respostas resultantes do DUV e do Modelo de Referência, para saber se elas são equivalentes. Ele compara esses dados automaticamente e, no caso de encontrar erros, emite uma mensagem para o engenheiro mostrando quando aconteceu o erro. Como a metodologia VeriSC utiliza a abordagem de *black-box*, ela não permite a visão dos componentes internos do DUV. Além disso, normalmente a cobertura também é realizada no C*hecker*. Em alguns casos, a cobertura também pode ser medida no *TDriver* e/ou no *TMonitor*. A cobertura deve indicar a todo instante quanto dos critérios especificados já foi alcançado e deve parar a simulação quando forem alcançados todos os critérios especificados.
- O Modelo de Referência contém a implementação ideal (especificada) do sistema. Por isso, ao receber estímulos, esse modelo deve produzir respostas corretas. Para efeito de comparação dos dados, após a conclusão das operações, o modelo deve passar os dados de saída para o módulo C*hecker* via FIFO(s). O Modelo de Referência deve ser implementado no nível de transações.

A metodologia VeriSC propõe uma melhor integração entre a fase de implementação e a fase de verificação. Nela cada um dos blocos do projeto terá exatamente um *testbench* associado e no nível superior da hierarquia há um *testbench* adicional para o DUV inteiro. Esta abordagem traz as seguintes vantagens [Silva 2007]:

- A verificação funcional pode ser feita a cada iteração do processo de refinamento hierárquico do projeto, corrigindo erros do projeto antes do nível mais baixo da hierarquia ser implementado em RTL;
- A subdivisão hierárquica, feita de uma perspectiva da verificação, tende a tornar o processo de verificação mais eficiente, conservando mais custos porque a maioria dos esforços de projeto é gasto no processo de verificação;
- Os *Drivers* e *Monitores* das interfaces podem ser reusados em um esforço sistemático para assegurar a consistência das interfaces acompanhando o processo de decomposição hierárquica;

- A diminuição do tempo total de verificação;
- A possibilidade de encontrar erros mais cedo no processo;
- O *testbench* fica pronto e depurado antes que o DUV seja implementado;
- A execução do RTL pode começar e inclusive ser feita em paralelo com os *testbench*.

Atualmente, a metodologia VeriSC está sendo utilizada na *Design House* CETENE (*Centro de Tecnologias Estratégicas do Nordeste*), que é membro do programa CI-Brasil, uma das iniciativas do MCT (Ministério da Ciência e Tecnologia), que tem por objetivo executar a política industrial de semicondutores do governo federal no projeto Brazil-IP (*Brazil-Intellectual Property*), que visa formar recursos humanos qualificados para projetar circuitos integrados e estruturar a área de projetos de módulos de IP *Core*.

3.3. Metodologia VMM

Esta metodologia está documentada no livro *Verification Methodology Manua*l (VMM) [Bergeron 2005]. VMM usa SystemVerilog como linguagem de verificação e de descrição de *hardware*.

A natureza orientada a objetos da metodologia VMM é apontada como o maior obstáculo para os engenheiros que desejam adotá-la, visto que classes, encapsulamento, herança, extensões entre outros aspectos da programação orientada a objetos, tornam o ambiente de verificação diferente do utilizado tradicionalmente para a construção dos *testbenches*.

Para tentar resolver este problema, a metodologia VMM possui quatro bibliotecas que ajudam no seu entendimento [Bergeron 2005]:

- A biblioteca VMM Standard que possui um conjunto de classes e *testbenches* em SystemVerilog.
- A biblioteca VMM Checker que possui um conjunto de *assertions*, que são descritores do projeto e do comportamento do ambiente, e *checkers* em SystemVerilog.
- A biblioteca XVC Standard que possui um conjunto de classes básicas em SystemVerilog.
- O Framework de Teste de Software que possui uma biblioteca C para verificação de *software*.

Estas bibliotecas têm por objetivo permitir o desenvolvimento de equipes, que passarão a criar mais facilmente o ambiente de verificação, utilizando a metodologia VMM.

A seguir, será apresentada uma breve descrição destas quatro bibliotecas.

3.3.1. Biblioteca VMM Standard

A metodologia VMM para SystemVerilog define uma arquitetura de *testbench* que suporta verificação avançada e promove a sua reutilização. O controle deste *testbench* e a execução dos casos de teste envolvem uma série de etapas bem definidas sob o controle de métodos virtuais. A biblioteca VMM Standard define a classe vmm_env para controlar a execução de cada caso de teste, como mostrado na Figura 03.

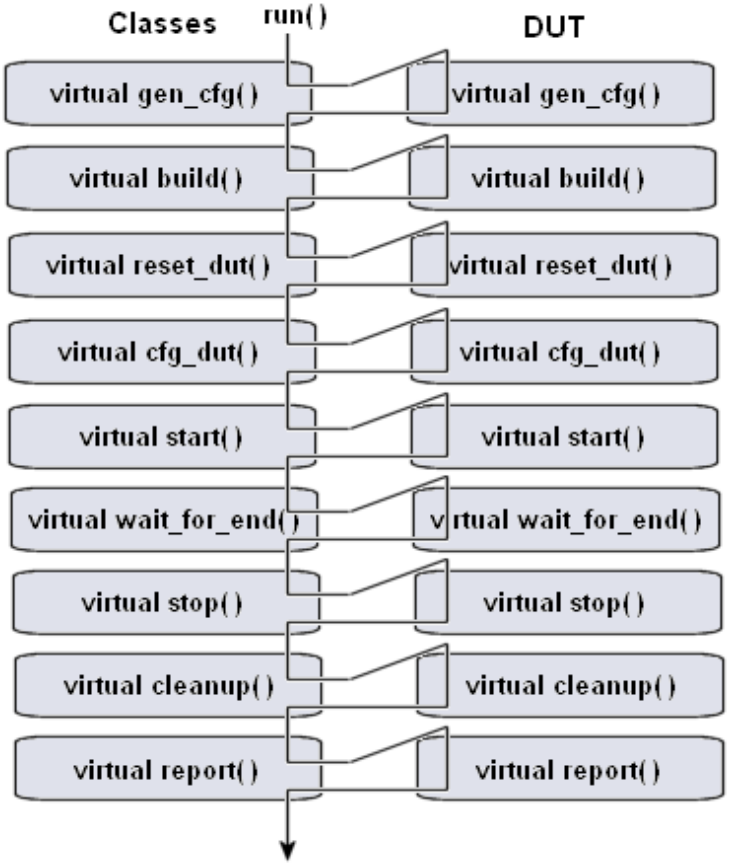

Figura 03: Definição da classe *vmm_env* com métodos virtuais.

O processo apresentado na Figura 03 envolve a geração da configuração de casos de teste, construindo *testbenches*, redefinindo e configurando o DUT (ver seção 2.7), executando os testes e, por fim, gerando relatórios. Os métodos virtuais na classe vmm_env fornecem a

infra-estrutura para cada uma destas etapas, porém muitos destes métodos têm que ser estendidos para realizar ações específicas do DUT [Bergeron 2006].

A classe vmm_log fornece uma interface para o serviço de mensagens da VMM, a fim de que todas as mensagens, independentemente da sua origem, possam ter uma visão comum. O serviço de mensagem descreve e controla mensagens, baseado em vários métodos: *Message source, Message filters, Message type, Message severity, Simulator message handling*.

A classe vmm_data é a base para todos os descritores de transações e do modelo de dados para os *testbenches*. Esta classe pode ser estendida para construir qualquer modelo adequado a um *testbench* especifico, como por exemplo, um modelo de dados para uma conexão Ethernet MAC ou para uma descrição de pacotes em um barramento serial. As transações são modeladas pelos descritores de transação, que também são estendidos da classe vmm_data. Isso torna mais fácil a criação de uma série de operações randômicas, como as chamadas a procedimentos tradicionais [Bergeron 2006].

Segue abaixo, outras classes importantes da biblioteca VMM Standard:

- Vmm_channel: fornece uma interface genérica no nível de transação.
- Vmm_broadcast: replica transações em múltiplos canais.
- Vmm_notify: fornece uma interface para a sincronização da execução concorrente de *threads*.
- Vmm_xactor: serve de base para todas as transações.

Juntas, todas essas classes provêem a construção de blocos essenciais para o ambiente de verificação, acelerando o desenvolvimento do *testbench,* ao mesmo tempo elas fornecem soluções personalizadas para atender às necessidades de verificação de DUTs específicos. A “estensibilidade” destas classes pré-definidas é a chave para a abordagem orientada a objetos com VMM, onde cada equipe de verificação pode personalizar seu ambiente de *testbench* e as suas operações sem ter que modificar as suas próprias classes [Bergeron 2006].

3.3.2. Biblioteca VMM Checker

O papel das *assertions* que consiste de encontrar erros, mais rapidamente, tem sido bem documentado há alguns anos. As *assertions* capturam as intenções do projeto, isolando os seus erros, acelerando o seu tempo de depuração, permitindo a análise formal para encontrar *bugs* que podem ser perdidos na simulação. Dadas estas vantagens, é normal que as equipes de verificação do projeto usem *assertions* [Bergeron 2006].

A biblioteca VMM Checker, que possui *assertion-checker*, é uma maneira de tornar mais fácil o uso de *assertions* em projetos RTL. Os C*heckers* são projetados para mapear os elementos comuns do projeto, tais como FIFOs, memórias, máquinas de estado e interfaces.

Os *Checkers* são implementados como módulos do SystemVerilog, podendo ser colocado em qualquer lugar do projeto ou do *testbench*. O seu uso é bem simples, bastando que o usuário ligue o *clock*, o *reset* e os outros sinais ao *checker*.

3.3.3. Biblioteca XVC Standard

A metodologia VMM define um componente de verificação estendido (XVC) e um sistema no nível transação produzido a partir de uma combinação de blocos no nível transação. O livro da metodologia VMM [Bergeron 2005] especifica a biblioteca XVC standard com um conjunto de classes e serem utilizadas na construção de um XVC para o sistema no nível de verificação.

O XVC gerencia um componente de verificação opcional responsável pelo alto nível de sincronização dos XVCs. A sincronização e os mecanismos de controle do XVC podem ser definidos pelo usuário de acordo com a necessidade do sistema ou de um teste especifico. A biblioteca XVC Standard especifica a classe xvc_manager, que pré-define o gerenciamento da classe vmm_xvc_manager [Bergeron 2006].

3.3.4. Framework de Teste de Software

A verificação utilizando *software* embutido é uma parte importante de qualquer sistema de verificação, em que um processador direciona aplicações de dados, controles de memórias e periféricos. Para um sistema no nível de testes, em que uma CPU ou um DSP faz parte do sistema a ser testado, é desejável a existência de um ambiente de verificação que suporte a execução de testes de *software*, a fim de demonstrar que o sistema suporta com sucesso a execução de um sistema operacional, uma aplicação de *software* ou algoritmo de controle de um DSP [Bergeron 2006].

A metodologia VMM define um ambiente de teste de *software* para complementar a infra-estrutura dos *testbenches*. Este ambiente é usado no lugar de um sistema operacional, cujo projeto é centrado na CPU. O ambiente de verificação do XVCs trabalha em conjunto com o *framework* de teste de *software*, de forma que tanto os estímulos internos quanto os externos podem ser gerados e sincronizados para criar condições para satisfazer os requisitos de

verificação de *hardware* e de *software*. Esta metodologia também define uma biblioteca C que inclui vários elementos que podem ser usados para implementar um ambiente de verificação e testes de *software* [Bergeron 2006].

3.4. Metodologia AVM

A Metodologia de Verificação AVM (*Advanced Verification Methodology*) é uma metodologia não proprietária que suporta verificação de sistemas no nível de RTL. Ela possui o livro AVM *Cookbook* [Mentor 2008], que inclui exemplos de código que podem ser utilizados na construção dos *testbenches*. Além disso, o código fonte das bibliotecas da metodologia AVM, com exemplos de implementação em SystemC e SystemVerilog, está disponível para o usuário.

As bibliotecas da metodologia AVM consistem de uma coleção de classes básicas que facilitam a construção de *testbenches,* incluindo classes para a construção de componentes, portas e a conexão de componentes com interfaces no nível de transação [Mentor 2008].

Nesta metodologia, os componentes fundamentais do *testbenches* são: geradores de estímulo no nível de transação, como os *Drivers* e *Monitores que* são complementares, pois os *Drivers* convertem o fluxo no nível de transação para nível de sinais e os *Monitores* convertem o fluxo no nível de sinais em fluxos no nível de transação. Os geradores de estímulo no nível transação provêem uma separação entre o processo de gerar estímulos para o DUT e o gerenciamento dos estímulos no nível de sinais [Mentor 2008].

Como a metodologia AVM suporta as linguagens SystemC e SystemVerilog, o seu código será *open-source,* podendo rodar em qualquer compilador SystemC ou em um simulador capaz de trabalhar com os recursos da linguagem SystemVerilog utilizados pela biblioteca AVM. Com base nas experiências efetuadas, somente o simulador da empresa *Mentor Graphics* trabalha com esses recursos. Ela se caracteriza por possuir um código orientado a objeto, que tem por objetivo reduzir a quantidade de código dos *testbenches,* possuindo também uma arquitetura modular que possibilita o reuso, mas não requer o uso de técnicas orientadas a objetos. Além disso, esta metodologia dispõe de uma camada abstrata a partir da linguagem SystemC que serve para unir modelos de alto-nível a modelos RTL [Mentor 2008].

Esta camada usa modelos no nível de transação (TLMs) podendo converter pacotes de alto-nível em sinais do nível RTL. A metodologia AVM segue o padrão *Open SystemC Internacional TLM 1.0,* nele as TLMs agem na transmissão de dados entre níveis de abstração altos e baixos. Esta metodologia também permite conectar equipes do nível arquitetural do projeto com as do nível RTL [Mentor 2008].

As operações *put*, *get* e *transport* são fundamentais para sincronizar processos paralelos e para a comunicação das informações no nível de transação entre esses processos. Estas idéias são bastante usadas no AVM para construir *testbenches* no nível de transação [Mentor 2008].

AVM possui controladores de teste, coletores de cobertura, analisadores de desempenho, geradores de estímulo, *constraints*, *drivers*, monitores e *responders*. Estes componentes da AVM usam algumas interfaces padrão, facilitando os *testbenches* modulares e o reuso de componentes.

A metodologia AVM oferece, a partir do SystemC, suporte à herança múltipla, já com o SystemVerilog ele suporta apenas herança simples. Além disso, o padrão das técnicas de programação orientada a objeto é usado para implementar algumas interfaces e para prover TLMs que são oferecidas pela herança múltipla do SystemC [Mentor 2008].

3.5. Metodologia IPCM

O tamanho e a complexidade de modelos no nível de sistema, aliada à geração de longas seqüências de execução de software embutido para fazer simulação lógica são insuficientes para realizar verificação no nível de sistema.

Com a metodologia IPCM, os engenheiros podem alcançar níveis mais altos de abstração com modelos no nível de transação (TLMs), usando SystemC e emulando a aceleração do hardware no nível de verificação, visto que ela oferece uma solução de verificação, baseada em transações que abrangem todo o processo de verificação para a validação arquitetural da simulação completa do RTL do sistema e da aceleração do *Hardware*.

A metodologia IPCM (*Incisive Plan-to-Closure Methodology)* promete reduzir os riscos na verificação dos *chips* e SoCs, por meio do fornecimento de um sistema com melhores práticas, princípios e procedimentos que visam aumentar a produtividade e previsibilidade do projeto, afim de assegurar a qualidade do sistema. Ela suporta a criação de um ambiente de verificação, o reuso, sistemas especialistas escritos em SystemC, modelos no nível de transação, co-verificação de hardware e de software, além de verificação baseada em transação e emulação de circuitos.

Esta metodologia aborda todo o processo de verificação de forma automatizada, reunindo um amplo espectro de necessidades de verificação, a partir dos projetistas de verificação, da equipe de projeto e das equipes especialistas que possuem o conhecimento de técnicas avançadas de verificação. Além disso, ela suporta várias linguagens: SystemVerilog, SystemC, C, C++, entre outras, possuindo exemplos de código fonte, bibliotecas com módulos

e utilitários, que automatizam o processo de eliminação de tarefas redundantes da codificação da verificação [Cadence 2006].

A Metodologia IPCM contém vários componentes metodológicos, tais como: ABV (*Assertion-Based Verification)*, SVM (*System Verification Methodology*), TBV (*Transaction Based Verification*), TBA (*Transaction Based Acceleration*) e URM (*Universal Reuse Methodology*) que inclui as metodologias eRM (*e Reuse Methodology*), MRM (*Mixed-Language Reuse Methodology*), UVC (*Unified Verification Component*) e OVM (*Open Verification Methodology*) entre outras. Esses componentes definem aproximações de verificação por meio de simulação funcional ou verificação com regras formais. Isso permite que a metodologia IPCM faça diferentes interpretações de uma especificação funcional ao longo do processo de tradução. Ela também provê o planejamento e o gerenciamento da verificação baseada em *Assertions*, em reuso e em *testbenches* automáticos, possibilitando a verificação completa do sistema [Cadence 2006].

A seguir, na Figura 04 é apresentada a arquitetura da metodologia *IPCM*.

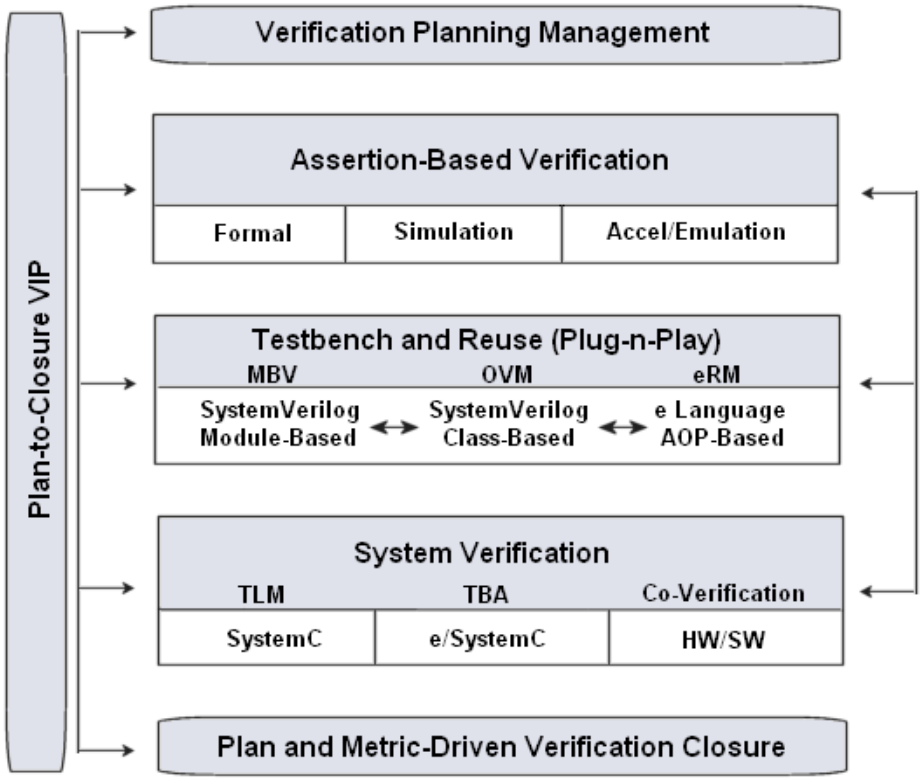

Figura 04: Arquitetura da metodologia IPCM.

O Plano de verificação da IPCM possui métricas de cobertura, oferecendo uma solução que automatiza os testes de regressão, falhas de análise e a agregação da cobertura, tanto

para a verificação do sistema no nível de transação, quanto para testes no nível de sinal para todo o fluxo do projeto [Cadence 2006].

A metodologia IPCM, por meio do componente metodológico ABV, faz uso de *assertions* como parte integrante do fluxo de verificação funcional, alcançando uma verificação mais completa em menor tempo. Elas servem como meio de documentação e de comunicação entre os engenheiros de verificação e os usuários dos IP; para especificações associadas ao código do projeto, que servem como verificação para análise formal; para executar elementos que localizam o comportamento do sistema durante a simulação, aceleração, e emulação; para sinalizar erros; e colecionar dados sobre pontos da cobertura funcional [Cadence ABV 2008].

O componente metodológico TBA possibilita a abstração dos *testbenches* que rodam em um simulador e se comunicam com o DUV que por sua vez roda em um emulador usando *transactors* que correspondem, por exemplo, aos *Drivers*, *Monitores e Checkers* da metodologia IPCM. Este componente também oferece um adaptador especial baseado em co-emulação que modela Interfaces. Este adaptador é usado pelo *transactors* para estabelecer o nível de comunicação entre mensagens da parte abstrata com a parte HDL do componente TBA [Cadence TBA 2006].

O componente metodológico SVM é responsável pelo tratamento do modelo TLM na metodologia IPCM, com ele é possível manipular transações temporais e não temporais utilizando a linguagem de descrição de *hardware* SystemC. Para utilizar o modelo TLM, esta metodologia dispõe de uma biblioteca chamada Cadence TLM que usa a biblioteca OSCI TLM [Cadence SVM 2008].

A metodologia URM é uma mistura entre metodologia e linguagem de verificação, ela utiliza os componentes metodológicos: MRM e UVC, além das linguagens: *e*, SystemVerilog e SystemC. O ambiente de verificação da URM é feito a partir do reuso dos componentes UVC. A metodologia URM estende a metodologia eRM para incluir os componentes de SytemVerilog e das linguagens da metodologia MRM [Cadence URM 2008].

O componente metodológico MRM faz parte da metodologia URM, definindo a criação dos componentes de verificação reutilizáveis, enquanto trabalha em um ambiente com várias linguagens. Tais componentes de verificação são chamados de UVC. Os UVCs são semelhantes aos eVC (*e Verification Component*), diferindo no fato do componente eVC ser restrito à linguagem *e*, enquanto os componentes UVC podem ser escritos em qualquer linguagem, além de poderem funcionar em ambientes com múltiplas linguagens.

UVC é um componente de verificação, configurável, encapsulavel e resusável para um protocolo de interface ou para um sistema completo [Cadence URM 2008].

A metodologia eRM tem por objetivo maximizar o reuso no código da verificação funcional escrito na linguagem *e*. O seu ambiente de verificação é composto pelo componente eVC, possuindo um pacote reutilizável escrito na linguagem *e* que é organizado como um componente eVC [Cadence eRM 2008].

Por fim, a metodologia IPCM também suporta a metodologia OVM que utiliza a linguagem SystemVerilog e faz reuso de metodologias que prometem possuir as melhores práticas no desenvolvimento dos componentes UVC focado na criação de SoCs. O ambiente de verificação da metodologia OVM é composto por componentes UVC reusáveis, em que cada UVC segue uma arquitetura, que consiste de um conjunto de elementos para estimular, controlar e recolher informações de cobertura para um específico protocolo ou projeto. O componente UVC é aplicado ao DUT, a fim de verificar a implementação de um protocolo ou de um projeto. Os UVCs agilizam a criação dos *testbenches* na linguagem SystemVerilog para o DUT [Cadence OVM 2008].

A metodologia OVM foi criada a partir das metodologias URM da *Cadence* e AVM da *Mentor* com o objetivo de substituir a metodologia URM, visto que ela utiliza as bibliotecas URM Class Lbrary e OVM Class Library, sendo muito similar à metodologia URM, tanto é que existe um script chamado urm2ovm que converte código escrito na metodologia URM em código para a metodologia OVM. Algumas das poucas diferenças entre URM e OVM são [Cadence OVM 2008]:

- O prefixo .urm_. foi substituído ppr .ovm_..
- Alguns nomes de tipos mudaram, como por exemplo, urm_unit passou a se chamar ovm_threaded_component.
- A ordem da API foi modificada.
- Algumas terminologias mudaram, como por exemplo, .BFM. passou a se chamar .driver.., enquanto a palavra .sequence driver., tornou-se .sequencer..

3.6. Metodologia UNISIM

A metodologia UNISIM (*UNIted SIMulation environment)* possui um ambiente de simulação modular, que proporciona um mapeamento intuitivo entre diagramas de blocos do *hardware* e o módulo do simulador, em que os blocos do *hardware* correspondem a módulos da simulação. Este ambiente de simulação é implementado em SystemC, sendo focado no reuso da lógica de controle, que corresponde a maior parte do código de um simulador. Além disso, ele suporta modelagem em um nível abstrato utilizando os modelos TLM e CLM (*Cycle-level*

Modeling). Sabendo-se que os simuladores TLM são menos precisos, mas muito mais rápidos do que os simuladores CLM, a metodologia UNISIM utiliza simuladores híbridos CLM/TLM, possuindo um sistema de simuladores funcionais que podem ser acoplados tanto em simuladores CLM quanto em simuladores TLM [August 2007].

Esta metodologia define um protocolo de comunicação comum compartilhado por todos os módulos, podendo interagir com outros simuladores, que podem se associar aos módulos da metodologia UNISIM. A idéia fundamental é evitar múltiplos padrões e assegurar que os módulos são facilmente interoperáveis, de forma que eles podem ser reusados em outro simulador. O principio básico da simulação modular é refletir a estrutura do processador na estrutura do *software*, definindo módulos na biblioteca para cada componente de *hardware*.

A metodologia UNISIM provê APIs para um conjunto de serviços, em que qualquer módulo que implemente um conjunto de chamadas padronizadas pode se beneficiar de serviços correspondentes, que são providos no nível de simulação de máquina, isto é, sendo independentes de qualquer simulador, eles podem ser facilmente modificados ou substituídos.

Ela também possui um repositório público que contém código fonte aberto onde os pesquisadores podem fazer *download* e *upload* da sua biblioteca, podendo dar contribuições para melhorias deste código fonte, além de propiciar o compartilhamento das suas pesquisas, a metodologia UNISIM permite o reuso e a comparação de idéias arquiteturais por meio dos componentes do simulador. [August 2007].

As diferenças básicas entre o simulador SystemC e o da metodologia UNISIM consistem no fato de que o simulador SystemC sugere o mapeamento dos blocos de *hardware* para os módulos do simulador, esquecendo-se do controle do *hardware*, que corresponde à menor quantidade de transistores, mas possui o maior número de linhas de código do simulador, isso desfavorece o reuso. As principais razões para isso são [Unisim 2008]:

- O Protocolo de comunicação entre módulos não é normalizado: ambientes como o SystemC provêem uma sintaxe para escrever interfaces de módulos, mas eles não especificam como dois módulos devem trocar informações. Como resultado, dois módulos desenvolvidos em separado normalmente não podem se comunicar.
- Não há nenhum esforço de reuso do controle: os seus módulos descrevem a funcionalidade dos blocos do *hardware*, mas normalmente, a maioria do código do simulador corresponde ao controle do *hardware*, visto que o código de controle dentro de um módulo descreve tipicamente as interações com os outros módulos. Assim, se outros módulos são modificados, este código de controle deve ser

reescrito, sendo quase nula a capacidade de reuso da simulação estrutural do SystemC.

Já o simulador da metodologia UNISIM é focado na reutilização de código de controle, fornecendo um módulo padronizado de um protocolo de comunicação e uma abstração do controle para essa proposta. A metodologia UNISIM provê [Unisim 2008]:

- Comunicações normalizadas e reuso do controle: a metodologia UNISIM define um protocolo de comunicação que serve para caracterizar interações entre módulos, embutindo tudo nas interfaces de comunicação dos módulos.
- Interoperabilidade: o simulador da metodologia UNISIM proporciona a interoperabilidade com simuladores existentes. Na prática, esta característica é importante porque grupos de pesquisa que investiram muitos esforços em desenvolver seus próprios simuladores podem envolver os seus simuladores com um módulo da metodologia UNISIM, tornando-o capaz de se comunicar com outros módulos do simulador UNISIM. Este mecanismo permite uma fácil transição entre os simuladores existentes e o ambiente de simulação da UNISIM.
- Biblioteca: a metodologia UNISIM vem com uma biblioteca *open source* que contém vários modelos e componentes do simulador.

Apesar de todas estas diferenças, não existe qualquer contradição entre os simuladores UNISIM e SystemC, pois o ambiente de simulação da metodologia UNISIM pode ser visto como uma camada no topo da arquitetura do SystemC.

Neste capítulo, apresentou-se um resumo a respeito das metodologias de verificação funcional (VeriSC, VMM, AVM, IPCM e UNISIM) estudadas neste trabalho. O próximo capítulo tem-se a apresentação do estudo de caso DigiSeal, apenas com a implementação do modelo não temporal.

Capítulo 4

Estudo de Caso - DigiSeal

Neste capítulo, apresenta-se o estudo de caso DigiSeal que foi utilizado para a criação e teste do novo mecanismo de transações temporais assíncronas, utilizando a metodologia de verificação funcional VeriSC.

4.1. Introdução

A distribuição das redes elétricas no Brasil favorece atos de vandalismo, tais como: deterioração de luminárias e lâmpadas, furtos de energia, travamento de medidores de energia, dentre outros. Uma das principais causas de perda de energia elétrica são, sem dúvida, as instalações irregulares ou desvios intencionais feitos pelos próprios consumidores finais. Assim, torna-se necessário a busca por soluções que visem minimizar os efeitos ou coibir ações desta natureza [Rocha 2007].

Os lacres convencionais de proteção utilizados atualmente são pouco confiáveis, de difícil manutenção e/ou verificação, sobretudo quando utilizados em redes aéreas. Os progressos recentes da microeletrônica possibilitam o surgimento de soluções inovadoras para os diversos problemas na área da segurança eletrônica, por meio do uso de câmeras, cartões eletrônicos, reconhecimento de padrões de voz e de imagem, rastreamento por GPS, dentre outros. Essas soluções se caracterizam por apresentar autonomia de funcionamento, confiabilidade e baixo custo operacional, características imprescindíveis para o bom funcionamento de um sistema de segurança digital [Rocha 2007].

4.2. Objetivos do Estudo de Caso

4.2.1. Objetivo Geral

O estudo de caso DigiSeal tem como objetivo conceber e implementar um protótipo de um circuito de um lacre digital destinado à detecção da violação de dispositivos instalados em redes aéreas de distribuição de energia elétrica [Rocha 2007]. O desenvolvimento deste estudo de caso deve proporcionar a minimização de furtos e adulteração dos dispositivos, a

minimização dos esforços de monitoramento, a redução de custos de manutenção, a redução de custos advindos da violação das redes nos pontos de instalação dos dispositivos, a redução do número de ocorrências de falhas provenientes da sobrecarga na rede de transmissão de energia e a agregação de valor ao processo de inspeção dos dispositivos nos quais o lacre digital estiver sendo utilizado.

4.2.2. Objetivos Específicos

Segue abaixo, os objetivos específicos do estudo de caso DigiSeal [Rocha 2007]:

1. Projetar um circuito integrado digital que implementa uma chave de criptografia utilizando o algoritmo DES [Burnett 2002] [Stinson 2002] e um mecanismo de correção de erros Golay [Morelos-Zaragoza 2002] [Usselmann 2008];

2. Incorporar ao projeto do circuito uma lógica de detecção de violação de lacre;

3. Acoplar ao circuito projetado um transceptor de rádio-freqüência [Texas 2002] para comunicação do sistema com dispositivos computacionais portáteis (palmtops);

4. Implementar um protótipo do sistema do lacre digital que incorpore todas as funcionalidades acima explicitadas.

4.3. Metodologia

O DigiSeal foi desenvolvido a partir da metodologia de construção de sistemas digitais integráveis, apresentada na Figura 05, possuindo as etapas descritas a seguir [Rocha 2007]:

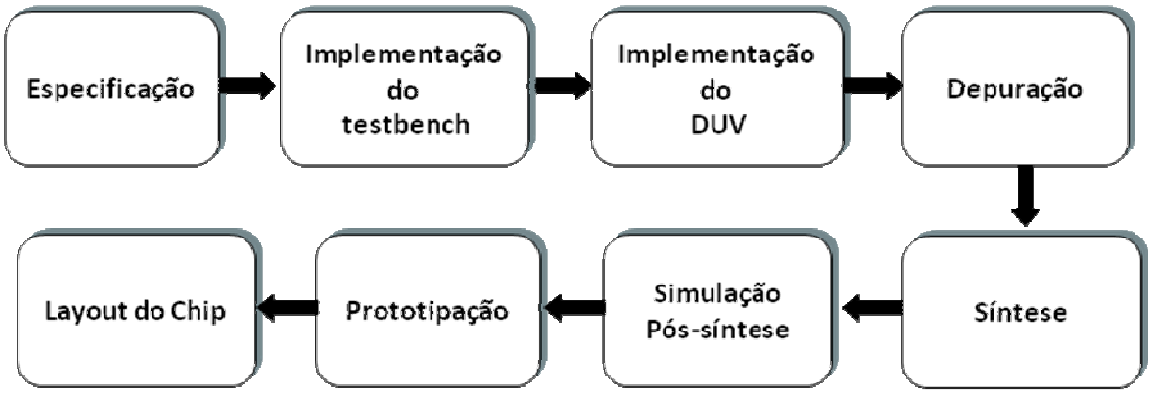

Figura 05: Fluxo do Projeto DigiSeal.

1. Especificação: Ocorre o levantamento de requisitos funcionais. Esta etapa é caracterizada pela aquisição de informações junto à empresa LIGHT (LIGHT - Serviços de

Eletricidade S.A.) para especificações dos requisitos necessários à especificação de um circuito integrado que incorpore as funcionalidades de detecção de violação do lacre digital. Ela pode ser dividida em especificação geral do circuito, em que serão caracterizados quantitativamente todos os parâmetros funcionais do sistema e em especificação detalhada que consiste na decomposição do circuito integrado do sistema em seus principais blocos funcionais, sendo especificados em detalhes os parâmetros relativos a cada bloco.

2. *Testbenches*: A metodologia de verificação VeriSC foi adotada para a escrita dos *testbenches* de cada um dos blocos funcionais. Esta etapa consiste da construção do ambiente de simulação do circuito.

3. RTL e depuração: Ocorre a implementação do código RTL sintetizável de cada bloco funcional. Em seguida, é feita a sua simulação e depuração usando os *testbenches* criados na etapa 2;

4. Síntese: Foi feita a síntese com a ferramenta de síntese automática (Altera Quartus 5.0 [Altera 2008]);

5. Simulação pós-síntese: Ocorre a verificação funcional da *netlist* que é o resultado da síntese de cada bloco, incluindo os atrasos. Por fim, é feita a síntese lógica e de layout de todo o circuito e ocorre a verificação funcional de toda a *netlist* do circuito, incluindo os atrasos de interconexão;

6. Prototipação: É feita a Implementação do circuito integrado do sistema em um dispositivo lógico programável (FPGA) usando Altera FPGA Ciclone II EP2C35 [Altera 2008] . Esta etapa visa a prototipação do circuito integrado do sistema para ser acoplado ao circuito transceptor de rádio-freqüência para comunicação do sistema com dispositivos computacionais portáteis (palmtops).

7. Layout: Criação do layout do circuito integrado do sistema do DigiSeal que incorpore todas as funcionalidades explicitadas.

4.4. Descrição Geral do Sistema

Os principais módulos do DigiSeal são (Figura 06) [Rocha 2007]:

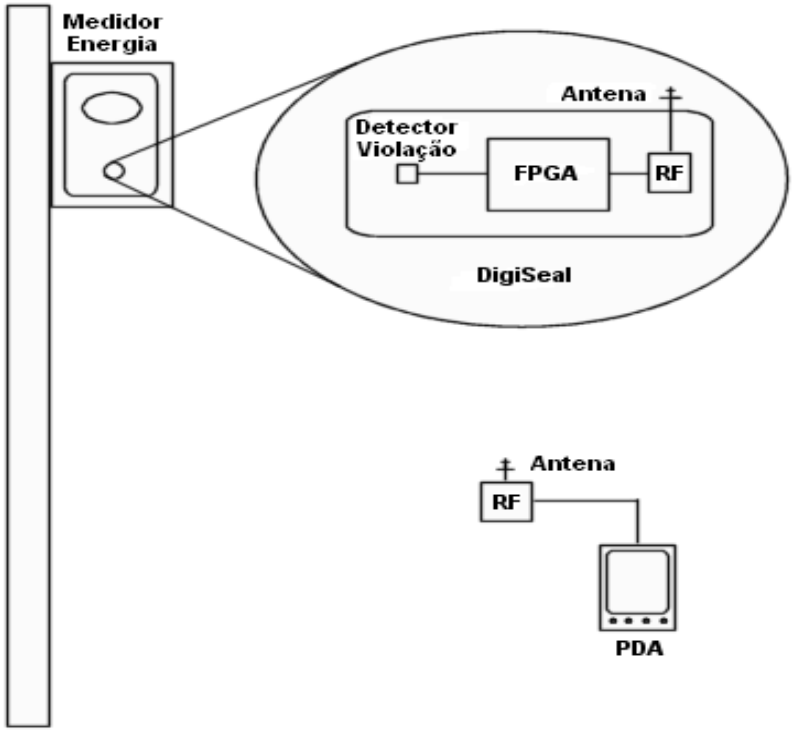

Figura 06: Esquema funcional do DigiSeal.

- Detector de Violação, que contém um contato elétrico responsável pela detecção da violação (abertura da caixa);
- O FPGA faz parte do DigiSeal e tem a função de armazenar o estado (Ok, Manutenção, Violado), a identificação do lacre, o histórico de situações, além de um relógio;
- Fonte de alimentação, responsável pela alimentação do DigiSeal, que será proveniente da própria rede elétrica e de uma bateria;
- Kit RF, módulo transceptor integrado responsável por fazer o transporte dos dados, através de radiofreqüência entre o FPGA e o PDA;

4.5. Operação do Sistema

O funcionamento do sistema segue os passos descritos a seguir [Rocha 2007].

- Ao ser ligado, o PDA passa a transmitir, de forma automática, solicitações de estado para o DigiSeal. O PDA deve estar na área de cobertura de um determinado lacre para que a solicitação chegue até o mesmo;

- O lacre instalado no medidor de energia ficará em modo de espera, sem transmitir dado algum, até receber uma solicitação de estado de um PDA nas proximidades;
- Ao receber uma solicitação, o lacre é ativado e envia uma informação ao PDA, contendo a sua identificação e o seu estado (“OK”, “VIOLADO” ou “EM MANUTENÇÃO”);
- Terminada a transmissão dos seus dados, o lacre entra novamente em modo de espera até uma nova solicitação;
- Ao receber a informação do lacre, o PDA checa se o estado é “VIOLADO”. Em caso afirmativo, ele emitirá um aviso sonoro e mostrará no display as informações de identificação do lacre e o seu estado. Se o estado do lacre for “OK” ou “EM MANUTENÇÃO”, o PDA armazenará o estado e data da inspeção na base local e não emitirá nenhum aviso sonoro, nem mostrará nada no display. Este processamento do PDA será feito de forma automática e passará totalmente despercebido pelo inspetor.
- Em qualquer momento da inspeção, o inspetor poderá consultar os lacres que não responderam (“Sem Comunicação”), bem como os que responderam (“OK”, “VIOLADO”, “EM MANUTENÇÃO”) à solicitação de estado do PDA. Esta solicitação específica será realizada quando o inspetor desejar confirmar um dado estado já recebido.
- O lacre armazena também um histórico com as mudanças de estados ocorridas nos últimos 30 dias e transmite esta coleção de estados para o PDA. Esta informação é solicitada por um inspetor autorizado e será enviada para o PDA que armazena os estados em sua base local. Não é permitido armazenar mais do que 30 informações entre dois sinais de *reset* consecutivos, em virtude das limitações do dispositivo de armazenamento.

4.6. Segurança do Sistema

A segurança dos dados ocorre a partir de um sistema híbrido de criptografia (simétrica + assimétrica), descrito da seguinte forma [Rocha 2007]:

- No FPGA é utilizada uma estratégia de criptografia simétrica (chave secreta), ficando a chave armazenada no referido dispositivo. Esta estratégia é suficiente, visto que a chave após ser gravada no FPGA, não estará mais acessível.

- No PDA, a chave esta protegida a partir da criptografia assimétrica (chave pública + chave privada), dado que a chave secreta poderia ser facilmente acessível através do acesso à memória do PDA.

4.7. Nível de Transação do Sistema

Na Figura 07, está ilustrado o diagrama de blocos no nível de FIFOs, para a implementação, a partir da especificação definida do DigiSeal.

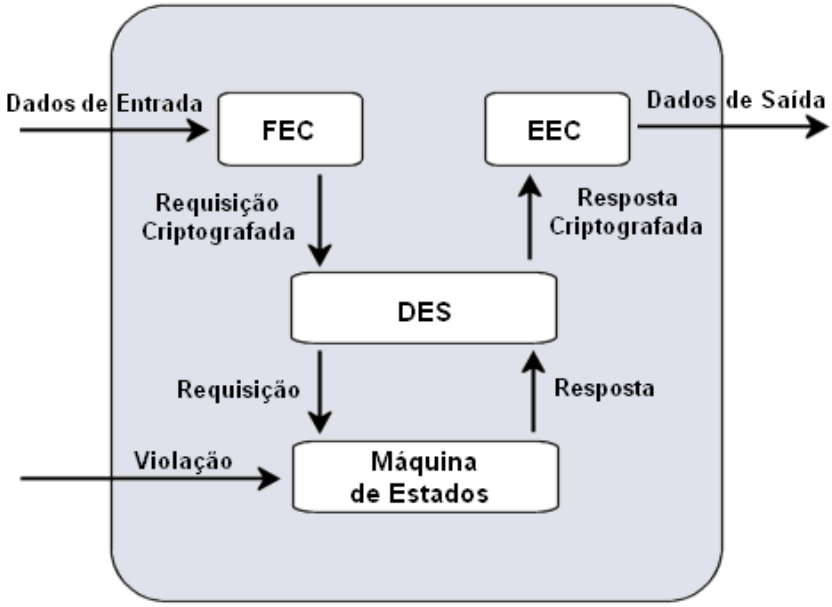

Figura 07: Diagrama em blocos com as FIFOs.

A seguir, a seqüência do caminho dos dados e definição dos blocos que são interligados por FIFOs, mostrados na Figura 07.

1º Passo: Os dados chegam criptografados pela FIFO de entrada (RFrequest) e entra no bloco chamado FEC que é o sistema do algoritmo Golay [Morelos-Zaragoza 2002] [Usselmann 2008] para a decodificação dos dados.

2 º Passo: Através da FIFO da requisição criptografada (CRIPTrequest) os dados entram no bloco DES que faz a descriptografia dos dados utilizando o protocolo DES.

3º Passo: Agora os dados passarão pela FIFO de requisição (Request) para entrar no bloco da máquina de estados, que contém um histórico de estados do DigiSeal e faz a verificação do tempo e da violação (Violate).

4º Passo: Ocorre a saída dos dados pela FIFO de resposta (Reply) para o bloco DES, afim de que eles sejam criptografados.

5º Passo: Os dados saem pela FIFO de resposta criptografada (CRIPTreply) em direção ao bloco EEC que é o sistema do algoritmo Golay que faz a codificação dos dados que sairão pela FIFO de saída (RFreply).

Os blocos que constituem o DigiSeal, apresentados na Figura 07, foram implementados em Linguagem de Descrição de Hardware (Verilog e VHDL) para a criação do DUV e nas Linguagens C/C++ e SystemC para a confecção dos modelos de referência e dos seus *testbenches*. Cada bloco do DigiSeal corresponde a um bloco do DUV que deve ser verificado de forma independente. Já os modelos de referência dos blocos devem ter todas as funcionalidades do DigiSeal implementadas de forma ideal.

4.8. Testbench do DUV Completo

O projeto de DigiSeal foi verificado usando uma aproximação hierárquica. Para isso, foi implementado o *testbench* para o DUV completo, que foi construído em sub-passos, conforme é mostrado na Figura 08 e explicado a seguir.

Primeiro, o Modelo de Referência foi testado para que ele possa interagir com o *testbenc. P*ara isso, criou-se um *Pre_Source* que é um Source que tem a função de gerar uma entrada para o Modelo de Referência e um *Pre_Sink* que recebe os dados de saída do Modelo de Referência. Em seguida, o *Source* e o *Checker* foram criados para evitar erros na criação dos estímulos. Para testá-los, foram utilizadas duas instancias do Modelo de Referência, então o Source estimulou estes modelos e o Checker recebeu as suas saídas durante a simulação, a fim de verificar se as respostas dos dois Modelos de Referência eram equivalentes. Se estes estímulos não acusarem erro no Checker, alguns erros deverão ser introduzidos em um dos Modelos de Referência para verificar se o Checker consegue detectar o problema. Por fim, os *TDrivers* e o *TMonitors* devem ser testados, como ainda não se tem um DUV, torna-se necessário utilizar um Modelo de Referência que junto com os *TDrivers* e *TMonitors* farão o mesmo papel do DUV na simulação. Para isso, foi necessário conectar os *TDrivers* e *TMonitors* ao Modelo de Referência, pois eles farão o mapeamento dos dados do nível de sinal para o nível de transação e vice-versa, a fim de que depois que o DUV estiver pronto, ele possa substituir os blocos: *TMonitorRfq*, *TMonitorVio*, Modelo de Referência e *TDriverRfp*, mostrados na Figura 08.

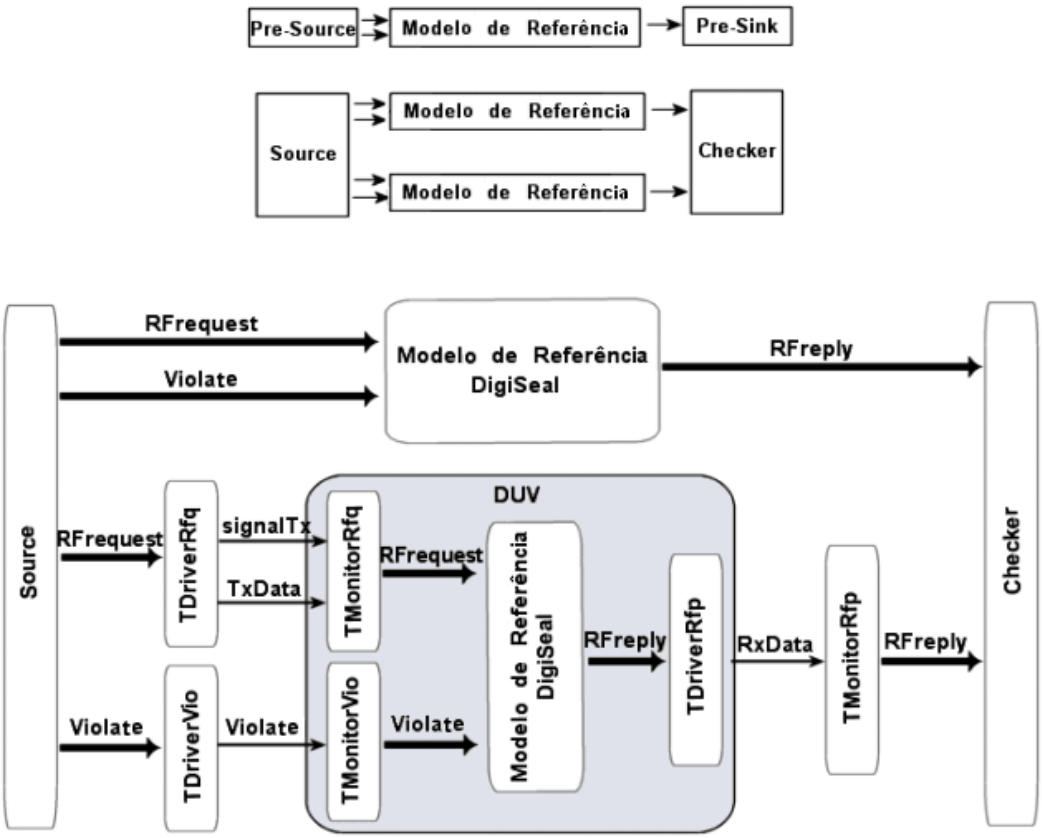

Figure 08: Testbench do DUV Completo.

Assim, os *testbenches* podem ser simulados enquanto o DUV ainda estiver indisponível, além disso, devido ao reuso do *TMonitorRfq*, *TMonitorVio* e *TDriverRfp* que será mostrado na seção 4.10, nenhum trabalho extra será necessário para implementar a substituição do DUV.

4.9. Decomposição Hierárquica do Modelo de Referência

A decomposição do Modelo de Referência do DigiSeal deve ser equivalente à decomposição hierárquica planejada para o DUV. Cada bloco que é o resultado da decomposição do Modelo de Referência é tratado como um Modelo de Referência que possui os seus próprios testes que usam *Pre_Source* e *Pre_Sink*. Esta geração de *testbenches* da decomposição hierárquica do Modelo de Referência é mostrada na Figura 08.

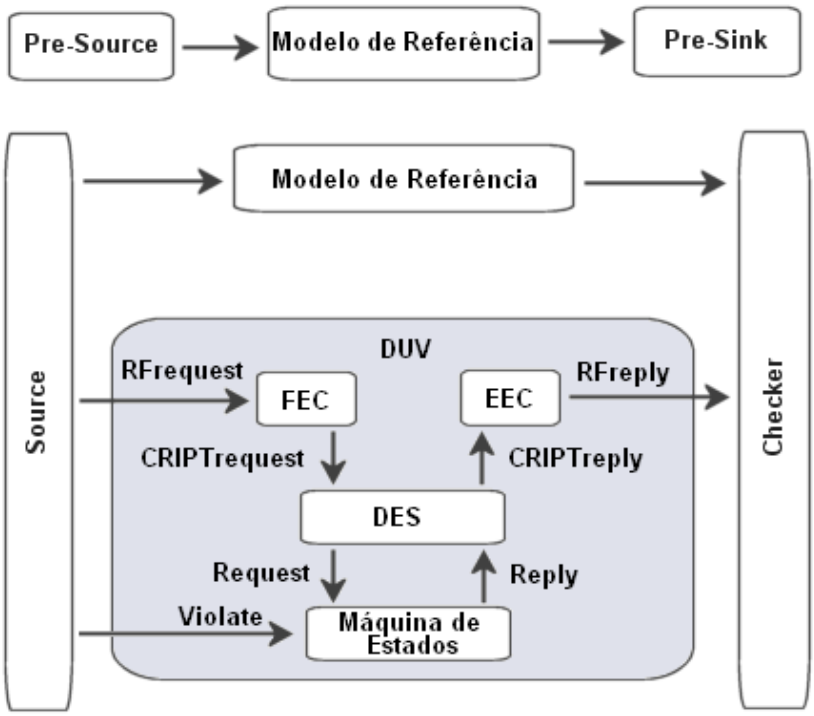

Figura 09: Decomposição Hierárquica do Modelo de Referência.

Este sub-passo usou um Modelo de Referência global, um *Source* e um *Checker* para verificar se o Modelo de Referência formado pela união de todos os Modelos de Referência dos blocos: FEC, EEC, DES e da Máquina de Estados, mostrado no bloco cinza da Figura 09, são funcionalmente equivalentes ao Modelo de Referência global. Para executar isto, o *Source* introduziu alguns dados no Modelo de Referência global e na composição dos Modelos de Referência do FEC, DES, EEC e da Máquina de Estados, enquanto o *Checker* comparou os resultados. Depois deste passo, cada bloco teve o seu próprio Modelo de Referência verificado e um *testbench* foi criado para cada um dos blocos individuais.

4.10. Testbenches para cada bloco do DUV

Este passo gerou os *testbenches* para cada bloco, que é o resultado da decomposição hierárquica de DUV e do Modelo de Referência. Os *testbenches* individuais foram criados de um modo semelhante ao *testbench* para o DUV completo (ver seção 4.8).

Neste sub-passo, criou-se um *testbench* para cada bloco: FEC, DES, EEC, e Máquina de Estados, em seguida, o *Source,* o *Checker, o TDriver(s)* e o *TMonitor(s)* de cada bloco foram testados. Os *testbenches* deste sub-passo são mostrados nas Figuras 10 a 13. O reuso do código ocorre no *testbench* do DES, onde são reusados o *TMonitorCrq* e o *TMonitorRep* do

testbench do FEC e da Máquina de Estados, visto que a interface de saída do FEC e da Máquina de Estados são iguais à interface de entrada do DES. O *testbench* do DES também reusa o *TDriverReq* e o *TDriverCrp* do *testbench* da Máquina de Estados e do EEC respectivamente, pois a interface de saída do DES é igual a interface de entrada da Máquina de Estados e do EEC. Todos os elementos podem ser reusados, caso seja necessário. O reuso é muito importante porque reduz o risco de inconsistências de interfaces ao unir os blocos.

Neste sub-passo, ocorre à substituição do *TDriver(s)*, Modelo de Referência e do *TMonitor(s)* pelo DUV correspondente, como indicado pelos blocos cinzas nas Figuras 10 a 13. Isto inclui o RTL que aparece na metodologia de verificação funcional VeriSC. O significado dos sinais e das FIFOs que aparecem nos *testbenches* das Figuras 10 a 14 está descrito no Apêndice B.

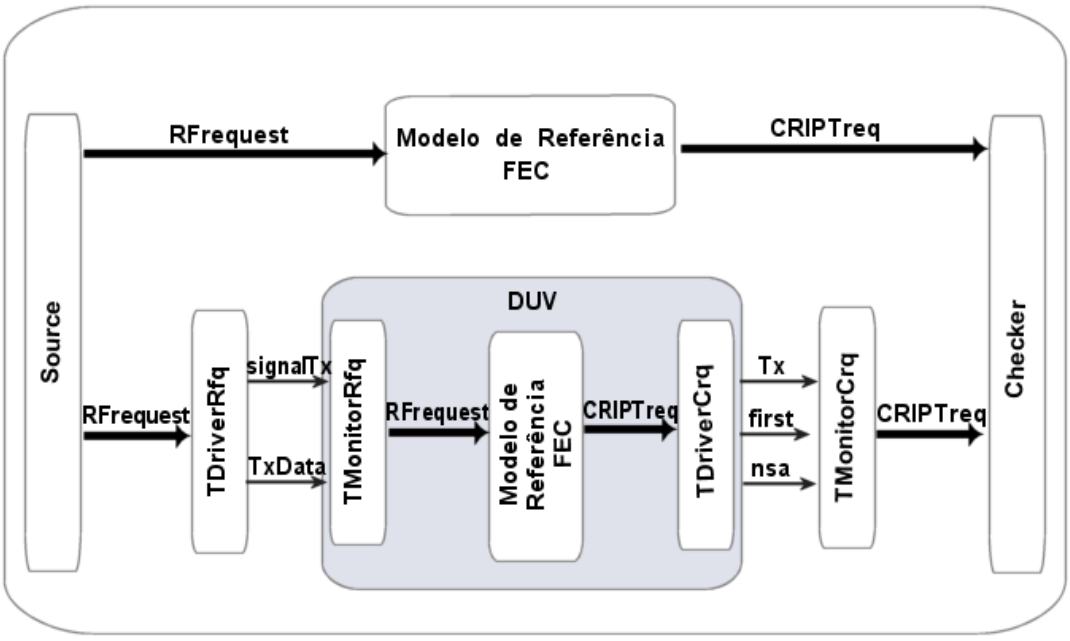

Figura 10: *Testbench* do bloco Golay FEC.

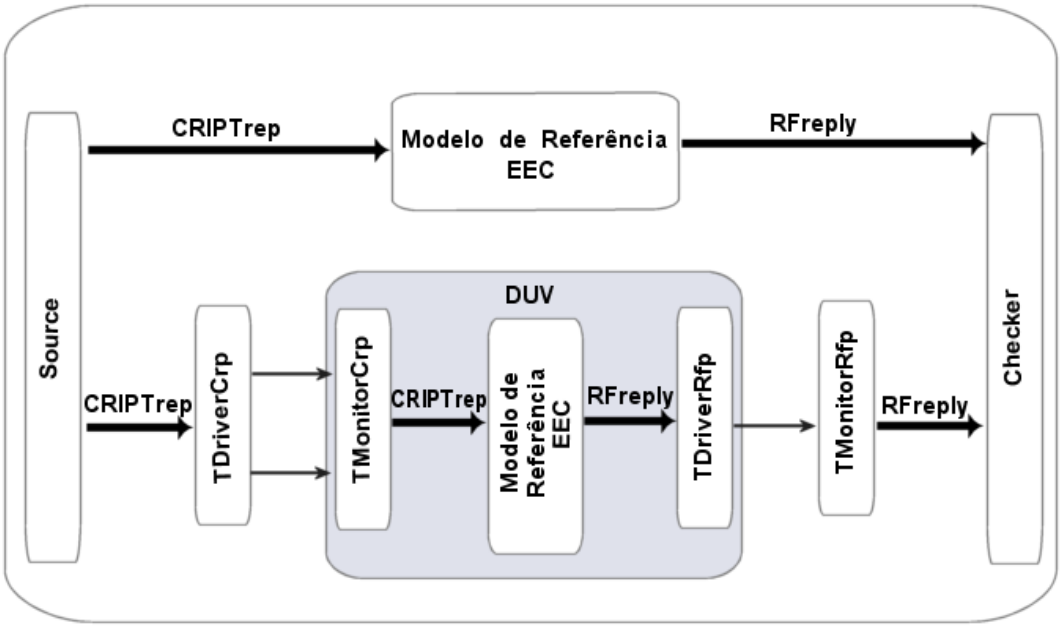

Figura 11: *Testbench* do bloco Golay EEC.

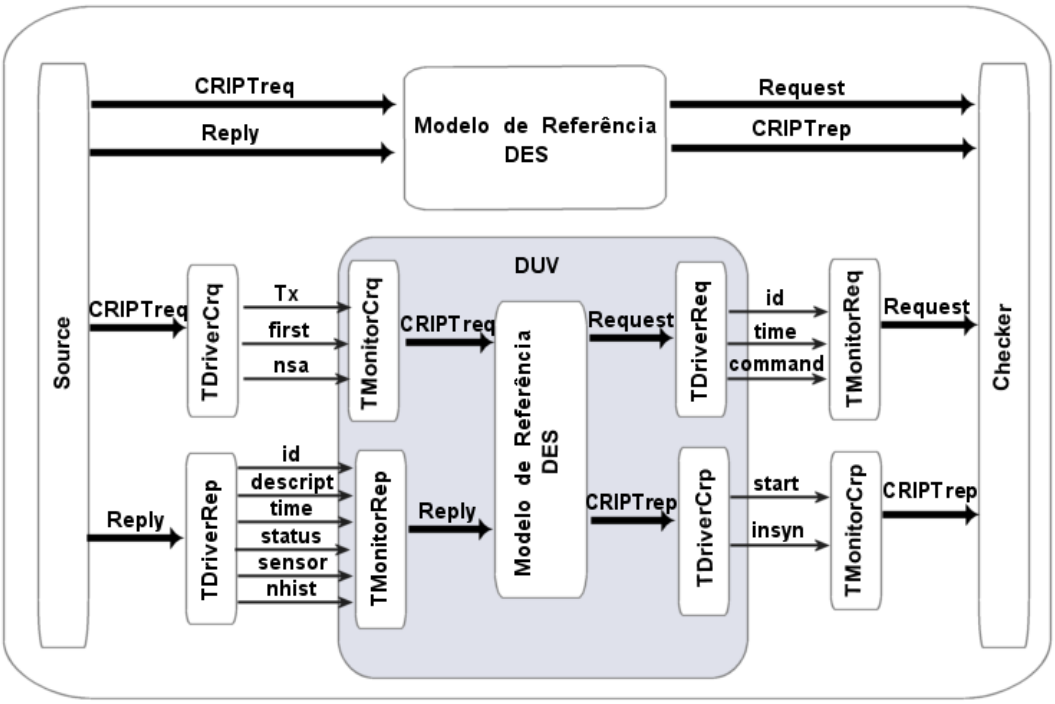

Figura 12: *Testbench* do bloco Criptografia.

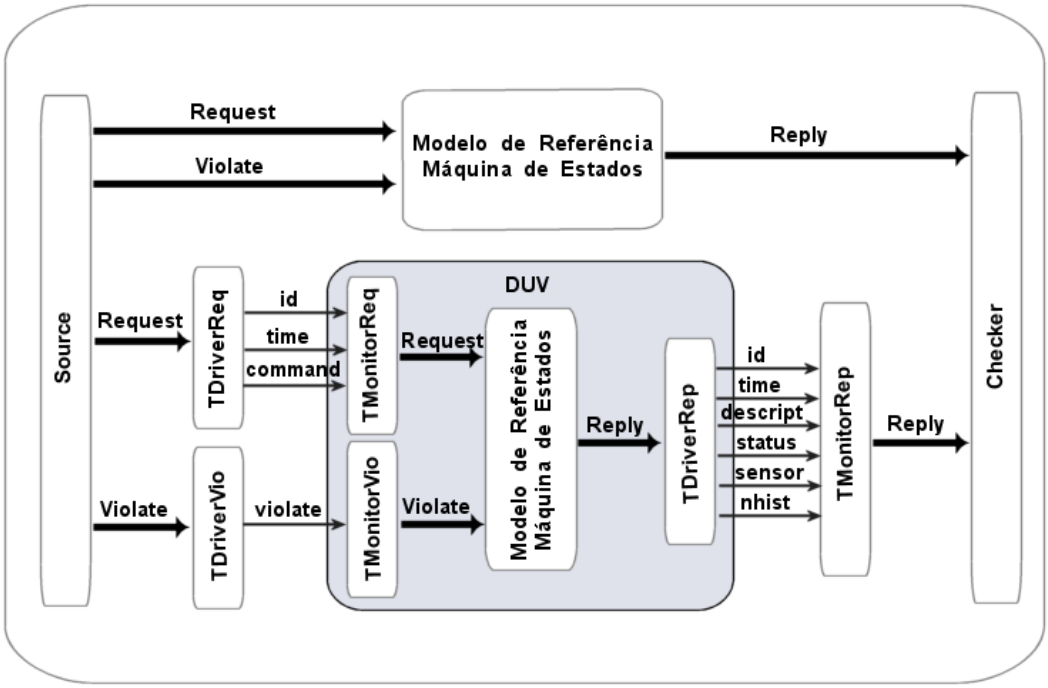

Figura 13: *Testbench* do bloco Máquina de Estados.

4.11. Substituição do DUV Completo

Finalmente, depois de verificar todos os blocos do projeto, eles foram ligados um ao outro para serem verificados. Então, no último passo os *TMonitorRfq*, *TMonitorVio*, Modelo de Referência e *TDriverRfp* de todo o *testbench* do DUV (seção 4.8, Figura 8) foram substituídos pelo DUV completo, conforme ilustrado na Figura 14, em que ocorreu a união dos códigos RTL de todos os blocos para a execução da verificação funcional do DUV completo. Deste modo, cada bloco do DUV foi verificado. Em seguida, um teste de regressão foi usado no *testbench* global para ver se a comunicação entre blocos não introduzia erros. O tratamento das transações temporais assíncronas utilizando VeriSC no DigiSeal será abordado no próximo capítulo.

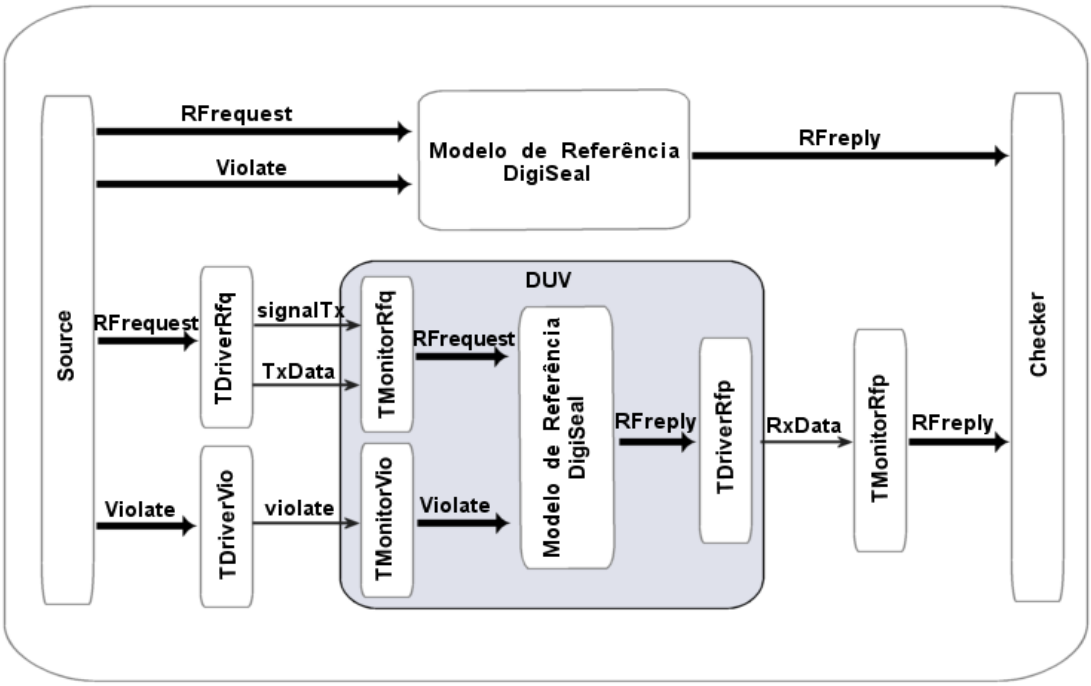

Figura 14: Testbench para o DUV Completo.

4.12. Montagem do Sistema

Na etapa de síntese, foi definido qual o dispositivo seria utilizado para a prototipação do sistema, a saber, placa Cyclone II EP2C35 DSP da Altera [Altera 2008]. Em seguida, o sistema foi prototipado nesta FPGA e funcionou de primeira, o que demonstra que o processo de verificação funcional do sistema foi bem feito. A fim de permitir a construção de uma placa específica para montagem do FPGA, foi gerado o layout para a medição da área do chip.

Neste capítulo, tem-se a apresentação do estudo de caso DigiSeal, apenas com a implementação do modelo não temporal. No próximo capítulo, será apresentado o modelo de transações temporais assíncronas criado para a metodologia de verificação funcional VeriSC, seguido da implementação deste modelo no estudo de caso DigiSeal.

Capítulo 5

Transações Temporais com a Metodologia VeriSC

Este capítulo traz uma descrição da implementação do mecanismo de transações temporais assíncronas proposto neste trabalho, mostrando como ele é inserido na metodologia VeriSC de forma genérica. Em seguida, é apresentada a implementação de transações temporais utilizando o estudo de caso DigiSeal.

5.1. Implementação de Transações Temporais com VeriSC

Normalmente, os modelos de referência não possuem noção de tempo. Este é um problema quando transações assíncronas têm que ser executadas de acordo com a ordem em que elas ocorreram. Na Figura 15, mostra-se um exemplo genérico de *testbench* na metodologia VeriSC que necessita de noção temporal em suas transações.

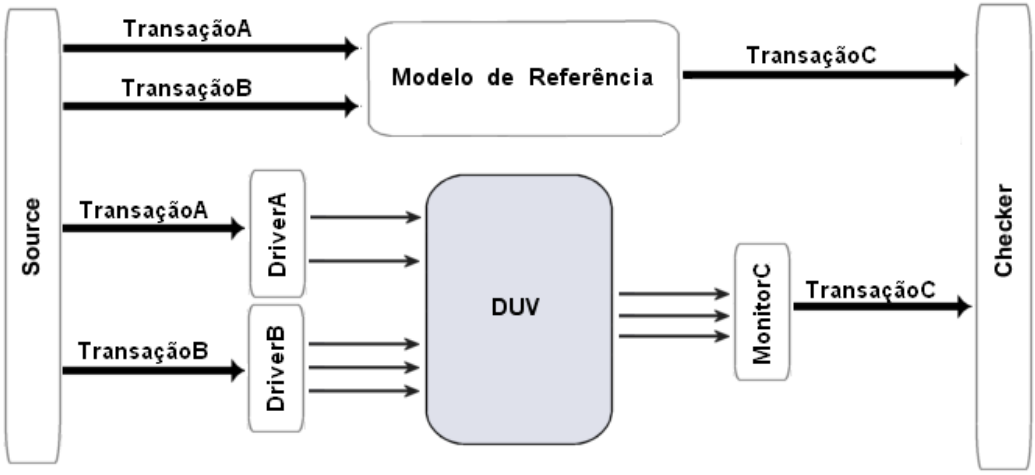

Figura 15: Testbench genérico com transações temporais.

Neste caso, para resolver o problema de transações temporais com VeriSC é necessário sincronizar as transações A e B durante a sua entrada no Modelo de Referência. Para isso, foram inseridas transações com noção de tempo no *testbench* apresentado na Figura 15.

Os procedimentos utilizados para a inserção de transações temporais neste *testbench* utilizando VeriSC são descritas a seguir:

No Source, foi criado um selo de tempo (*timestamp)* para cada uma das transações A e B;

No Modelo de Referência, foi inserido o comando *wait* que espera a chegada de transações por meio de uma das FIFOs de entrada. Isto é feito com o auxilio de um método que detecta a chegada de um evento por uma FIFO, por exemplo: wait(fifo_inA.data_written_event() | fifo_inB.data_written_event()). O comando *wait()* permite bloquear o SC_THREAD no qual ele está inserido até que qualquer uma das duas FIFOs tenha a chegada de uma transação. Logo, as transações podem ser tratadas de forma assíncrona uma em relação à outra. Isso preenche um dos requisitos para tratamento de transações temporais formulados no Capítulo 2.

Uma vez que o SC_THREAD [Bhasker 2002] sai do método *wait()*, deve-se verificar em qual ou quais FIFOs existe a chegada de uma transação, executando o seguinte procedimento:

1. Se chegar uma transação em uma determinada FIFO e não existir nenhuma transação anteriormente armazenada nesta FIFO, a nova transação será armazenada. Vale ressaltar que existe espaço apenas para o armazenamento de uma única transação para cada FIFO.
2. Entre as transações armazenadas, a transação com menor valor de *timestamp* será tratada, fazendo com que o tempo simulado avance até o valor do *timestamp* desta transação. A necessidade de retirar uma transação da FIFO e armazená-la separadamente no passo 1, se dá para que se possa acessar o campo *timestamp* neste passo, visto que não é possível acessar campos de transações que ainda estão dentro de uma FIFO.
3. Feito os itens 1 e 2, esta transação é descartada.
4. Repetem-se os passos 1 a 3 até que não existam mais transações armazenadas nem transações nas FIFOs.
5. Em seguida, o Modelo de Referência volta a esperar até que alguma FIFO tenha uma transação para ser entregue. Assim, o tempo de simulação da transação será sempre o mesmo valor do menor *timestamp* encontrado dentre as transações. Isto garante a sincronização destas transações que originalmente eram assíncronas. Nos *Drivers* e Monitores ocorre a geração de sinais conforme o *timestamp* da transação.

Para o caso de três ou mais FIFOs com transações assíncronos, acrescentam-se as correspondentes expressões *fifo_inX.data_written_event()* no comando *wait()*. Neste caso, os passos 1 até 5 ocorrem conforme descrito acima.

5.2. Implementação de Transações Temporais com VeriSC no DigiSeal

No DigiSeal o detector de violação e o armazenamento do número de eventos de violação contendo variáveis de tempo e data é feito na Máquina de Estados (ver *testbench* na Figura 13). Quando os dados chegam pela transação *Request*, ela guarda o estado do DigiSeal no histórico de eventos de violação. A Máquina de Estados também reage à chegada da transação de Violação que modifica o histórico de eventos de violações. Esta última transação é assíncrona em relação à transação *Request*.

Para resolver este problema, foram inseridas transações com noção de tempo no *testbench* da Máquina de Estados. Isto tem o objetivo de sincronizar as transações *Request* e Violação durante a sua passagem pelas FIFOs do Modelo de Referência, que na prática é implementada da seguinte forma: Todo o *testbench* foi escrito em SystemC, com exceção do DUV que foi escrito na linguagem Verilog. No Source da Máquina de Estados foram criadas transações randomizadas para o intervalo de tempo da simulação usando restrições do SystemC, como no exemplo a seguir:

```
01. class Req_constraint_class: public scv_constraint_base { (...)
02.  simulation_time_distrib.push(pair<int,int>(15000, 40000), 100);
03.  req_sptr->simulation_time.set_mode(simulation_time_distrib);
04.  req_sptr->time.set_mode(time_distrib); (...)};
05. class viol_constraint_class: public scv_constraint_base { (...)
06.  viol_sptr->simulation_time.set_mode(simulation_time_distrib); }};
07.
08. SC_MODULE(source) {
09.  Req_constraint_class Req_constraint; viol_constraint_class viol_constraint;
10.  sc_fifo_out <Violate *> violate_modref, violate_duv;
11.  unsigned int last_simulation_time = 0; Violate vio;
12.  while(1) { if(type=="Violate") { vio = viol_constraint.viol_sptr.read();
13.   vio.simulation_time += last_simulation_time
14.   last_simulation_time = vio.simulation_time;
15.  if(last_sensor==CLOSED) last_sensor=OPEN; else last_sensor=CLOSED;
16.   violate_modref.write(new Violate(vio)); violate_duv.write(new
17.   Violate(vio));
18.   } else if(type=="Request") {
19.      Request req = Req_constraint.req_sptr.read();
20.      req.simulation_time += last_simulation_time;
21.      last_simulation_time = req.simulation_time;
22.      req.time = req.simulation_time;
23.      req_modref.write(new Request(req)); req_duv.write(new Request(req));
24.   }} SC_CTOR(source){{SC_THREAD(p)}};
```

O código das linhas 13, 14, 20, 21 e 22 mostram a criação de uma distribuição de probabilidade do tempo de simulação, isto garante que o tempo de simulação está aumentando e que está sendo armazenado. No *TDriver_vio* e *TDriver_req* as linhas 28, 29, 30, 37, 38 e 39

mostram a recuperação do tempo simulado no *Source* e a captura do tempo atual de simulação. Neste momento, eles são comparados e se forem assíncronos, o comando de espera (*wait*) permitirá sincronizar o tempo de simulação em relação ao tempo da transação.

```
25. SC_MODULE(TDriver_vio) {
26. sc_fifo_in <Violate *> violate; Violate *viol_ptr; sc_out <bool> viol ;
27. while(1) {  viol_ptr = violate.read();
28.    sc_time trans_time(viol_ptr->simulation_time, SC_US);
29.    sc_time time_now = sc_time_stamp();
30.    if( trans_time > time_now ) wait( trans_time - time_now );
31.    if(viol_ptr-> sensor == OPEN) viol = true; else viol = false;
32. }} SC_CTOR(TDriver_vio){ (...}};
33.
34. SC_MODULE(TDriver_req) { (...)
35. sc_fifo_in<Request *> request; Request *rep_ptr;
36. while(1) { req_ptr = request.read();
37.      sc_time trans_time(req_ptr->simulation_time, SC_US);
38.      sc_time time_now  = sc_time_stamp();
39.      if( trans_time > time_now ) wait( trans_time - time_now );
40. (...) } SC_CTOR(TDriver_req{{SC_THREAD(p)}};
```

No *TMonitor_vio* e *TMonitor_req* as linhas 44 e 50 mostram que o tempo de simulação já está sincronizado em relação ao tempo atual da simulação. Então, eles estão prontos para enviar os dados para o Modelo de Referência.

```
41. SC_MODULE(TMonitor_vio) { (...)
42. sc_in<bool> violate; Violate viol;
43. while(1) { if(violate.read()){ viol.sensor = violate.read() ? OPEN : CLOSED;
44.      viol.simulation_time = sc_time_stamp().value();
45. }  else (...)}SC_CTOR(TMonitor_vio){ (...) };
46.
47. SC_MODULE(TMonitor_req) {
48. sc_fifo_out <Request *> request; Request req;
49. while(1) {
50.     req.simulation_time = sc_time_stamp().value();
51. (...)} SC_CTOR(TMonitor_req){{SC_THREAD(p)}};
```

No Modelo de Referência da Máquina de Estados, a linha 53 apresenta as funções que executam as funcionalidades das transações Request e Violação, respectivamente. A linha 55 retorna uma referência para um evento que será notificado no momento de uma escrita na FIFO. Neste caso, o evento pode ser de Request ou de Violação. Nas linhas 60, 61, 62 e 63 ocorre a captura sincronizada do evento Request e do evento de Violação. Quando a transação Request existe, mas a transação de Violação não existe ou o tempo simulado da transação Request é menor ou igual ao tempo simulado da transação de Violação, tem-se que o tempo

simulado da transação que será propagada terá o mesmo valor da transação Request. Este processo ocorre para a transação de Violação de forma análoga. Este procedimento verifica entre as transações armazenadas, qual é a transação com menor valor de *timestamp* que será tratada, fazendo com que o tempo simulado da transação que será propagada avance até o valor do *timestamp* desta transação. Isto garante a sincronização destas transações que originalmente eram assíncronas. Mais informações a respeito do código do *testbench* da Máquina de Estados estão no Apêndice A.

```
52. SC_MODULE(StateMachine_RM) {
53. void exec_request() {(...)} void exec_violate(){(...)}
54. while(1) {
55.  wait(violate.data_written_event() | request.data_written_event());
56. while(req_ptr || viol_ptr || request.num_available()+violate.num_available()
>    57. 0){
58.   if(req_ptr==NULL && request.num_available() > 0) req_ptr = request.read();
59.   if(viol_ptr==NULL && violate.num_available() > 0) viol_ptr =
violate.read();
60.   if(req_ptr && (viol_ptr==NULL || req_ptr->simulation_time <= viol_ptr-   >
61. simulation_time)) exec_request();
62.     else if(viol_ptr && (req_ptr==NULL || viol_ptr->simulation_time <=
req_ptr-  63. > simulation_time)) exec_violate();
64. } SC_CTOR(State_Machine_RM){{SC_THREAD(p)}};
```

No capítulo 4, apresentou-se o projeto DigiSeal que possui cinco *testbenches*: FEC, EEC, DES, Máquina de Estados e do bloco Principal. A partir da análise dos seus *testbenches*, percebe-se que dois deles necessitam do tratamento de suas transações temporais: o da Máquina de Estados com as FIFOs Request e Violate e do bloco Principal do DigiSeal que engloba a funcionalidade de todos os blocos com as FIFOs FRrequest e Violate. Nesta seção, foi apresentada a implementação do mecanismo para transações temporais assíncronas com a metodologia de verificação funcional VeriSC para o *testbench* da Máquina de Estados. Este mecanismo funciona de forma análoga para o *testbench* do bloco Principal do DigiSeal, visto que, a exemplo do *testbench* da Máquina de Estados, ele também apresenta apenas duas FIFOs de entrada que necessitam do processo sincronização (ver o código do *testbench* do bloco Principal no Apêndice A). Sabendo-se que o DigiSeal possui cinco *testbenches* e em dois deles ocorrem transações assíncronas, pode-se afirmar que em 40% dos *testbenches* do projeto DigiSeal houve a necessidade da utilização do mecanismo proposto neste trabalho.

Neste capítulo, apresentou-se o modelo de transações temporais assíncronas criado para a metodologia de verificação funcional VeriSC, seguido da implementação deste modelo no estudo de caso DigiSeal. No próximo capítulo, tem-se o resumo das soluções TTLM com as outras metodologias estudadas.

Capítulo 6

Trabalhos Relacionados

Neste capítulo, apresenta-se algumas formas de implementação de transações temporais utilizadas nas principais metodologias de verificação funcional: UNISIM, AVM, VMM e IPCM, com o objetivo de se fazer uma análise comparativa das técnicas existentes.

6.1. Transações Temporais com UNISIM

Na metodologia de verificação funcional UNISIM, os módulos no nível TLM correspondem principalmente a modelos funcionais. Esses modelos normalmente não envolvem transações temporais, assim eles não são usados para avaliação de desempenho.

Na Figura 16, representa-se um pequeno simulador composto por dois módulos no nível TLM: O módulo Inicial que direciona a simulação e o módulo Final que responde as chamadas do módulo Inicial [Pérez 2006].

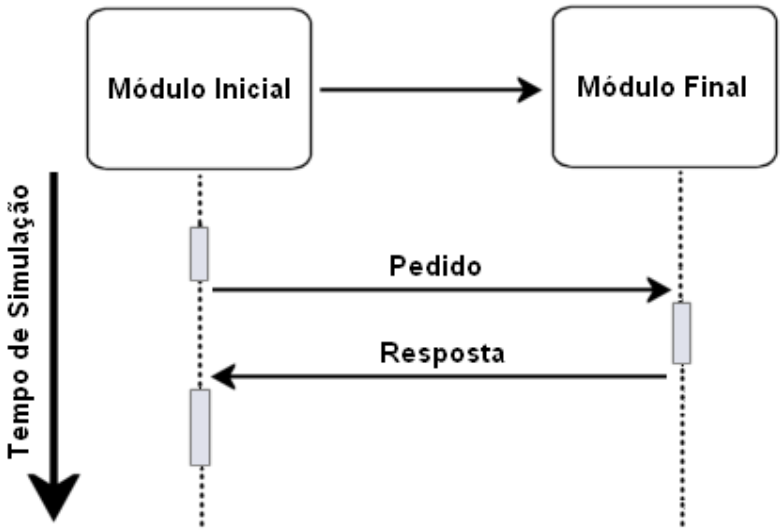

Figura 16: Esquema de simulação TLM entre dois módulos.

A execução dos passos se dá da seguinte forma [Pérez 2006]: o módulo Inicial é o primeiro a ser executado durante a simulação, em seguida, ele envia um pedido ao módulo Final e fica à espera de uma resposta.

O módulo Final fica à espera de um pedido e ao recebê-lo faz o processamento das informações para, em seguida, enviar o resultado de volta para o módulo Inicial, que ao receber a reposta do módulo Final pode continuar o processo de simulação.

6.1.1. Módulos Untimed TLM com UNISIM

Modelos *Untimed* TLM na metodologia UNISIM são projetados usando a Interface TLM. Em um modelo *Untimed* TLM (UTLM), os atrasos para atender ao pedido não são considerados. No processo de simulação de circuitos escritos em linguagens HDL pode haver vários ciclos de simulação antes do tempo de simulação avançar. Estes ciclos que ocorrem enquanto o tempo de simulação não avança são chamados ciclos delta [Umamageswaran 1999]. Na Figura 17, apresenta-se uma situação em que o tempo simulado tende para zero e só os ciclos delta ocorrem durante a simulação [Pérez 2006].

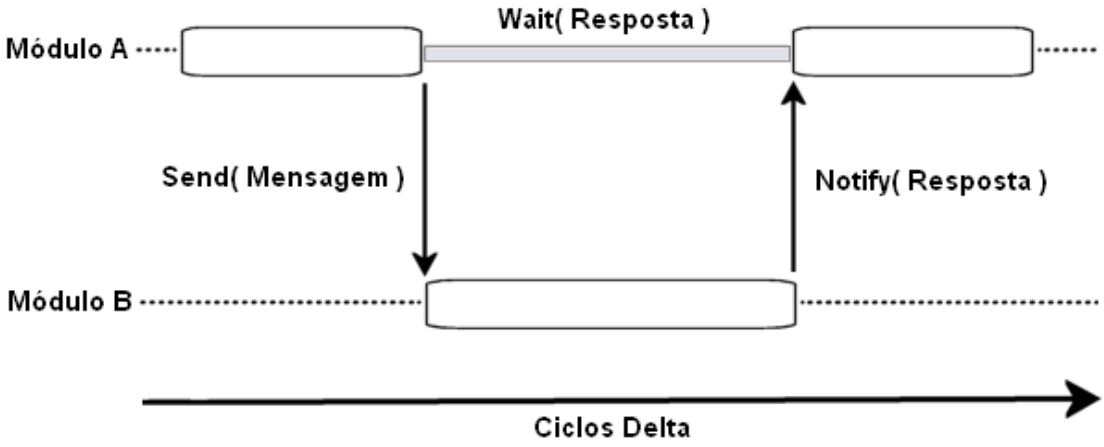

Figura 17: Esquema de simulação Untimed TLM entre dois módulos.

O módulo A envia um pedido ao módulo B utilizando o método *Send()*. Em seguida, o módulo A fica esperando por meio do método *Wait()*. O módulo B envia a resposta ao módulo A fazendo uso do método Notify().

6.1.2. Módulos Timed TLM com UNISIM

Os módulos *Timed* TLM na metodologia UNISIM são projetados usando a Interface TLM. Ao utilizar o modelo *Timed* TLM (TTLM) algumas informações de tempo são adicionadas às mensagens durante a comunicação. A idéia é tirar proveito do método *wait()*, conforme mostrado na Figura 18 [Pérez 2006]:

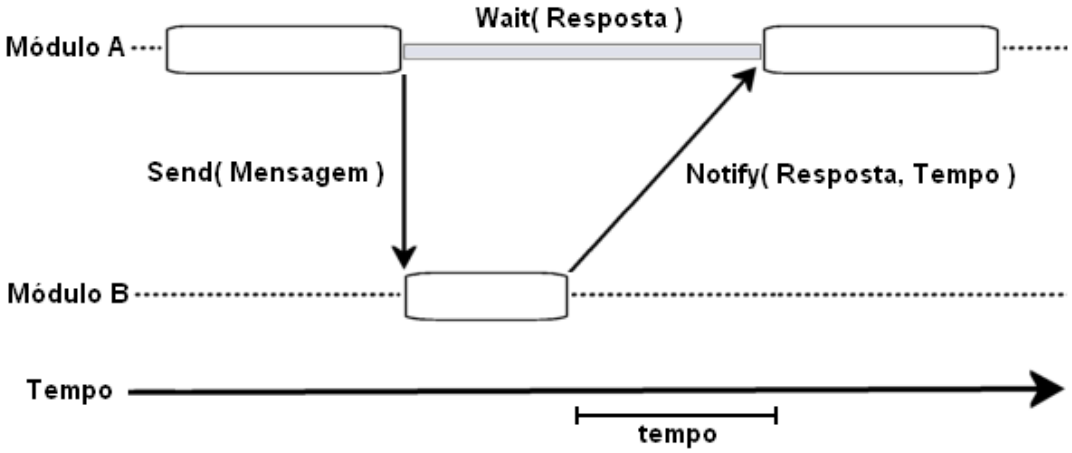

Figura 18: Esquema de simulação Timed TLM entre dois módulos.

Primeiro, o módulo A envia um pedido ao módulo B com o método *Send()*. Em seguida, o módulo B responde ao módulo A utilizando o método *Notify(),* que tem a função de adicionar informação de tempo a esta resposta. Fixado o valor da variável de tempo, o módulo B informa ao módulo A que tempo é necessário para enviar a sua resposta. Graças ao método W*ait()*, o módulo A só usará a resposta depois de alcançado o tempo pré-estabelecido. Em outra situação, o módulo B pode ativar o evento de resposta assim que receber o método Send do módulo A sem que ocorram atrasos ou durante o envio do pedido pelo módulo A, ou ainda, em outro delta ciclo pode ocorrer o envio de um pedido originado de um módulo C, antes que o módulo B tenha ativado a resposta do módulo A. Neste caso, o módulo C ficará esperando até que possa ser atendido pelo módulo B [Pérez 2006].

Note-se que a diferença básica entre o seu modelo UTLM e TLLM da metodologia UNISIM é a modificação da chamada ao método Notify que agora indica o tempo que o módulo Final precisa para dar um conjunto de respostas, ou seja o valor do tempo de atraso destas respostas [Pérez 2006].

6.2. Transações Temporais com AVM

Na metodologia de verificação funcional AVM, os *testbenches* são organizados em forma de camadas de componentes, conforme mostrado na Figura 19. Componentes são caixas pretas que se conectam e se comunicam apenas por meio de interfaces. Eles são escritos em linguagens HDL, SystemC, SystemVerilog, C++, entre outras [Mentor 2008]. Existem vários tipos de componentes de verificação que são utilizados pela metodologia AVM: Controle, Análise, Operacional, transacionais, DUT.

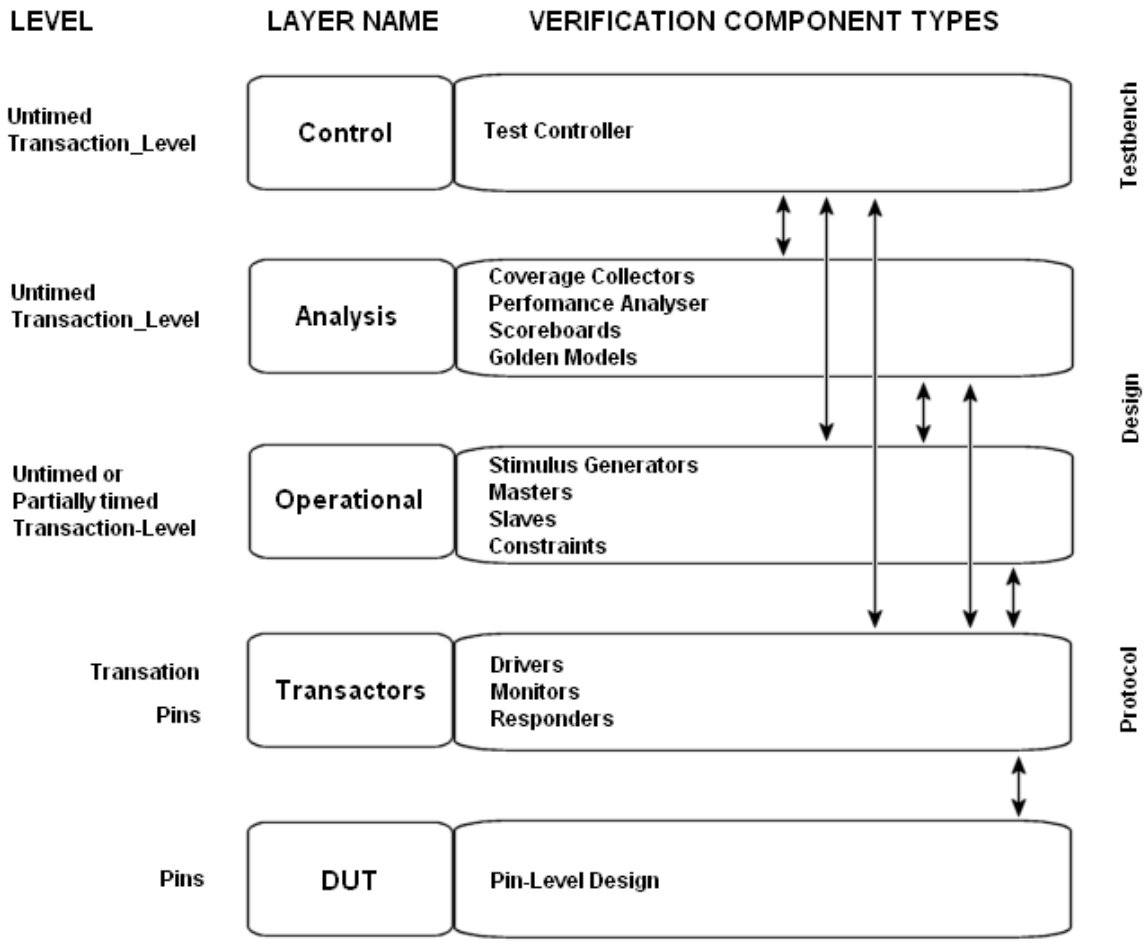

Figura 19: Camadas da arquitetura do *testbench* com AVM.

A partir da Figura 19, nota-se que existem camadas que lidam com transações temporais e/ou não temporais nesta metodologia. Para implementar os *testbenches* existentes em cada uma destas camadas, a metodologia AVM dispõe da biblioteca AVM que possui um conjunto de classes que facilitam a sua construção [Mentor 2008].

Além das interfaces unidirecionais, a metodologia AVM provê interfaces bidirecionais entre os seus componentes, conforme mostrado na Figura 20.

Figura 20: Interface bidirecional entre dois componentes.

As interfaces bidirecionais permitem que as transações sejam enviadas a partir do módulo A para o módulo B e, em seguida, possa voltar do módulo B para o módulo A. Normalmente, usa-se este tipo de transação para modelos que necessitem de uma solicitação seguida por uma resposta [Mentor 2008].

Existem também outros tipos de interfaces que são utilizadas em processos não temporais, que são chamadas de interface de bloqueio, pois elas esperam pacientemente até que a operação solicitada esteja completa, não importando quanto tempo isso possa levar, logo elas são úteis para operar entre dois componentes síncronos. Estas interfaces são implementadas com o uso do método wait() do SystemC.

Além deste tipo de interface, tem-se também as chamadas interfaces não bloqueantes, onde as chamadas são retornadas imediatamente, retornando no mesmo ciclo de tempo em que foi emitido. A implementação desta interface não permite o uso do método *wait()*. Este tipo de interface é bastante útil em comunicações assíncronas [Mentor 2008]. A metodologia de verificação funcional AVM faz uso de interfaces unidirecionais e bidirecionais não bloqueantes para implementar as suas transações assíncronas utilizando a linguagem SystemVerilog. Quando utiliza a linguagem SystemC, a metodologia AVM usa interfaces TLM com *timestamp*s para construir os seus Modelos de Referência temporais, podendo interagir com outras interfaces TLM. Isso possibilita a sua interação com transações vindas de vários canais [Mentor 2008].

6.3. Transações Temporais com VMM

A metodologia de verificação funcional VMM utiliza a linguagem SystemVerilog e possui uma interface própria para tratar transações assíncronas chamada de *timing* interface, cujo esquema é mostrado na Figura 21 [Bergeron 2005].

Figura 21: Interface de Notificação.

Na metodologia VMM, *transactor* é o termo utilizado para identificar componentes do ambiente de verificação que servem de interface entre dois níveis de abstrações para um

determinado protocolo ou para gerar protocolos de transações. Exemplo: *Drivers,* Monitores, *Checker* [Bergeron 2005].

No esquema da Figura 21, as informações temporais em transações assíncronas podem ser trocadas entre dois *transactors* por meio de uma instância de uma interface de serviço de notificação. Com um canal instanciado, uma única instância é compartilhada nos dois *transactors.* Um *transactor* produz indicações de notificação, enquanto o outro espera por informações relevantes. Um canal é utilizado em paralelo para transferir informações de qualquer transação, desde que ela esteja completa [Bergeron 2005].

Os *testbenches* que possuem transações temporais devem ser capazes de verificar o tempo da resposta de um pedido. Com SystemVerilog isto pode ser feito comparando *timestamps.* Como não é interessante que a variável temporal da transação modifique a informação a ser enviada entre os dois *transactors,* SystemVerilog permite estender o objeto original para adicionar a variável temporal, deixando o objeto original com a sua informação intacto [Bergeron 2003]. Além disso, os canais de notificação da metodologia VMM possuem *timestamps* associados a eles que podem ser utilizados para identificar o tempo total de execução de uma transação [Bergeron 2005].

A seguir, algumas regras básicas que são utilizadas pela metodologia VMM no código fonte do esquema mostrado na Figura 21 [Bergeron 2005].

- A classe *vmm_notify* deve ser utilizada para a troca de notificações entre dois *transactors.* A extensão da classe vmm_notify define todas as notificações que podem ser trocadas entre os dois *transactors* [Bergeron 2005].
- A classe *vmm_channel* possui um mecanismo que pode ser usado para implementar uma interface flexível que permite múltiplas transações em um canal.
- As referências ao serviço de notificação das instâncias devem ser armazenadas em uma classe com propriedade pública. Esta estrutura permite que a conexão entre dois *transactors* seja feita em uma ordem arbitrária. A primeira cria o serviço de notificação da instância. Em seguida, a segunda utiliza a referência à instância na primeira [Bergeron 2005].
- O canal de notificação do serviço de instâncias deve ser especificado como opcional nos argumentos do construtor. Assim como os canais, a ligação entre dois *transactors* exige que eles compartilhem uma referência para o mesmo serviço de notificação da instância. Deve ser possível especificar o serviço de notificação de instâncias para se conectar aos argumentos do construtor como opcional. Se não forem especificadas, novas instâncias são atribuídas internamente [Bergeron 2005].

- Um *transactor* não deve realizar uma referência interna a um serviço de notificação de uma instância enquanto ela estiver parada ou reiniciada. Se um *transactor* detém uma cópia da referência a um serviço de notificação de uma instância em uma variável interna, o serviço de notificação da instância não pode ser substituído por outro para modificar a saída ou a entrada de um *transactor* e dinamicamente reconfigurar a estrutura do ambiente de verificação. Embora inevitável durante o funcionamento normal, um *reset* ou uma parada do *transactor* deve liberar todas essas referências internas para deixar que o serviço de notificação da instância seja substituído [Bergeron 2005].

6.4. Transações Temporais com IPCM

A metodologia de verificação funcional IPCM possui um componente metodológico chamado SVM (*System Verification Methodology*) que provê transações temporais utilizando o modelo TLM. A metodologia IPCM com o auxilio do componente metodológico SVM utiliza o modelo PVT (*Programmer's view Timing*) para definir os tempos de atrasos existente no processo em que um módulo A faz a solicitação de um pedido a um módulo B, este pedido será transferido por meio de um barramento para um módulo B que por sua vez, retornará uma resposta ao módulo A por meio do barramento. O tempo de atraso global deste processo será a soma dos tempos de atraso do pedido, do barramento e da resposta. Na metodologia IPCM todos estes atrasos são centralizados no barramento, que é implementado com a utilização do pacote vr_xbus_sc_tlm, que foi desenvolvido a partir do modelo OSCI TLM. Entre outras coisas, este barramento monitora a sincronização dos pedidos e das respostas que estão em andamento, além de possuir um centralizador de atrasos, que centraliza os atrasos dos pedidos e das respostas, e um modelo dinâmico de atrasos que permite ao usuário definir um valor *defaut* de atraso no pedido e na resposta, que será processado durante a sua associação com o barramento, que também possibilita ao usuário definir dinamicamente os valores destes atrasos utilizando o método set_timing_mode() [Cadence SVM 2008].

O modelo para transações temporais da metodologia IPCM gerencia as variáveis temporais de um SoC no nível de microarquitetura, conforme o esquema da Figura 22.

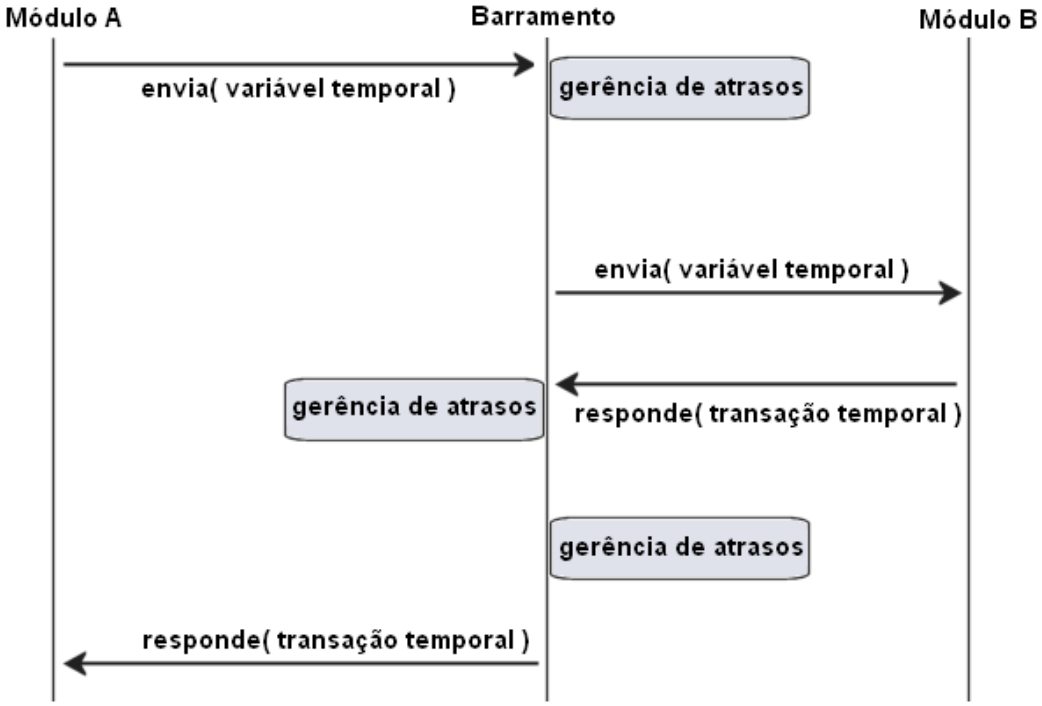

Figura 22: Seqüência de Mensagens *Timed* TLM com IPCM.

Inicialmente, o módulo A envia um pedido que além da mensagem, possui uma transação temporal que corresponde a um valor de atraso definido pelo módulo A para a transação do pedido. Esta transação será capturada por um barramento de dados que fará o gerenciamento dos atrasos gerados pelas transações do pedido, da resposta e do próprio barramento. Em seguida, o módulo B recebe a transação enviada pelo módulo A e após processar os dados recebidos, envia uma resposta ao módulo A. Esta resposta também passará pelo gerenciamento de atrasos do barramento de dados antes de chegar ao módulo A. Se durante este processo, o barramento receber um pedido enviado por um módulo C, solicitando uma reposta do módulo B, será o barramento que também irá gerenciar os atrasos gerados pela espera do módulo C, que ficará esperando até que os processos de pedido e resposta do módulo A terminem, para que ele seja atendido. Neste método o barramento fará a sincronização das transações assíncronas e garantirá que a informação temporal inicialmente enviada não será perdida ao longo de todo o processo de envio e recebimento dos dados [Cadence SVM 2008].

6.5. Análise Comparativa

Esta análise comparativa tem por objetivo destacar as principais características das metodologias UNISIM, VMM, IPCM, AVM e VeriSC, no que diz respeito a implementação de transações temporais assíncronas.

A metodologia UNISIM possui um princípio equivalente de utilização do selo de tempo (*timestamp*) em conjunto com o tempo simulado. Ela também permite o tratamento de transações assíncronas vindas de diferentes canais, mas a forma de implementação do mecanismo é diferente, pois utiliza os métodos *wait*() e *notify*() (ver Seção 6.1).

Já na metodologia VMM, o princípio de utilização de um canal de notificação é uma abordagem diferente do mecanismo proposto neste trabalho. Este canal de notificação se torna necessário porque na linguagem SystemVerilog utilizada por esta metodologia, não existe uma forma simples de realizar, por exemplo, a funcionalidade do método *wait*() que considera vários canais simultâneos. Para isso, a metodologia VMM utiliza o canal de notificação que possui associação com *timestamps* e a classe *vmm_channel* que permite múltiplas transações em um único canal.

A metodologia AVM por possibilitar a utilização de duas linguagens: SystemC e SystemVerilog, possui duas formas para o tratamento das suas transações temporais assíncronas. O princípio usado com SystemC também possui um selo de tempo em conjunto com o tempo simulado para fazer a sincronização. Já com SystemVerilog, ela utiliza interfaces não bloqueantes para sincronizar as suas transações, mas não necessita do selo de tempo. Apesar da sua documentação [Mentor 2008] não mostrar detalhes sobre como deve ser tratado o caso de transações assíncronas vindas por diferentes canais, ela sugere que isso é possível com a metodologia AVM utilizando a interface TLM do SystemC.

A metodologia IPCM possui uma forma diferente para gerenciar suas transações temporais assíncronas, utilizando um barramento que gerencia todos os atrasos ocorridos durante o processo e sincroniza estas transações. Assim, como no mecanismo proposta neste trabalho, ela permite a sincronização de transações vindas de canais diferentes.

Por fim, todas as metodologias estudadas ([Bergeron 2005] [Mentor 2008] [Cadence 2008] [August 2007]) possuem um mecanismo diferente para as suas transações temporais assíncronas. Pode-se afirmar que o mecanismo proposto para resolver transações temporais assíncronas com VeriSC é equivalente aos mecanismos das metodologias de verificação estudadas que também utilizam a linguagem SystemC, visto que este mecanismo além de ser restrito a metodologia VeriSC, também é limitado pela linguagem SystemC, por ser dependente

dos mecanismos disponíveis nesta linguagem. Por outro lado, percebe-se que o mecanismo que melhor soluciona o problema de transações temporais é o utilizado pela metodologia AVM com SystemVerilog por ter a vantagem de não necessitar de selo de tempo, visto que a interface não bloqueante consegue sincronizar as transações sem a utilização de nenhum outro artifício. No Quadro 01, apresenta-se um resumo das principais características encontradas nos mecanismos para transações temporais assíncronas das metodologias estudadas.

Quadro 01: Resumo da análise comparativa das metodologias.

	VeriSC	AVM	IPCM	UNISIM	VMM
Utilização de selo de tempo.	Sim	Sim	Sim	Sim	Sim
Forma de sincronização das transações.	Wait()	Interfaces não bloqueantes	Barramento	Wait() e Notify()	Canal
Permite transações vindas de diferentes canais.	Sim	Sim	Sim	Sim	Sim

Neste capítulo, apresentou-se análise comparativa tem por objetivo destacar as principais características das metodologias de verificação funcional, no que diz respeito à implementação de transações temporais assíncronas.. No próximo capítulo, têm-se as considerações finais deste trabalho.

Capítulo 7

Considerações Finais

Neste capítulo, apresenta-se as considerações finais da dissertação e indicam-se algumas sugestões para trabalhos futuros para complementar o trabalho descrito ao longo deste documento.

7.1. Contextualização geral da dissertação

O objetivo geral deste trabalho de dissertação foi desenvolver uma forma de implementação de transações temporais assíncronas na metodologia de verificação funcional VeriSC. A princípio, fez-se necessário adquirir conhecimento sobre o estado da arte no tocante ao uso de técnicas de verificação com transações temporais nas principais metodologias de verificação funcional existentes na literatura. Então, uma revisão bibliográfica foi realizada, assim como o levantamento dos dados necessário para este trabalho.

Após a revisão bibliográfica, os modelos de transações temporais existentes foram estudados e foi então possível a inserção do modelo de transações temporais na metodologia VeriSC, que foi testado no estudo de caso DigiSeal, que é um circuito destinado à detecção da violação de dispositivos instalados em redes aéreas de distribuição de energia elétrica. Por fim, foram feitas análises comparativas entre as diversas formas de implementação de transações temporais em outras metodologias, a fim de destacar as principais características das metodologias UNISIM, VMM, IPCM, AVM e VeriSC, no que diz respeito a implementação de transações temporais assíncronas.

7.2. Contribuições da Pesquisa

O trabalho desenvolvido consiste da criação de um mecanismo para resolver transações temporais assíncronas para a metodologia de verificação funcional VeriSC, utilizando como estudo de caso um projeto destinado à detecção da violação de dispositivos instalados em redes aéreas de distribuição de energia elétrica que foi solicitado pela empresa Light Serviços de Eletricidade S.A, além da comparação do mecanismo proposto com outros existentes na

literatura (VMM [Bergeron 2005], AVM [Mentor 2008], IPCM [Cadence 2006] e UNISIM [August 2007]).

Ao término deste trabalho, pode-se dizer que as atividades desenvolvidas foram bem sucedidas, pois as principais metas definidas inicialmente foram cumpridas com sucesso dentro do prazo.

7.3. Conclusões

Ao término deste trabalho, pode-se afirmar que o estudo de caso DigiSeal foi realizado com sucesso, visto que após a execução dos procedimentos da metodologia VeriSC, o circuito do Lacre digital foi testado em uma FPGA, funcionando na primeira tentativa.

Vale ressaltar que VeriSC foi a metodologia escolhida para fazer a verificação funcional do estudo de caso proposto, por ser possível a criação do ambiente de verificação (testbench) antes da implementação do circuito (DUV). Após a constatação de que VeriSC não possuía nenhum mecanismo para fazer o tratamento das transações temporais assíncronas, que era necessário a implementação do DigiSeal, surgiu a necessidade da criação de um mecanismo de transações temporais assíncronas com VeriSC.

Após o estudo das outras metodologias ([Bergeron 2005] [Mentor 2008] [Cadence 2008] [August 2007]), percebeu-se que cada uma delas possui um mecanismo diferente para as suas transações temporais assíncronas. Podendo-se dizer que o mecanismo proposto para resolver transações temporais assíncronas com VeriSC é equivalente aos mecanismos das metodologias de verificação estudadas que também utilizam a linguagem SystemC, visto que este mecanismo além de ser restrito a metodologia VeriSC, também é limitado pela linguagem SystemC, por ser dependente dos mecanismos disponíveis nesta linguagem.

7.4. Sugestões para Trabalhos Futuros

Devido ao fato do mecanismo para transações temporais assíncronas com SystemVerilog na metodologia AVM ter se mostrado bastante interessante por não necessitar de selo de tempo, visto que as suas interfaces não bloqueantes conseguem resolver todo o problema de sincronização das transações, aliado ao crescente empenho das principais metodologias de verificação (AVM, VMM, IPCM) em utilizar a linguagem SystemVerilog na verificação dos seus circuitos digitais. Percebe-se a necessidade da metodologia VeriSC também se atualizar para não ficar ultrapassada em relação as estas outras metodologias.

Têm-se como sugestões para trabalhos futuros:

- A migração da metodologia VeriSC da linguagem SystemC para SystemVerilog, incluindo a adaptação do modelo de transações temporais assíncronas para a linguagem SystemVerilog.
- A criação de um mecanismo que propicie a existência de interfaces bidirecionais que também permita transações temporais assíncronas na metodologia VeriSC.
- Extensão da abrangência do mecanismo para transações temporais assíncronas proposto, a fim de poder utilizá-lo em muitos outros projetos de circuitos digitais em que a metodologia VeriSC esteja sendo utilizada e em que seja necessária a sincronização de transações assíncronas.

Referências Bibliográficas

[Altera 2008] Altera Corporation**.** Disponível em: **<**http://www.altera.com>. Acesso em 11 mar. 2008.

[August 2007] AUGUST, D. et al. UNISIM: An Open Simulation Environment and Library for Complex Architecture Design and Collaborative Development. IEEE Computer Architecture Letters, 2007.

[Bergeron 2003] BERGERON, J. **Writing Testbenches: Functional verification of hdl models**. 2nd Edition. Kluwer Academic Publisher, Boston, USA, 2003.

[Bergeron 2005] BERGERON, J. et al. **Verification Methodology Manual for SystemVerilog**. 1nd Edition, Synopsys Inc, EUA, 2005.

[Bergeron 2006] BERGERON, J. et al. SystemVerilog Reference Verification Methodology: VMM Adoption. **EETimes: Design News**, 2006.

[Bhasker 2002] BHASKER, J. **A SystemC Primer**. Star Galaxy Publisher, 1nd Edition, 2002.

[Black 2004] BLACK, D. and DONOVAN, J. **SystemC: From the ground up**. Kluwer Academic Publishers. New York, EUA, 2004.

[Brahme 2000] BRAHME, S. et al. The transaction-based verification methodology. Technical Report CDNL-TR-2000-0825, Cadence Berkeley Labs, 2000.

[Burnett 2002] BURNETT, R. and PAINE, S. **Criptografia e Segurança - O guia oficial RSA**. Editora Campus, São Paulo, SP, 2002.

[Cadence 2006] CADENCE. Incisive Plan-to-Closure Methodology Overview. Cadence Design Systems, Inc 2006.

[Cadence 2008] CADENCE. Incisive Plan-to-Closure Methodology Introduction. Cadence Design Systems, Inc 2008

[Cadence ABV 2008] CADENCE. Incisive® Plan-to-Closure Methodology, Assertion-Based Verification (Early Access). Cadence Design Systems, Inc 2008.

[Cadence eRM 2008] CADENCE. Incisive® Plan-to-Closure Methodology, *e* Reuse Methodology (eRM) Developer Manual. Cadence Design Systems, Inc 2008.

[Cadence OVM 2008] CADENCE. Incisive® Plan-to-Closure Methodology, Open Verification Methodology (OVM) for SystemVerilog User Guide. Cadence Design Systems, Inc 2008.

[Cadence SVM 2008] CADENCE. Incisive® Plan-to-Closure Methodology, System Verification Methodology (SVM). Cadence Design Systems, Inc 2008.

[Cadence TBA 2006] CADENCE. Incisive® Plan-to-Closure Methodology, Incisive XE TBA Transaction Modeling Guide. Cadence Design Systems, Inc 2006.

[Cadence URM 2008] CADENCE. Incisive® Plan-to-Closure Methodology, Introduction to Universal Reuse Methodology (URM). Cadence Design Systems, Inc 2008.

[Cai 2003] CAI, L. and GAJSKI, D. Transaction level modeling in system level design. Technical report, Center for Embedded Computer Systems, University of California, EUA, 2003.

[Caldari 2003] CALDARI, M. et al. Transaction-Level Models for AMBA Bus Architecture using SystemC 2.0. In: DATE'03 - DESIGN, AUTOMATION AND TEST IN EUROPE, Munich, Germany, 2003.

[Carro 2003] CARRO, L. and WAGNER, F. R. Sistemas Computacionais Embarcados. In: JAI'03 – XXII JORNADAS DE ATUALIZAÇÃO EM INFORMÁTICA, Campinas, SP, 2003.

[Donlin 2004] DONLIN, A. Transaction Level Modeling: Flows and use Models. In: CODES ISS' 04, Stockholm, Sweden, 2004.

[Flóres 2006] FLÓRES, M. J. S. **Estimativa de desempenho de uma NoC a partir de seu modelo SystemC TLM**. 172 f. Dissertação (Mestrado em Microeletrônica) – Escola Politécnica da USP, Universidade de São Paulo, São Paulo, SP, 2006.

[Ghenassia 2005] GHENASSIA, F. and MAILLET-CONTOZ, L. **Transaction Level Modeling with SystemC TLM Concepts and Aplications for Embedded Systems**. Chapter 2. Springer, 2005.

[IEEE 1993] IEEE Institute of Electrical and Electronics Engineers. IEEE standard VHDL language reference manual. **IEEE Standards Office**, New York, NY, USA, 1993.

[IEEE 1996] IEEE Institute of Electrical and Electronics Engineers. IEEE Standard Description Language Based on the VERILOG Hardware Description Language. **IEEE Standards Office**, New York, NY, USA, 1996.

[Lavagno 2006] LAVAGNO, L.; MARTIN, G.; SHEFFER, L. **Electronic Design Automation for Integrated Circuits**. Handbook, t. CRC Press, Inc., Boca Raton, Florida, USA, 2006.

[Mentor 2008] Mentor Graphics. Advanced Verification Methodology Cookbook. Disponível em: <http://www.mentor.com/products/fv/_3b715c/>. Acesso em 16 mar. 2008.

[Micheli 2002] MICHELI, G. and BENINI, L. Networks-on-Chip: A New Paradigm for Systems-on-Chip Design. In: DATE'02 – DESIGN, AUTOMATION AND TEST IN EUROPE, France, 2002. Proceedings, IEEE Computer Society Press, 2002.

[Midorikawa 2007] MIDORIKAWA, T. E. Uma Introdução às Linguagens de Descrição de Hardware. Escola Politécnica da USP - Universidade de São Paulo – USP, São Paulo, SP, 2007.

[Moraes 2004] MORAES, F., et. al.. Dynamic and Partial Reconfiguration in FPGA SoCs: Requirements Tools and a Case Study. In: ROSENSTIEL, Wolfgang. Reconfigurable Computing. New York, USA, 2004.

[Morelos-Zaragoza 2002] MORELOS-ZARAGOZA, R. H. **The Art of Error Correcting Coding.** John Wiley & Sons, 2002.

[Moussa 2003] MOUSSA, I.; GRELLIER, T.; NGUYEN, G. Exploring SW performance using SoC Transaction level modeling. In: DATE'03 - DESIGN, AUTOMATION AND TEST IN EUROPE, Munich, Germany, 2003.

[Munden 2005] MUNDEN, R. Asic and FPGA Verification: A guide to component modeling. ELSEVIER, 2005.

[Pasricha 2002] PASRICHA, S. Transaction level modeling of SOC with SystemC 2.0. In: SYNOPSYS USERS GROUP CONFERENCE – SNUG 2002. India, 2002.

[Pérez 2006] PÉREZ, D. G.; MOUCHARD, G.; CARLOGANU, A. UNISIM Methodology for Transaction Level Modeling, 2006. Disponível em: <http://unisim.org/site/>. Acesso em: 21 set. 2008.

[Pessoa 2007] PESSOA, I. M. **Geração semi-automática de Testbenches para Circuitos Integrados Digitais**. 52 f. Dissertação (Mestrado em Sistemas Distribuídos) – Universidade Federal de Campina Grande – UFCG, Campina Grande, PB, 2007.

[Piziali 2004] PIZIALI, A. **Functional verification Coverage Measurement and Analysis**. Kluwer Academic Publishers, Boston, USA, 2004.

[Randjic 2002] RANDJIC, A. et al. Complex ASICs verification with SystemC. In: INTERNATIONAL CONFERENCE ON MICROELECTRONICS, Nis, Yugoslavia, 2002.

[Rocha 2007] ROCHA, A. K. O. and MELCHER, E. U. K. DigiSeal - An application using Functional Verification with timed transactions. In: EMBEDDED WORLD 2007, Nuernberg, Germany, 2007.

[Silva 2004] SILVA, K. R. G.; MELCHER, E. U. K.; ARAUJO, G. An Automatic Testbench Generation Tool for a SystemC Functional Verification Methodology. In: SBCCI 2004 – 17th SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEM DESIGN, Porto de Galinhas, PE, 2004.

[Silva 2007] SILVA, K. R. G. **Uma Metodologia de Verificação Funcional para Circuitos Digitais**. 2007. 119 f. Tese (Doutorado em Microeletrônica) - Universidade Federal de Campina Grande, Campina Grande, PB, 2007.

[Stinson 2002] STINSON, D. **Cryptography Theory and Practice**. 2nd Edition, CRC Press, 2002.

[Texas 2002] TEXAS, Instruments Incorporated. Data Sheet - Single Chip RF Transceiver TRF 5901. Mailing Address P.O. Box 655303 Dallas, Texas 75265. Disponível em: <http://www.ti.com>. Acesso em: 05 dez. 2002.

[Unisim 2008] UNISIM: *UNIted SIMulation environment*. Disponível em: <https://unisim.org/tutorial/may_2007_tutorial/html/basics/Basics%20of%20UNISIM.html>. Acesso em: 02 mar. 2008.

[Umamageswaran 1999] UMAMAGESWARAN, K.; PANDEY, S. L.; WILSEY, P. **A. Formal Semantics and Proof Techniques for Optimizing VHDL Models**. Kluwer Academic Publishers. Springer, 1999.

[Usselmann 2008] USSELMANN, R. DES/Triple DES IP Cores. Disponível em: <http://www.opencores.org/projects.cgi/web/des/overview>. Acesso em: 15 jan. 2008.

[Wagner 2005] WAGNER, I.; BERTACCO, V.; AUSTIN, T. Stresstest: An automatic approach to test generation via activity monitors. In: DESIGN AUTOMATION CONFERENCE, Anaheim, CA, 2005.

[Wile 2005] WILE, B.; GOSS, J. C.; WOLFGANG, R. Comprehensive Functional Verification: The Complete Industry Cycle. ELSEVIER, 2005.

[Wolf 2001] WOLF, W. **Computers as Components**. McGraw-Hill, 2001.

Apêndice - A

A.1. Código Fonte da Máquina de Estados

A seguir, será apresentado o código fonte de todos os componentes do *testbench* da Máquina de Estados do DigiSeal que possui a implementação do mecanismo para sincronização das suas transações assíncronas proposta neste trabalho.

Source.h

```
class viol_constraint_class: public scv_constraint_base {
  scv_bag<pair<unsigned int, unsigned int> >
simulation_time_distrib ;
 public:
  scv_smart_ptr<Violate> viol_sptr;
  SCV_CONSTRAINT_CTOR(viol_constraint_class) {
    // simulation_tima aqui é na verdade um intervalo de tempo
    simulation_time_distrib.push(pair<int,int>(1, 36), 100);
    viol_sptr->simulation_time.set_mode(simulation_time_distrib);
  }
};
class Req_constraint_class: public scv_constraint_base {
  scv_bag<pair<unsigned int, unsigned int> >
simulation_time_distrib;
  scv_bag<pair<int,int> > id_distrib;
  scv_bag<pair<int,int> > time_distrib;
  scv_bag<pair<int,int> > comando_distrib;
  int i;

 public:
  scv_smart_ptr<Request> req_sptr;
  SCV_CONSTRAINT_CTOR(Req_constraint_class) {
  // simulation_time aqui é na verdade um intervalo de tempo
  // não pode ser menos do que 15 ms para dar tempo o request ir e
o reply voltar.
  simulation_time_distrib.push(pair<int,int>(15000, 40000), 100);
  req_sptr->simulation_time.set_mode(simulation_time_distrib);
  id_distrib.push(pair<int,int>(1024, 1024), 90);
  id_distrib.push(pair<int,int>(0,9000), 10);
  req_sptr->id.set_mode(id_distrib);
  time_distrib.push(pair<int,int>(0, 300), 100);
  req_sptr->timeStamp.set_mode(time_distrib);
  comando_distrib.push(pair<int,int>(0, 3), 100);
  req_sptr->comando.set_mode(comando_distrib);
  }
};
```

```
SC_MODULE(source) {

  sc_fifo_out <Request *> req_resposta;
  sc_fifo_out <Request *> req_resposta_duv;
  Req_constraint_class req_constraint;

  sc_fifo_out <Violate *> viol_resposta;
  sc_fifo_out <Violate *> viol_resposta_duv;
  viol_constraint_class viol_constraint;

  void proc () {
    ifstream sfi("stimuli.txt");
    string type;
    unsigned int last_simulation_time = 0;
    int last_sensor = FECHADO;

    // Vetores de teste lidos de arquivo
    while(!sfi.fail() && !sfi.eof()) {
      sfi >> type;

      if(type=="Violate") {
        Violate vio;
        sfi >> vio;
        last_simulation_time = vio.simulation_time;
        last_sensor = vio.sensor;
        viol_resposta.write(new Violate(vio));//preenche a
transacao de saida da violacao
        viol_resposta_duv.write(new Violate(vio));//preenche a
transacao de saida da violacao
      } else if(type=="request") {
        Request req;
        sfi >> req;
        last_simulation_time = req.simulation_time;
        req_resposta.write(new Request(req));//preenche a
transacao de saida do request
        req_resposta_duv.write(new Request(req));//preenche a
transacao de saida do request
      }

      sfi.ignore(225, '\n');
    }

// Vetores de teste gerados
    while(1) {
      type = rand() % 3 == 0 ? "Violate" : "request";

      if(type=="Violate") {
        viol_constraint.next();
        Violate vio = viol_constraint.viol_sptr.read();
```

```
      // garante que o simulation time cresce sempre
        vio.simulation_time += last_simulation_time;
        last_simulation_time = vio.simulation_time;
        // um evento de violação não pode se repetir
        if(last_sensor==FECHADO) last_sensor=ABERTO; else
last_sensor=FECHADO;
        vio.sensor = last_sensor;
        viol_resposta.write(new Violate(vio));
        viol_resposta_duv.write(new Violate(vio));
      } else if(type=="request") {
        req_constraint.next();
        Request req = req_constraint.req_sptr.read();
        // garante que o simulation time cresce sempre
        req.simulation_time += last_simulation_time;
        // para evitar um erro espurio do timestamp
        if((req.simulation_time % 1000) < 400) req.simulation_time
+= 400;
        last_simulation_time = req.simulation_time;
        // em 90% dos casos o timeStamp é correto
        if(rand() % 10 > 0) req.timeStamp = req.simulation_time /
TIME_STAMP_PER_SIMULATION_TIME;
        req_resposta.write(new Request(req));//preenche a transacao
de saida do request
        req_resposta_duv.write(new Request(req));//preenche a
transacao de saida do request
      }
    }
  }


SC_CTOR(source):req_constraint("req_constraint"),viol_constraint("v
iol_constraint")  {
    SC_THREAD(proc);
  }
};
```

Modref.h

```
SC_MODULE(modref) {

  sc_fifo_in <Violate *> viol_estimulo;
  sc_fifo_in <Request *> req_estimulo;
  sc_fifo_out <Reply *> rep_resposta;

  // variáveis a serem setados antes de iniciar a simulação
  long long chave;//valor da chave de critografia
  int id;//valor do id do lacre

  KitLacreTeste *lacre;
  Reply rep;
  bool verif;

  Request *req_ptr;
  Violate *viol_ptr;

  void violate() {
    lacre->setTime(viol_ptr-
>simulation_time/TIME_STAMP_PER_SIMULATION_TIME);
    lacre->doSensor(viol_ptr);
    delete viol_ptr;
    viol_ptr=NULL;
  }

void request() {

    lacre->setTime(req_ptr-
>simulation_time/TIME_STAMP_PER_SIMULATION_TIME);
    if(lacre->getId() == req_ptr->id){
      lacre->reply_request(req_ptr,&rep);
      rep.simulation_time = req_ptr->simulation_time;
      rep_resposta.write(new Reply(rep));//preenche a transação de
saida
    }
    delete req_ptr;
    req_ptr=NULL;
  }

void proc () {

    lacre = new KitLacreTeste(chave, id);
    req_ptr=NULL;
    viol_ptr=NULL;
```

```
    while(1) {
      wait(viol_estimulo.data_written_event() |
req_estimulo.data_written_event());
      while(req_ptr || viol_ptr || req_estimulo.num_available() +
viol_estimulo.num_available() > 0) {
        if(req_ptr==NULL && req_estimulo.num_available() > 0)
req_ptr = req_estimulo.read();
        if(viol_ptr==NULL && viol_estimulo.num_available() > 0)
viol_ptr = viol_estimulo.read();

        if(req_ptr && (viol_ptr==NULL || req_ptr->simulation_time
<= viol_ptr->simulation_time))
          request();
        else if(viol_ptr && (req_ptr==NULL || viol_ptr-
>simulation_time <= req_ptr->simulation_time))
          violate();
      }
    }
  }

  SC_CTOR(modref) {
    SC_THREAD(proc);
  }
};
```

Checker.h

```
SC_MODULE(checker) {


  sc_fifo_in<Reply *> rep_modref;
  sc_fifo_in<Reply *> rep_duv;

  sc_signal<unsigned int> error_count;


  void proc () {
    unsigned int trans_count=0;

    Reply *rep_mr_ptr, *rep_duv_ptr;

    while(1) {
      rep_mr_ptr = rep_modref.read();
      rep_duv_ptr = rep_duv.read();


      if( !(*rep_mr_ptr == *rep_duv_ptr) ) {
        char title[200], msg[2000];
        ofstream ts(title);
        ts << "rep Transaction " << trans_count << ends;
        ofstream ms(msg);
        ms << "expected: " << *rep_mr_ptr << endl
           << "    received: " << *rep_duv_ptr << ends;
        SCV_REPORT_ERROR(title,msg);
        error_count = error_count.read()+1;
      }
      trans_count++;
    }
  }

  SC_CTOR(checker)
    { SC_THREAD(proc); }
};
```

Driver_Req.h

```
   SC_MODULE(driver_req) {

  sc_fifo_in <Request *> req_estimulo;

  sc_in <bool> clk;
  sc_in <bool> reset;

  //declaracao dos sinais de saida
  sc_out <bool> req_valid;//serve para validação fica ativo durante
1 ciclo de clock
                          //e fica em zero durante pelo menos 16
ciclos de clock
  sc_out <sc_uint<30> > id;
  sc_out <sc_uint<3> > comando;
  sc_out <sc_uint<23> > timeStamp;

  Request *req_ptr;
  scv_tr_stream stream;
  scv_tr_generator < Request, bool > gen;

  void p() {

    id = 0;
    comando = 0;
    timeStamp = 0;
    wait();
    ESPERE(!reset);

int w;
    while(1) {

      req_ptr = req_estimulo.read();
      sc_time trans_time(req_ptr->simulation_time, SC_US);
      sc_time agora = sc_time_stamp();
      if( trans_time > agora ) wait( trans_time - agora );
      wait();

scv_tr_handle h = gen.begin_transaction(*req_ptr);

      id = req_ptr->id;
      comando = req_ptr->comando;
      timeStamp = req_ptr->timeStamp;
      req_valid = 1;
      wait();
      req_valid = 0;
      for(w=0; w<16; w++) wait();
```

```
       // id = rand();
      // comando = rand();
      // timeStamp = rand();

      gen.end_transaction(h);
      delete req_ptr;

    }

  }

  SC_CTOR(driver_req):
  stream(name(), "Transactor"),
    gen("driver_req", stream)
    { SC_THREAD(p); sensitive << clk.pos(); }
};
```

Monitor_Req.h

```
  SC_MODULE(monitor_req) {

  sc_fifo_out <Request *> req_resposta;

  sc_in <bool> clk;
  sc_in <bool> req_valid;
  sc_in <sc_uint<30> > id;
  sc_in <sc_uint<3> > comando;
  sc_in <sc_uint<23> > timeStamp;

  Request req;

  scv_tr_stream stream;
  scv_tr_generator <bool, Request> gen;

  void p() {

    while(1) {

        if(req_valid){
        scv_tr_handle h =  gen.begin_transaction ();

        req.simulation_time = sc_time_stamp().value() /
SIMULATION_TIME_SC_TIME;

        req.id = id.read();
        req.timeStamp = timeStamp.read();
        req.comando = comando.read();

        wait();

        gen.end_transaction(h,req);
        req_resposta.write(new Request(req));
        }

      else wait();
    }
  }


  SC_CTOR(monitor_req):
    stream(name(), "Transactor"),
    gen("monitor_req", stream)
    { SC_THREAD(p); sensitive << clk.pos(); }
};
```

Driver_Vio.h

```
SC_MODULE(driver_vio) {

  sc_fifo_in <Violate *> viol_estimulo;

  sc_in <bool> clk;
  sc_in <bool> reset;

  Violate *viol_ptr;

  //declaracao dos sinais de saida
   sc_out <bool> Violate;//sinal para acionar a violação

  scv_tr_stream stream;
  scv_tr_generator<Violate, bool> gen;

  void p() {

    Violate = 0;
    wait();
    ESPERE(!reset);

    while(1) {

        viol_ptr = viol_estimulo.read();
        sc_time trans_time(viol_ptr->simulation_time, SC_US);
        sc_time agora = sc_time_stamp();
        if( trans_time > agora ) wait( trans_time - agora );

        scv_tr_handle h = gen.begin_transaction(*viol_ptr);

        if(viol_ptr->sensor == ABERTO) {
          Violate = 1;//seta o sinal para Violado
        } else {
          Violate = 0;
        }

        wait();wait();

        gen.end_transaction(h);
        delete viol_ptr;
      }
    }


  SC_CTOR(driver_vio):
    stream(name(), "Transactor"),
    gen("driver_vio", stream)
    { SC_THREAD(p); sensitive << clk.pos(); }
};
```

Monitor_Vio.h

```
SC_MODULE(monitor_vio) {


  sc_fifo_out <Violate *> viol_resposta;

  sc_in<bool> clk;
  sc_in<bool> Violate;//sinal para acionar a violação

  Violate viol;

  scv_tr_stream stream;
  scv_tr_generator< bool, Violate> gen;

  void p() {
    bool Violate_ = 0; // estado de violado no ciclo de clk
anterior

    while(1) {

      if(Violate.read() != Violate_){
        //tratamento da transacao Violado
        scv_tr_handle h = gen.begin_transaction ();
        viol.sensor = Violate.read() ? ABERTO : FECHADO;
        viol.simulation_time = sc_time_stamp().value() /
SIMULATION_TIME_SC_TIME;
        Violate_ = Violate.read();
        wait();
        gen.end_transaction(h, viol);
        viol_resposta.write(new Violate(viol));
        wait(viol_resposta.data_read_event());
      }
      else {
        Violate_ = Violate.read();
        wait();
      }

    }
  }

  SC_CTOR(monitor_vio):
    stream(name(), "Transactor"),
    gen("monitor_v", stream)
      { SC_THREAD(p); sensitive << clk.pos(); }
};
```

Driver_Rep.h

```
SC_MODULE(driver_rep) {

  sc_fifo_in <Reply *> rep_estimulo;

  sc_in <bool> clk;
  sc_in <bool> reset;

  //declaracao dos sinais de saida
  sc_out <bool> descript;//em 0 faz criptografia
  sc_out <sc_uint<4> > round_sel; // seleciona o passo da
criptografia

  sc_out <sc_uint<30> > id;
  sc_out <sc_uint<23> > timeStamp;
  sc_out <sc_uint<3> > status;
  sc_out <sc_uint<2> > sensor;
  sc_out <sc_uint<6> > nhist;

  Reply *rep_ptr;

  scv_tr_stream stream;
  scv_tr_generator < Reply , bool > gen;

  void p() {

    descript = 1;
    wait();
    ESPERE(!reset);


    while(1) {

rep_ptr = rep_estimulo.read();

      sc_time trans_time(rep_ptr->simulation_time, SC_US);
      sc_time agora = sc_time_stamp();
      if( trans_time > agora ) wait( trans_time - agora );
      wait();
      scv_tr_handle h = gen.begin_transaction(*rep_ptr);

      id = rep_ptr->id;
      timeStamp = rep_ptr->timeStamp;
      status = rep_ptr->status;
      sensor = rep_ptr->sensor;
      nhist = rep_ptr->nHist;
      descript = 0;
```

```
 for(int i=0; i<16; i++) {
        round_sel = i;
        wait();
      }
      round_sel = 1;

      descript = 1;


      gen.end_transaction(h);
      delete rep_ptr;

    }

  }


  SC_CTOR(driver_rep):
  stream(name(), "Transactor"),
    gen("driver_rep", stream)
    { SC_THREAD(p); sensitive << clk.pos(); }
};
```

Monitor_Rep.h

```
SC_MODULE(monitor_rep) {

  sc_fifo_out <Reply *> rep_resposta;

  sc_in <bool> clk;
  sc_in <bool> descript;
  sc_in <sc_uint<4> > round_sel;
  sc_in <sc_uint<30> > id;
  sc_in <sc_uint<23> > timeStamp;
  sc_in <sc_uint<3> > status;
  sc_in <sc_uint<2> > sensor;
  sc_in <sc_uint<6> > nhist;

  Reply rep;

  scv_tr_stream stream;
  scv_tr_generator < bool, Reply > gen;

  void p() {
    scv_tr_handle h;

    while(1) {

      wait();
       if(!descript && round_sel.read()==1) {
        h =  gen.begin_transaction();
        rep.simulation_time = sc_time_stamp().value() /
SIMULATION_TIME_SC_TIME;
        rep.id = id.read();
        rep.timeStamp = timeStamp.read();
        rep.status = status.read();
        rep.sensor = sensor.read();
        rep.nHist = nhist.read();
       }

      if(!descript && round_sel.read()==15){
      gen.end_transaction(h,rep);
     rep_resposta.write(new Reply(rep));
      }
    }
  }

  SC_CTOR(monitor_rep):
    stream(name(), "Transactor"),
    gen("monitor_rep", stream)
    { SC_THREAD(p); sensitive << clk.pos(); }
};
```

A.2. Código Fonte do Bloco Principal

A seguir, será apresentado o código fonte de todos os componentes do *testbench* do bloco Principal do DigiSeal que possui a implementação do mecanismo para sincronização das suas transações assíncronas proposta neste trabalho, com exceção do Driver e Monitor da FIFO de Violação, por já terem sido apresentados dentre os elementos do *testbench* da Máquina de Estados, isto ocorre devido ao reuso permitido pela metodologia VeriSC.

Source.h

```
class Req_constraint_class: public scv_constraint_base {

  scv_bag<pair<unsigned int, unsigned int> >
simulation_time_distrib ;
  scv_bag<pair<int,int> > id_distrib;
  scv_bag<pair<int,int> > time_distrib;
  scv_bag<pair<int,int> > comando_distrib;
  int i;

 public:
  scv_smart_ptr<Request> req_sptr;
  SCV_CONSTRAINT_CTOR(Req_constraint_class) {
  // simulation_tima aqui é na verdade um intervalo de tempo
  // não pode ser menos do que 15 ms para dar tempo o request ir e
o reply voltar.
  simulation_time_distrib.push(pair<int,int>(15000, 40000), 100);
  req_sptr->simulation_time.set_mode(simulation_time_distrib);

  id_distrib.push(pair<int,int>(1024, 1024), 90);
  id_distrib.push(pair<int,int>(0,9000), 10);
  req_sptr->id.set_mode(id_distrib);

  time_distrib.push(pair<int,int>(0, 3), 100);
  req_sptr->timeStamp.set_mode(time_distrib);

  comando_distrib.push(pair<int,int>(0, 1), 100);
  req_sptr->comando.set_mode(comando_distrib);

  }
};
  class viol_constraint_class: public scv_constraint_base {

  scv_bag<pair<unsigned int, unsigned int> >
simulation_time_distrib ;
 public:
  scv_smart_ptr<Violate> viol_sptr;
  SCV_CONSTRAINT_CTOR(viol_constraint_class) {
```

```
    // simulation_tima aqui é na verdade um intervalo de tempo
    simulation_time_distrib.push(pair<int,int>(3000, 30000), 100);
    viol_sptr->simulation_time.set_mode(simulation_time_distrib);
  }
};

SC_MODULE(source) {

  sc_fifo_out <RFrequest *> RFreq_estimulo_modref;
  sc_fifo_out <RFrequest *> RFreq_estimulo_duv;
  Req_constraint_class Req_constraint;

  sc_fifo_out <Violate *> viol_estimulo_modref;
  sc_fifo_out <Violate *> viol_estimulo_duv;
  viol_constraint_class viol_constraint;

  // variáveis a serem setados antes de inciar a simulação
  long long chave; // valor da chave de critografia
  bool trans; // indica se a simulação é só de transações, sem
tempo

void proc () {
    ifstream sfi("stimuli.txt");
    string type;
    RFrequest RFreq;
    Violate vio;
    des cript(chave);
    unsigned int last_simulation_time = 0;
  int last_sensor = FECHADO;

    // Vetores de teste lidos de arquivo
    while(!sfi.fail() && !sfi.eof()) {
      sfi >> type;

      if(type=="RFrequest") {
        RFreq.monta(&cript, sfi);
        last_simulation_time = RFreq.simulation_time;
       RFreq_estimulo_duv.write(new RFrequest(RFreq));
        RFreq_estimulo_modref.write(new
RFrequest(RFreq));//preenche a transacao de saida
        if(trans) wait(RFreq_estimulo_modref.data_read_event() &
RFreq_estimulo_duv.data_read_event());

      } else if(type=="Violate") {
        sfi >> vio;
        last_simulation_time = vio.simulation_time;
        last_sensor = vio.sensor;
        viol_estimulo_duv.write(new Violate(vio));
        viol_estimulo_modref.write(new Violate(vio));//preenche a
transacao de saída.
```

```
        if(trans) wait(viol_estimulo_modref.data_read_event() &
viol_estimulo_duv.data_read_event());
      }
      sfi.ignore(225, '\n');
    }
// Vetores de teste gerados
    while(1) {
      type = rand() % 3 == 0 ? "Violate" : "RFrequest";
      if(type=="RFrequest") {
        Req_constraint.next();
        Request req = Req_constraint.req_sptr.read();
        // garante que o simulation time cresce sempre
        req.simulation_time += last_simulation_time;
        // para evitar um erro espurio do timestamp
        if((req.simulation_time % 1000) < 100) req.simulation_time
+= 400;
        last_simulation_time = req.simulation_time;
        RFreq.monta(&cript, &req);
        RFreq_estimulo_modref.write(new RFrequest(RFreq));
        RFreq_estimulo_duv.write(new RFrequest(RFreq));
        if(trans) wait(RFreq_estimulo_modref.data_read_event() &
RFreq_estimulo_duv.data_read_event());

      } else if(type=="Violate") {
        viol_constraint.next();
        vio = viol_constraint.viol_sptr.read();
        // garante que o simulation time cresce sempre
        vio.simulation_time += last_simulation_time;
        last_simulation_time = vio.simulation_time;
        // um evento de violação não pode se repetir
        if(last_sensor==FECHADO) last_sensor=ABERTO; else
last_sensor=FECHADO;
        vio.sensor = last_sensor;
        viol_estimulo_duv.write(new Violate(vio));
        viol_estimulo_modref.write(new Violate(vio));
        if(trans) wait(viol_estimulo_modref.data_read_event() &
viol_estimulo_duv.data_read_event());
}
    }
  }


SC_CTOR(source):Req_constraint("Req_constraint"),viol_constraint("v
iol_constraint")
 {
    SC_THREAD(proc);

  }
};
```

Modref.h

```
#ifndef MODREF_H
#define MODREF_H

SC_MODULE(modref) {

  sc_fifo_in <RFrequest *> RFreq_estimulo;
  sc_fifo_in <Violate *> viol_estimulo;
  sc_fifo_out<RFReply *> RFrep_resposta;


  // estes dois variáveis devem ser setadas antes de iniciar a
simulação com sc_start()
  long long chave;//valor da chave de critografia
  int id;//valor do id do lacre
  bool trans; // indica se a simulação é só de transações, sem
tempo

  KitLacreTeste *lacre;
  RFrequest *RFreq_ptr;
  Violate *viol_ptr;
  RFReply RFrep;

  void violate() {
    lacre->setTime(viol_ptr-
>simulation_time/TIME_STAMP_PER_SIMULATION_TIME);
    lacre->doSensor(viol_ptr);
    delete viol_ptr;
    viol_ptr=NULL;
  }

void request() {
    lacre->setTime(RFreq_ptr-
>simulation_time/TIME_STAMP_PER_SIMULATION_TIME);
    if(lacre->RFreply_request(RFreq_ptr, &RFrep)) {
      RFrep_resposta.write(new RFReply(RFrep));//preenche a
transacao de saida
    }
    #ifdef _DEBUG
    else cout << "lacre.RFreply_request sem resposta." << endl;
    #endif
    delete RFreq_ptr;
    RFreq_ptr=NULL;
  }

  void p1 () {

    lacre = new KitLacreTeste(chave, id);
    RFreq_ptr=NULL;
    viol_ptr=NULL;
```

```
while(1) {
      wait(RFreq_estimulo.data_written_event() |
viol_estimulo.data_written_event());
      while(RFreq_ptr || viol_ptr || RFreq_estimulo.num_available()
+ viol_estimulo.num_available() > 0) {
        if(RFreq_ptr==NULL && RFreq_estimulo.num_available() > 0)
RFreq_ptr = RFreq_estimulo.read();
        if(viol_ptr==NULL && viol_estimulo.num_available() > 0)
viol_ptr = viol_estimulo.read();

        if(RFreq_ptr && (viol_ptr==NULL || RFreq_ptr-
>simulation_time <= viol_ptr->simulation_time))
          requisicao();
        else if(viol_ptr && (RFreq_ptr==NULL || viol_ptr-
>simulation_time <= RFreq_ptr->simulation_time))
          violacao();
      }
    }
  }

  SC_CTOR(modref) {
    SC_THREAD(p1);

  }
};
#endif
```

Checker.h

```
SC_MODULE(checker) {

  sc_fifo_in<RFReply *> RFrep_modref;
  sc_fifo_in<RFReply *> RFrep_duv;

  sc_signal<unsigned int> error_count;

  bve_cover_bucket out_checker_cv; //declaração da instância

  void proc () {

    RFReply *RFrep_mr_ptr, *RFrep_duv_ptr;

    while(1) {
      RFrep_mr_ptr = RFrep_modref.read();
      RFrep_duv_ptr = RFrep_duv.read();

      out_checker_cv.begin();
      BVE_COVER_BUCKET(out_checker_cv, true, 400); //definir a
parada
      out_checker_cv.end();

      if( !(*RFrep_mr_ptr == *RFrep_duv_ptr) ) {
        ostringstream  ms;

        ms << "expected: " << *RFrep_mr_ptr  << endl
           << "received: " << *RFrep_duv_ptr << ends;

        SCV_REPORT_ERROR("out" , ms.str().c_str() );

        error_count = error_count.read()+1;

      } else { cout << "+"; fflush(NULL); }


    }
  }

  SC_CTOR(checker): out_checker_cv("out_checker_cv")
    { SC_THREAD(proc); }
};
```

Driver_rfp

```
SC_MODULE(driver_rfp) {

  sc_fifo_in<RFReply *> RFrep_estimulo;

  sc_in <bool> clk;
  sc_in <bool> reset;

  RFReply *RFrep_ptr;

  //declaracao dos sinais de saida
  sc_out<bool> rx_data; //saida com a informação do RFreply

  scv_tr_stream stream;
  scv_tr_generator<RFReply,bool> gen;

  void p() {

    rx_data = 1;
    ESPERE(!reset);

    int w;
    while(1) {


      RFrep_ptr = RFrep_estimulo.read();

      //antes do primeiro bit do pacote envia 10 stop bits
      for(int i=0; i<10; i++){
        rx_data = STOP_BIT;
        for(w=0; w<CLKS_PER_BIT; w++) wait();
      }

scv_tr_handle h = gen.begin_transaction(*RFrep_ptr);

      for(int by=0; by<18; by++) {
        unsigned char c = RFrep_ptr->RFreply[by];

        rx_data = START_BIT;
        for(w=0; w<CLKS_PER_BIT; w++) wait();

        int yi = 0x01;  // lsb primeiro
        for(int bi=0; bi<8; bi++) {
          rx_data = (c & yi) != 0;
          yi <<=1;
          for(w=0; w<CLKS_PER_BIT; w++) wait();
        }
```

```
//entre o primeiro e os demais bytes envia 2 Stop bits
        rx_data = STOP_BIT;
        for(w=0; w<CLKS_PER_BIT; w++) wait();
        rx_data = STOP_BIT;
        for(w=0; w<CLKS_PER_BIT; w++) wait();
      }


      gen.end_transaction(h);
      delete RFrep_ptr;

    }
  }

  SC_CTOR(driver_rfp):
    stream(name(), "Transactor"),
    gen("driver_rfp", stream)
      { SC_THREAD(p); sensitive << clk.pos(); }
};
```

Monitor_rfp

```
SC_MODULE(monitor_rfp) {

  sc_fifo_out<RFReply *> RFrep_resposta;

  sc_in <bool> clk;


  sc_in <bool> rx_data;//entrada com a informação do RFreply

  scv_tr_stream stream;
  scv_tr_generator < bool , RFReply > gen;

  RFReply RFrep;

  void p() {

    int w;
    while(1) {

      for(int i=0; i<18; i++) {//zerando o array
        RFrep.RFreply[i]=0;
      }

      //antes do primeiro bit do pacote recebe 10 stop bits
int i=0;
      while(i<10*CLKS_PER_BIT || rx_data.read()) {
        wait();
        i++;
      }

      ESPERE(!rx_data);  // espere o start bit
      scv_tr_handle h =  gen.begin_transaction ();

for(int by=0; by<18; by++) {

        ESPERE(!rx_data);  // espere o start bit
        for(w=0; w<CLKS_PER_BIT+CLKS_PER_BIT/2; w++) wait();  //
espera pelo tempo de 1.5 bits

        for(int bi=0; bi<8; bi++) {  // lsb primeiro

          if(rx_data.read()==1) {
            RFrep.RFreply[by] |= (1<<bi);
          }
          for(w=0; w<CLKS_PER_BIT; w++) wait();
        }

      }
```

```
gen.end_transaction(h,RFrep);
      RFrep_resposta.write(new RFReply(RFrep));

    }
  }

  SC_CTOR(monitor_rfp):
    stream(name(), "Transactor"),
    gen("monitor_rfp", stream)
    { SC_THREAD(p); sensitive << clk.pos(); }
};
```

Driver_rfq

```
SC_MODULE(driver_rfq) {

  sc_fifo_in <RFrequest *> RFreq_estimulo;

  sc_in <bool> clk;
  sc_in <bool> reset;

  //declaracao dos sinais de saida
  sc_out <bool> sinal_tx;//sinal que indica se o RFreq está sendo
transmitido
  sc_out <bool> tx_data;//sinal de saída com a informação do
RFrequest

  RFrequest *RFreq_ptr;

  scv_tr_stream stream;
  scv_tr_generator < RFrequest, bool > gen;

  void send_char(char c) {
    int w, bi;

    tx_data = START_BIT;
    for(w=0; w<CLKS_PER_BIT; w++) wait();

    int yi=0x01;  // lsb primeiro
    for(bi=0; bi<8; bi++) {
      tx_data = (c & yi) != 0;
      yi <<= 1;
      for(w=0; w<CLKS_PER_BIT; w++) wait();
    }

//entre o primeiro e os demais bytes envia 2 (entre 1 e 5) stop
bits
    tx_data = STOP_BIT;
    for(bi = 2/* (rand() % 5) +1 */; bi>0; bi--)
      for(w=0; w<CLKS_PER_BIT; w++) wait();
  }

  void p() {

    sinal_tx = 0;
    tx_data = 1;

    ESPERE(!reset);

    int w;

    sinal_tx=1; // acionar o sinal (portadora RF)
    }
```

```
for(int i=0; i<15; i++){
   tx_data = STOP_BIT;
for(w=0; w<CLKS_PER_BIT; w++) wait();

for(int by=0; by<6; by++) {
      send_char( 1 );
    }

    sinal_tx=0; // desligar o sinal (portadora RF)

    while(1) {

      RFreq_ptr = RFreq_estimulo.read();
      if(RFreq_ptr->simulation_time < PACKAGE_TIME)
         SCV_REPORT_FATAL("RFrequest", "simulation time less than
time to send the package");
      sc_time trans_time(RFreq_ptr->simulation_time - PACKAGE_TIME,
SC_US);
      sc_time agora = sc_time_stamp();
      if( trans_time > agora ) wait( trans_time - agora );
      wait();

      sinal_tx=1; // acionar o sinal (portadora RF)

      //antes do primeiro bit do pacote envia 10 stop bits
      for(int i=0; i<10; i++){
        tx_data = STOP_BIT;
        for(w=0; w<CLKS_PER_BIT; w++) wait();
      }

      scv_tr_handle h = gen.begin_transaction(*RFreq_ptr);

      for(int by=0; by<18; by++) {
        send_char( RFreq_ptr->RFRequest[by] );
      }

      sinal_tx=0; // desligar o sinal

      gen.end_transaction(h);
      delete RFreq_ptr;
    }

  }

  SC_CTOR(driver_rfq):
    stream(name(), "Transactor"),
    gen("driver_rfq", stream)
    { SC_THREAD(p); sensitive << clk.pos(); }
};
```

Monitor_rfq

```
SC_MODULE(monitor_rfq) {


  sc_fifo_out <RFrequest *> RFreq_resposta;

  sc_in<bool> clk;
  sc_in<bool> tx_data;//sinal de saída com a informação do
RFrequest

  sc_in<bool> sinal_tx;

  RFrequest RFreq;

  scv_tr_stream stream;
  scv_tr_generator< bool, RFrequest > gen;

  void p() {

    int i,w;
    while(1) {

      for(i=0; i<18; i++) {//zerando o array
        RFreq.RFRequest[i]=0;
      }

      i=0;
      //descartando os 10 Stop bits iniciais
      while(i < 10*CLKS_PER_BIT || tx_data.read() ) {
        wait();
        i++;
      }

scv_tr_handle h =  gen.begin_transaction ();

      for(int by=0; by<18; by++) {

        ESPERE(!tx_data);  // espere start bit
        for(w=0; w<CLKS_PER_BIT+CLKS_PER_BIT/2; w++) wait();  //
espera pelo tempo de 1.5 bits

        for(int bi=0; bi<8; bi++) {  // lsb primeiro

          if(tx_data.read()) {
            RFreq.RFRequest[by] |= (1<<bi);
          }
          for(w=0; w<CLKS_PER_BIT; w++) wait();
        }
      }
```

```
RFreq.simulation_time = sc_time_stamp().value() / 1000000;
      gen.end_transaction(h,RFreq);
      RFreq_resposta.write(new RFrequest(RFreq));

    }
  }

  SC_CTOR(monitor_rfq):
    stream(name(), "Transactor"),
    gen("monitor_rfq", stream)
      { SC_THREAD(p); sensitive << clk.pos(); }
};
```

Apêndice - B

A seguir, será apresentada uma descrição mais detalhada dos requisitos, dos módulos e das variáveis do Lacre Digital (DigiSeal).

B.1. Descrição Geral do Sistema

Os principais módulos do Lacre digital são (Figura 23):

- O detector de Violação que contém um contato elétrico responsável pela detecção da violação (abertura da caixa);
- O FPGA que detém o estado (Ok, Manutenção, Violado), a identificação do Lacre e o histórico de situações, além de um relógio;
- A fonte de alimentação, responsável pela alimentação do Lacre digital, que será proveniente da própria rede elétrica e de uma bateria;
- O kit RF, módulo transceptor integrado responsável por fazer o transporte dos dados, através de radiofreqüência, entre o FPGA e o PDA;

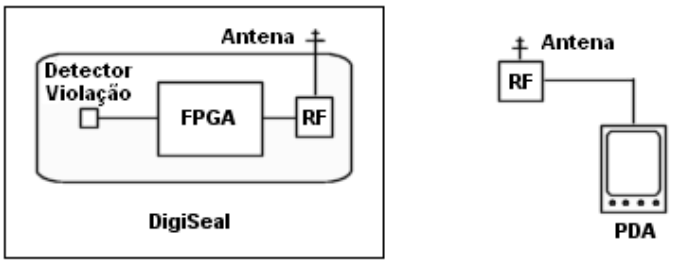

Figura 23: Componentes do Sistema.

B.2. Requisitos do Sistema

A fim de que os objetivos deste projeto sejam alcançados é necessário que se atinja um conjunto de metas secundárias relativas aos seguintes requisitos:

- Projetar um circuito integrado digital que implementa uma chave de criptografia e um mecanismo de quebra de chave [Burnett 2002] [Stinson 2002];
- Incorporar ao projeto do circuito uma lógica de detecção de violação do Lacre Digital;
- Acoplar ao circuito projetado um transceptor de rádio-freqüência [Texas 2002] para comunicação do sistema com dispositivos computacionais portáteis (palmtops);
- Implementar um protótipo do sistema de Lacre digital que incorpore todas as funcionalidades acima explicitadas.

B.3. Descrição dos Módulos

Nas Figuras 24, 25 e 26, apresentam-se os diagramas de blocos de cada módulo que constitui o Lacre digital.

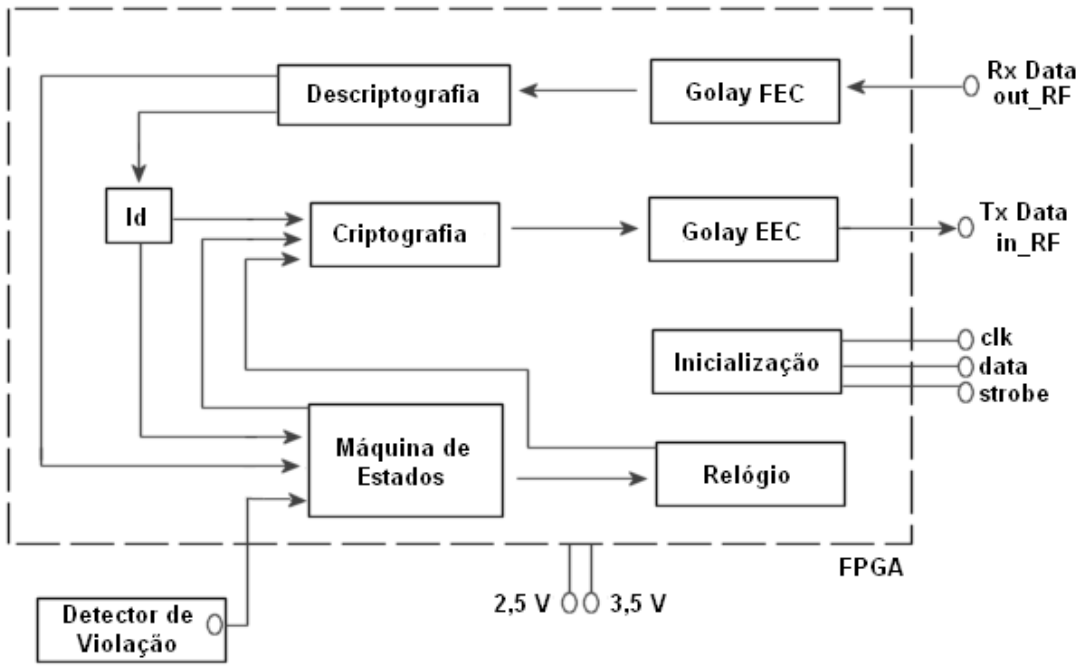

Figura 24: Diagrama de Blocos do Lacre Digital.

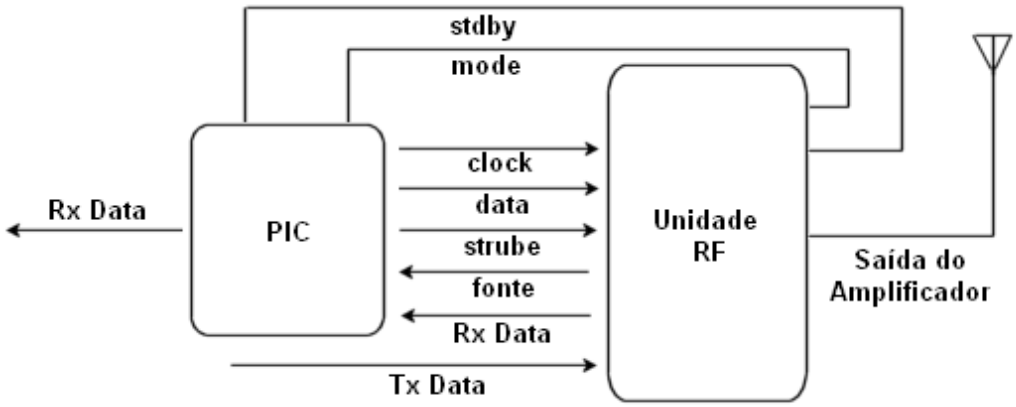

Figura 25: Diagrama de Blocos do kit RF.

B.4. Integração dos Módulos

- **Interface do FPGA com o kit RF**

O kit RF poderá ser programado pelo FPGA, configurando-o inicialmente, antes da sua operação normal. A operação do Kit RF (transceptor TRF5901+fonte de alimentação+componentes eletrônicos) poderá ser configurada através do FPGA, que passará a receber o estado do Lacre quando este for acionado pelo PDA. O kit RF receberá, através do pino TX-data, os dados do FPGA. Esses bits serão, então, transmitidos pelo kit RF para o PDA. O kit RF enviará os dados transmitidos pelo PDA, através do pino RX-data, para o FPGA. A transmissão e a recepção de dados serão implementadas de forma serial. Após o envio da informação do Lacre, o mesmo, bem como o kit RF, entrarão novamente no modo de espera. Na Figura 26, apresenta-se a interface de integração da FPGA com o kit RF.

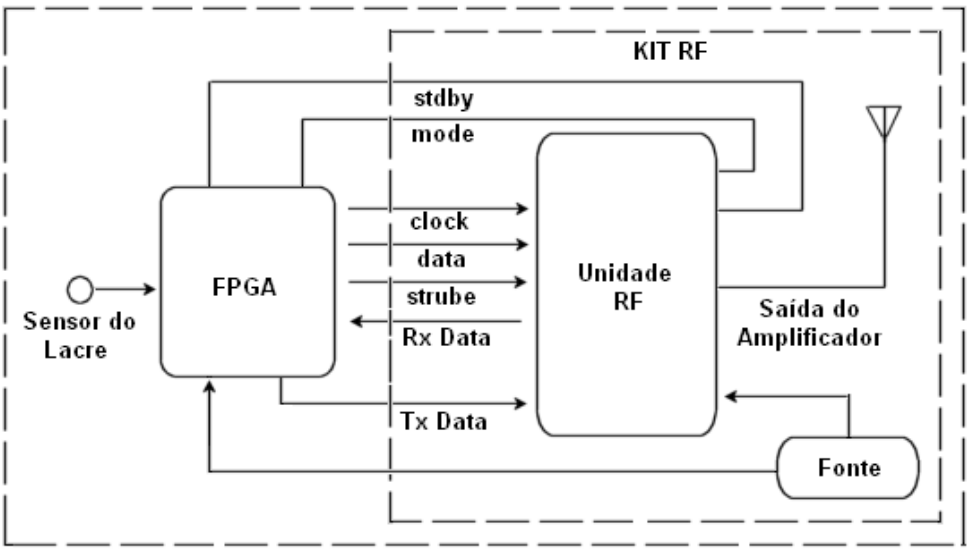

Figura 26: Interface do FPGA com o kit RF.

- **Interface do kit RF com PDA**

A conexão entre o kit RF e o PDA será feita por meio de uma porta USB presente no aparelho, de forma serial. Das quatro linhas de comunicação disponibilizadas pela USB, duas serão usadas para o tráfego de dados: uma para a transmissão dos dados enviados do PDA para o Lacre e outra para a recepção dos dados enviados pelo Lacre para o PDA. As duas linhas restantes serão usadas para alimentação e determinação do pólo negativo (terra).

A configuração do kit RF conectado ao PDA será feita por um micro-controlador integrado (PIC). Desta maneira, não haverá necessidade de se integrar uma funcionalidade de configuração de parâmetros do kit RF (como potência e freqüência de transmissão) na

aplicação do PDA. O kit RF respeitará as especificações do PDA no tocante a tensão de funcionamento na porta USB, definição de sinais lógicos e freqüência de transmissão de dados, de maneira a permitir a integração com um mínimo de esforço de adaptação no *hardware* do PDA. Na Figura 27, apresenta-se a interface de integração do Kit RF com o PDA, em que o Vcc é a tensão de alimentação, Rx é a linha de recepção dos dados, Tx é a linha de transmissão dos dados e GND é o terra (pólo negativo).

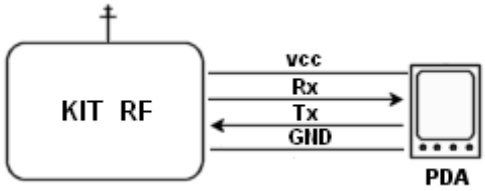

Figura 27: Interface do kit RF com PDA.

B.5. Transmissão dos Pacotes de Dados

Na Figura 28, apresenta-se o mecanismo utilizado para a transmissão do pacote de dados.

Figura 28: Transmissão dos Pacotes de Dados.

O protocolo de comunicação do Lacre digital consiste em:

- Taxa de Transmissão entre o FPGA e o PDA: 19.200 baud;
- Configuração da transmissão: Sem paridade, 1 start Bit (nível lógico baixo), oito bits de dados, 2 stop bits (nível lógico alto), sem *handshake*;
- Pacote de dados: Antes de cada pacote, deve haver a transmissão de pelo menos 10 bits em nível lógico alto. O pacote de dados consiste de 18 repetições de 1 start bit (nível lógico baixo), oito bits de dados, começando pelo bit menos significativo, e pelo menos 2, mas não mais do que 10 stop bits (nível lógico alto). O pacote nesse formato já estará criptografado, contendo o código de identificação e recuperação de

erros. Para receber o histórico dos estados, o PDA deverá enviar comandos repetidos para obter cada um dos registros do histórico.

Na Figura 29, apresenta-se o mecanismo de transmissão de dados utilizado pelas variáveis TX-data e RX-data.

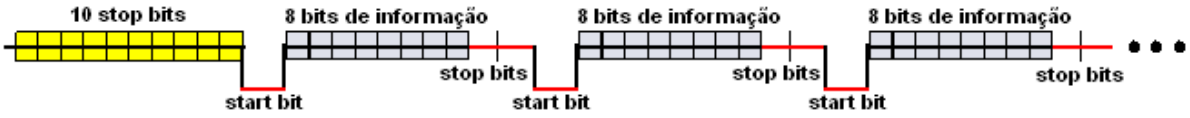

Figura 29: Mecanismo de Transmissão de Dados.

O tamanho dos dados das variáveis que aparecem nas Figuras (11 a 15) dos *testbenches* do Lacre digital é dado por: o pacote RFrequest e RFreply possui tamanho de 18 bytes, o pacote de criptografia (CRIPTreq e CRIPTrep) possui 64 bits, a variável nhist é o número de registros no histórico e possui 6 bits, o Id é o identificado do Lacre e possui 30 bits, a variável estado gerencia o estado do Lacre e possui 3 bits, o sensor possui 2 bits, a variável command possui 3 bits, o time possui 23 bits e os pacotes Reply e Request possuem 64 bits e 56 bits respectivamente.

B.6. Descrição das Variáveis do Sistema

Na Figura 30, ilustra-se o diagrama de blocos do sistema contendo as principais variáveis do Lacre Digital.

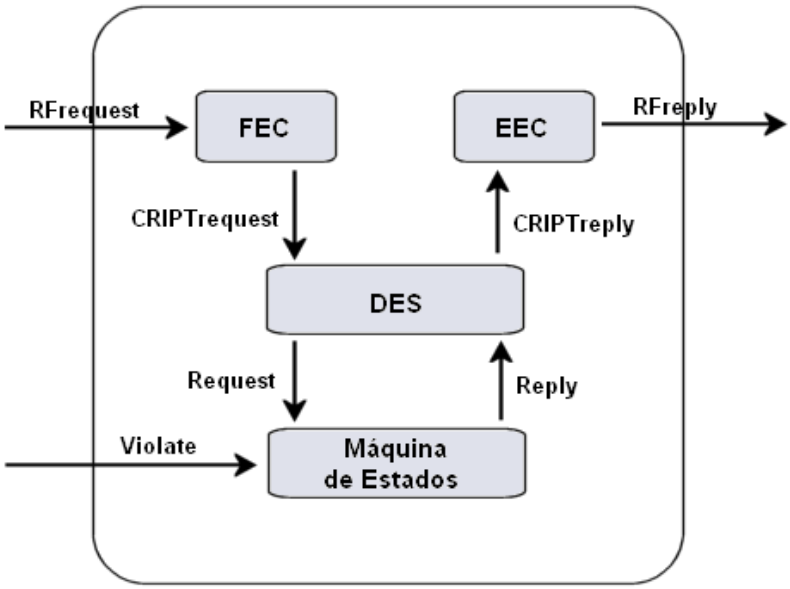

Figura 30: Diagrama de Blocos com Variáveis do Sistema.

A seguir, a definição da codificação dos sinais em nível de FIFOs:

- RFrequest: Este pacote possui como entrada o TX-data com 1 bit e o sinal_tx com 1 bit. O TX-data antes de enviar o primeiro pacote, envia 10 stop bits, entre o primeiro e os demais bits, ele envia 2 start bits, este pacote consiste em 18 bytes de informação contendo o sinal RF. O bit sinal_tx indica que o sinal RFrequest esta sendo transmitido. No RFrequest estão codificadas as variáveis id, time e comando. A transação do RFrequest contém um vetor de 18 bytes.
- CRIPTrequest: Possui o sinal ts que é a palavra corrigida do Golay que possui 12 bits, o sinal nsa possui 1 bit que serve para a validação, o sinal first que possui 1 bit para indicar que a primeira palavra foi enviada. A primeira palavra ocupa os bits mais significativos do CRIPTrequest. O CRIPTrequest possui 64 bits criptografados e os 8 bits do último (sexto) símbolo são descartados.
- Request: Este pacote possui os sinais id e req-valid com 1 bit, time com 23 bits e o comando com 3 bits. O req-valid fica em 1 durante um ciclo de *clock*, e em 0 durante pelo menos dezesseis ciclos de *clock*.
- Reply: Possui as variáveis: id, round-sel, time, estado, sensor, decrypt, e nhist. O round-sel possui 4 bits, o decrypt possui 1 bit que seleciona o passo de criptografia, começando com 0 e indo até 15. Quando o decrypt fica em 1 o algoritmo faz a criptografia. O sinal time possui 23 bits, o estado 3 bits, o sensor 2 bits e o nhist 6 bits.
- RFreply: Este pacote possui o sinal rx-data com 1 bit. Ele possui 18 bytes de informação criptografada, sendo do mesmo formato do pacote RFrequest.
- CRIPTreply: Possui os sinais ready, start e o insym O start tem 1 bit e serve para fazer a validação. Ele fica em 1 durante alguns ciclos de *clock*, depois fica em 0 por mais alguns ciclos de *clock*. O sinal start só pode ser acionado se o ready estiver em 1. Logo, após o sinal start ficar em 1, o sinal ready deve ficar em 0. O insym corresponde aos dados que serão codificados. Este pacote possui 64 bits de informação que passou pelo Golay e pela criptografia. Os bits mais significativos estão no primeiro símbolo. Assim, os 8 bits menos significativos do último símbolo são zero.
- Violate: Este sinal possui 1 bit. O valor 1 indica que o Lacre está fechado, enquanto o valor 0 indica que o Lacre está aberto.

É importante lembrar que: o intervalo de tempo entre uma solicitação (Request) e um atendimento (Reply) deve ser de, no mínimo, 7.000 microssegundos.