UNIVERSIDADE FEDERAL DE SANTA CATARINA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Wesley Gonçalves Silva

UMA NOVA ABORDAGEM PARA GERAÇÃO AUTOMÁTICA DE PROPRIEDADES PARA VERIFICAÇÃO FORMAL DE SISTEMAS DIGITAIS EM HDL

Florianópolis

2013

Wesley Gonçalves Silva

UMA NOVA ABORDAGEM PARA GERAÇÃO AUTOMÁTICA DE PROPRIEDADES PARA VERIFICAÇÃO FORMAL DE SISTEMAS DIGITAIS EM HDL

Dissertação submetida ao Programa de Pós-Graduação em Ciência da Computação para a obtenção do Grau de Mestre em Ciência da Computação.
Orientador: Prof. Dr. Djones Lettnin

Florianópolis

2013

Catalogação na fonte elaborada pela biblioteca da
Universidade Federal de Santa Catarina

A ficha catalográfica é confeccionada pela Biblioteca Central.

Tamanho: 7cm x 12 cm

Fonte: Times New Roman 9,5

Maiores informações em:
http://www.bu.ufsc.br/design/Catalogacao.html

Wesley Gonçalves Silva

UMA NOVA ABORDAGEM PARA GERAÇÃO AUTOMÁTICA DE PROPRIEDADES PARA VERIFICAÇÃO FORMAL DE SISTEMAS DIGITAIS EM HDL

Esta Dissertação foi julgada aprovada para a obtenção do Título de "Mestre em Ciência da Computação", e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação.

Florianópolis, 28 de agosto 2013.

Prof. Dr. Ronaldo dos Santos Mello
Coordenador

Banca Examinadora:

Prof. Dr. Djones Vinícius Lettnin
Presidente

Prof. Dr. Eduardo Augusto Bezerra

Prof. Dr. Jean-Marie Farines

Prof. Dr. José Luís Güntzel

À minha família e à minha namorada Ana Beatriz.

AGRADECIMENTOS

Agradeço à minha família, amigos e parceiros de trabalho que me acompanharam durante esta etapa. Gostaria agradecer a algumas pessoas em especial.

Ao Professor Djones Lettnin por sua ótima orientação e suporte. Suas sugestões e indicações foram fundamentais para o progresso do trabalho. Esses anos de trabalho foram, sem dúvida, de aprendizado e contribuíram para meu avanço como profissional.

Ao Professor Eduardo Bezerra, pela orientação em parte do mestrado e por seu apoio. Sem sua ajuda a minha vinda para Florianópolis provavelmente não teria acontecido.

Aos colegas do LISHA, em especial ao Professor Antônio Fröhlich, o Guto, ao Hugo Marcondes, ao Tiago Mück e Mateus Ludwich que ajudaram constantemente com ferramentas e discutindo conceitos de desenvolvimento de sistemas embarcados e verificação formal.

Aos alunos Douglas Schroeder, Hiago Horstmann e Victor Menegon, que formam comigo o Grupo de Verificação Formal, parte do Grupo de Sistemas Embarcados.

À Universidade Federal de Santa Catarina e ao Programa de Pós-Graduação em Ciência da Computação, que proveram as boas condições de trabalho e auxiliaram financeiramente na participação em eventos. À Katiana (secretária do programa), ao Professor Ronaldo Mello (coordenador), e ao Professor Mario Dantas (antigo coordenador).

Ao Conselho Nacional de Pesquisa (CNPq), pela ajuda financeira indispensável.

À Cadence Design Systems, Inc., pelas licenças das ferramentas utilizadas no desenvolvimento do trabalho.

Aos meus pais, Edmilson e Tânia, por seu apoio e incentivo durante todas as etapas de minha vida. Com palavras sábias e confortadoras me ajudaram a progredir e me tornar a pessoa que sou hoje. E também agradeço às minhas irmãs.

À minha namorada Ana Beatriz, por seu amor, companheirismo e apoio, que me ajudou a permanecer firme em momentos de dificuldade.

Agradeço principalmente a Deus, pela vida, saúde e força.

Não tenhas medo, pois estou contigo. Não olhes em volta, pois eu sou teu Deus. Vou fortificar-te. Vou realmente ajudar-te. Vou deveras segurar-te firmemente com a minha direita de justiça.

Isaías 41:10, Tradução do Novo Mundo

RESUMO

A flexibilidade de FPGAs baseadas em SRAM é uma opção atrativa para o projeto de sistemas embarcados. Contudo, estes sistemas críticos requerem a verificação funcional do projeto em HDL (*Hardware Description Language*) para assegurar o seu correto funcionamento. A verificação formal utilizando *model checking* representa um sistema em um modelo formal que pode ser automaticamente gerado por ferramentas de síntese. No entanto, as propriedades que descrevem o comportamento esperado, necessárias para provadores de modelo, são usualmente elaboradas de forma manual, o que é mais suscetível a erro humano, aumentando custo e tempo de verificação. Este trabalho apresenta uma nova abordagem para geração automática de propriedades para verificação de sistemas descritos em HDL. O estudo de caso industrial é o subsistema de comunicação de um satélite artificial que foi desenvolvido em parceria com o Instituto Nacional de Pesquisas Espaciais (INPE).

Palavras-chave: Sistemas Embarcados. Sistemas Críticos. Verificação Formal. Geração de propriedades.

ABSTRACT

The flexibility of Commercial-Off-The-Shelf (COTS) SRAM-based FPGAs is an attractive option for the design of embedded systems. However, the functional verification of HDL-based designs is required and is of fundamental importance. Formal verification using model checking represents a system as formal model that are automatically generated by synthesis tools. On the other hand, the properties are represented by temporal logic expressions and are traditionally elaborated by hand, which is susceptible to human errors thus increasing the costs and verification time. This work presents a new method for automatic property generation for formal verification of Hardware Description Language (HDL) based systems. The industrial case study is a communication subsystem of an artificial satellite, which was developed in cooperation with the Brazilian Institute of Space Research (INPE).

Keywords: Embedded Systems. Critical Systems. Formal Verification. Properties generation.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

SUMÁRIO

1 INTRODUÇÃO

Nos últimos anos sistemas embarcados têm sido frequentemente utilizados na indústria de sistemas eletrônicos devido a sua flexibilidade de operação. Sistemas embarcados são compostos por módulos de hardware, software e outros (e.g., mecânicos) com a finalidade de executar alguma tarefa específica. Alguns exemplos práticos nesta área são os sistemas de controle interno de veículos, piloto-automático, produtos de telecomunicações e dispositivos médicos.

Recentemente, a facilidade de acesso aos FPGAs (*Field-Programmable Gate Array*) comerciais (*Commercial-Off-The-Shelf* – COTS) abriu um novo mercado de sistemas embarcados destinados a aplicações aeroespaciais, militares e multimídia. Comparado ao processo tradicional de desenvolvimento de ASIC (*Application-Specific Integrated Circuit*), o uso de FPGA oferece baixos custos de desenvolvimento, uma vez que é um tipo de hardware que pode ser descrito através de linguagens conhecidas como HDL (*Hardware Description Languages*). Esta característica torna-o mais atraente para sistemas de baixa produção.

Sistemas do tipo embarcado e crítico requerem uma atenção especial no que se refere ao seu comportamento funcional. Estes estão entre o grupo de sistemas em que uma falha é inaceitável (JR.; GRUMBERG; PELED, 1999). Os resultados de tais falhas podem variar de leve aborrecimento até grandes catástrofes, acarretando em custos – para os indivíduos, para as empresas, para a sociedade como um todo – que podem ser imensos (WILE; GOSS; ROESNER, 2005). O aumento da complexidade de tais sistemas torna as falhas mais suscetíveis de ocorrerem. Assim, a aplicação de técnicas que garantam o funcionamento correto do sistema são mandatórios. Para atingir este objetivo existem diferentes técnicas de verificação.

Na atual era de ASICs com multimilhões de portas, reutilização de módulos de hardware com propriedade intelectual (IP, *Intellectual Properties*) e projetos de sistema em um único chip (*System-on-a-Chip*, SoC), o processo de verificação pode tornar-se muito complexo, consumindo até 70% dos custos de um projeto. Além disso, o código destinado para a verificação (i.e., desenvolvimento de testbenches) pode atingir até 80% de um projeto (BERGERON, 2000).

Assim, este trabalho de mestrado apresenta três novas abordagens para automatização do processo de verificação formal através da geração automática de propriedades a serem verificadas: (1) Análise em nível de módulo isolado; (2) Análise de módulos concorrentes; e (3) Análise em nível de sistema.

O restante do capítulo dedica-se a delinear a motivação para verificação

de sistemas embarcados e as técnicas mais utilizadas para verificação, bem como seus pontos fortes e fracos. Por fim, as principais contribuições deste trabalho são apresentadas.

1.1 POR QUE VERIFICAR?

Quando se trata de sistemas de natureza crítica, tais como sistema de bordo de aeronaves e satélites, controle de caldeira e controle de usinas nucleares, um erro de projeto pode levar a prejuízos materiais e físicos. Se um sistema de controle de bordo de um avião falhar, por exemplo, vidas podem ser colocadas em risco. No caso de uma falha em um sistema de bordo de satélite, uma enorme perda financeira pode ocorrer, além de resultar no término mal sucedido da missão espacial. Um exemplo bastante conhecido de um erro em um sistema crítico foi a explosão, logo após a decolagem, do foguete Ariane-5, em junho de 1996, devido a uma conversão de um ponto flutuante de 64 bits em um inteiro de 16 bits (BAIER; KATOEN, 2008). Portanto, técnicas para aumentar a qualidade funcional de sistemas são imprescindíveis.

Entretanto, mesmo em sistemas não críticos, como MP3 *players*, aparelhos de DVD, computadores, câmeras digitais, e outro dispositivos eletrônicos, existe a crescente necessidade de garantir a confiabilidade do sistema, pois cada vez mais requerem um rápido tempo para mercado (i.e., *time-to-market*) e espera-se que seja livre de erros. O erro na divisão de ponto flutuante do processador Pentium custou à Intel quase meio bilhão de dólares (BAIER; KATOEN, 2008). Uma prévia detecção deste erro poderia ter evitado este prejuízo. Este exemplo ilustra a importância de um projeto à prova de erros na era da alta produtividade.

À medida que o projeto vai sendo desenvolvido, os custos de uma falha e os esforços realizados para verificação do sistema aumentam. Dados estatísticos apontam que mais de 40% dos erros em todo o projeto se encontram nas fases iniciais do projeto, por exemplo, durante a definição da especificação (KIEVIET, 2013; BRAUN et al., 2010). Por esta razão, quanto antes o processo de verificação for iniciado, menor será o custo do projeto em termos de recursos e tempo.

As abordagens mais utilizadas para verificação de projetos em HDL são baseadas em simulação e em verificação formal, por meio da técnica conhecida como *model checking*. Abordagens baseadas em simulação têm a vantagem de utilizar soluções para *debug*. No entanto, resultam em um grande esforço para a criação de vetores de teste, uma vez que os estímulos providos para exercitar as funcionalidades do projeto podem exercitar apenas um caminho no espaço de estados por vez. Dessa forma, caminhos críticos

fora do fluxo principal de execução do sistema podem ser esquecidos, resultando no conhecido problema de cobertura (LETTNIN et al., 2009). Além do mais, conforme observado por Dijkstra (DIJKSTRA, 1972), teste de programas podem ser usados para mostrar a presença de *bugs*, mas nunca mostram sua ausência.

Por outro lado, o *model checking* utiliza linguagens de especificação baseadas em lógica temporal para definir propriedades que fazem asserções (i.e., afirmações) sobre o comportamento esperado do sistema. Então, um procedimento percorre o espaço de estados de forma exaustiva e verifica se as propriedades são satisfeitas ou não (CLARKE; EMERSON; SIFAKIS, 2009).

1.2 IDENTIFICAÇÃO DO PROBLEMA

O processo de verificação utilizando propriedades de um sistema está descrito na Figura 1. A partir da especificação do que é esperado do sistema, propriedades estabelecem condições que devem ser provadas durante a execução da verificação.

A especificação pode estar descrita em maneira textual, em um formato de documento de requisitos, e por meio de diagramas, como por exemplo o *Unified Modeling Language* – UML (UML, 2013). O engenheiro de verificação é responsável por analisar esses documentos e diagramas e criar proposições sobre o sistema, isto é, as propriedades. Essas propriedades deverão ser escritas em um formato reconhecido pela ferramenta que será utilizada para explorar o sistema e comparar se as proposições são verdadeiras ou falsas.

O problema é que a elaboração de propriedades geralmente é feita manualmente, dependendo da habilidade do engenheiro em entender a modelagem e a implementação do sistema e reconhecer o que precisa ser verificado. Isso não é uma tarefa trivial, uma vez que à medida que o sistema torna-se mais complexo, a análise da interação entre diversos módulos torna-se mais difícil de ser descrita. Outra dificuldade na elaboração manual de propriedades é que se requer do engenheiro de verificação o conhecimento de lógica temporal e sua utilização em propriedades. Todas essas dificuldades levam ao aumento do custo e do consumo de tempo no projeto, além de ser mais suscetível ao erro humano.

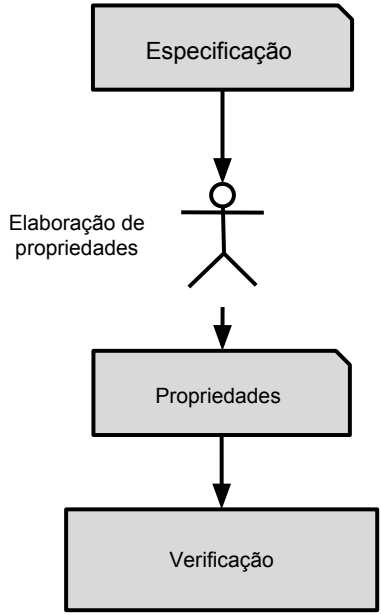

Figura 1: Processo de verificação de sistemas

1.3 OBJETIVOS E ESCOPO

Para lidar com os problemas mencionados, este trabalho propõe uma nova abordagem para definir propriedades por meio de um processo automatizado.

1.3.1 Objetivo geral

O objetivo geral é gerar propriedades automaticamente utilizando o modelo de máquinas de estados finitas (FSM, *Finite State Machine*) e diagramas de sequência, definidos na especificação do sistema, e refinadas durante toda a fase de projeto, à medida que o entendimento do sistema vai aumentando.

1.3.2 Escopo do trabalho

O escopo deste trabalho deve atender as seguintes características:

- As propriedades devem estar descritas em um formato utilizado na indústria para verificação de HDL, tais como o PSL (*Property Specification Language*) (IEEE. . . , 2005) ou SVA (*SystemVerilog Assertions*) (SVA, 2013), a fim de viabilizar a validação do método proposto usando verificadores (*model checkers*) industriais.

- A verificação das propriedades ocorrerá com o uso de ferramentas de verificação de terceiros baseadas em asserções (ABV, *Assertion Based Verification*).

1.3.3 Objetivos específicos

1. Obter o aumento da confiabilidade do sistema embarcado crítico do estudo de caso deste trabalho através da aplicação das técnicas propostas. O estudo de caso em questão é a Unidade de Telemetria e Telecomando (UTMC), um sistema concebido a partir do trabalho de pesquisa e desenvolvimento intitulado "Projeto, fabricação e teste de protótipo da Unidade de Telemetria e Telecomando do Computador de Bordo do SISCAO (SISTEMA DE CONTROLE DE ATITUDE E ORBITA DE UM SATÉLITE ESTABILIZADO EM TRÊS EIXOS)", financiado pelo Instituto Nacional de Pesquisas Espaciais (INPE).

2. Elaborar um *framework* para verificação, propondo de forma detalhada um fluxo de atividades para verificar sistemas com características semelhantes ao estudo de caso.

3. Gerar propriedades locais, para verificação do funcionamento interno de um módulo, e propriedades globais, para verificação em nível de sistema.

4. Encontrar erros de especificação e implementação.

5. Aprimorar o conhecimento de projeto de sistemas embarcados.

6. Aprimorar a habilidade de desenvolvimento de sistemas descritos em HDL.

1.4 ESTRUTURA DA DISSERTAÇÃO

Os principais conceitos relacionados ao *model checking* são abordados no Capítulo 2, começando com o surgimento da técnica na década de 1980 até os últimos avanços, com esforços feitos principalmente para lidar com um problema chamado de *problema de explosão de estados*. O Capítulo 3 aborda trabalhos relacionados à geração de propriedades para verificação de sistemas.

O Capítulo 4 pormenoriza a proposta para geração de propriedades, tendo como alvo a verificação formal de sistemas baseados em FPGA. A partir de um pseudo-algoritmo, a abordagem é detalhada, e um exemplo didático é estabelecido. O Capítulo 5 mostra a implementação desta abordagem e o desenvolvimento de uma ferramenta cujo nome é PropGen.

O Capítulo 6 apresenta a Unidade de Telemetria e Telecomando, a UTMC. As características do sistema, as normas obedecidas e os detalhes de implementação são tratados.

O Capítulo 7 apresenta a verificação dos módulos do fluxo de telemetria da UTMC. A verificação é feita com base nas propriedades geradas utilizando a abordagem proposta neste trabalho. Por fim, o Capítulo 8 sumariza as contribuições do presente trabalho e aponta para os trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresentará uma visão geral do processo de verificação de sistemas, tendo como foco o *model checking*, que corresponde ao mecanismo de verificação de sistemas adotado neste trabalho.

2.1 VISÃO GERAL

O processo de verificação de sistemas é ilustrado na Figura 2.

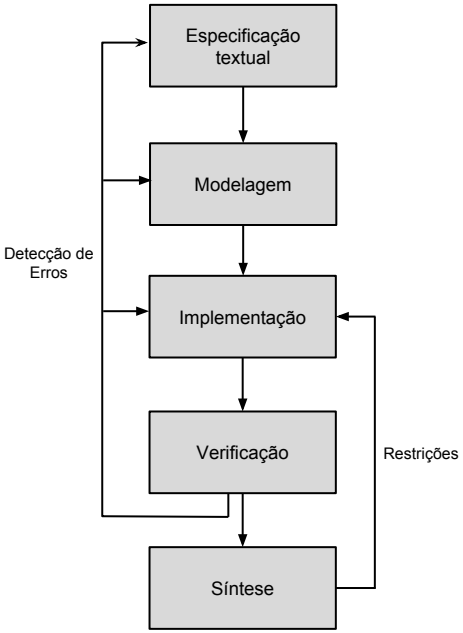

Figura 2: Fluxo de projeto da indústria, adaptado de (DRECHSLER; FEY, 2004)

A primeira fase do projeto tem como produto a especificação textual do sistema. Nessa fase é feito o levantamento de requisitos e especificação do sistema, na qual são gerados documentos que apresentam as características esperadas do sistema e é definido o que o sistema deverá ou não fazer. Geralmente, isso é feito em linguagem natural.

Na segunda fase, os requisitos são analisados e começa-se a projetar as funcionalidades do sistema e o resultado são modelos que descrevem o sistema de maneira abstrata, ou seja, sem maiores detalhes de implementação. Alguns exemplos são os modelos UML (UML, 2013), como os diagramas de sequência e diagramas de máquinas de estados.

Em sistemas de hardware, a partir do modelo o sistema é codificado em uma linguagem de descrição de hardware (HDL, *Hardware Description Language*). Na sequência, o sistema é verificado por simulação, utilizando *testbenches*, ou por verificação formal (*model checking*). Quando inconsistencias não são mais encontradas, a síntese física [1] do código HDL pode ser feita (DRECHSLER; FEY, 2004).

Para permitir o uso de um *model checker* como o mecanismo de verificação, a implementação resultante do processo de desenvolvimento deve ser representada através de um modelo formal (i.e., um modelo baseado em máquinas de estados), através de ferramentas de síntese. Em sistemas de hardware baseados em FPGAs, o modelo formal é gerado através da síntese do código descrito em HDL. Esse processo pode ser visto na Figura 3.

O modelo formal é confrontado com propriedades, premissas elaboradas sobre o sistema que descrevem seu comportamento esperado. O sistema é dito "provado" se o conjunto de propriedades são satisfeitas. Caso contrário, um contra-exemplo é gerado (BAIER; KATOEN, 2008). Contra-exemplo descreve o caminho da execução do sistema que leva a violação da propriedade.

A seção 2.2 abordará com mais detalhes o que são as propriedades. A seção 2.3 aprofundará o método de verificação utilizando *model checking*.

2.2 PROPRIEDADES

Os *model checkers* utilizam uma linguagem de especificação baseada em lógica temporal para descrever o comportamento esperado de um sistema. As definições estabelecidas em tais linguagens são chamadas de propriedades. Por meio das propriedades são feitas afirmações sobre a execução do sistema ao longo do tempo.

A verificação de um sistema que implementa uma estrutura de fila do tipo FIFO (*First In, First Out*), por exemplo, poderia ter as seguintes propriedades:

[1] O fluxo de projeto para FPGAs consiste em diversas etapas realizadas por diversas ferramentas. O fluxo central de projeto é composto por: descrição do circuito (HDLs); Síntese lógica (geração de *netlist*); Síntese física (*placement and routing*); Simulação funcional; e Análise temporal. O nível de projeto considerado neste trabalho envolve a etapa de síntese lógica para a geração do modelo formal.

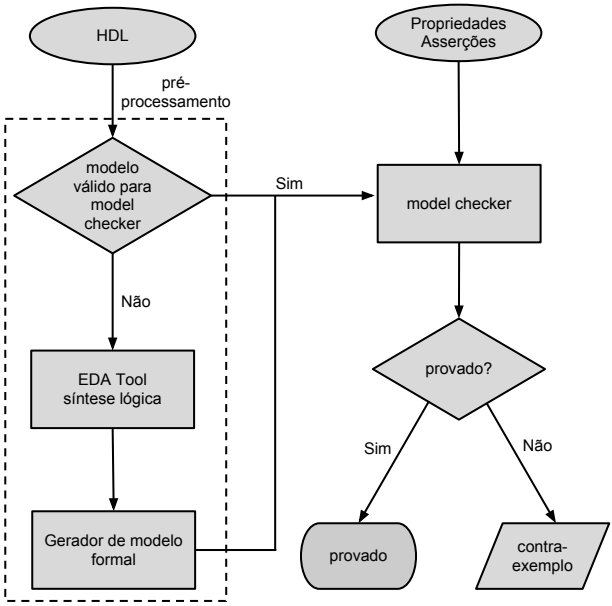

Figura 3: Geração de modelo formal através do pré-processamento e sua utilização no fluxo do *model checking*

- Sempre que houver uma inserção, o número de elementos da fila nunca ultrapassará o limite máximo (*overflow*);
- Sempre que houver uma remoção, o número de elementos da fila nunca será negativo (*underflow*).

As propriedades são divididas em três camadas, conforme a Figura 4. A camada booleana contém uma expressão lógica, uma proposição, sobre determinado comportamento do sistema. A camada temporal informa *quando* espera-se que esta proposição seja verdadeira. A camada de verificação é utilizada para informar qual diretiva de verificação será usada e depende da ferramenta utilizada.

2.2.1 Lógica Temporal

Lógica Temporal (LT) é usada para descrever a ordem de eventos sem introduzir medidas de tempo explícitas (CLARKE; GRUMBERG; LONG,

Figura 4: Estrutura básica de uma propriedade

1994).

As LTs são classificadas de acordo com a maneira em que as sequências de eventos são consideradas, se são vistas como uma estrutura linear ou com uma estrutura de ramificação.

Em uma sequência linearmente ordenada, cada instante tem um único evento sucessor (BAIER; KATOEN, 2008). Por exemplo, para os estados s e t, ou $s < t$, ou $s = t$ ou $t < s$, onde o operador $<$ pode ser lido como "ocorre primeiro" e o operador $=$ pode ser lido como "ocorre ao mesmo tempo" (MCMILLAN, 1992). Este tipo de lógica é conhecido como *Linear Time Logic* (LTL).

Por outro lado, um tempo em ramificação é aquele que o ordenador temporal $<$ define uma árvore cuja ramificação avança para o futuro, de forma que cada instante tem um único passado, mas um futuro indeterminado. Para os estados s, t e u, se $t < s$ e $u < s$, então $t < u$, $t = u$ ou $t > u$, onde o operador $>$ pode ser lido como "ocorre após". Neste caso o passado de cada estado é linearmente ordenado (MCMILLAN, 1992). Este tipo de lógica é conhecido como *Computation Tree Logic* (CTL).

Os operadores básicos de LTL são: F (*eventually*, *future*), G (*always*, *globally true*), X (*next*) ou U (*until*) (MCMILLAN, 1992; CLARKE; EMERSON; SIFAKIS, 2009). Por exemplo, $F\,q$ afirma que um evento q acontecerá eventualmente no futuro. Da mesma forma, $G\,p$ afirma que um evento p sempre ocorre no futuro G. A lógica CTL conta com outros dois operadores que representam o conceito de que algo *necessariamente* ocorre no futuro (operador A) e o conceito de que algo *possivelmente* ocorre no futuro (operador E). A CTL utiliza A ou E seguido dos operadores básicos de LTL.

A Figura 5 mostra alguns exemplos do uso destes operadores em condições booleanas.

2.2.2 Tipos de Propriedades

Propriedades são classificadas em duas principais classes: *safety* e *liveness*. Propriedades *safety* afirmam que algo ruim **nunca** acontece ou que

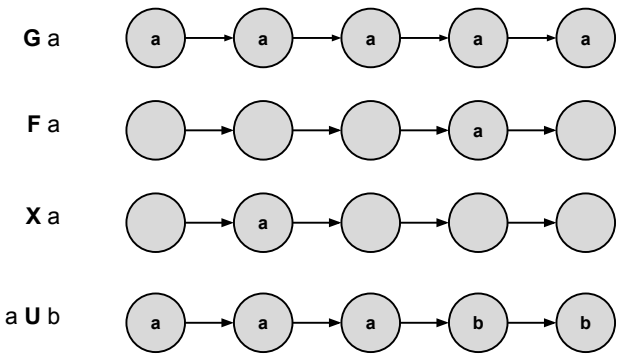

Figura 5: Exemplo de operadores temporais

algo bom **sempre** acontece. Propriedades *liveness* declaram que algo acontece **eventualmente** (FUJITA; GHOSH; PRASAD, 2008). Em termos de máquinas de estados, propriedades *safety* verificam se um estado é sempre ou nunca verdadeiro. Por outro lado, propriedades *liveness* verificam se um estado às vezes é verdadeiro ao longo da execução da máquina.

Além disso, (BAIER; KATOEN, 2008) classifica outros tipos de propriedades:

- Corretude funcional - o sistema faz o que ele deveria fazer?
- Alcançabilidade - é possível acabar em um estado de *deadlock*? A execução alcança o estado desejado?
- *Fairness* - em certas condições, o evento ocorre repetidamente?
- Tempo-real - o sistema está agindo em tempo?

As descrições de propriedades para verificação formal em módulos de hardware podem ser expressas de diversas maneiras. Duas linguagens bastante utilizadas são o PSL (*Property Specification Language*) (IEEE..., 2005), e o SVA (*SystemVerilog Assertions*)(SVA, 2013), um subconjunto da linguagem SystemVerilog. Ambas permitem fazer pressuposições sobre o comportamento do sistema e são utilizadas tanto para verificações baseadas em simulação quanto para verificação formal. Por meio de asserções é possível inferir sobre o comportamento interno do sistema durante a sua execução e averiguar se ocorre conforme o previsto (LONG; SEAWRIGHT; KAVALIPATI, 2009). Também é possível examinar se um dado sinal tem um determinado valor após “n” passos na execução.

2.3 MÉTODOS DE VERIFICAÇÃO

2.3.1 Model Checking

Nos anos 80 os pesquisadores Edmund Clarke e Allen Emerson propuseram uma abordagem automatizada de verificar sistemas concorrentes. Esta recebeu o nome de *model checking*. Na mesma época outros dois pesquisadores, que trabalhavam isoladamente, Jean-Pierre Quielle e Joseph Sifakis, propuseram uma abordagem semelhante (CLARKE; EMERSON; SIFAKIS, 2009).

Os provadores de modelos (i.e., *model checkers*) possuem três componentes: (1) uma linguagem de especificação baseada em lógica temporal (propriedades), (2) um modelo formal que representa o sistema a ser verificado por meio de máquinas de estado e (3) um procedimento de verificação que usa uma busca exaustiva dos espaços de estados para determinar se uma especificação é verdadeira ou não (CLARKE; EMERSON; SIFAKIS, 2009). Caso uma propriedade falhe durante a verificação, o *model checker* retorna um contra-exemplo que permite, através de simulação, replicar o cenário em que a falha ocorreu (BAIER; KATOEN, 2008).

Nos últimos trinta anos, o foco dos pesquisadores na área tem sido atenuar o problema da busca exaustiva do espaço de estados, um problema conhecido como "problema da explosão de estados". A primeira abordagem para busca de estados no *model checking* representava as relações de transição por meio de uma lista de adjacências, ou em uma grande tabela *hash*, memorizando os estados alcançáveis em um percurso em profundidade (CLARKE; EMERSON; SISTLA, 1986). Este modo explícito de exploração do espaço de estados demandava muito recurso computacional e apenas pequenos sistemas podiam ser verificados. Em 1987, McMillan, então estudante de graduação em Carnegie Mellon, sugeriu que em vez de usar uma estrutura explícita para representação das máquinas de estados, fosse usada uma estrutura chamada *Binary Decision Diagram* (BDD), proposto por Bryant (BRYANT, 1986), uma estrutura de dados para representação de funções booleanas. A utilização desta estrutura permitiu uma representação mais compacta do sistema, tornando possível verificar sistemas que manipulavam mais de 10^{120} estados (CLARKE et al., 2001), e viabilizando o uso do *model checking* em sistemas reais.

2.3.1.1 Symbolic Model Checking

Como mencionado anteriormente, em 1987 McMillan propôs representar as transições de estados em uma forma simbólica baseada nos BDDs, não com lista de adjacência (i.e., hash). O fruto deste trabalho foi a concepção da técnica conhecida como *Symbolic Model Checking* (WANG et al., 2009).

Essa técnica usa fórmulas em lógica temporal para descrever propriedades. Tais fórmulas são parâmetros para um procedimento que retorna um BDD que representa todos os estados que satisfazem à fórmula (CLARKE; GRUMBERG; LONG, 1994).

Uma ferramenta para *model checking* chamada *Symbolic Model Verifier* (SMV), desenvolvida por McMillan (MCMILLAN, 1992), implementou estes conceitos. Esta permite uma verificação automática de programas escritos em uma linguagem também chamada SMV. Esta linguagem descreve sistemas representados por estados finitos e protocolos.

2.3.1.2 Bounded Model Checking

Muito embora a utilização de BDDs tenha sido um avanço na área, existem ainda algumas limitações. Para manipular sistemas na ordem de centenas de variáveis de estado, essa técnica era suficiente. Porém, em sistemas maiores as estruturas BDD geradas durante o *model checking* tornavam-se muito grandes para serem processados pelos computadores, devido aos seus limites de memória. Por isso, uma abordagem alternativa ao uso de BDDs foi proposta (BIERE et al., 1999b). Tal abordagem consiste na utilização de técnicas de solvabilidade (“satisfiability”, em inglês) booleana, conhecida como SAT. Esta abordagem é também classificada como *symbolic model checking*, mas devido a sua estratégia de verificação, foi chamada de *bounded model checking*.

Como os BDDs, a técnica baseada em SAT também opera expressões booleanas, mas de uma forma não canônica. Isso possibilita inferir sobre problemas na ordem de milhares de variáveis. A abordagem consiste em analisar o caminho do espaço de estados limitados ao tamanho k, procurando por contra-exemplos. A cada passo, o limite k é incrementado. Depois de um certo número de iterações, caso não existam contra-exemplos, é concluído que o sistema está correto para o espaço de estados de tamanho k (BIERE et al., 1999a).

2.4 FERRAMENTAS

O *model checker* VIS (VIS, 2013) utiliza como principal entrada o modelo formal no formato de netlist BLIF-MV (BRAYTON et al., 1991), uma linguagem projetada para descrever sistemas sequenciais e sua comunicação. O ABC (MISHCHENKO, 2013) é uma ferramenta para síntese e verificação de circuitos lógicos que utiliza o formato AIGER (BIERE, 2013) como principal entrada. O formato AIGER representa circuitos sequenciais e combinacionais e utiliza a estrutura AIG (*And-Inverters Graph*), uma rede booleana composta de portas AND e inversores. O *model checker* CadenceSMV (MIR; BALAKRISHNAN; TAHAR, 2000) utiliza o modelo no formato SMV, mas conta com um procedimento que aceita Verilog como entrada e gera o modelo formal no modelo SMV.

O VIS e o ABC também aceitam como entrada a linguagem Verilog. Contudo, elas trabalham apenas com um subconjunto restrito da linguagem, tornando inapropriado para sistemas de maior complexidade. A linguagem VHDL, utilizada no estudo de caso deste trabalho, não é uma entrada válida para esses provadores de modelo.

A ferramenta *Incisive Formal Verifier* (IFV) (CADENCE, 2013), da empresa Cadence Design Systems, Inc., realiza a Verificação Baseada em Asserções (Assertion-based Verification, ABV) de projetos RTL escritos em Verilog e VHDL. Não é necessário que o usuário gere o modelo formal através do pré-processamento descrito na Figura 3. O IFV suporta os principais padrões de descrição de propriedades da indústria, incluindo SVA e PSL. A ferramenta realiza a análise estática, em que as asserções são confrontadas com a implementação de uma maneira exaustiva. No caso de falha em alguma asserção a ferramenta retorna um contra-exemplo de fácil entendimento, no formato de gráfico de ondas, sendo possível detectar com maior facilidade o que levou à falha.

Todas essas ferramentas foram avaliadas para o uso neste trabalho. No entanto, devido a algumas dificuldades, descritas no Apêndice A, a ferramenta comercial IFV foi a escolhida para realizar a verificação no fluxo proposto.

3 TRABALHOS RELACIONADOS

Os estudos recentes referentes à geração de propriedades seguem duas principais vertentes. A primeira obtém informações dos dados levantadas durante a simulação, isto é, dos *testbenches* desenvolvidos para estimular as funcionalidades do sistema. A outra vertente adquire informações dos documentos e modelos desenvolvidos na fase de especificação do sistema. Neste capítulo serão apresentados e comparados os principais trabalhos correlatos, bem como, as principais contribuições do presente trabalho.

3.1 ABORDAGENS BASEADAS EM SIMULAÇÃO

No trabalho de Drechsler e Fey (2004) a geração de propriedades é proposta para mitigar os custos envolvidos no processo de verificação do sistema utilizando *model checking*. A geração é feita com base no conjunto de sinais em dados de simulação do sistema. Os autores propõem que assim que se tenha um bloco de sistema simulado, já se inicie o processo de geração, o que contribuirá para uma detecção precoce de erros e diminuirá o tempo de construção do ambiente de verificação. No processo de verificação, caberá ao engenheiro de verificação avaliar se a propriedade gerada está de acordo com a especificação.

A heurística adotada no trabalho de Drechsler e Fey (2004) para a extração das propriedades é chamada de *Pattern Matching*. A heurística consiste em analisar os sinais de entrada, saída e de *flip-flops* em um vetor de simulação do sistema, e procurar por padrões dentro de uma janela de tempo.

O trabalho de Rogin et al. (2008) concentra-se na geração de propriedades complexas. O processo de geração de propriedades é dividido em duas etapas. Na primeira etapa faz-se o levantamento de propriedades candidatas com base na proposta de Drechsler e Fey (2004). Na segunda etapa, as propriedades candidatas são combinadas e a dependência temporal entre elas é analisada. Com esse procedimento, os autores conseguiram definir propriedades de alta complexidade. Uma ferramenta chamada *Dianosis* foi implementada para manipular vetores de simulação e gerar propriedades em SVA (SVA, 2013) e OVL (*Open Verification Library*) (ACCELLERA, 2013).

A ferramenta GoldMine (VASUDEVAN et al., 2010) faz a geração de propriedades para a verificação de modelo de sistemas descritos em HDL. Além dos dados de simulação, a ferramenta desenvolvida realiza a análise estática do sistema e a mineração de dados. Os dados de simulação são fornecidos pelo usuário ou gerados pela ferramenta. Na fase de análise estática,

informações específicas do domínio do sistema são extraídas. A mineração de dados produz propriedades candidatas. O último passo no fluxo da ferramenta é a avaliação humana das propriedades.

3.2 ABORDAGENS BASEADAS NA ESPECIFICAÇÃO

Em um contexto de desenvolvimento baseado em modelo, Balasubramanian et al. (2011) menciona a dificuldade de definir propriedades para verificar o sistema em modelos abstratos e utilizar o mesmo grupo de propriedades para verificar a implementação gerada a partir dos modelos, em vista da diferença de níveis de abstração. Conforme observado pelos autores, a correspondência entre os elementos do modelo e a sua geração de código não é óbvia. Para tratar esse problema, eles propuseram uma abordagem de geração de propriedades através de padrões de especificação, baseados no trabalho de Dwyer, Avrunin e Corbett (1999).

Os padrões de especificação aplicados em propriedades descrevem requisitos comumentes observados a respeito do comportamento do sistema. Foi identificado um padrão de *ocorrência*, que define a frequência com que um estado ocorre: se *nunca*, *sempre eventualmente*, ou *repetidamente* em um número finito de tempo. Outro padrão é o de *ordem*, que define a sequência de um evento (DWYER; AVRUNIN; CORBETT, 1999).

No trabalho de Balasubramanian et al. (2011), os padrões de especificação são utilizados em um autômato observador parametrizado, os quais podem ser utilizados para descrever diferentes aspectos da especificação do sistema. Os autores também propõem o uso de contratos, que estipulam pré-condições, pós-condições e constantes, com respeito aos valores de entrada e saída. Neste caso, o comportamento interno do sistema não é relevante. O gerador de código analisa estas informações e gera as propriedades no nível de implementação, para sistemas de software.

Campos, Machado e Seabra (2008) também utilizaram a abordagem de padrões (DWYER; AVRUNIN; CORBETT, 1999), e introduziram mais dois novos padrões: o padrão de *possibilidade*, indicando que um estado pode ocorrer durante toda a execução; e o padrão *fairness*, expressando que um estado ocorre frequentemente durante a execução do sistema. Em 2009 os autores apresentaram um outro padrão, o *liveness*, representando que um estado Q pode ocorrer com frequência depois de um outro estado P. O trabalho apresentou uma ferramenta que exibe uma coleção de padrões que estabelecem um modelo cujas informações pertinentes ao domínio do sistemas são inseridas em forma de proposições. Desta forma, a ferramenta auxilia a seleção de um padrão que seja mais adequado para a propriedade que se queira elabo-

rar. Como saída, a ferramenta retorna a propriedade no formato LTL ou CTL, e cabe ao usuário inserir a propriedade no ambiente de verificação apropriado (CAMPOS; MACHADO, 2009), seja para um sistema de hardware ou de software.

No trabalho de Zhang e Liu (2010) máquinas de estado em UML são usadas para gerar modelos para a linguagem formal CSP (BROOKES; HOARE; ROSCOE, 1984). Xu, Miao e Philbert (2009) propuseram a geração do modelo CSP a partir de diagramas de atividade. Em ambas abordagens, a verificação é aplicada sobre modelos de alto-nível do sistema de software.

3.3 COMPARATIVO ENTRE AS ABORDAGENS

Após análise das características de cada abordagem é possível identificar pontos positivos e negativos. A Tabela 1 apresenta um comparativo entre os trabalhos relacionados.

As abordagens baseadas em simulação têm mais proximidade com a implementação, e como resultado, uma implementação mais confiável. Conforme Baier e Katoen (2008): *"Any verification using model-based techniques is only as good as the model of the system"*. De fato, abordagens que enfocam apenas na verificação do modelo de alto nível do sistema não podem garantir que a implementação também está verificada.

Tabela 1: O estado da arte na geração de propriedades

	(Semi-) automático	**Abordagem**	**Aplicado em hardware**	**Nível de módulo / sistema / ambos**
Drechsler e Fey (2004)	não	simulação	sim	módulo
Vasudevan et al. (2010)	sim	simulação	sim	ambos
Balasubramanian et al. (2011)	sim	especificação	não	sistema
Zhang e Liu (2010)	não	especificação	não	sistema
Xu, Miao e Philbert (2009)	não	especificação	não	sistema
Campos, Machado e Seabra (2008)	não	especificação	sim	sistema

No entanto, estas dependem de grande esforço na elaboração de *testbenches*. A geração de propriedades é comprometida caso o gerador não tenha uma grande base de dados de simulações do sistema. As abordagens baseadas na especificação não requerem esse esforço, podendo em alguns casos descartar totalmente a necessidade de se simular o sistema.

A análise sobre a especificação, por sua vez, influencia no refinamento e amadurecimento das especificações do sistema. Vale ressaltar que 40% dos erros no projeto são mais prováveis de ocorrerem na especificação (KIEVIET, 2013; BRAUN et al., 2010). Naturalmente, o exercício das funcionalidades descritas na especificação auxiliam na identificação precoce desses erros.

As abordagens baseadas na especificação também têm como ponto forte a direta observância dos requisitos do sistema, isto é, do comportamento esperado. As abordagens baseadas em simulação não garantem que as propriedades geradas estarão de acordo com os comportamentos descritos da especificação. Na abordagem proposta em Drechsler e Fey (2004) é necessário que engenheiro de verificação examine o resultado da verificação das propriedades e avalie a observância do sistema com a especificação. Este exame manual pode levar a erros.

As abordagens também diferem com relação a que aspectos do sistema são verificados. A coluna **Nível de módulo / sistema / ambos** da Tabela 1, apresenta se as propriedades são geradas para verificação do funcionamento isolado de um módulo do sistema, se são geradas para verificação da interação entre os módulos do sistema, ou se para os dois casos. De fato, a análise apenas em nível de módulo pode fazer com que funcionalidades importantes deixem de ser verificadas. O trabalho de Vasudevan et al. (2010) foi o único entre os trabalhos relacionados capaz de produzir propriedades para os dois níveis.

3.4 CONTRIBUIÇÃO

A proposta deste trabalho de mestrado adota a abordagem baseada na especificação. A geração de propriedades ocorre sobre o comportamento descrito em máquinas de estados, para verificação em nível de módulo, e em diagramas de sequência, para verificação em nível de sistema, considerando a interação entre os módulos.

Ao contrário das abordagens baseadas em simulação, o esforço na criação de *testbenches* não é necessário. No entanto, a proposta aqui não se refere à verificação do modelo de alto nível do sistema (i.e, máquinas de estados e diagramas de sequência), mas na verificação da implementação do módulo de hardware. A geração é fruto da combinação entre os modelos

utilizados para descrever o comportamento esperado e a implementação. O resultado da geração são propriedades descritas em PSL (IEEE..., 2005) e SVA (SVA, 2013) que serão inseridas junto da implementação e verificadas em um contexto de verificação baseada em asserções (ABV).

Os trabalhos do estado da arte para a geração de propriedades com base na especificação são, em sua grande maioria, aplicados em módulos de software. O trabalho desta dissertação tem como alvo sistemas de hardware descritos em HDL.

4 DESENVOLVIMENTO DO TRABALHO: GERAÇÃO DE PROPRIEDADES PARA A VERIFICAÇÃO FORMAL

Este capítulo apresenta uma abordagem para geração de propriedades para verificação formal. Desta forma, espera-se diminuir a suscetibilidade ao erro humano, reduzindo custo e tempo do projeto.

A geração de propriedades é feita através de máquinas de estados finitas e diagramas de sequência, dois modelos utilizados na fase de especificação do sistema. Estes modelos foram escolhidos para descrição da especificação pois são mais consistentes do que linguagem natural ao expressarem o comportamento do sistema. O fluxo de trabalho proposto, comparado com a elaboração manual de propriedades, é apresentado na Figura 6.

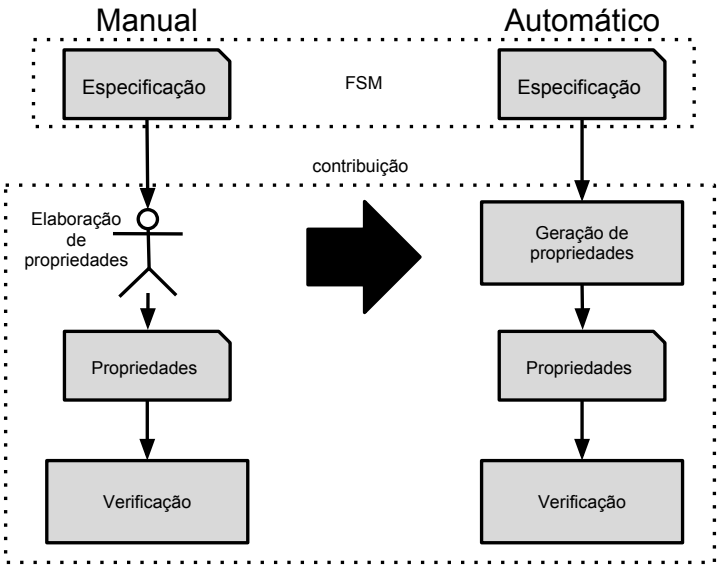

Figura 6: Elaboração manual vs Geração automática

Os modelos são utilizados na geração de propriedades de acordo com três diferentes heurísticas descritas a seguir.

4.1 ANÁLISE EM NÍVEL DE MÓDULO

A primeira heurística realiza a geração de propriedades para um módulo do sistema. Para isso, são utilizadas máquinas de estados que modelam o funcionamento do módulo. O estratégia é a seguinte:

Estratégia 1. *A geração de propriedades segue o princípio da síntese de projeto de circuitos sequenciais utilizando tabelas de transições de estados (VAHID, 2010). Esta tabela é utilizada para definir propriedades* safety *que verifiquem a alcançabilidade dos estados de acordo com as variáveis de controle do módulo. As propriedades* liveness *são geradas nos casos em que o estados têm transição para si mesmos ou existe uma transição para um estado final.*

O Algoritmo 1 descreve a estratégia proposta para analisar a máquina de estados do módulo.

O algoritmo tem uma máquina de estados como entrada. Na função ***setNextProperty*** o operador temporal *Next* é empregado na propriedade e verificará se a partir do estado atual a execução alcançará o estado esperado, levando em consideração se existem condições para transição ou se a transição é incondicional.

A função ***setNotLockedProperty*** define uma propriedade que verificará se o fluxo de execução não permanecerá em *loop* infinitamente em um estado que têm transição para si mesmo. Duas variações da função podem ser observadas nas linhas 5 e 7, onde em um caso a transição é condicional e em outro caso não. Este teste é importante visto que em sistemas reativos é comum a existência de estados *idle*, estados em que o sistema se mantém ocioso enquanto espera por eventos externos.

Se há estados de erro, a função ***setStErrorProp*** gera uma propriedade *safety*, afirmando que o estado de erro nunca será alcançado.

A função ***setReachableFinalSt*** é chamada sempre que o algoritmo alcança um estado final. Essa função define uma propriedade que verificará se estados finais são alcançáveis a partir do estado inicial.

4.1.1 Exemplo

Na máquina de estados da Figura 7, $s1$ é o estado inicial e tem transições para $s2$ e para ele mesmo. $T2$, $T3$, $T4$ e $T5$ são transições condicionais. $T1\_unc$ é uma transição incondicional. *error* e $s3$ são estados finais.

Executando o algoritmo, são obtidas as propriedades na Tabela 3. A propriedade 1 é gerada pela função ***setNotLockedProperty*** sobre o estado $s1$,

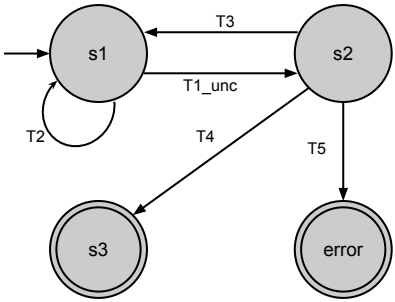

Figura 7: FSM exemplo

Tabela 2: Tabela de transição de estados (VAHID, 2010) do exemplo da Figura 7

Estado inicial	Entrada	Próximo estado
$s1$	T1_unc	$s2$
$s1$	T2	$s1$
$s2$	T3	$s1$
$s2$	T4	$s3$
$s2$	T5	*error*

uma vez que $s1$ tem transições para ele 1mesmo. A propriedade 2 é gerada pela função ***setNextProperty*** na linha 11, pois $s1$ tem uma transição incondicional para $s2$. As propriedades 3, 4 e 5 são geradas pela função ***setNextProperty*** na linha 13. A propriedade 6 é gerada pela função ***setStErrorProp***. A propriedade 7 é gerada pela função ***setReachableFinalSt***.

4.2 ANÁLISE EM NÍVEL DE CONCORRÊNCIA ENTRE MÓDULOS

A geração de propriedades em módulos isolados pode não incluir propriedades importantes relacionadas com a concorrência e interação entre os módulos ou máquinas de estados. Um exemplo onde a concorrência entre máquinas de estados é uma séria preocupação é quando elas compartilham a mesma região de memória, podendo levar a problemas de consistência durante a leitura e escrita. Propriedades que verifiquem este cenário são de grande relevância.

Tabela 3: Exemplo de geração de propriedades

	Property
1	$s1 \rightarrow (\mathbf{F\ not}\ T2)$
2	$s1 \rightarrow \mathbf{X}s2$
3	$s2\ \mathbf{and}\ T3\ \rightarrow \mathbf{X}s1$
4	$s2\ \mathbf{and}\ T4\ \rightarrow \mathbf{X}s3$
5	$s2\ \mathbf{and}\ T5\ \rightarrow \mathbf{X}error$
6	$\mathbf{G}(\mathbf{not}\ error)$
7	$s1 \rightarrow \mathbf{F}(s3)$

Conforme observado na literatura (BAIER; KATOEN, 2008), uma abordagem para lidar com este cenário é realizando o produto cartesiano das máquinas de estados, resultando em uma única e complexa máquina. A proposta deste trabalho é utilizar diagramas de sequência para prover informação sobre a interação entre módulos e assim, gerar propriedades adicionais, conforme estratégia a seguir.

Estratégia 2. *Quando dois módulos manipulam dados compartilhados, diagramas de sequência são usados para definir propriedades. Este diagrama descreve o cenário de quando existe uma dependência entre módulos.*

O Algoritmo 2 descreve essa heurística. Dois módulos do sistema são analisados; o Algoritmo *ModulePropGen* (Algoritmo 1) é chamado para cada módulo (linhas 1 e 2) a fim de definir propriedades locais de cada um. Então, cada evento no diagrama de sequência (uma transição para si mesmo ou uma troca de mensagem) é considerado. Se uma interação entre eles é detectada, uma propriedade que verifica a ordem dos eventos é gerada.

4.2.1 Exemplo

A Figura 8 exemplifica o caso citado anteriormente. Os módulos $frame\_mounter$ e $buffer\_controller$ implementam um suavizador de transições entre *frames* de vídeo. A principal função deles é que o vídeo só seja mostrado quando todos os *pixels* de vídeo no *frame* já foram carregados.

O comportamento de cada módulo pode ser observado na Figura 9. Os módulos compartilham a mesma região de memória, isto é, leem e escrevem no mesmo sinal. Neste caso, quando o módulo $frame\_mounter$ termina de montar o próximo bloco a ser exibido pelo controlador de *frame buffer*, ele

avisa ao módulo *buffer_controller* por meio do dado de controle *buf_ready* que o *buffer* está pronto. Por sua vez, o módulo *buffer_controller* disponibiliza o dado e confirma ao *frame_mounter* por meio do dado de controle *reader_done* que a leitura já foi realizada.

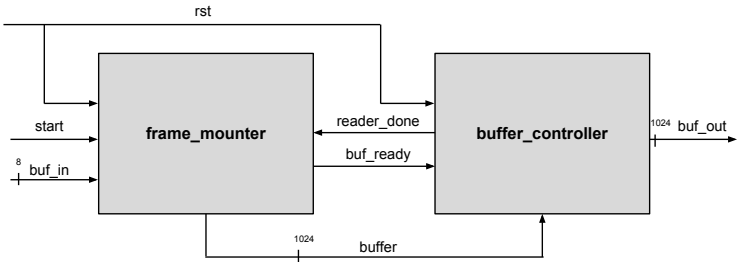

Figura 8: Diagrama de blocos de um sistema de suavização de transição de *framebuffer* de video

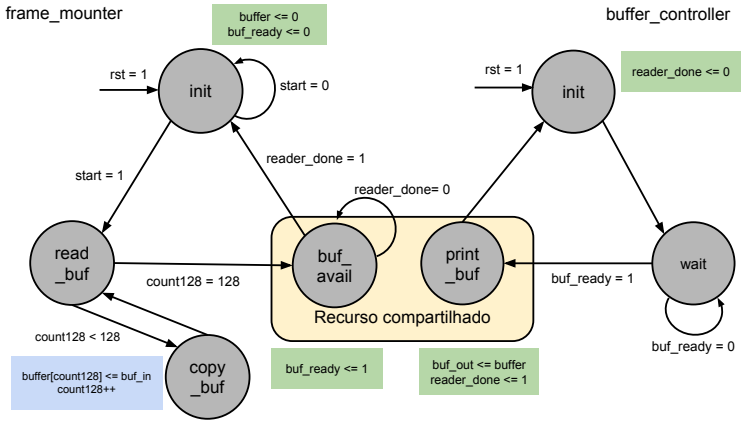

Figura 9: Exemplo de máquinas de estados concorrentes

Diagramas de sequência são usualmente utilizados para modelar a troca de mensagens entre processos. Neste trabalho, os diagramas de sequência são utilizados para modelar esta interação entre módulos de hardware. A Figura 10 mostra a modelagem em diagrama de sequência do comportamento dos módulos descritos na Figura 9.

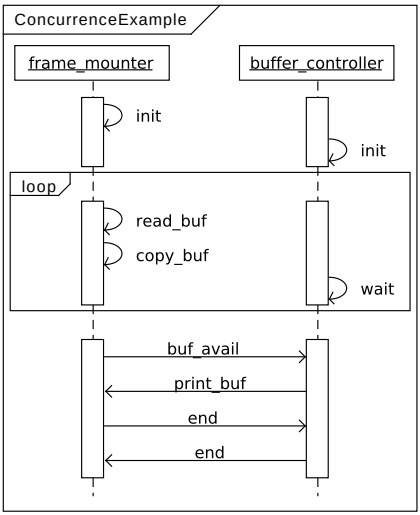

Figura 10: Exemplo de diagrama de sequência

Utilizando a abordagem proposta no exemplo citado, além das propriedades geradas para cada módulo com a execução do Algoritmo *ModulePropGen*, duas propriedades adicionais são geradas, conforme a Tabela 4.

Tabela 4: Exemplo de propriedades geradas a partir de diagramas de sequência

	Property
1	$\mathbf{G}(frame\_mounter.buf\_avail$ **AND** $buf\_ready = 1 \rightarrow \mathbf{X}\ buffer_controller.print\_buf)$
2	$\mathbf{G}(frame\_mounter.end$ **AND** $reader\_done = 1 \rightarrow \mathbf{X}\ buffer\_controller.end)$

4.3 ANÁLISE EM NÍVEL DE SISTEMA

Para realizar a verificação em nível de sistema é necessário aplicar os Algoritmos *ModulePropGen* e *ParseSequenceDiagram*, conforme a seguir.

Estratégia 3. *As propriedades são geradas para todos os módulos do sistema por meio da Estratégia 1. Caso dois módulos, ou dois processos em*

um mesmo módulo, realizem leitura e escrita em recursos compartilhados concorrentemente, a geração de propriedades é feita por meio da Estratégia 2.

O Algoritmo 3 descreve a heurística para executar a verificação em nível de sistema. O algoritmo visita cada módulo do sistema e averígua se este módulo faz mapeamento de portas com outros módulos. Quando o módulo faz o mapeamento das portas e realiza leitura/escrita em um recurso compartilhado (linha 4), consistindo de dois módulos com sinais que escrevem e leem em ambos, o Algoritmo *ParseSequenceDiagram* (Algoritmo 2) é chamado. Caso contrário, o módulo é analisado pelo Algoritmo *ModulePropGen*.

Algorithm 1 ModulePropGen(*FSM*)

```
1: for FSM.states as state do
2:     for state.transitions as tr do
3:         if state == tr.target then
4:             if tr.condition == NULL then
5:                 setNotLockedProperty(state)
6:             else
7:                 setNotLockedProperty(state, tr.condition)
8:             end if
9:         else
10:            if tr.condition == NULL then
11:                setNextProperty(state, tr.target)
12:            else
13:                setNextProperty(state, tr.target, tr.condition)
14:            end if
15:        end if
16:    end for
17:    if state.isError() then
18:        setStErrorProp(state)
19:    end if
20:    if state.isFinal() then
21:        setReachableFinalSt(state)
22:    end if
23: end for
```

Algorithm 2 ParseSequenceDiagram(*mod*1, *mod*2)

```
1: ModulePropGen(mod1)
2: ModulePropGen(mod2)
3: for (events as event) do
4:     if hasInteraction(event) then
5:         setNextProperty(event.from, event.to)
6:     end if
7: end for
```

Algorithm 3 PropertyGen(*modules*)

```
1:  for modules as module do
2:      mod = hasPortMapping(module)
3:      if mod <>NULL then
4:          if not rwSameResource(module, mod) then
5:              ModulePropGen(mod)
6:          else
7:              ParseSequenceDiagram(module, mod)
8:          end if
9:      else
10:         ModulePropGen(module)
11:     end if
12: end for
```

5 PROPGEN: FRAMEWORK PARA GERAÇÃO AUTOMÁTICA DE PROPRIEDADES

Este capítulo descreve a ferramenta PropGen, o protótipo que implementa a abordagem de geração de propriedades proposta nesta dissertação. Seu fluxo de operação pode ser observado na Figura 11.

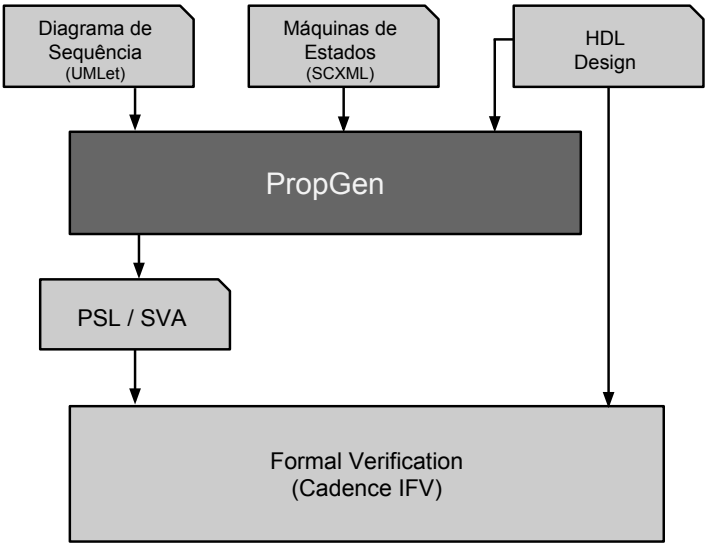

Figura 11: Fluxo de verificação usando o PropGen e o IFV model checker (CADENCE, 2013)

Para manipular as máquinas de estados e diagramas de sequência em um programa computacional é necessário representá-los em uma forma textual. O PropGen reconhece uma linguagem de máquinas de estados baseadas em evento conhecida como *State Chart XML* (SCXML) (SCXML, 2013). O SCXML permite modelar sistemas de alta complexidade devido a sua capacidade de manipular estados atômicos, paralelos e hierárquicos. Existem algumas ferramentas, de código aberto e proprietárias, que permitem a edição gráfica de máquinas de estado e que as geram no formato SCXML. Neste trabalho uma ferramenta de código aberto chamada *scxmlgui* (MORBINI, 2013) foi utilizada.

Para manipular diagramas de sequência o PropGen utiliza o formato

de texto utilizado pela ferramenta UMLet (UMLET, 2013). O código a seguir é a descrição textual do diagrama de sequência da Figura 10, no Capítulo 4.

```
title: ConcurrenceExample
_frame_mounter~id1_|_buffer_controller~id2_

id1->id1:id2,id1:init
id2->id2:id1,id2:init

iframe{:loop
id1->id1:id2,id1:read_buf
id1->id1:id2,id1:copy_buf
id2->id2:id1,id2:wait
iframe}

id1->id2:id1,id2:buf_avail
id2->id1:id2,id1:print_buf

id1->id2:id1,id2:end
id2->id1:id2,id1:end
```

5.1 ESTRUTURA

O ProGen é uma ferramenta de linha de comando, multi-plataforma e modular. Os módulos do sistema encapsulam classes em três diferentes componentes: analisador de VHDL, analisador de modelos, e gerador de propriedades.

O PropGen foi implementado na linguagem Java. Os motivos da escolha da linguagem foram considerando a disponibilidade e facilidade de uso de bibliotecas para manipulação de XML e de criação de interface gráfica de usuário, uma vez que inicialmente pretendia-se criar uma ferramenta gráfica.

O diagrama de classes do projeto pode ser observado na Figura 12. As classes *VHDLAnalyzer*, *VHDLPort*, *VHDLSignal*, *VHDLPortMap* compõem o analidor VHDL. As classes *StateMachine*, *State*, *Transition*, *SequenceDiagram*, e *Interaction* compõem o analisador de modelos. E as classes *PropGen*, *ModulePropGen* e *SystemPropGen* compõem o gerador de propriedades.

O analisador VHDL é responsável por ler o código em VHDL e reconhecer as estruturas de cada módulo, isto é, as portas de entrada e saída, sinais internos, o tipo destes, e o mapeamento entre portas e sinais. Estas informações serão utilizadas posteriormente nas propriedades em PSL ou

SVA.

O analisador de modelos interpreta a descrição dos modelos de máquinas de estado e diagramas de sequência através do SCXML e arquivo do UMLet, respectivamente.

Por fim, o gerador de propriedades percorre os modelos interpretados pelo analisador de modelos e identifica as propriedades. Para gerá–las em PSL e SVA para verificação da implementação em VHDL é necessário fazer a correspondência entre os estados dos modelos e os sinais que os representam na implementação.

5.2 FUNCIONAMENTO

O PropGen solicita algumas informações do sistema de hardware, tais como os sinais de *clock* e *reset*, e sinais de estado. Sinais de estados sinalizam o estado corrente na máquina de estados da execução do módulo. Com esses dados a ferramenta realiza geração das propriedades de acordo com os algoritmos apresentados no Capítulo 4. A saída da ferramenta são as propriedades descritas em PSL ou SVA.

Os principais comandos da ferramenta são:

set_design_dir – informa ao PropGen qual é o diretório em que a implementação está localizada;

set_scxml_file – define o arquivo SCXML correspondente a algum módulo no diretório da implementação;

mod_propgen – realiza a geração de propriedades para determinado módulo;

set_sd_file – define o arquivo do diagrama de sequência correspondente a interação entre dois módulos;

sys_propgen – realiza a geração de propriedades que verificam a interação entre módulos.

As figuras 13, 14 e 15 mostram exemplos de utilização de alguns dos comandos.

Pode-se observar na Figura 14 que após informar o módulo e seu respectivo arquivo SCXML, uma série de opções são exibidas ao usuário. Assim, a ferramenta faz a ligação entre o modelo e a implementação e possibilita gerar propriedades que verificam a implementação com ferramentas que realizam verificação baseadas em asserções, como é o caso do IFV (CADENCE, 2013).

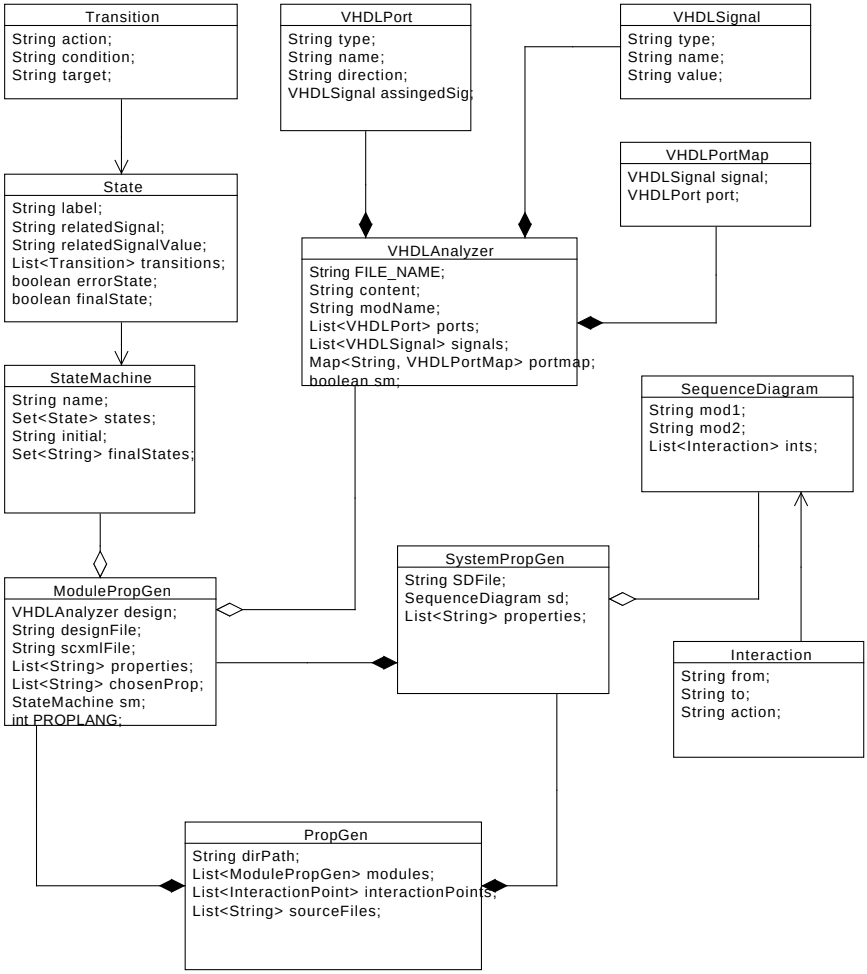

Figura 12: Diagrama de classes do PropGen

```
PropGen> set_design_dir /home/wesley/Msc/PropertyExtractor/data/tm_ack_layer/
Read file: /home/wesley/Msc/PropertyExtractor/data/tm_ack_layer//tm_ack_layer.vhd
Entity: tm_ack_layer
Read file: /home/wesley/Msc/PropertyExtractor/data/tm_ack_layer//utmc.vhd
Read file: /home/wesley/Msc/PropertyExtractor/data/tm_ack_layer//reset_sync.vhd
Entity: reset_sync
Done.
PropGen>
```

Figura 13: Definição do diretório de implementação

```
PropGen> set_scxml_file tm_ack_layer /home/wesley/Msc/PropertyExtractor/data/tm_ack_layer/scxml/tm_ack_layer.scxml
Please set the corresponding signal for the state 'set_ack':
(0) CS : ack_layer_state
(1) NS : ack_layer_state
(2) ack_type_s : std_logic
(3) ack_data_sv : std_logic_vector (39 downto 0)
(4) packet_ack_sv : std_logic_vector (183 downto 0)
(5) sequence_count_sv : std_logic_vector (13 downto 0)
(6) busy_s : std_logic
(7) rst_sync_n_s : std_logic
: 0
Selected: 0
Please set the corresponding CS value for the state 'set_ack':
Type ack_layer_state
(0) st_rst : ack_layer_state
(1)  st_idle : ack_layer_state
(2)  st_get_ack : ack_layer_state
(3)  st_set_ack : ack_layer_state
(4)  st_end_ack : ack_layer_state
(5)  st_inc_num_sequence : ack_layer_state
(6)  st_create_data_header ) : ack_layer_state
: 3
Selected: 3
CS =  st_set_ack
```

Figura 14: Importação de arquivo SCXML

```
PropGen> mod_propgen tm_ack_layer
module: tm_ack_layer
Print PSL properties
Module: tm_ack_layer
scxml: /home/wesley/Msc/PropertyExtractor/data/tm_ack_layer/scxml/tm_ack_layer.scxml
-- psl assert_p0: assert always ( (CS =  st_set_ack) -> next (CS =  st_end_ack) );
-- psl assert_p1: assert always ( (CS =  st_end_ack) -> next (CS =  st_idle) );
-- psl assert_p2: assert always ( (CS =  st_get_ack) -> next (CS =  st_idle) );
-- psl assert_p3: assert always ( (CS =  st_idle) -> eventually! (not (CS =  st_idle)) );
-- psl assert_p4: assert always ( ( (ready_i = '1' and busy_s = '0') and (CS =  st_idle) ) -> next (CS =  st_get_ack) );
-- psl assert_p5: assert always ( ( (ready_frame_i = '1') and (CS =  st_idle) ) -> next (CS =  st_set_ack) );
-- psl assert_p6: assert always ( (CS = st_rst) -> next (CS =  st_idle) );
PropGen>
```

Figura 15: Geração de propriedades do módulo selecionado

6 ESTUDO DE CASO

A Unidade Telemetria e Telecomando (UTMC)[1] é um sistema em chip (*System-on-a-Chip*, SoC) cuja função é realizar a comunicação entre estação terrestre e um satélite. A UTMC foi desenvolvida de acordo com os requisitos fornecidos pelo INPE, normas da *European Space Agency* (ESA) e recomendações do *Consultative Comitte for Space Data Systems* (CCSDS). A próxima seção abordará com mais detalhes o funcionamento da UTMC.

6.1 DESCRIÇÃO DA UTMC

A UTMC faz parte de um sistema chamado *Attitude, Control and Data Handler* (ACDH), um sistema de plataforma orbital (satélite), que tem a função de troca de mensagens de serviço com estações de controle de missão na Terra. O ACDH é composto por um computador de bordo (OBC, *on-board computer*), sensores e atuadores, como pode ser visto na Figura 16 (BEZERRA; SILVA; ROCHAT, 2009). O sistema de comunicação, a UTMC, recebe os dados da estação terrestre por meio de rádio frequência (RF). Esses dados, chamados de Telecomando (TC), incluem comandos para correção de órbita e atitude do veículo espacial, e comandos para ligar e desligar motores. A UTMC também envia dados, chamados Telemetria (TM), coletados pelo OBC, dos diversos instrumentos ligados a ele, ou uma resposta a algum telecomando. O OBC empacota-os e envia-os à UTMC, que transmite via RF para a Terra. A Figura 17 mostra o *hardware* final do subsistema.

É importante ressaltar que a UTMC processa informações referentes ao estado de funcionamento do veículo espacial. Desta forma, uma falha neste sistema poderá resultar no fim da missão. Esta característica a torna um componente crítico.

As recomendações CCSDS (ECSS, 2003) criam uma política que determina como estas informações (TC e TM) devem ser trocadas. As recomendações sugerem uma série de camadas, num formato de "pilha de protocolos", onde são definidas tarefas para cada camada. O fluxo de comunicação para telecomando e telemetria utiliza as camadas de codificação, transferência e empacotamento, de acordo com os requisitos delimitados pelo INPE. A Figura 18 (BEZERRA; SILVA; ROCHAT, 2009) descreve o fluxo de TC e TM, exibindo as camadas que foram implementadas.

O módulo de RF encaminha os dados recebidos da estação terrestre

[1] O SoC foi desenvolvido inteiramente na linguagem de descrição de hardware VHDL (IEEE..., 1994), não sendo assim um sistema que combina software e hardware.

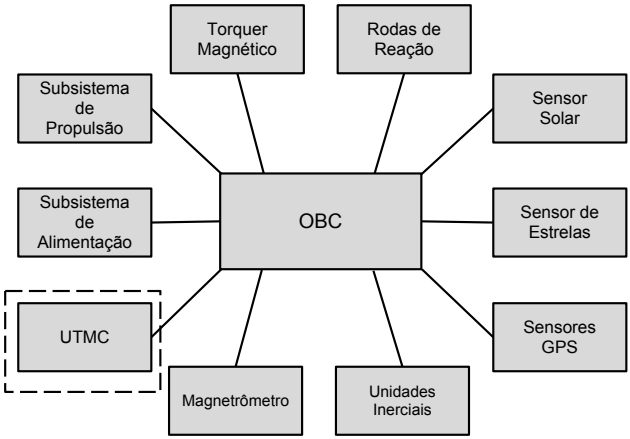

Figura 16: Diagrama de blocos do ACDH

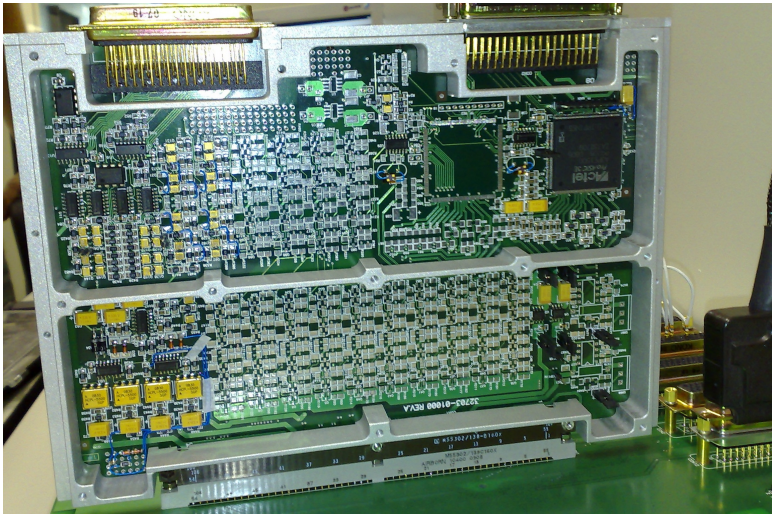

Figura 17: UTMC em FPGA

em forma de *Command Link Transfer Units* (CLTUs) para o fluxo de TC. As CLTUs são repassadas para a camada de codificação, onde é aplicado o

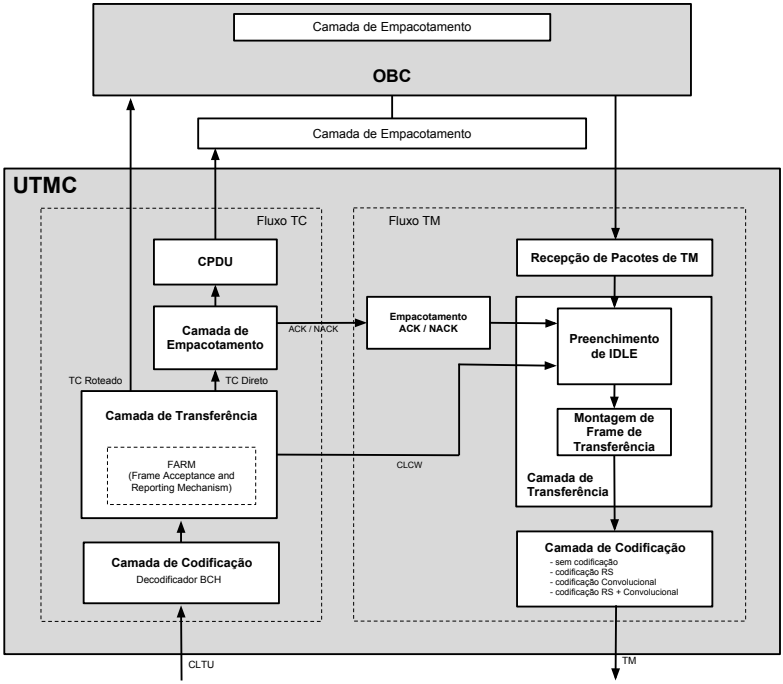

Figura 18: Fluxo de telecomando e telemetria

algoritmo BCH (BOSE; RAY-CHAUDHURI, 1960), que é responsável por decodificar o *frame* recebido e corrigir possíveis erros de transmissão.

Após a decodificação, os dados vão para o módulo *Frame Acceptance and Reporting Mechanism* (FARM), que situa-se na camada de transferência. A função do FARM é aceitar ou rejeitar o frame recebido. Caso o *frame* seja rejeitado, o fluxo de TM é acionado, solicitando o reenvio da transmissão por parte da estação terrestre. Caso o *frame* seja aceito há duas variações de fluxo. A primeira variação, se o telecomando for do tipo "direto" (um telecomando de alta prioridade deve ser encaminhado diretamente aos instrumentos sem a intervenção do OBC) este é enviado à camada de empacotamento. A segunda variação é o tipo "roteado", que é encaminhado ao OBC para processamento.

No fluxo de TM a camada de empacotamento é utilizada exclusivamente para a geração de pacotes ACK/NACK referente ao *status* dos telecomandos diretos enviados para os instrumentos. Os dados processados pelo OBC são empacotados no próprio OBC e enviados para a camada de trans-

ferência. A camada de transferência no fluxo de TM recebe os pacotes (do OBC ou da camada de empacotamento) e constrói o *frame* para telemetria. Este *frame* é enviado para a camada de codificação, onde uma das quatro opções de codificação são aplicadas: sem codificação, codificação Reed-Solomon (RS), codificação convolucional, ou codificação RS + convolucional. O *frame* resultante é enviado via RF para a estação terrestre. Este processo pode ser visualizado na Figura 18.

6.2 VALIDAÇÃO DA UTMC

Para realizar a validação da UTMC, a equipe de desenvolvimento projetou um equipamento conhecido como Equipamento de Suporte Elétrico (ESE), cuja finalidade é prover uma comunicação da UTMC com um dispositivo externo através de portas seriais assíncronas e disponibilizar um painel de eventos (FERREIRA; SILVA; VILLA, 2009).

Além da validação elétrica, foi desenvolvido no ESE um software que gera vetores de entrada para simulação dos módulos da UTMC. A Figura 19 mostra o ESE da UTMC.

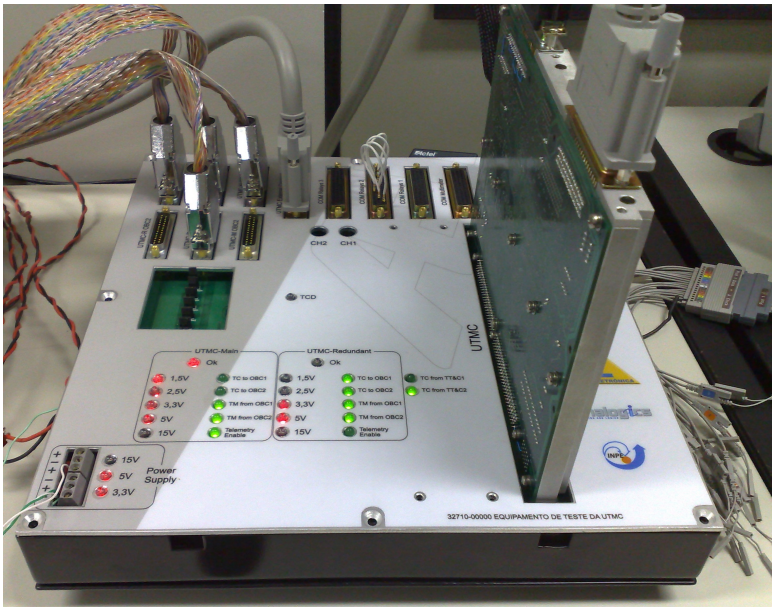

Figura 19: Placa de testes

6.3 FLUXO DE TELEMETRIA

Como estudo de caso para a abordagem de geração de propriedades, foi delimitada a realização da verificação de módulos da UTMC no fluxo de telemetria, de forma que a verificação do fluxo de telecomando é um trabalho posterior. Esta seção explanará com mais detalhes o funcionamento dos módulos que realizam a telemetria da UTMC.

O quadro à direita na Figura 18 descreve as camadas implementadas no fluxo de TM, bem

A camada de empacotamento separa os dados de TM em pacotes para a entrega fim-a-fim dos dados de aplicação. Na camada de transferência os pacotes são encapsulados em uma estrutura chamada *frame* de transferência, de forma que os dados sejam entregues à estação terrestre de maneira confiável. A camada de codificação recebe o *frame* de transferência e prepara-o de forma a ser protegido contra erros concernentes a um meio de comunicação sujeito a ruídos. A camada física não está descrita na figura pois não foi parte do escopo de implementação da UTMC.

6.3.1 Camada de empacotamento

Os pacotes manipulados pelo fluxo de TM são os vindos do OBC e os pacotes de ACK e NACK em resposta a telecomandos diretos (TCD). Conforme definido pelo INPE, os pacotes podem ter o campo de dados no tamanho máximo de 1097 bytes. O pacote contém ainda outros 6 bytes referentes ao cabeçalho do pacote, onde são descritas informações como a origem, o tamanho e a sequência do pacote. Assim, os pacotes manipulados pela UTMC são de tamanho máximo de 1103 bytes.

A maior parte das funcionalidades da camada de empacotamento não se encontram na UTMC, mas sim no OBC, conforme pode ser observado na Figura 18. Os únicos pacotes montados na UTMC são os pacotes de ACK e NACK, em confirmação de recebimento de TCD.

6.3.2 Camada de Transferência

A função da camada de transferência é prover uma estrutura para o transporte confiável de pacotes na comunicação entre o satélite e a estação terrestre. Os dados recebidos da camada de empacotamento, oriundos do OBC ou do módulo de Empacotamento ACK / NACK, são encapsulados em *frames* na camada de transferência e repassados à camada de codificação.

6.3.3 Camada de Codificação

Os *frames* formados na camada de transferência são repassados à camada de codificação. Esses dados são codificados, ou não, e encaminhados para envio serial. Existem quatro alternativas de codificação:

1. Codificação Reed Solomon (RS);
2. Codificação Convolucional;
3. Codificação Reed Solomon (RS) e convolucional;
4. Nenhuma codificação.

Para cada codificação são gerados um arquivo de configuração diferente para a UTMC. Neste caso, antes de cada missão é definida a abordagem de codificação a ser utilizada, e o arquivo de configuração respectivo é utilizado no FPGA.

7 RESULTADOS

O fluxo de verificação proposto neste trabalho (Figura 11) foi utilizado para verificar os módulos no fluxo de telemetria da UTMC. Os experimentos foram conduzidos em um computador Intel Pentium Dual-Core 2.10 GHz, 3 GB RAM, com o sistema operacional Linux.

As propriedades foram geradas utilizando o PropGen e verificadas pelo IFV. Os resultados da verificação das propriedades, apresentados nas tabelas que seguem, se dividem em três diferentes casos:

- Provado – a propriedade foi verificada em todo o espaço de estados do projeto e não foi encontrado falha.

- Falha – a propriedade foi violada em algum momento na execução do sistema. Neste caso o IFV provê o contra–exemplo, tornando possível encontrar o motivo do erro por meio de simulação.

- Explorado – o IFV não conseguiu percorrer todo o espaço de estados para definir se a propriedade foi provada ou não. Este resultado varia de acordo com a capacidade de processamento do computador que está sendo utilizado.

- Eliminado – durante a combinação entre modelo e implementação no PropGen averigua-se a presença de estados no modelo que não tenham correspondente na implementação. Para estes casos as propriedades são descartadas.

As tabelas apresentam também o tempo usado pela CPU para verificar a propriedade e a memória consumida pelos mecanimos de verificação. Estes dados são informados pela ferramenta IFV.

7.1 VERIFICAÇÃO DOS MÓDULOS DE TELEMETRIA

Esta seção apresentará o comportamento dos módulos descritos no Capítulo 6 referentes à montagem do pacote de telemetria, sua descrição em máquinas de estados e as propriedades geradas e verificadas.

As legendas para os estados e transições das máquinas de estado nesta seção podem ser encontradas no Anexo A.

7.1.1 Módulo Empacotamento ACK / NACK

O módulo de Empacotamento ACK / NACK reside na camada de empacotamento e é responsável por: (1) Receber pacotes de Telecomandos Diretos; (2) Realizar o empacotamento e a geração de pacotes ACK e NACK; e (3) Armazenar em um *buffer* para subsequente entrega para a camada de transferência. O comportamento deste módulo foi modelado em máquina de estado conforme a Figura 20.

As máquinas de estados utilizadas na geração e apresentadas neste capítulo foram retiradas do trabalho apresentado por Madeira e Bezerra (2008).

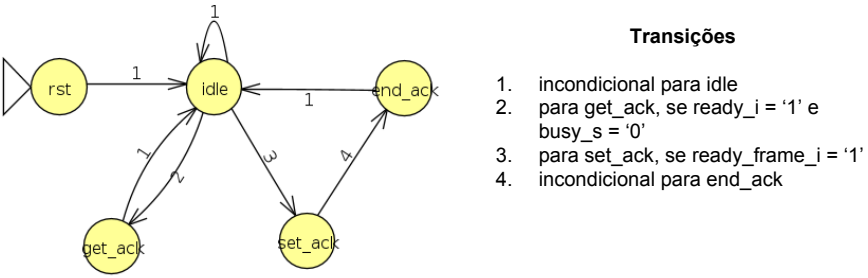

Figura 20: Máquina de estados para o módulo Empacotamento ACK / NACK

O estado **rst** é o estado inicial e é responsável pela inicialização do módulo. O próximo estado após um ciclo de *clock* é o **idle**. Se o sinal *ready_i* é *verdadeiro* e o sinal *busy_s* é $falso$, existe um pacote (i.e., ACK ou NACK) pronto para ser recebido do fluxo de telecomando. Assim, o próximo estado será o **get_ack**. Por outro lado, se o sinal *ready_frame_i* é *verdadeiro*, então existe um pacote pronto para ser enviado pelo fluxo de telemetria e assim o próximo estado será o **set_ack**. Se nenhuma destas condições forem *verdadeiras* o módulo permanece no estado **idle**.

A partir desta máquina de estados a ferramenta PropGen gerou propriedades em PSL, que podem ser vistas na Tabela 5. Por ser um módulo pequeno, a verificação demandou pouco tempo e memória, conforme os resultados observados na Tabela 6.

Tabela 5: Propriedades geradas no módulo Empacotamento ACK / NACK

Propriedade	Resultado
assert_p0: assert always ((CS = st_end_ack) ->next (CS = st_idle));	Provado
assert_p1: assert always ((CS = st_idle) ->next eventually! ((CS = st_get_ack) or (CS = st_set_ack)));	Provado
assert_p2: assert always (((ready_i = '1') and (busy_s = '0') and (CS = st_idle)) ->next (CS = st_get_ack));	Provado
assert_p3: assert always (((ready_frame_i = '1') and (CS = st_idle) and not ((ready_i = '1') and (busy_s = '0'))) ->next (CS = st_set_ack));	Provado
assert_p4: assert always ((CS = st_get_ack) ->next (CS = st_idle));	Falha
assert_p5: assert always ((CS = st_set_ack) ->next (CS = st_end_ack));	Provado
assert_p6: assert always ((CS = st_rst) ->next (CS = st_idle));	Provado

Tabela 6: Tempo e memória consumidos pela verificação do módulo Empacotamento ACK / NACK

Propriedade	Tempo (s)	Mem. (MB)
assert_p0	0.24	256
assert_p1	0.12	256
assert_p2	0.12	256
assert_p3	0.13	256
assert_p4	0.07	256
assert_p5	0.21	256
assert_p6	0.18	256

A Figura 21 mostra o contra–exemplo da falha encontrada na propriedade *assert_p*4. A propriedade afirma que *st_idle* é o próximo estado após *st_get_ack*, mas na verdade, o próximo estado é *st_create_data_header*. Entretanto, o estado *st_create_data_header* não está na especificação do sistema, existindo assim uma inconsistência entre implementação e especificação. É necessário fazer a análise da falha e identificar qual é a fonte do problema, se

é a especificação, a implementação ou a propriedade.

Neste caso, o resultado da análise foi que durante a implementação os engenheiros de desenvolvimento perceberam a necessidade de criar outros estados, mas a especificação deixou de ser atualizada.

Figura 21: Contra–exemplo da propriedade assert_p4

7.1.2 Módulo Montagem de Frame e Transferência

Este módulo está situado na camada de transferência e realiza três tarefas: (1) preenchimento do cabeçalho do frame de TM; (2) inserção do CLCW no frame; e (3) cálculo do CRC e posterior inserção deste no frame. A Figura 22 apresenta a máquina de estados desse módulo. A seção A.1, no Anexo A indicam as legendas da Figura 22.

A Tabela 7 mostra as propriedades geradas para este módulo. A Tabela 8 apresenta o processamento (tempo e memória) gasto para verificação de cada propriedade.

Tabela 7: Propriedades geradas no módulo Montagem de Frame de Transferência

Propriedade	Resultado
assert_p0: assert always ((CS = st_mem_write_crc4) ->next (CS = st_update_crc))	Explorado
assert_p1: assert always (((mem_id_i = ID_TM_FRAME and mem_ack_i = '1') and (CS = st_mem_write2)) ->next (CS = st_mem_write3))	Provado
assert_p2: assert always ((CS = st_mem_write2) ->eventually! not (mem_id_i /= ID_TM_FRAME or mem_ack_i /= '1'))	Provado
assert_p3: assert always ((CS = st_update_pointers) ->next (CS = st_pre_write))	Provado
continuado na próxima página	

Tabela 7 – continuado da página anterior

Propriedade	Resultado
assert_p4: assert always ((CS = st_end_frame) ->next (CS = st_idle))	Explorado
assert_p5: assert always ((CS = st_mem_write4) ->next (CS = st_update_pointers))	Provado
assert_p6: assert always ((CS = st_rst) ->next (CS = st_idle))	Provado
assert_p7: assert always ((CS = st_update_crc) ->next (CS = st_pre_write_crc))	Explorado
assert_p8: assert always (((count_read_sv <1113) and (CS = st_pre_read)) ->next (CS = st_read1))	Explorado
assert_p9: assert always (((count_read_sv >= 1113) and (CS = st_pre_read)) ->next (CS = st_pre_write_crc))	Explorado
assert_p10: assert always ((CS = st_mem_write1) ->next (CS = st_mem_write2))	Provado
assert_p11: assert always (((crc_index_sv = 0) and (CS = st_crc_calc2)) ->next (CS = st_pre_read))	Falha
assert_p12: assert always (((crc_index_sv /= 0) and (CS = st_crc_calc2)) ->next (CS =))	Eliminado
assert_p13: assert always (((count_bytes_sv >0) and (CS = st_pre_write)) ->next (CS = st_mem_write1))	Provado
assert_p14: assert always (((count_bytes_sv <= 0) and (CS = st_pre_write)) ->next (CS = st_end_write))	Provado
assert_p15: assert always ((CS = st_idle) ->eventually! not (ready_i = '0'))	Provado
assert_p16: assert always (((ready_i = '1') and (CS = st_idle)) ->next (CS = st_get_variables))	Provado
assert_p17: assert always ((CS =) ->next (CS = st_crc_pre_calc))	Eliminado
assert_p18: assert always (((mem_id_i = ID_TM_FRAME and mem_ack_i = '1') and (CS = st_mem_write_crc2)) ->next (CS = st_mem_write_crc3))	Explorado
assert_p19: assert always ((CS = st_mem_write_crc2) ->eventually! not (mem_id_i /= ID_TM_FRAME or mem_ack_i /= '1'))	Provado
assert_p20: assert always (((mem_ack_i = '0') and (CS = st_mem_write_crc3)) ->next (CS = st_mem_write_crc4))	Explorado
assert_p21: assert always ((CS = st_mem_write_crc3) ->eventually! not (mem_ack_i = '1'))	Provado

continuado na próxima página

Tabela 7 – continuado da página anterior

Propriedade	Resultado
assert_p22: assert always ((CS = st_read1) ->next (CS = st_read2))	Explorado
assert_p23: assert always ((CS = st_mem_write3) ->eventually! not (mem_ack_i /= '0'))	Provado
assert_p24: assert always (((mem_ack_i = '0') and (CS = st_mem_write3)) ->next (CS = st_mem_write4))	Provado
assert_p25: assert always ((CS = st_end_write) ->next (CS = st_pre_read))	Provado
assert_p26: assert always ((CS = st_pre_write_crc) ->eventually! not (crc_count_byte_sv >= 2 and busy_i = '1'))	Explorado
assert_p27: assert always (((crc_count_byte_sv <2) and (CS = st_pre_write_crc)) ->next (CS = st_mem_write_crc1))	Explorado
assert_p28: assert always (((crc_count_byte_sv >= 2 and busy_i = '0') and (CS = st_pre_write_crc)) ->next (CS = st_coding_ready))	Explorado
assert_p29: assert always ((CS = st_mem_write_crc1) ->next (CS = st_mem_write_crc2))	Explorado
assert_p30: assert always ((CS = st_read4) ->next (CS = st_crc_pre_calc))	Falha
assert_p31: assert always ((CS = st_read3) ->eventually! not (mem_ack_i = '1'))	Provado
assert_p32: assert always (((mem_ack_i = '0') and (CS = st_read3)) ->next (CS = st_read4))	Explorado
assert_p33: assert always ((CS = st_coding_ready) ->next (CS = st_end_frame))	Explorado
assert_p34: assert always ((CS = st_read2) ->eventually! not (mem_id_i /= ID_TM_FRAME or mem_ack_i /= '1'))	Provado
assert_p35: assert always (((mem_id_i = ID_TM_FRAME and mem_ack_i = '1') and (CS = st_read2)) ->next (CS = st_read3))	Explorado
assert_p36: assert always ((CS = st_get_variables) ->next (CS = st_pre_write))	Provado
assert_p37: assert always ((CS = st_crc_calc1) ->next (CS = st_crc_calc2))	Explorado
assert_p38: assert always ((CS = st_crc_pre_calc) ->next (CS = st_crc_calc1))	Falha

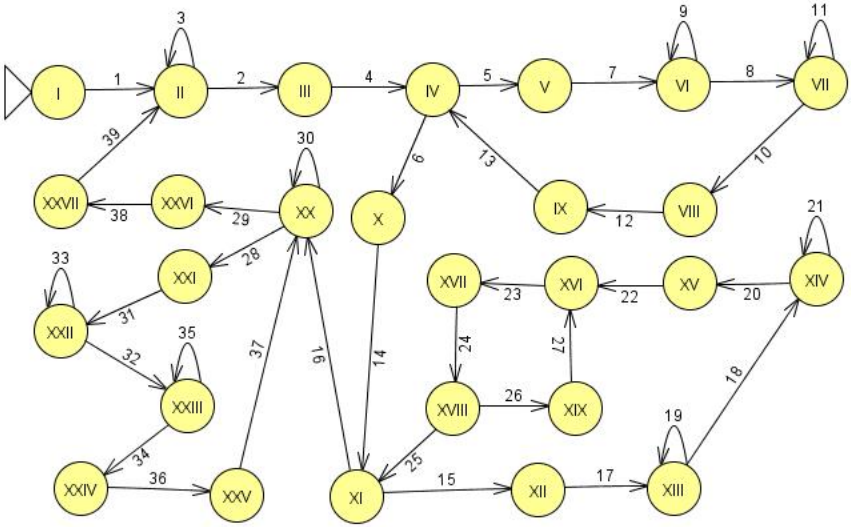

Figura 22: Máquina de estados para o módulo Montagem de Frame de Transferência

Tabela 8: Tempo e memória consumidos pela verificação do módulo Montagem de Frame de Transferência

Propriedade	**Tempo (s)**	**Mem. (MB)**
assert_p0	621.29	967
assert_p1	0.90	707
assert_p2	0.29	706
assert_p3	1.85	707
assert_p4	624.68	980
assert_p5	3.25	746
assert_p6	0.12	717
assert_p7	624.11	985
assert_p8	333.80	1242
assert_p9	675.17	1169
assert_p10	0.77	734
assert_p11	12.51	734
assert_p12	-	-
assert_p13	0.76	734
assert_p14	218.49	1245
continuado na próxima página		

Tabela 8 – continuado da página anterior

Propriedade	Time (s)	Mem. (MB)
assert_p15	0.28	724
assert_p16	16.53	724
assert_p17	-	-
assert_p18	806.05	968
assert_p19	772.47	779
assert_p20	658.44	1009
assert_p21	672.19	804
assert_p22	322.24	1265
assert_p23	0.34	743
assert_p24	1.64	743
assert_p25	326.26	1129
assert_p26	748.85	790
assert_p27	764.15	1023
assert_p28	687.27	1278
assert_p29	718.51	1011
assert_p30	11.49	702
assert_p31	3.54	702
assert_p32	425.55	1128
assert_p33	643.48	797
assert_p34	3.58	256
assert_p35	315.82	1274
assert_p36	0.57	754
assert_p37	315.62	1013
assert_p38	8.66	754

Entre as propriedades geradas, 3 apresentaram o resultado “Falha” devido a inconsistência entre implementação e especificação. A propriedade *assert_p*11 falhou pois após o estado *st_crc_calc*2, de acordo com a implementação o próximo estado é *st_crc_last_bit*, e não *st_pre_read*, como descrito na máquina de estados. A propriedade *assert_p*30 falhou pois após o estado *st_read*4, o próximo estado é *st_crc_set*, e não *st_crc_pre_calc*. A falha na propriedade *assert_p*38 ocorreu pois o próximo estado após *st_crc_pre_cal* é *crc_set*.

7.1.3 Módulo da camada de codificação

A camada de codificação na versão sem codificação tem a função de inserir uma marca de sincronização (*Attached Synchronization Mark*) em cada *frame* vindo da camada de transferência lido da memória e fazer a comunicação com o módulo responsável pelo envio serial. A FSM descrita na Figura 23 descreve o comportamento do módulo. Quando a camada de transferência finaliza a montagem de um frame, ela envia o sinal *start* para a camada de codificação. Recebendo esse sinal, a camada de codificação inicia o processo de envio da marca de sincronização. O envio é controlado nos estados V e $VIII$, por meio do contador *sync_idx*. Quando o contador chega em 4, a FSM encerrou o envio da marca de sincronização, passando então ao envio dos dados.

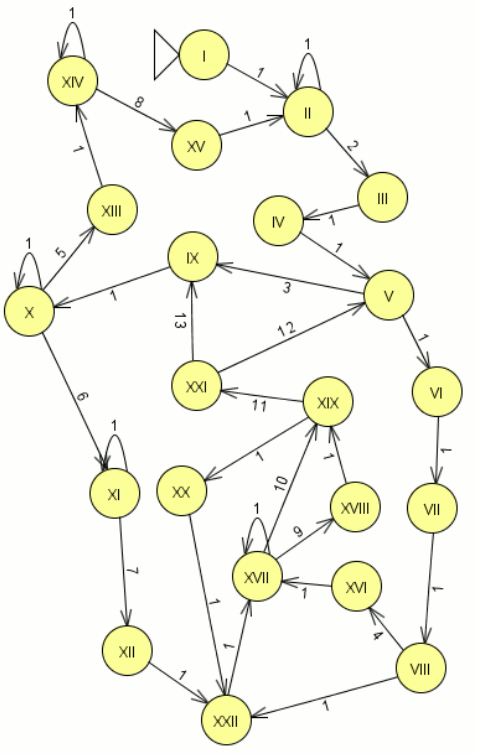

Figura 23: Máquina de estados para o módulo da camada de codificação

A Tabela 9 mostra as propriedades geradas para este módulo. O tempo e memória utilizados na verificação das propriedades pode ser observado na Tabela 10. Não foi encontrada nenhuma falha. Entretanto, várias propriedades apresentaram o resultado “Explorado”.

Tabela 9: Propriedades geradas no módulo da camada de codificação

Propriedade	**Resultado**
assert_p0: assert always (((mem_ack_i = ’0’) and (CS = st_read3)) ->next (CS = st_read4));	Explorado
assert_p1: assert always ((CS = st_read3) ->eventually! not (mem_ack_i /= ’0’));	Explorado
assert_p2: assert always ((CS = st_read4) ->next (CS = st_serial_send_bit));	Explorado
assert_p3: assert always ((CS = st_frame_sent3) ->next (CS = st_idle));	Explorado
assert_p4: assert always ((CS = st_getInfo) ->next (CS = st_validateInfo));	Provado
assert_p5: assert always ((CS = st_inc_index) ->next (CS = st_send_sync_serial));	Provado
assert_p6: assert always ((CS = st_read1) ->next (CS = st_read2));	Explorado
assert_p7: assert always ((CS = st_validateInfo) ->next (CS = st_pre_sync));	Provado
assert_p8: assert always (((serial_bitcount_sv /= 0) and (CS = st_serial_finish)) ->next (CS = st_dec_bit_count));	Provado
assert_p9: assert always (((serial_bitcount_sv = 0) and (CS = st_serial_finish)) ->next (CS = st_stop_byte));	Explorado
assert_p10: assert always ((CS = st_frame_sent) ->next (CS = st_frame_sent2));	Explorado
assert_p11: assert always (((sync_idx_sv = 1) and (CS = st_send_sync_serial)) ->next (CS = st_serial_start));	Provado
assert_p12: assert always (((sync_idx_sv /= 1) and (CS = st_send_sync_serial)) ->next (CS = st_serial_send_bit));	Explorado
assert_p13: assert always ((CS = st_idle) ->eventually! not (ready_i /= ’1’));	Explorado
assert_p14: assert always (((ready_i = ’1’) and (CS = st_idle)) ->next (CS = st_getInfo));	Provado
assert_p15: assert always ((CS = st_serial_start) ->next (CS= st_serial_wait_event));	Provado

continuado na próxima página

Tabela 9 – continuado da página anterior

Propriedade	Resultado
assert_p16: assert always ((CS = st_rst) ->next (CS = st_idle));	Provado
assert_p17: assert always ((CS = st_dec_bit_count) ->next (CS = st_serial_send_bit));	Provado
assert_p18: assert always ((CS = st_sendSync) ->next (CS = st_inc_index));	Provado
assert_p19: assert always ((CS = st_serial_wait_event) ->eventually! not (serial_busy_s = reg_event_serial_s));	Explorado
assert_p20: assert always (((serial_start_s = '1' and serial_busy_s = not reg_event_serial_s) and (CS = st_serial_wait_event)) ->next (CS = st_start_off));	Provado
assert_p21: assert always (((serial_start_s = '0' and serial_busy_s = not reg_event_serial_s) and (CS = st_serial_wait_event)) ->next (CS = st_serial_finish));	Provado
assert_p22: assert always ((CS = st_serial_send_bit) ->next (CS = st_serial_wait_event));	Provado
assert_p23: assert always ((CS = st_stop_byte) ->next (CS = st_read1));	Explorado
assert_p24: assert always ((CS = st_stop_byte) ->next (CS = st_pre_sync));	Explorado
assert_p25: assert always ((CS = st_start_off) ->next (CS = st_serial_finish));	Provado
assert_p26: assert always (((sync_idx_sv /= 4) and (CS = st_pre_sync)) ->next (CS = st_sendSync));	Provado
assert_p27: assert always (((sync_idx_sv = 4) and (CS = st_pre_sync)) ->next (CS = st_read1));	Explorado
assert_p28: assert always (((mem_id_i = ID_TM_CODING and mem_ack_i = '1' and sent_count_sv /= TAM_FRAME_TM) and (CS = st_read2)) ->next (CS = st_read3));	Explorado
assert_p29: assert always (((sent_count_sv = TAM_FRAME_TM) and (CS = st_read2)) ->next (CS = st_frame_sent));	Explorado
assert_p30: assert always ((CS = st_read2) ->eventually! not (mem_id_i /= ID_TM_CODING and mem_ack_i /= '1'));	Explorado
assert_p31: assert always ((CS = st_frame_sent2) ->eventually! not (buffer_done_i /= '1'));	Explorado
assert_p32: assert always (((buffer_done_i = '1') and (CS = st_frame_sent2)) ->next (CS = st_frame_sent3));	Explorado

Tabela 10: Tempo e memória consumidos pela verificação do módulo da camada de codificação

Propriedade	Tempo (s)	Mem. (MB)
assert_p0	1105.27	2744
assert_p1	601.67	207
assert_p2	600.87	230
assert_p3	601.29	200
assert_p4	0.84	165
assert_p5	2.53	165
assert_p6	600.14	229
assert_p7	0.87	271
assert_p8	334.15	1250
assert_p9	688.47	2155
assert_p10	636.31	2668
assert_p11	2.20	1746
assert_p12	536.60	2655
assert_p13	322.77	1729
assert_p14	0.63	1433
assert_p15	2.89	1433
assert_p16	0.16	1433
assert_p17	330.33	1894
assert_p18	1.30	1433
assert_p19	323.27	1724
assert_p20	327.06	1961
assert_p21	339.74	1958
assert_p22	324.71	1528
assert_p23	648.38	1875
assert_p24	648.11	1252
assert_p25	335.80	1354
assert_p26	1.40	271
assert_p27	600.44	256
assert_p28	600.28	226
assert_p29	601.09	215
assert_p30	600.86	196
assert_p31	600.52	197
assert_p32	642.10	2677

7.1.4 Discussão

De maneira geral as falhas encontradas no processo de verificação são divididas em duas classes: falha de implementação e falha de inconsistência com a especificação. Os módulos verificados apresentaram a segunda classe de falha, tornando necessário reavaliar o que foi implementado e averiguar a inconsistência entre especificação e implementação.

Com base nos resultados apresentados neste capítulo é possível observar uma grande proporção de propriedades com o resultado "Explorado", com aproximadamente 50% das propriedades verificadas. Um possível motivo para esse resultado foi a falta de recurso computacional para que o IFV fizesse a análise da implementação. Outro motivo seria o detalhamento da descrição do sistema no nível RT, sendo necessário utilizar abstrações em mais alto nível.

Entre as propriedades geradas, nota-se a falta de resultados com respeito à verificação de módulos concorrentes. Esta falta se deve a uma limitação da ferramenta IFV em lidar com os sinais internos de mais de um módulo na verificação. Assim, não foi possível criar uma propriedade que utilize um sinal interno de um módulo, relacionando-o com um sinal interno de outro módulo.

8 CONCLUSÃO

Este trabalho de mestrado tratou de um problema da verificação formal decorrente da elaboração manual das propriedades, o fato de estas serem mais suscetíveis ao erro humano, além de demandarem mais tempo e acarretarem em maior custo. Para isso, a proposta foi atenuar o problema através da geração automática de propriedades.

A dificuldade inerente em elaborar propriedades manualmente foi identificada já na fase inicial deste trabalho, quando o foco era aprender técnicas e ferramentas para verificação formal. Foram selecionados exemplos didáticos de sistemas escritos em VHDL e para verificá-los era necessário identificar algumas propriedades. O levantamento de propriedades era manual, o que a tornava uma atividade difícil pois havia a necessidade de adquirir conhecimentos de lógica temporal e elaboração de propriedades que ainda não haviam sido adquiridos.

8.1 CONTRIBUIÇÕES CIENTÍFICAS

Durante o estudo do estado da arte notou-se uma ênfase na simulação do sistema como pré-requisito para a geração automática de propriedades para verificação formal. A simulação do sistema é um passo importante no desenvolvimento de um projeto, mas um grande esforço em elaborar *testbenches* é sem dúvida um aspecto negativo de tais abordagens.

Outra abordagem observada no estado da arte é a utilização dos dados da especificação para geração de propriedades, em que os autores utilizam uma série de modelos e diagramas, como por exemplo, diagramas UML. Nesta abordagem exige-se esforço em modelagem, mas isto é justificado, pois, conforme descrito anteriormente, 40% dos erros no projeto são mais prováveis de ocorrerem na especificação (KIEVIET, 2013; BRAUN et al., 2010). A concentração de esforço nas etapas iniciais do projeto contribuem para seu aprimoramento.

Por essa razão, a abordagem proposta nesta dissertação de mestrado foi gerar as propriedades a partir da especificação, expressas em termos de máquinas de estados e diagramas de sequência. O grande fator motivador em trabalhar com as máquinas de estados foi o fato de que o estudo de caso deste trabalho, a Unidade de Telecomando e Telemetria (UTMC), tinha sido previamente modelada em máquinas de estados nas fases iniciais do projeto e a medida que o entendimento do sistema fosse aumentando, a especificação ia sendo aprimorada. O Capítulo 6 descreveu com detalhes as características

da UTMC.

A contribuição principal deste trabalho de mestrado são três heurísticas, apresentadas no Capítulo 4, para realizar a geração de propriedades tanto em nível de módulo como em nível de sistema. Asserções sobre o comportamento do sistema são definidas por meio de algoritmos que percorrem máquinas de estados e diagramas de sequência.

8.2 CONTRIBUIÇÕES TÉCNICAS

A ferramenta PropGen foi desenvolvida a partir dos algoritmos propostos. Conforme descrito no Capítulo 5, a ferramenta recebe como entrada os modelos expressos textualmente e a implementação descrita em VHDL. O PropGen faz a análise e geração de propriedades para serem verificadas, de acordo com o comportamento da máquina e os sinais manipulados pelo código VHDL. Por fim, a ferramenta gera as propriedades descritas em LTL, SVA e PSL.

A abordagem proposta foi utilizada para verificar formalmente os módulos do fluxo de telemetria da UTMC. As propriedades geradas e verificadas foram apresentadas no Capítulo 7. Os resultados apresentaram muitas propriedades com o estado "Explorado", em torno de 50% das propriedades, o que é um fator negativo. Conforme observado anteriormente, "Explorado" é um caso em que há explosão de estados e que o *model checker* não pode determinar se a propriedade é provada ou não.

8.3 TRABALHOS FUTUROS

O tratamento de propriedades com o estado "Explorado" é o principal ponto a ser considerado em trabalhos futuros. Um fator importante a se considerar é o nível de abstração utilizado na verificação do sistema. Todas as propriedades foram criadas para verificar a implementação em nível RT (RTL). Futuramente é necessário avaliar a abstração do sistema e a criação de propriedades para verificar comportamentos em um nível mais alto de detalhamento do sistema.

A geração de propriedades foi realizada com base nas transições de estados, resultando em um grande número de propriedades em módulos maiores. Isto aponta para a necessidade de gerar e selecionar propriedades mais relevantes. No entanto, como determinar a importância de uma transição apenas com os dados de máquinas de estados, ou com a concorrência entre elas? Isto é, a relevância de uma transição pode depender muito de projeto para

projeto. Uma alternativa seria a análise pelo engenheiro do sistema para determinar quais propriedades serão verificadas. Além disso, o PropGen poderia requerer informações extras sobre o comportamento do sistema com o objetivo de extrair propriedades de maior complexidade.

8.4 CONSIDERAÇÕES FINAIS

Dentre as falhas encontradas, haviam apenas falhas de inconsistência entre especificaçao e implementação. O desejável era que fossem encontradas falhas de implementaçao, no entanto, a UTMC já havia sido extensivamente verificada por meio de simulação, de modo que a maioria das falhas simples já haviam sido detectadas e corrigidas. Entretanto, falhas de implementação foram encontradas em outros sistemas que foram verificados durante este trabalho sem que houvesse a necessidade de se criar *testbenches*.

As características da UTMC enquadram-a no tipo de sistemas críticos e embarcados. A utilização da geração automática de propriedades a partir de máquinas de estados e diagramas de sequência mostraram-se muito úteis e aprimoraram o processo de verificação do fluxo de telemetria da UTMC. Tendo sido possível aplicar a proposta deste trabalho de mestrado em um sistema de tamanha complexidade, como é o caso da UTMC, e tendo alcançado os objetivos traçados, este trabalho concluiu todos os objetivos pré-definidos.

REFERÊNCIAS BIBLIOGRÁFICAS

ACCELLERA. *Open Verification Language*. 2013. Disponível em: <http://www.accellera.org/downloads/standards/ovl>.

BAIER, C.; KATOEN, J.-P. *Principles of Model Checking (Representation and Mind Series)*. [S.l.]: The MIT Press, 2008. ISBN 026202649X, 9780262026499.

BALASUBRAMANIAN, D. et al. Rapid property specification and checking for model-based formalisms. In: *Rapid System Prototyping (RSP), 2011 22nd IEEE International Symposium on*. [S.l.: s.n.], 2011. p. 121 –127. ISSN Pending.

BERGERON, J. *Writing testbenches : functional verification of HDL models*. Boston: Kluwer Academic, 2000. ISBN 0792377664.

BEZERRA, E.; SILVA, E.; ROCHAT, D. *Relatório técnico 72709-90000 - DESCRIÇÃO DO PROJETO DA UTMC*. Instituto Nacional de Pesquisas Espaciais (INPE), Coordenação Geral de Engenharia e Tecnologia Espacial, Divisão de Eletrônica Aeroespacial, São José dos Campos, SP, Brasil, Outubro 2009.

BIERE, A. *AIGER*. 2013. Disponível em: <http://fmv.jku.at/aiger/>.

BIERE, A. et al. Symbolic model checking using sat procedures instead of bdds. In: *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*. New York, NY, USA: ACM, 1999a. (DAC '99), p. 317–320. ISBN 1-58113-109-7. Disponível em: <http://doi.acm.org/10.1145/309847.309942>.

BIERE, A. et al. Symbolic model checking without bdds. In: *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*. London, UK, UK: Springer-Verlag, 1999b. (TACAS '99), p. 193–207. ISBN 3-540-65703-7. Disponível em: <http://dl.acm.org/citation.cfm?id=646483.691738>.

BLIF. *Berkeley Logic Interchange Format (BLIF)*. 2013. Disponível em: <http://www1.cs.columbia.edu/ cs4861/s07-sis/blif/index.html>.

BOSE, R. C.; RAY-CHAUDHURI, D. K. On a class of error correcting binary group codes. *Information and Control*, v. 3, n. 1, p. 68–79, 1960.

BRAUN, A. et al. Simulation-based verification of the most netinterface specification revision 3.0. In: *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*. [S.l.: s.n.], 2010. p. 538 –543. ISSN 1530-1591.

BRAYTON, R. K. et al. *BLIF-MV: An Interchange Format for Design Verification and Synthesis*. [S.l.], 1991. Disponível em: <http://www.eecs.berkeley.edu/Pubs/TechRpts/1991/1861.html>.

BROOKES, S. D.; HOARE, C. A. R.; ROSCOE, A. W. A theory of communicating sequential processes. *J. ACM*, ACM, New York, NY, USA, v. 31, n. 3, p. 560–599, jun. 1984. ISSN 0004-5411. Disponível em: <http://doi.acm.org/10.1145/828.833>.

BRYANT, R. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on*, C-35, n. 8, p. 677 –691, aug. 1986. ISSN 0018-9340.

CADENCE. *Incisive Formal Verifier*. 2013. Disponível em: <http://www.cadence.com/products/ld/formal_verifier/pages/default.aspx>.

CAMPOS, J. C.; MACHADO, J. Pattern-based analysis of automated production systems. In: *13th IFAC Symposium on Information Control Problems in Manufacturing (INCOM 2009)*. [S.l.]: Elsevier, 2009.

CAMPOS, J. C.; MACHADO, J.; SEABRA, E. Property patterns for the formal verification of automated production systems. In: *Proceedings of the 17th IFAC World Congress*. [S.l.]: IFAC, 2008. p. 5107–5112.

CLARKE, E. M.; EMERSON, E. A.; SIFAKIS, J. Model checking: algorithmic verification and debugging. *Commun. ACM*, ACM, New York, NY, USA, v. 52, p. 74–84, November 2009. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/1592761.1592781>.

CLARKE, E. M.; EMERSON, E. A.; SISTLA, A. P. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, ACM, New York, NY, USA, v. 8, p. 244–263, April 1986. ISSN 0164-0925. Disponível em: <http://doi.acm.org/10.1145/5397.5399>.

CLARKE, E. M. et al. Progress on the state explosion problem in model checking. In: *Informatics - 10 Years Back. 10 Years Ahead.* London, UK: Springer-Verlag, 2001. p. 176–194. ISBN 3-540-41635-8. Disponível em: <http://dl.acm.org/citation.cfm?id=647348.724453>.

CLARKE, E. M.; GRUMBERG, O.; LONG, D. E. Verification tools for finite-state concurrent systems. In: *A Decade of Concurrency, Reflections and Perspectives, REX School/Symposium*. London, UK: Springer-Verlag, 1994. p. 124–175. ISBN 3-540-58043-3. Disponível em: <http://portal.acm.org/citation.cfm?id=648145.750141>.

DIJKSTRA, E. W. Structured programming. In: DAHL, O. J.; DIJKSTRA, E. W.; HOARE, C. A. R. (Ed.). London, UK, UK: Academic Press Ltd., 1972. cap. Chapter I: Notes on structured programming, p. 1–82. ISBN 0-12-200550-3. Disponível em: <http://dl.acm.org/citation.cfm?id=1243380.1243381>.

DRECHSLER, R.; FEY, G. Design understanding by automatic property generation. In: *Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI)*. [S.l.: s.n.], 2004. p. 274–281.

DWYER, M. B.; AVRUNIN, G. S.; CORBETT, J. C. Patterns in property specifications for finite-state verification. In: *Proceedings of the 21st international conference on Software engineering*. New York, NY, USA: ACM, 1999. (ICSE '99), p. 411–420. ISBN 1-58113-074-0. Disponível em: <http://doi.acm.org/10.1145/302405.302672>.

ECSS, E. C. f. S. S. *ECSS-E-70-41A, Ground Systems and Operations - Telemetry and Telecommand Packet Utilization*. January 2003.

FERREIRA, C.; SILVA, F. A. da; VILLA, P. R. C. *Concepção do On-Board Data Handling com Sensor Inercial para Aplicações Espaciais*. 2009.

FUJITA, M.; GHOSH, I.; PRASAD, M. *Verification Techniques for System-Level Design*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008. ISBN 0123706165, 9780080553139, 9780123706164.

IEEE Standard for Property Specification Language (PSL). *IEEE Std 1850-2005*, p. 1–143, 2005.

IEEE Standard VHDL Language Reference Manual. *ANSI/IEEE Std 1076-1993*, p. i, 1994.

JR., E. M. C.; GRUMBERG, O.; PELED, D. A. *Model Checking*. [S.l.]: The MIT Press, 1999. ISBN 0262032708.

KAHN, H. J.; GOLDMAN, R. F. The electronic design interchange format edif: present and future. In: *Proceedings of the 29th ACM/IEEE Design Automation Conference*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1992. (DAC '92), p. 666–671. ISBN 0-89791-516-X. Disponível em: <http://dl.acm.org/citation.cfm?id=113938.149663>.

KIEVIET, M. *IEC 61508 - IEC 61508 - Umfassende Sicherheit für Maschinen und Anlagen*. mar. 2013. Disponível em: <http://www.innotecsafety.de/>.

KRATOCHVíLA, T.; REHáK, V.; SIMECEK, P. *Verification of COMBO6 VHDL Design*. Prague, Czech Republic, November 2003.

LETTNIN, D. et al. Semiformal verification of temporal properties in automotive hardware dependent software. In: *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.* [S.l.: s.n.], 2009. p. 1214 –1217. ISSN 1530-1591.

LONG, J. et al. *Enhancing ABC for LTL Stabilization Verification of SystemVerilog/VHDL Models*. 2011.

LONG, J.; SEAWRIGHT, A.; KAVALIPATI, P. Multi-clock sva synthesis without re-writing. In: *Design Automation Conference, 2009. ASP-DAC 2009. Asia and South Pacific*. [S.l.: s.n.], 2009. p. 648 –653.

MADEIRA, A.; BEZERRA, E. Verificação da implementação do protocolo ccsds do puc#sat usando promela/spin. B.Sc Thesis, PUC/RS. Julho 2008.

MCMILLAN, K. L. *Symbolic model checking: an approach to the state explosion problem*. Tese (Doutorado), Pittsburgh, PA, USA, 1992. UMI Order No. GAX92-24209.

MENTORGRAPHICS. *LeonardoSpectrum*. 2013. Disponível em: <http://www.mentor.com/products/fpga/synthesis/leonardo_spectrum/>.

MIR, A.; BALAKRISHNAN, S.; TAHAR, S. Modeling and verification of embedded systems using cadence smv. In: *Electrical and Computer Engineering, 2000 Canadian Conference on*. [S.l.: s.n.], 2000. v. 1, p. 179 –183 vol.1.

MISHCHENKO, A. *ABC - A System for Sequential Synthesis and Verification*. 2013. Disponível em: <http://www.eecs.berkeley.edu/ alanmi/abc/>.

MORBINI, F. *scxmlgui: A graphical editor for SCXML finite state machines*. 2013. Disponível em: <http://code.google.com/p/scxmlgui/>.

ROGIN, F. et al. Automatic generation of complex properties for hardware designs. In: *Proceedings of the conference on Design, automation and test in Europe*. New York, NY, USA: ACM, 2008. (DATE '08), p. 545–548. ISBN 978-3-9810801-3-1. Disponível em: <http://doi.acm.org/10.1145/1403375.1403506>.

SCXML. *State Chart XML (SCXML): State Machine Notation for Control Abstraction*. 2013. Disponível em: <http://www.w3.org/TR/scxml/>.

SVA. *System Verilog*. 2013. Disponível em: <http://www.systemverilog.org/>.

UML. *Unified Modeling Language*. 2013. Disponível em: <http://www.omg.org/spec/UML/>.

UMLET. *Free UML Tool for Fast UML Diagrams*. 2013. Disponível em: <http://www.umlet.com/>.

VAHID, F. *Digital Design with RTL Design, Verilog and VHDL*. John Wiley & Sons, 2010. ISBN 9780470531082. Disponível em: <http://books.google.com.br/books?id=-YayRpmjc20C>.

VASUDEVAN, S. et al. Goldmine: Automatic assertion generation using data mining and static analysis. In: *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*. [S.l.: s.n.], 2010. p. 626 –629. ISSN 1530-1591.

VIS. *VIS - Verification Interacting with Synthesis*. 2013. Disponível em: <http://vlsi.colorado.edu/ vis/>.

WANG, J. et al. Survey on formal verification methods for digital ic. In: *Internet Computing for Science and Engineering (ICICSE), 2009 Fourth International Conference on*. [S.l.: s.n.], 2009. p. 164 –168.

WILE, B.; GOSS, J.; ROESNER, W. *Comprehensive Functional Verification: The Complete Industry Cycle (Systems on Silicon)*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005. ISBN 0127518037.

XU, D.; MIAO, H.; PHILBERT, N. Model checking uml activity diagrams in fdr. In: *Computer and Information Science, 2009. ICIS 2009. Eighth IEEE/ACIS International Conference on*. [S.l.: s.n.], 2009. p. 1035 –1040.

ZHANG, S. J.; LIU, Y. An automatic approach to model checking uml state machines. In: *Secure Software Integration and Reliability Improvement Companion (SSIRI-C), 2010 Fourth International Conference on*. [S.l.: s.n.], 2010. p. 1 –6.

APÊNDICE A – Geração do modelo formal para projetos em VHDL

Várias ferramentas foram testadas para realizar a verificação dos módulos da UTMC, descritos em VHDL. No entanto, dentre as ferramentas disponíveis no meio acadêmico, não há um *model checker* que tenha como entrada descrição de hardware em VHDL. Assim, conforme mencionado no Capítulo 2, é necessário fazer o pré-processamento do sistema e gerar uma entrada válida para algum provador de modelo.

A primeira tentativa foi gerar o modelo formal para utilização na ferramenta CadenseSMV. Visto que a ferramenta manipula a linguagem Verilog, o desafio foi converter VHDL em Verilog. A ferramenta ***vhd2vl*** prometia fazer esta conversão em nível RTL, mas não foi possível devido à incapacidade da ferramenta de lidar com os tipos de dados utilizados no projeto, como por exemplo, os da biblioteca IEEE.1164 do VHDL.

No relatório técnico (KRATOCHVíLA; REHáK; SIMECEK, 2003) foi mostrado um caso em que um sistema em VHDL foi verificado utilizando a ferramenta CadenseSMV (MIR; BALAKRISHNAN; TAHAR, 2000). Utilizando a ferramenta LeonardoSpectrum (MENTORGRAPHICS, 2013), desenvolvida pela Mentor Graphics, os autores fizeram a síntese do VHDL em um *netlist* na linguagem Verilog e procederam com a verificação.

Diversas tentativas de replicar o experimento com a ferramenta LeonardoSpectrum foram feitas, todas sem sucesso. A primeira barreira foi quanto à licença da ferramenta, pois é uma ferramenta proprietária. Uma vez que a ferramenta CadenceSMV aceita Verilog sintetizado, outras ferramentas proprietárias foram utilizadas. Podemos citar as ferramentas Quartus (Altera), ISE Project (Xilinx) e Design Compiler (Synopsys). Nenhum dos *netlists* gerados foram corretamente aceitos como entrada pelo CadenseSMV.

Diante da dificuldade de gerar um modelo correto no formato SMV, outra ferramenta recebeu a atenção, o ABC (MISHCHENKO, 2013). Como observado no Capítulo 2, esta ferramenta tem como principal formato de entrada é o modelo AIGER (BIERE, 2013). Um subconjunto bastante restrito de Verilog estrutural e o formato BLIF (BLIF, 2013) também são aceitos como entrada. A ferramenta não era compatível com os *netlists* gerados anteriormente, assim, o formato BLIF tornou-se então o foco na etapa de pré-processamento. A ferramenta ***edif2blif*** foi então utilizada para fazer a conversão do formato EDIF (KAHN; GOLDMAN, 1992), que foi corretamente gerado a partir do código original em VHDL por meio da ferramenta ISE Project, para o formato BLIF. O ***edif2blif*** trabalha com uma tabela de conversão que lê as células do *netlist* EDIF e os escreve no formato BLIF. No entanto, algumas das células não foram reconhecidas, havendo a necessidade de se criar uma tabela própria de conversão. Visto que a proposta deste trabalho é ter um processo automatizado, estava fora do escopo criar uma tabela manualmente. Além disso outras diversas ferramentas acadêmicas foram testadas

mas todas sem sucesso.

Por fim, a ferramenta Verific foi capaz de gerar o formato BLIF diretamente do código VHDL. O formato AIGER foi gerado por uma ferramenta, ainda em desenvolvimento por pesquisadores de Berkeley, chamada VeriABC (LONG et al., 2011). Esta última utiliza uma biblioteca disponibilizada pela ferramenta Verific, um analisador genérico para plataformas HDLs, e diretamente do código VHDL gera o modelo AIGER durante o processo de síntese. O fluxo de trabalho com a ferramenta VeriABC (LONG et al., 2011) processa o sistema em HDL e as propriedades geradas (em SVA), produz o modelo AIGER, que por sua vez, são verificadas com o ABC.

Todavia, a ferramenta ABC realiza apenas o *equivalence checking*, de forma que as propriedades não são avaliadas. Desta forma, a solução para verificação dos módulos da UTMC foi utilizar a ferramenta comercial Incisive Formal Verifier (CADENCE, 2013).

ANEXO A – Legendas das máquinas de estados

A.1 MÓDULO MONTAGEM DE FRAME E TRANSFERÊNCIA

Estados:

I: rst

II: idle

III: get_variables

IV: pre_write

V: mem_write1

VI: mem_write2

VII: mem_write3

VIII: mem_write4

IX: update_pointers

X: end_write

XI: pre_read

XII: read1

XIII: read2

XIV: read3

XV: read4

XVI: crc_pre_calc

XVII: crc_calc1

XVIII: crc_calc2

XIX: crc_dec

XX: pre_write_crc

XXI: mem_write_crc1

XXII: mem_write_crc2

XXIII: mem_write_crc3

XXIV: mem_write_crc4

XXV: update_crc

XXVI: coding_ready

XXVII: end_frame

Transições:

1: incondicional para idle

2: para get_variables se ready_i = ’1’

3: para idle se ready_i = ’0’

4: incondicional para pre_write

5: para mem_write1 se count_bytes_sv >0

6: para end_write se count_bytes_sv <= 0

7: incondicional para mem_write2

8: para mem_write3 se mem_id_i = ID_TM_FRAME e mem_ack_i = ’1’

9: para mem_write2 se condição anterior não for atendida

10: para mem_write4 se mem_ack_i = ’0’

11: para mem_write3 se mem_ack_i <>’0’

12: incondicional para update_pointers

13: incondicional para pre_write

14: incondicional para pre_read

15: para read1 se count_read_sv <1113

16: para pre_write_crc se count_read_sv >= 1113

17: incondicional para read2

18: para read3 se mem_id_i = ID_TM_FRAME e mem_ack_i = ’1’

19: para read2 se condição anterior não for atendida

20: para read4 se mem_ack_i = ’0’

21: para read3 se mem_ack_i = ’1’

22: incondicional para crc_pre_calc

23: incondicional para crc_calc1

24: incondicional para crc_calc2

25: para pre_read se crc_index_sv = 0

26: para crc_dec se crc_index_sv <>0

27: incondicional para crc_pre_calc

28: para mem_write_crc1 se crc_count_byte_sv <2

29: para coding_ready se crc_count_byte_sv >= 2 e busy_i = ’0’

30: para pre_write_crc se crc_count_byte_sv >= 2 e busy_i = ’1’

31: incondicional para mem_write_crc2

32: para mem_write_crc3 se mem_id_i = ID_TM_FRAME e mem_ack_i = ’1’

33: para mem_write_crc2 se a condição anterior não for atendida

34: para mem_write_crc4 se mem_ack_i = ’0’

35: para mem_write_crc3 se mem_ack_i = ’1’

36: incondicional para update_crc

37: incondicional para pre_write_crc

38: incondicional para end_frame

39: incondicional para idle

A.2 MÓDULO DA CAMADA DE CODIFICAÇÃO

Estados:

I: rst

II: idle

III: getInfo

IV: validateInfo

V: pre_sync

VI: sendSync

VII: inc_index

VIII: send_sync_serial

IX: read1

X: read2

XI: read3

XII: read4

XIII: frame_sent

XIV: frame_sent2

XV: frame_sent3

XVI: serial_start

XVII: serial_wait_event

XVIII: start_off

XIX: serial_finish

XX: dec_bit_count

XXI: stop_byte

XXII: serial_send_bit

Transições:

1: transição incondicional

2: ready_i = '1'

3: sync_idx = 4

4: sync_idx = 1

5: sent_count = TAM_FRAME_TM '1'

6: mem_id_i = ID_TM_CODING and mem_ack_i = '1' and sent_count <>TAM_FRAME_TM

7: mem_ack_i = '0'

8: buffer_done_i = '1'

9: serial_start = '1' and serial_busy = not reg_event_serial_s

10: serial_start = '0' and serial_busy = not reg_event_serial_s

11: serial_bitcount_sv = 0

12: caso tenha chegado no estado atual através do estado VIII

13: caso tenha chegado no estado atual através do estado XII