CARLOS IVÁN CASTRO MÁRQUEZ

**Checagem de equivalência de sequências de estados de projetos digitais em RTL com
modelos de referência em alto nível e de protocolo de comunicação**

São Paulo

2014

CARLOS IVÁN CASTRO MÁRQUEZ

**Checagem de equivalência de sequências de estados de projetos digitais em RTL com
modelos de referência em alto nível e de protocolo de comunicação**

Tese apresentada à Escola Politécnica da
Universidade de São Paulo para obtenção do
título de Doutor em Ciências

São Paulo

2014

CARLOS IVÁN CASTRO MÁRQUEZ

**Checagem de equivalência de sequências de estados de projetos digitais em RTL com
modelos de referência em alto nível e de protocolo de comunicação**

Tese apresentada à Escola Politécnica da
Universidade de São Paulo para obtenção do
título de Doutor em Ciências

Área de Concentração: Microeletrônica

Orientador: Prof. Dr. Wang Jiang Chau

São Paulo

2014

Este exemplar foi revisado e corrigido em relação à versão original, sob responsabilidade única do autor e com a anuência de seu orientador.

São Paulo, de abril de 2014.

Assinatura do autor _____

Assinatura do orientador _____

CATALOGAÇÃO-NA-PUBLICAÇÃO

Castro Marquez, Carlos Ivan

Checagem de equivalência de sequências de estados de projetos digitais em RTL com modelos de referência em alto nível e de protocolo de comunicação / C.I. Castro Marquez. – versão corr. -- São Paulo, 2014. 110 p.

Tese (Doutorado) - Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia de Sistemas Eletrônicos.

1.Verificação formal 2.Checagem de equivalência sequencial 3.Sequências de estados 4.Projeto sob verificação 5.Implementação RTL 6.Modelo de referência 7.Protocolos de comunicação I.Universidade de São Paulo. Escola Politécnica. Departamento de Engenharia de Sistemas Eletrônicos II.t.

À minha mãe, Jeannette,

Ao meu pai, Ciro,

Aos meus irmãos, Ciro Walter, Juan José, e Adriana María,

À minha família de amigos da Colômbia,

À minha família de amigos do Brasil.

AGRADECIMENTOS

A minha mãe, meu pai, e meus irmãos, pelo ânimo, carinho, e apoio constantes. Sem eles este trabalho não poderia ter sido realizado.

Um sincero agradecimento ao Professor Wang Jiang Chau, pelo seu constante apoio e dedicação. Ter aprendido ao longo destes anos, junto com ele, foi uma experiência inestimável.

A todos meus colegas do Grupo de Sistemas Eletrônicos e Software Aplicado, GSEIS, com quem compartilhei muitos momentos de enriquecimento pessoal e intelectual.

A Kevin Chanemougame, quem contribuiu na obtenção de casos de teste para as experiências realizadas como parte deste trabalho.

À Escola Politécnica, e ao CNPq, por todos os recursos e incentivos disponibilizados. Também, ao Brasil por disponibilizar as oportunidades que tornam possíveis este tipo de pesquisas e programas de Pós-Graduação.

RESUMO

A verificação funcional é o conjunto de tarefas destinado a descobrir erros gerados durante o projeto de circuitos integrados, e representa um importante desafio ao influenciar fortemente a eficiência do ciclo inteiro de produção. Estima-se que até 80% dos custos totais de projeto são devidos à verificação, tornando esta atividade o gargalo principal para reduzir o *time-to-market*. Tal problemática tem provocado a aparição de diversas estratégias para diminuir o esforço, ou para aumentar a capacidade de cobertura da verificação. Por um lado existe a simulação, que permite descobrir um número razoável de erros de projeto; porém, a lentidão da simulação de descrições RTL torna mínima a cobertura real de estados. Por outro lado, os métodos formais de verificação fornecem alta cobertura de estados. Um deles é a checagem de modelos, que checa a validade de um conjunto de propriedades para todos os estados do projeto sob verificação. No entanto, esta técnica padece do problema de explosão de estados, e da dificuldade de especificar um conjunto robusto de propriedades. Outra alternativa formal é a checagem de equivalência que, ao invés de verificar propriedades, compara o projeto com um modelo de referência. No entanto, a checagem de equivalência tradicional é aplicável, unicamente, a descrições no mesmo nível de abstração, e com interfaces idênticas. Como fato importante, não foram encontrados registros na literatura de sobre a verificação formal de descrições RTL, considerando ambos os aspectos computacionais (presentes no modelo de referência) e de comunicação às interfaces (provenientes da especificação funcional de protocolo). Neste trabalho apresenta-se uma metodologia de verificação formal, através do uso de técnicas de checagem de equivalência para determinar a validade de uma implementação em RTL, comparando-a com um modelo de referência em alto nível, e com um modelo formal do protocolo de comunicação. Para permitir tal checagem, a metodologia baseia-se no conceito de sequências de estados, ao invés de estados individuais como na checagem de equivalência tradicional. As discrepâncias entre níveis diferentes de abstração são consideradas, incluindo alfabetos diferentes, mapeamento entre estados, e dessemelhanças temporais. A caracterização e solução do problema são desenvolvidas através de um quadro teórico, onde se apresentam conceitos, e definições, cuja validade é provada formalmente. Uma ferramenta para aplicação prática da metodologia foi desenvolvida e aplicada sobre diferentes tipos de descrições RTL, escritas nas linguagens VHDL e SystemC. Os resultados demonstram efetividade e eficiência na verificação formal de circuitos digitais que incluem, mas não se limitam à correção de erros, encriptação, processamento de imagens, e funções matemáticas. Também, evidencia-se a capacidade da ferramenta para descobrir erros de tipo

combinatório e sequencial injetados propositalmente, relacionados com a funcionalidade do modelo de referência, assim como, com a da especificação do protocolo de comunicação, dentro de tempos e número de iterações praticáveis em casos reais.

Palavras-Chave: Verificação formal. Checagem de equivalência sequencial. Sequências de estados. Projeto sob verificação. Implementação RTL. Modelo de referência. Protocolo de comunicação. Algoritmo. Ferramenta.

ABSTRACT

Functional verification is the group of tasks aiming the discovery of bugs created during integrated circuit design, and represents an important challenge by its strong influence on efficiency throughout production cycles. As an estimative, up to 80% of the whole design costs are due to verification, which makes verification the greatest bottleneck while attempting to reduce *time-to-market*. Such problem has given rise to a series of techniques to reduce the effort, or to increase verification coverage capability. On the one side, simulation allows finding a good number of bugs, but it is still far from reaching high state coverage because of RTL cycle-accurate slowness. On the other side, formal approaches supply high state coverage. Model checking, for instance, checks the validness of a set of properties for all design's states. However, a strong disadvantage resides in defining and determining the quality of the set of properties to verify, not to mention state explosion. Sequential equivalence checking, which instead of checking properties compares the design with a reference model. Nevertheless, traditionally it can only be applied between circuit descriptions where a one-to-one correspondence for states, as well as for memory elements, is expected. As a remarkable issue, no works were found in literature that dealt with formal verification of RTL designs, while taking care of both computational aspects, present in the high-level reference model, and interface communication aspects, which proceed from the protocol functional specification. This work presents a formal verification methodology, which uses equivalence checking techniques, to validate RTL descriptions through direct comparison with a high-level reference model, and with formal model of the communication protocol. It is based on extracting and comparing complete sequences of states, instead of single states as in traditional equivalence checking, in order to determine if the design intention is maintained in RTL implementation. The natural discrepancies between system level and RTL code are considered, including non-matching interface and memory elements, state mapping, and process concurrency. For the complete problem characterization and solution, a theoretical framework is introduced, where concepts and definitions are provided, and whose validity is formally proved. A tool to apply systematically the methodology was developed and applied on different types of RTL descriptions, written in VHDL and SystemC languages. The results show that the approach may be applied effectively and efficiently to verify formally digital circuits that include, but are not limited to error correction, encryption, image processing, and math functions. Also, evidence has been obtained about the capacity of

the tool to discover both combinatory and sequential bugs injected on purpose, related with computational and protocol functionalities, on real scenarios.

Keywords: Formal verification. Sequential equivalence checking. State sequences. Design under verification. RTL Implementation. Reference model. Communication protocol. Algorithm. Software tool.

LISTA DE FIGURAS

Figura 1 . Fluxo de projeto de CIs digitais generalizado.....	18
Figura 2 . Modelo geral de testbench.	28
Figura 3 . Diferentes BDDs. (a) BDD sem reduzir para a função $f = ab+c$. (b) ROBDD para a função $f = ab+c$. (c) ROBDD para a função $f = a \text{ xor } b \text{ xor } c \text{ xor } d$	32
Figura 4 . Modelagem com CDFG. (a) Diagrama completo com as relações de controle. (b) Um dos blocos básicos em (a), com especificação temporal de tarefas.....	34
Figura 5 . <i>Miter</i> utilizado para checagem de equivalência de dois circuitos.....	39
Figura 6 . Simplificação do problema de checagem de equivalência entre duas FSMs.....	40
Figura 7 . Exemplo de FSMs equivalentes e distinguíveis, de acordo com a definição tradicional.	42
Figura 8 . Técnica de checagem com ferramentas de HLS e SEC no nível RTL.....	44
Figura 9 . Parte do grafo de circuito (a) que representa uma função em C (b).	45
Figura 10 . Transformações realizadas em cima dos estados para elevar o nível de abstração desde RTL até TLM.	47
Figura 11 . Inclusão de comportamentos de comunicação, e de computação, no projeto em RTL.....	51
Figura 12 . Pseudocódigo SystemC para um exemplo de implementação de calculador de GCD: (a) Modelo HL, (b) Especificação de protocolo, (c) Descrição RTL.	52
Figura 13 . Exemplo de FSM composta por uma única sequência de estados.....	54
Figura 14 . Representação gráfica dos conjuntos de variáveis adotados.....	57
Figura 15 . Exemplo de redução de uma sequência k_{RTL} até k_{RTL}^2	58
Figura 16 . Casos hipotéticos que fariam com que a operação de propagação violasse a terceira condição do Postulado 1.....	60
Figura 17 . Configuração inicial de uma sequência k_{RTL}^2 , junto aos dois casos hipotéticos que fariam com que a operação de agrupamento infringisse a primeira condição do Postulado 1.	62
Figura 18 . Exemplos de sequências de estados que podem existir (à esquerda), ou não (à direita), em qualquer conjunto K_{HL}	65

Figura 19 . Exemplos sequências que satisfazem (k_{RTL1}), ou não (k_{RTL2}), a definição de sequência estendida.	68
Figura 20 . Diagrama de fluxo da metodologia global de verificação funcional formal baseada em SEC entre sequências de estados.	74
Figura 21 . Sequências construídas em desconsideração (a), ou em consideração (b) da regra 4 para exploração de estados.	76
Figura 22 . Algoritmos desenvolvidos para construção dos conjuntos de sequências de estados (a), e para checagem de equivalência entre estas (b).	77
Figura 23 . Código VHDL dos processos de um calculador de GCD em alto nível (a) e em RTL (b).	81
Figura 24 . Representações FSMD para os códigos do GCD em alto nível (a), e RTL (b).	82
Figura 25 . Sequências resultantes para a FSMD do calculador de GCD em alto nível (a), e em RTL (b).	83
Figura 26 . Sequências de entradas e saídas para as sequências k_{HL1} e k_{RTL2}	84
Figura 27 . Operações de propagação, remoção, e agrupamento, realizadas pela função R_{HL}	85
Figura 28 . Formas de onda da especificação do protocolo para o calculador de GCD.	86
Figura 29 . Máquina de estados, M_E , que especifica o comportamento temporal do circuito de cálculo de GCD.	86
Figura 30 . Sequências de entradas e saídas para M_E	87
Figura 31 . Obtenção da sequência k_{RTL1}^R , equivalente a k_{E1} , através da função R_E	87
Figura 32 . Três sequências de estados, k_0 , k_1 , e k_2 , obtidas a partir de três submáquinas concorrentes, M_0 , M_1 , e M_2	89
Figura 33 . Arquitetura representada através de blocos correspondente à ferramenta desenvolvida.	90
Figura 34 . Dois casos em que um ordenamento diferente produz comportamentos parcialmente (k_{RTL1}) e totalmente inequivalentes (k_{RTL2}).	91
Figura 35 . Exemplo do uso de CVC3 para checagem da igualdade de duas expressões.	92

LISTA DE ABREVIATURAS E SIGLAS

*BMD	<i>Multiplicative Binary Moment Diagram</i>
AES	<i>Advanced Encryption Standard</i>
AST	<i>Abstract Syntax Tree</i>
ATPG	<i>Automatic Test Pattern Generation</i>
BDD	<i>Binary Decision Diagram</i>
BMC	<i>Bounded Model Checking</i>
CCDFG	<i>Clocked CDFG</i>
CDFG	<i>Diagramas de Fluxo de Controle e Dados</i>
CEC	<i>Checação de Equivalência Combinatorial</i>
CI	<i>Circuito integrado</i>
CNF	<i>Conjunctive Normal Form</i>
CORDIC	<i>Coordinate Rotation Digital Computer</i>
CRC	<i>Cyclic Redundancy Check</i>
CTL	<i>Computation Tree Logic</i>
DPLL	<i>Davis-Putnam-Logemann-Loveland</i>
DUV	<i>Design Under Verification</i>
EFSM	<i>Extended FSM</i>
ESL	<i>Electronic System Level</i>
ExSDG	<i>Extended System Dependence Graph</i>
FBDD	<i>Free-BDD</i>
FIFOs	<i>First In First Out</i>
FSM	<i>Finite State Machine</i>
FSMD	<i>FSM com dados</i>
GCD	<i>Greatest Common Divisor</i>
GOTCHA	<i>Generator of Test Cases for Hardware Architectures</i>
HDL	<i>Hardware Description Language</i>
HL	<i>High-Level</i>
HLS	<i>High-Level Synthesis</i>
IP	<i>Intellectual Property</i>
LTL	<i>Linear Time Logic</i>
MDCT	<i>Modified Discrete Cosine Transform</i>
MPSoC	<i>SoC-Multiprocessador</i>

MTBDD	<i>Multi-Terminal BDD</i>
OVL	<i>Open Verification Library</i>
PSL	<i>Property Specification Language</i>
ROBDD	BDD Reduzido e Ordenado
RTCTL	<i>Real-Time Computation Tree Logic</i>
RTL	<i>Register Transfer Level</i>
SAT	Satisfatibilidade
SEC	Checagem de Equivalência Sequencial
SoC	<i>System-on-Chip</i>
VHDL	<i>Very High Speed Integrated Circuit</i> HDL
ZBDD	<i>Zero-suppressed BDD</i>

TABELA DE CONTEÚDO

1	INTRODUÇÃO.....	17
1.1	Motivação	19
1.2	Justificativa.....	22
1.3	Objetivos.....	23
1.4	Contribuições.....	24
1.5	Organização da Tese.....	24
2	VERIFICAÇÃO POR SIMULAÇÃO E VERIFICAÇÃO FORMAL.....	26
2.1	Conceitos Preliminares	26
2.1.1	Verificação, Validação e Depuração.....	26
2.1.2	Erro, Falha e Defeito.....	26
2.2	Verificação baseada em Simulação	27
2.2.1	Estratégias de Estimulação	28
2.2.2	Cobertura Funcional	29
2.3	Verificação Formal	30
2.3.1	Formas de Representação para Verificação Formal	31
2.3.2	Checagem de Modelos	35
2.3.3	Checagem de Equivalência	37
3	TRABALHOS CORRELATOS	43
4	VERIFICAÇÃO FUNCIONAL DE UMA IMPLEMENTAÇÃO NO NÍVEL RTL ATRAVÉS DE MÉTODOS FORMAIS DE SEC.....	51
4.1	As Máquinas de Estados.....	51

4.2	Verificação da Região Computacional	56
4.3	Verificação do Protocolo de Comunicação	71
5	METODOLOGIA, ALGORITMOS E IMPLEMENTAÇÃO	74
5.1	Decomposição de FSMs em Sequências de Estados	75
5.2	Checação de Equivalência entre Sequências de Estados	78
5.3	Verificação Funcional de um Projeto de Calculador de Máximo Divisor Comum.....	80
5.3.1	Checação do Projeto de GCD com Respeito a um Modelo em Alto Nível...	80
5.3.2	Checação do Comportamento do Protocolo de Comunicação do Projeto de GCD	85
5.4	Extensão a Máquinas de Estados que Representam Processos Concorrentes	87
5.5	Implementação e Ferramentas	89
5.6	Aplicabilidade, Observações, e Restrições da Metodologia de Checagem de Equivalência baseada em Sequências.....	92
6	RESULTADOS	94
6.1	Casos de Estudo.....	94
6.1.1	Blocos de Hardware descritos em VHDL.....	94
6.1.2	Blocos de Hardware descritos em SystemC	96
6.2	Aplicação das Ferramentas de Extração e Comparação de Sequências	97
6.2.1	Resultados para os Modelos em VHDL	97
6.2.2	Resultados para Modelos em SystemC.....	99
7	CONCLUSÕES	102
8	TRABALHOS FUTUROS	104
9	REFERÊNCIAS	105

1 INTRODUÇÃO

Existem dois fenômenos, decorrentes um do outro, que caracterizam o grande avanço dos dispositivos digitais, especialmente nos últimos anos. Primeiramente, a demanda constante do mercado impõe aos fabricantes a necessidade de lançarem produtos novos, e melhores, embora com fortes restrições no tempo do ciclo de desenvolvimento. Ademais, a evolução da tecnologia de circuitos integrados (CIs) tem permitido, hoje em dia, se chegar a escalas de fabricação abaixo dos 20 nm, o que se traduz em ordens de bilhões de componentes por cm² (SCHUEGRAF et al., 2013). Esta tendência de miniaturização (e otimização em geral) dos processos possibilita a criação de CIs mais complexos, o que, por sua vez, permite a construção de aparelhos eletrônicos com mais e melhores serviços. Tal fato não somente satisfaz as exigências do mercado tecnológico, mas também aumenta os padrões esperados para futuros produtos no que diz respeito às suas características, tanto físicas, como funcionais.

Em coerência e proporção com o aumento de serviços e funções, o nível de complexidade de todas as tarefas de projeto de CIs digitais tem aumentado de forma considerável. Visando solucionar isto, metodologias modulares de desenvolvimento têm sido adotadas, as quais se baseiam na ideia de integrar, em um único circuito, diferentes macroblocos funcionais que descrevem um comportamento específico de forma predefinida; tais blocos são comumente chamados de *Intellectual Property* (IP), disponíveis em bibliotecas a partir de projetos prévios, ou fornecedores externos (GAJSKI et al., 2009). Estes blocos IP, através do seu uso, e reuso, têm demonstrado ser altamente produtivos e vantajosos na hora de gerenciar circuitos com altos níveis de densidade e integração.

Exemplos típicos são as arquiteturas multifuncionais como os sistemas em um chip (SoC, do inglês: *System-on-Chip*) ou SoC-Multiprocessador (MPSoC). Estes sistemas reúnem, em um único CI, vários componentes com funções especializadas como processadores, memórias e barramentos. Contudo, dadas suas dimensões em relação ao número de elementos e funções internas, a probabilidade de inserção de erros é consideravelmente alta. Um dos mecanismos que pode causar tal efeito é a intervenção e refinamento manual ao longo do fluxo de desenvolvimento do projeto *Top-down*, com a redução progressiva do nível de abstração. O processo de refinamento pode ser dividido em quatro grandes tarefas, como mostrado na Figura 1 (VAHID; GIVARGIS, 2013), indicado pela seta à esquerda.

Figura 1. Fluxo de projeto de CIs digitais generalizado.

Inicialmente, a funcionalidade completa do sistema é especificada, elaborando, assim, um documento que inclui todos os detalhes que qualquer implementação eventual deve respeitar. Em seguida, o comportamento global do sistema é capturado na forma de uma especificação formal que, geralmente, é descrita em linguagens de alto nível como, por exemplo, C/C++, MatLab (MATHWORKS, 2013), ou SystemC/TLM (ACCELLERA, 2013). A partir destas descrições comportamentais é possível se obter uma implementação no nível de transferência entre registradores (RTL, do inglês: *Register Transfer Level*), seja manualmente, ou através de ferramentas de síntese de alto nível (HLS, do inglês: *High-Level Synthesis*). A última etapa do projeto de CIs consiste da síntese lógica e implementação física, para o qual existem ferramentas e técnicas específicas.

A estratégia *Top-down* está caracterizada por uma notável proporção de tarefas realizadas pelo projetista, podendo-se notar que as fontes de erro não são escassas. A Figura 1 ainda faz referência a uma segunda alternativa, correspondente à técnica conhecida como *Bottom-up*, onde vários IPs são integrados, em um único circuito, para se chegar a um comportamento geral. Dado que este método se baseia no reuso de IPs, a existência de erros em tais blocos implicaria propagação de comportamentos funcionalmente incoerentes ao longo do circuito.

Desta forma, considerando o tamanho e a complexidade dos projetos atuais e a grande chance de se inserir erros funcionais, ou *bugs*, que deterioram a qualidade dos projetos e podem torná-los inválidos, é necessário adotar tarefas robustas de validação, de forma a prevenir a execução de futuras iterações que representam altos custos no ciclo de projeto. Por

exemplo, no caso do processador Power5 da IBM, relata-se uma economia de entre US\$ 50 milhões e US\$150 milhões, ao detectar adequada e antecipadamente todas as inconsistências do projeto (NAVEH et al., 2007). O conjunto de tarefas que visam identificar, isolar e corrigir tais erros é denominado de verificação funcional.

1.1 Motivação

Diante da complexidade dos CIs digitais atuais, a verificação funcional representa um importante desafio, pois ela influi de forma considerável sobre a eficiência de desenvolvimento do projeto. Estima-se que até 80% dos custos totais de projeto são devidos à verificação (DRECHSLER; FEY, 2010), situação que tem se mantido durante as duas últimas décadas, tornando esta atividade no principal gargalo na tentativa de reduzir os recursos de produção (humanos, computacionais, tempo).

Tal problemática tem provocado a aparição de diversas estratégias para diminuir o esforço requerido para a validação de sistemas digitais. Existem, assim, duas técnicas principais para realizar a verificação de blocos de hardware digitais, no nível RTL. Uma delas consiste na construção de um ambiente de execução, chamado de *testbench*, dentro do qual o projeto sob verificação (DUV, do inglês: *Design Under Verification*) de um determinado bloco IP integra, com certo tipo e quantidade de restrições, um meio de operação equivalente àquele em que funcionaria na realidade. Isto significa que o modelo é simulado, e suas respostas a um grupo de estímulos de entrada de especial interesse são examinadas para verificar a sua validade. Especificamente, a técnica é denominada verificação funcional por simulação, e sua maior vantagem se explica a partir da sua capacidade para achar uma quantidade razoável de erros, especialmente em etapas iniciais da verificação; no entanto, é uma alternativa de verificação que consegue mostrar a existência de erros no código na linguagem de descrição de hardware (HDL, do inglês: *Hardware Description Language*) do DUV, mas não comprova sua ausência, sob qualquer hipótese, em circuitos de tamanho médio (>10.000 transistores) ou grande (>100.000 transistores) (WILE; GOSS; ROESNER, 2005).

A verificação funcional baseada em simulação com estimulação aleatória, ou pseudoaleatória, é aquela com maior aceitação (especialmente no setor industrial de CIs) devido à sua notável eficiência, com respeito ao tempo e os recursos computacionais requeridos para a sua realização (BERGERON, 2003). Entretanto, a natureza intrinsecamente aleatória dos estímulos de entrada, que são utilizados nos *testbenches* para a simulação do DUV, faz necessário especificar critérios que permitam conhecer o nível de cobertura a ser

atingido das diversas funcionalidades do sistema. Para isto utilizam-se modelos de cobertura funcional, definidos como métricas que indicam quais características e capacidades do DUV foram exercitadas durante a simulação, com referência ao documento de especificação funcional (PIZIALI, 2004).

É possível identificar vários problemas que diminuem a produtividade da verificação por simulação. Primeiro, a tarefa de programar os *testbenches* é, de fato, suscetível a erros, podendo fornecer informações erradas sobre a simulação. Uma segunda dificuldade está na tarefa de especificar um modelo de cobertura funcional que forneça a melhor informação sobre os valores observados que correspondam às características funcionais do DUV. Isto se explica pelo fato de tais modelos serem criados manualmente pelo projetista de verificação, de forma cuidadosa, e a partir de uma base de conhecimentos que inclui a análise do documento de especificação funcional, experiência, e juízo próprio relacionado com as funções consideradas relevantes (FUJITA; GHOSH; PRASAD, 2008). Por último, tem-se a incapacidade dos *testbenches* em percorrer o espaço de estados completo do DUV, em projetos de complexidade industrial, dentro de um tempo de execução razoável; isto é consequência direta do fato de as ferramentas de simulação em RTL precisarem de uma quantidade de ciclos de processamento de várias ordens de magnitude maior do que a quantidade de ciclos realmente simulados para o DUV (COUSSY; MORAWIEC, 2008).

Dadas as dificuldades da simulação, seria vantajoso, de um ponto de vista mais rigoroso da verificação, utilizar técnicas que permitam determinar, com certeza total, a exatidão do comportamento implementado no DUV. Isto é possível através da verificação formal, em que algoritmos dedicados para percorrer, de forma exaustiva, o espaço de estados de um modelo abstrato do circuito, podendo assim, assinalar todos os erros nele existentes (KERN; GREENSTREET, 1999). Existem duas abordagens formais, adotadas tanto no mundo acadêmico quanto no industrial: a checagem de modelos e a checagem de equivalência.

Na checagem de modelos, um conjunto de propriedades (na forma de asserções) é analisado para todos os estados do modelo do DUV; caso uma das propriedades não for verdadeira, o algoritmo de verificação produzirá como resultado uma sequência de entradas, conhecida como contraexemplo, que conduz à ocorrência do erro a partir de algum estado específico. Vale notar que as ferramentas reais para checagem de modelos de hardware não realizam uma varredura completa do espaço de estados, mas, apenas, sobre uma região limitada do modelo, ou seja, efetuam uma checagem de modelos restringida (BMC, do inglês: *Bounded Model*

Checking) (WILE; GOSS; ROESNER, 2005). Isto é uma consequência direta do alto nível de consumo de recursos computacionais que caracteriza os métodos formais e que compromete a sua eficiência em circuitos de grande complexidade, o que é, de fato, a sua principal desvantagem. Outra questão que pode ser considerada desvantajosa na técnica de checagem de modelos é o problema de se definir o conjunto ótimo de propriedades a se verificar. De um lado, tal conjunto pode ser pequeno até o ponto de não descrever a comportamento do sistema de forma satisfatória, mas, de outro lado, quantidade demasiada de propriedades pode facilmente levar a redundâncias, o que acabaria reduzindo ainda mais a eficiência do algoritmo de checagem ao precisar de mais recursos para se testar propriedades desnecessárias.

De maneira similar à checagem de modelos, a checagem de equivalência é um método que faz uso de um modelo abstrato do projeto para verificar sua validade, mas, ao invés de se testar propriedades, utiliza um modelo adicional do mesmo projeto, e checa-se se os dois modelos são equivalentes funcionalmente, ou não (DRECHSLER, 2004). Portanto, a checagem de equivalência pode ser considerada uma alternativa de verificação mais produtiva, uma vez que não há necessidade de definir um modelo de cobertura funcional (no caso da verificação por simulação), ou conjunto adequado de propriedades (utilizando BMC). Também, no caso de projetos de hardware com refinamentos sucessivos, a checagem de equivalência pode ser aplicada de maneira “natural”, entre duas variantes do DUV, de forma a verificar se a funcionalidade da especificação se mantém. A grande parte das técnicas para checagem de equivalência foi desenvolvida para verificar a consistência entre descrições de circuitos até o nível de abstração RT, usualmente, pela checagem da lógica Booleana, em esquemas em nível de portas lógicas (MOLITOR; MOHNKE, 2004).

Em projetos digitais desenvolvidos com estratégias *top-down*, na evolução do circuito do nível de sistema eletrônico (ESL, do inglês *Electronic System Level*) para o RTL, existe o risco de se inserir um número indeterminado de erros. Tal situação torna a checagem de equivalência entre estas descrições altamente desejável, entretanto, as metodologias tradicionais para checagem de equivalência são aplicáveis, unicamente, a casos em que duas descrições de hardware apresentam nível de abstração similar, ou que, no mínimo, compartilham conjuntos de entradas e saídas idênticos (MOLITOR; MOHNKE, 2004). Com o propósito de prover a checagem de equivalência entre descrições ESL e RTL, extensões foram propostos em (HAO; RAY; XIE, 2012) e (KOELBL et al., 2009), onde informação e/ou código são adicionados ao modelo de referência para incluir aspectos temporais mais

detalhados e torná-lo comparável à representação RTL, o que acaba por adicionar complexidade ao processo de checagem de equivalência. Outra alternativa seria explorar as similaridades entre as descrições em alto nível e RTL através da simplificação do último. É exemplo de tal alternativa o trabalho de (BOMBIERI et al., 2007), onde FIFOs e outras construções em alto-nível são acrescentadas para adaptar a descrição RTL ao nível de abstração do modelo de referência. De fato, com a inclusão de mais código, tal abordagem reduz a potencial vantagem em se aplicar a checagem de equivalência em descrições mais abstratas.

Em abordagem semelhante, em (LEE et al., 2011), sequências de estados são comparadas, consistindo de caminhos curtos entre cada dois pontos de controle na máquina de estado correspondente, o que acaba tornando o número de caminhos muito grande. O método foi concebido para um ambiente de ferramenta de escalonamento em alto nível, não sendo de aplicação geral. Em se tratando de checagem de equivalência entre níveis diferentes de abstração, existem ainda outros problemas adicionais a se resolver como, por exemplo, a diferença entre: interfaces de entrada e saída, sinais e elementos internos de memória, correspondência entre estados, e disparidade dos comportamentos sequenciais e temporais entre cada descrição.

1.2 Justificativa

A microeletrônica continua avançando no mundo inteiro, e, na mesma tendência, o Brasil se encontra em uma etapa de investimento para o fomento da educação e ressurgimento da indústria de semicondutores. A evidência disto é o programa CI Brasil (CIBRASIL, 2013), a partir do qual, os setores acadêmico, industrial e governamental, têm conseguido impulsionar a atividade econômica, assim como a formação de novos projetistas de CIs. Outro exemplo mais ligado à área acadêmica é o programa Brazil-IP (BRAZILIP, 2013), resultado de uma parceria entre universidades brasileiras para o desenvolvimento de metodologias, técnicas, e capacitação para o projeto e a validação de sistemas digitais. No contexto deste programa, e de forma específica à área da verificação funcional, foi desenvolvida a metodologia VeriSC (DA SILVA et al., 2005), caracterizada pelo uso de um modelo de referência em alto nível, como ponto de comparação, para se estabelecer a validade funcional de uma implementação RTL descrita na linguagem SystemC.

Igualmente, em vista da necessidade de metodologias de projeto de CIs digitais, o Grupo de Sistemas Eletrônicos Integrados e Software Aplicado (GSEIS) (GSEIS, 2013), ligado ao

Laboratório de Microeletrônica da Universidade de São Paulo, tem desenvolvido, desde meados da década anterior, trabalhos de pesquisa sobre verificação funcional em resposta ao problema, em continuo crescimento, do tamanho e da complexidade dos CIs digitais e sistemas embarcados.

Assim, o presente trabalho visa satisfazer a demanda nacional e global de metodologias eficientes de verificação funcional que permitam diminuir o ciclo e os recursos de projeto, permitindo, ao mesmo tempo, garantir a validade de descrições de hardware no nível RTL com altos níveis de exaustão.

1.3 Objetivos

O objetivo principal do presente trabalho é estabelecer uma metodologia de verificação funcional formal através do uso de técnicas de modelagem, análise, e exploração direcionadas à checagem de equivalência, para, assim, determinar a validade funcional de uma implementação no nível RTL, comparando-a com um modelo de referência descrito em um nível de abstração superior, e com um modelo formal do comportamento do protocolo de comunicação.

Adicionalmente, aponta-se ao atendimento dos seguintes objetivos específicos:

- analisar e caracterizar as dissimilaridades existentes entre a checagem de equivalência tradicional, restrita a circuitos no mesmo nível de abstração e com conjuntos idênticos de entradas e saídas, e a abordagem própria da presente metodologia, a qual deve considerar níveis e interfaces diferentes;
- implementar uma técnica que permita representar em um modelo formal a funcionalidade do DUV, o modelo de referência, e o protocolo, para seu posterior tratamento formal. Especificamente, deseja-se chegar a uma modelagem, na forma de máquinas de estados finitos, que considere as diferenças mencionadas de forma apropriada e eficiente;
- criar um quadro teórico formalizado, específico para a checagem de equivalência de uma implementação RTL, com um modelo de referência em alto nível e uma especificação do protocolo de comunicação, levando em conta as restrições necessárias como tipos de circuitos alvo, limitações, e escalabilidade;
- desenvolver algoritmos que permitam resolver o problema proposto, através da aplicação sistematizada do formalismo teórico anterior, visando automatizar, e portanto facilitar, o processo de verificar se o DUV é coerente com a especificação funcional;

- desenvolver um ambiente de verificação, coerente com a metodologia proposta, que permita aplicar e aproveitar os benefícios dos algoritmos e modelos formais, integrando-os com métodos e técnicas já conhecidos, quando necessário, para se obter uma ferramenta de checagem integral de equivalência;
- apresentar resultados da aplicação da metodologia sobre circuitos reais, de forma a demonstrar sua efetividade e contribuição ao problema geral da eficiência da verificação e do projeto de sistemas digitais.

1.4 Contribuições

A contribuição principal do presente trabalho de doutorado é a obtenção de uma metodologia de verificação funcional formal, que aproveite as vantagens da checagem formal de equivalência entre modelos, para tornar possível a verificação funcional de uma implementação RTL de forma coerente com as tendências atuais de projetos de CIs em alto nível.

Contribuições adicionais incluem:

- um novo quadro teórico formalizado que poderá servir como base para a futura constituição, e unificação, de uma metodologia geral de checagem de equivalência entre níveis diferentes de abstração;
- uma técnica de representação interna, tanto do DUV como do modelo de referência, na forma de máquinas de estados finitos, para modelar e analisar aspectos computacionais comuns a ambos modelos, assim como detalhes específicos e únicos do DUV;
- algoritmos que permitam explorar, com eficiência razoável, o espaço de estados dos modelos formais, e assim, estabelecer sua conformidade funcional;
- uma ferramenta de software que permita integrar as técnicas de checagem de equivalência, próprias da metodologia presente, com outras técnicas e ferramentas já existentes. Com esta ferramenta será possível verificar todos os aspectos, de alto e baixo nível, do projeto sob verificação.

1.5 Organização da Tese

O resto do corpo da presente tese de doutorado está organizado da forma seguinte:

- Capítulo 2. Introdução às teorias, conceitos, e definições gerais, relacionados com a área da verificação funcional, incluindo simulação, e técnicas formais como checagem de

modelos e checagem de equivalência. É feita uma revisão da teoria tradicional sobre checagem de equivalência sequencial que serve de base para a teoria do Capítulo 4;

- Capítulo 3. O estado da arte relevante ao presente trabalho é apresentado. Citam-se, em especial, trabalhos sobre checagem de equivalência entre descrições com níveis diferentes de abstração;
- Capítulo 4. Apresentam-se todos os detalhes teóricos da metodologia proposta neste trabalho. Mostra-se, inicialmente, o tratamento formal para o caso da checagem de equivalência do DUV com respeito a um modelo de referência em alto nível. A seguir, incluem-se as definições necessárias para verificar os aspectos de baixo nível do DUV, relacionados com o protocolo de comunicação;
- Capítulo 5. Proporcionam-se todos os aspectos práticos e técnicos que possibilitam a verificação de descrições em RTL, de acordo com a teoria do Capítulo 4. Este capítulo inclui algoritmos, ferramentas, e um exemplo de aplicação;
- Capítulo 6. Exibem-se, aqui, os resultados de aplicação da ferramenta desenvolvida sobre vários tipos de circuitos, descritos nas linguagens VHDL e SystemC;
- Capítulo 7. Apresentam-se as conclusões finais da presente tese, de acordo com o exposto em todos os capítulos anteriores;
- Capítulo 8. Sugestão breve sobre trabalhos futuros para melhora da metodologia apresentada neste trabalho.

2 VERIFICAÇÃO POR SIMULAÇÃO E VERIFICAÇÃO FORMAL

A tarefa de verificação é um dos grandes gargalos no projeto de circuitos integrados, consumindo grande quantidade de recursos humanos e materiais. Desta forma, inúmeras e diferentes abordagens foram propostas como tentativa para aumentar a eficiência da verificação. Neste capítulo, apresentam-se os casos mais representativos de tais abordagens. A primeira seção introduz algumas definições básicas. A segunda seção trata a verificação por simulação e *testbenches*; a terceira e última seção mostra as estratégias mais conhecidas para verificação formal, passando pelas formas de representação mais comuns, algoritmos binários de verificação, checagem de modelos, e checagem de equivalência.

2.1 Conceitos Preliminares

No capítulo anterior foram estabelecidos alguns conceitos fundamentais sobre teoria, técnica, e situação atual da verificação funcional de CIs digitais. A seguir, incluem-se conceitos considerados necessários para o melhor entendimento das seções e capítulos posteriores.

2.1.1 Verificação, Validação e Depuração

A verificação é o conjunto de tarefas que objetivam garantir que uma determinada implementação atinge os parâmetros descritos, unicamente, na especificação do sistema (KROPF, 1999); isto significa que, se a especificação divergir da intenção inicial, a correspondente implementação também o fará. Caso a verificação sinalize a presença de erros, estes devem ser corrigidos através da depuração, um processo composto geralmente por várias iterações.

A validação, de forma mais abrangente, visa demonstrar que ambas a implementação e a especificação são coerentes com a intenção (KROPF, 1999). Desta forma, a validação engloba todos os processos como verificação, depuração, e teste físico, que inclui a análise completa do sistema junto com o software.

2.1.2 Erro, Falha e Defeito

Entende-se por defeito qualquer condição explícita que leva um sistema em operação ou execução a estados (observáveis) que diferem da especificação. Tais estados recebem o nome de falha. Em termos de hardware, os defeitos são consequência de erros, os quais se encaixam em dois tipos principais. Por um lado, erros funcionais são aqueles produzidos quando o

código em HDL descreve comportamentos incoerentes com a especificação. Por outro lado, erros estruturais são consequência da instanciação de componentes inapropriados no código em HDL, ou de uma ferramenta de síntese defeituosa (BEHROOZ, 1997).

2.2 Verificação baseada em Simulação

A simulação tem sido, durante décadas, a opção mais utilizada para detectar rapidamente uma boa parte dos erros de projeto, particularmente em etapas iniciais de desenvolvimento. Normalmente, os circuitos digitais são descritos utilizando linguagens como Verilog, VHDL, ou SystemC, tendo como objetivo a geração de arquivos de código RTL sintetizável, os quais podem ser simulados utilizando ferramentas projetadas especificamente para tal. Independentemente do fato de ser aberta ou comercial, genérica ou específica, toda ferramenta para simulação RTL está composta por um *testbench*, ou ambiente de simulação, que modela, sob determinadas restrições, o ambiente real de aplicação e utilização do DUV. O *testbench* mais básico consiste de um gerador de estímulos, componentes de determinação de erro, e o circuito ou lógica que está sendo verificada (o DUV) (WILE; GOSS; ROESNER, 2005).

Um dos modelos de *testbench* mais genéricos e utilizados é o proposto em (BERGERON, 2003), ilustrado na Figura 2. Este modelo, como mostrado, apresenta dois tipos de sinais: os canais abstratos, que representam transações ou valores dos parâmetros de simulação em alto nível, independentes do tempo; e sinais de tempo, que transportam informações de baixo nível correspondentes aos sinais que o DUV utilizaria na sua operação real.

O ponto de partida deste modelo é o gerador de estímulos. Neste bloco, estímulos são gerados no nível de transações, ou seja, representações abstratas das entradas do DUV. Há basicamente duas funções no bloco de amarração: primeiramente, o *Driver*, onde as transações provenientes do gerador de estímulos são traduzidas em sinais sincronizados com relógio, para que possam ser inseridas no DUV (o objeto do teste); depois, o *Monitor*, onde as reações ou respostas do DUV em sinais de tempo são observadas e transcritas de novo em termos de transações para fins de comparação no bloco de autochecagem. A funcionalidade é testada através deste último bloco, o qual pode conter, por exemplo, um modelo de referência, normalmente descrito em linguagens de alto nível como o C++, MatLab, SystemC, SystemVerilog, etc., cujas respostas aos mesmos sinais de estímulos inseridos no *Driver* são comparadas aos resultados do *Monitor* (ROMERO, 2005).

Figura 2. Modelo geral de testbench.

2.2.1 Estratégias de Estimulação

Os estímulos a serem gerados dentro do bloco que efetua as funções de teste (Figura 2) podem se enquadrar em quatro categorias:

- estimulação direta: esta é uma forma de verificar características específicas do projeto através de estímulos programados para tal. Muitas vezes o conjunto de estímulos é recomendado na especificação do IP. A maior desvantagem desta alternativa é a sua falta de escalabilidade, tornando-se incontável para quantidades elevadas de casos de teste;
- estimulação por casos reais: os casos reais de teste são aqueles compostos, unicamente, por estímulos que representam dados reais de operação do sistema. A relevância deste tipo de estimulação reside na necessidade de se monitorar, e verificar, o funcionamento do modelo do sistema, sob condições reais de operação. Exemplos incluem cadeias de bits correspondentes a áudio, imagens, e vídeo; dados de autenticidade, segurança, e qualidade de informação; sequências de dados de controle, etc;
- estimulação de casos extremos: considerada um tipo especial de estimulação direta, na qual tenta-se descobrir erros de projeto que possivelmente seriam detectados somente após um grande número de ciclos redundantes de simulação;
- estimulação aleatória: com as restrições apropriadas para não gerar estados ilegais, estes estímulos são efetivos ao gerar uma grande quantidade de valores, que abrangem, embora com baixa densidade, a maior parte do espaço de estados do sistema.

Em (BERGERON, 2003) recomenda-se que os *testbenches* devam ter autochecagem e operar utilizando-se transações, enquanto os estímulos devem ser gerados de forma aleatória, com restrições (limites de valores) definidas. Também é possível restringir ou dirigir os

estímulos gerados, para otimizar os critérios de cobertura funcional. (Seção 2.2.2) Como estratégia geral de verificação, a combinação da estimulação aleatória com a estimulação dirigida produz um aumento da eficiência significativo, ao aumentar a taxa de ocorrência daqueles estímulos que exercem funções do sistema que são de especial interesse.

2.2.2 Cobertura Funcional

A tarefa de validação de projetos de hardware deve ser eficiente e planejada com ajuda de métricas que forneçam uma visão clara e objetiva das propriedades que são verdadeiramente relevantes, sem se desperdiçar tempo e recursos em processos de verificação desnecessários. A cobertura funcional é a métrica utilizada para medir o progresso da verificação, baseando-se na estimativa da funcionalidade coberta do DUV, na simulação do modelo em HDL (WILE; GOSS; ROESNER, 2005). Sendo as métricas de cobertura adequadas, quanto mais alta a cobertura obtida, mais aspectos relevantes do projeto terão sido monitorados.

No entanto, independentemente da forma em que os estímulos são gerados, o fato de alcançar 100% de cobertura funcional não significa, necessariamente, testar a funcionalidade completa do sistema. Este é um aspecto problemático da verificação por simulação, pois dada a complexidade dos projetos digitais atuais, é difícil se definir o modelo de cobertura funcional que melhor reflita o comportamento integral do DUV. Esta é uma tarefa que se baseia na experiência, visão, e entendimento que a equipe de verificação possui sobre a especificação funcional do projeto. Outro problema é o alto grau de subjetividade em se determinar o ponto até o qual é necessário exercitar o DUV para garantir que um determinado tipo ou quantidade de erros de projeto tenham sido descartados, ou que certo nível de cobertura funcional tenha sido atingido.

Há várias alternativas para se organizar a cobertura funcional, que ajudam a atingir o maior nível de depuração possível; isto é, que providenciem um nível mínimo e aceitável de confiabilidade. Algumas delas são:

- cobertura funcional por itens - consiste de um conjunto de eventos relevantes e de especial interesse que se pretende observar à entrada, dentro, ou à saída do DUV. Tais eventos devem ser definidos em função dos valores, ou conjuntos de valores, que os parâmetros de simulação do DUV podem adquirir, e do número de itens (quantidade de vezes que o evento deve ocorrer para se considerar satisfeito) correspondente a cada um dos eventos definidos (BERGERON, 2003). Parâmetros podem ser entradas, saídas, sinais, ou

variáveis do sistema, e incluem endereços de memória, comprimento de pacotes de dados, variáveis de dados ou controle, tipos de interrupções, e estado de terminação de transações. Um problema importante está relacionado ao tamanho numérico de cada um dos eventos de cobertura, já que a faixa de valores de cada evento pode ser ampla demais para se estabelecer níveis exaustivos de cobertura. A forma de resolver este problema é se limitar a um número suficientemente representativo de itens a serem monitorados para cada faixa;

- cobertura funcional por combinações - conhecida também por cobertura cruzada, é necessária quando houver dependências entre os parâmetros à entrada, saída, ou no interior do DUV. O método básico para se definir eventos de cobertura cruzada consiste no uso dos operadores lógicos elementares AND e OR. Assim, objetiva-se medir unicamente a coincidência de valores específicos de vários parâmetros de simulação;
- cobertura funcional por transições - existe um item de cobertura para cada possível transição, de acordo ao conjunto de casos de teste que são a ela associados. Existem propostas de ferramentas para simulação, focalizadas em modelos de cobertura por transições, como, por exemplo, GOTCHA (do inglês, *Generator of Test Cases for Hardware Architectures*), apresentada em (BENJAMIN et al., 1999), a qual utiliza testes de cobertura por transições como componente de um motor de simulação. Tem como vantagem a sua grande correspondência funcional com o projeto.

2.3 Verificação Formal

A verificação formal é notavelmente interessante, no que diz respeito à sua capacidade para provar (na teoria) a exatidão do DUV exaustivamente, isto é, fornece 100% de certeza na verificação. A ideia geral da verificação formal é provar, algoritmicamente, que as funções do projeto estão implementadas de forma correta, ao invés de simular e aplicar vetores de teste ao projeto em RTL. Inúmeras técnicas têm sido propostas para realizar o processo de prova formal. Geralmente, elas estão caracterizadas pela utilização de uma forma de representação interna específica para modelar as descrições em HDL como, por exemplo, os diagramas de decisão binária (BDD, do inglês *Binary Decision Diagram*), e a aplicação, sobre tais representações, de algoritmos solucionadores como os de satisfatibilidade (SAT), ou geração automática de padrões de teste (ATPG, do inglês *Automatic Test Pattern Generation*) (DRECHSLER, 2004).

A existência de técnicas variadas é uma resposta direta ao problema da complexidade computacional, típica das metodologias de verificação formal, e decorrente da grande quantidade de estados dos projetos digitais atuais, a incapacidade dos algoritmos para percorrer eficientemente o espaço de estados, e a limitação em recursos computacionais. A síntese desses três fatores se conhece como “o problema de explosão de estados”, e é um dos principais problemas a ser contornado dentro de qualquer ambiente de verificação formal (KERN; GREENSTREET, 1999).

A seção seguinte introduz algumas das alternativas mais comuns de representação e modelagem funcional de circuitos digitais (Seção 2.3.1). Posteriormente, apresentam-se os cenários típicos de verificação formal de hardware: a checagem de modelos, conhecida também como checagem de propriedades (Seção 2.3.2), e a checagem de equivalência (Seção 2.3.3). Existe também a prova de teoremas, que visa demonstrar que uma expressão lógica é condizente com axiomas e teorias carregados previamente no provador (a ferramenta de prova) (WILE; GOSS; ROESNER, 2005). Esta técnica tem uma função importante na verificação formal de alguns tipos específicos de hardware como, por exemplo, unidades de ponto flutuante em processadores. No entanto, a aplicação desta técnica é limitada devido à complexidade das ferramentas, e da própria teoria matemática. Assim, é uma alternativa de verificação formal que vai além do escopo deste trabalho, e portanto, detalhes adicionais não são fornecidos.

2.3.1 Formas de Representação para Verificação Formal

Os modelos de hardware em HDL incluem diversos tipos de lógica descrita em termos das instruções e expressões próprias de tais linguagens. O primeiro passo, nas técnicas formais, consiste da modelagem de tal lógica por meio de representações computacionais que permitam, posteriormente, a aplicação dos eventuais algoritmos para efetuar a própria verificação. A seguir, expõem-se as formas mais comuns de representação interna.

2.3.1.1 Forma Normal Conjuntiva

As formas canônicas são uma das formas mais utilizadas para implementar algoritmos e ferramentas de verificação formal, uma vez que elas representam cada função Booleana de maneira única. Uma forma canônica bastante popular é a forma normal conjuntiva (CNF, do inglês *Conjunctive Normal Form*), que aparece frequentemente na implementação de inúmeras ferramentas para análise e verificação formal de modelos como, por exemplo,

ZChaff (ZCHAFF, 2013), Berkmin (BERKMIN, 2013), e GRASP (MARQUES-SILVA; SAKALLAH, 1999). Entretanto, o uso de CNF apresenta fortes desvantagens, como a perda de toda informação estrutural do projeto, adição de variáveis desnecessárias, e crescimento exponencial no caso de transformações requeridas a partir de uma fórmula que não seja CNF (WILLIAMS; ANDERSEN; HULGAARD, 2001).

2.3.1.2 Diagramas de Decisão Binária

Os BDDs também têm sido usados, de forma abrangente, na criação de estratégias de verificação formal. Os BDDs são formas de representação gráfica que permitem manipular fórmulas Booleanas, de forma mais compacta, e com uma eficiência superior àquela de outras formas canônicas. Particularmente, os BDDs reduzidos e ordenados (ROBDDs) apresentam um nível máximo de redução e minimização, portanto, o ROBDD para qualquer função Booleana (WILE; GOSS; ROESNER, 2005). A Figura 3 mostra graficamente a vantagem de se utilizar BDDs para representar funções lógicas. A Figura 3^a corresponde ao BDD, sem ordenar ou reduzir, da função “ $f = ab + c$ ”; ao lado, a Figura 3b mostra um ROBDD que requer muito menos espaço para a mesma função. Para muitos casos, os ROBDDs são representações que são muito mais eficientes, se comparados a outras formas como as tabelas de decisão ou

Figura 3. Diferentes BDDs. (a) BDD sem reduzir para a função $f = ab + c$. (b) ROBDD para a função $f = ab + c$. (c) ROBDD para a função $f = a \oplus b \oplus c \oplus d$.

CNF. Na Figura 3c, por exemplo, o ROBDD para a operação ‘XOR’ entre quatro variáveis, somente cresce linearmente com o número de variáveis.

Existem, porém, muitos casos em que o uso de BDDs (ou ROBDDs) não é a escolha mais eficiente ou vantajosa. Funções aritméticas, como a multiplicação ou divisão, crescem de forma exponencial em uma representação de BDD. Adicionalmente, a complexidade e ordem de crescimento dos BDDs são facilmente afetadas segundo o ordenamento imposto às variáveis, o que, por sua vez, é um problema de complexidade exponencial (isto é, achar uma sequência de variáveis ótima). Existem diversos tipos de BDD mais elaborados, que surgiram em decorrência da necessidade de otimizar o tratamento de funções lógicas com tais diagramas. Exemplos de tais elaborações incluem o FBDD (Free-BDD), o MTBDD (do inglês, *Multi-Terminal BDD*), e o ZBDD (do inglês, *Zero-suppressed BDD*). Todas estas variações permitem manipular funções lógicas com níveis maiores de abstração do que é possível com o BDD puro (binário), diminuindo, por conseguinte, a complexidade dos problemas anteriormente mencionados (DRECHSLER, 2004).

2.3.1.3 Máquinas de Estados Finitos

Em geral, fórmulas CNF e BDDs são formas de representação que podem ser aproveitadas com ajuda de algoritmos eficientes, ainda que, devido à sua natureza puramente combinatória, torna-se necessário trabalhar com métodos alternativos para o caso da lógica sequencial. As máquinas de estados finitos (FSM, do inglês *Finite State Machine*) são um modelo computacional aplicado de forma intensiva para representar características sequenciais dos sistemas digitais, e assim, construir ambientes e ferramentas de verificação que possibilitam a checagem exaustiva das funções relacionadas com este tipo de lógica.

O sistema de verificação proposto em (ZHU; BIAN; WU, 2006), por exemplo, envolve a construção de uma FSM que representa a lógica de controle do sistema a partir do modelo descrito em VHDL, e de um conjunto preestabelecido de propriedades a serem verificadas; separada, mas paralelamente, simulam-se todas as outras funções de processamento que requerem mecanismos exatos de modelagem temporal. O sistema de teste UniTESK, inicialmente criado para verificação de software, utiliza também uma FSM que informa o estado atual do sistema, para gerar sequências otimizadas de vetores de teste e, como resultado, obter um equilíbrio entre escalabilidade e exaustividade (KAMKIN, 2006). Um tipo especial de FSM, denominadas EFSM (do inglês: *extended FSM*), aparece em (FUMMI et al., 2007) para, também, gerar melhores casos de teste. No trabalho, os processos do

circuito são modelados de forma concorrente por um grupo de EFSMs em execução simultânea, sobre o qual opera um algoritmo ATPG, para criar estímulos de entrada ao DUV que possuem uma alta probabilidade de achar estados errôneos. Existem adicionalmente implementações de ferramentas que efetuam uma modelagem do DUV na forma de FSMs para levar a cabo tarefas de verificação formal. Um exemplo é apresentado em (GROBE; DRECHSLER, 2005), e consiste da ferramenta chamada CheckSyC, a qual transforma a descrição do sistema, da linguagem SystemC, para uma representação FSM, e executa um algoritmo BMC em base às propriedades que se desejam verificar.

2.3.1.4 Diagramas de Fluxo de Controle e Dados

Os diagramas de fluxo de controle e dados (CDFG) são especialmente vantajosos para se modelar aspectos temporais precisos como, por exemplo, o clock do circuito, e o escalonamento das tarefas. Portanto, são uma forma eficiente de se representar as dependências de dados entre operações sucessivas e concorrentes (com especificação detalhada do clock), assim como, as relações de controle descritas no código HDL do projeto (GAJSKI et al., 2009). A Figura 4a ilustra um exemplo de tal modelagem, onde sentenças de controle como *if*, *while*, e *for*, são tratadas como blocos de decisão que podem formar laços iterativos; os blocos básicos (BB, exemplificado na Figura 4b) consistem de grupos de tarefas

Figura 4. Modelagem com CDFG. (a) Diagrama completo com as relações de controle. (b) Um dos blocos básicos em (a), com especificação temporal de tarefas.

com dependências de dados, que devem ser executadas sob restrições de recursos físicos do sistema, junto a um escalonamento bem definido. Dado que os CDFGs compreendem todas as relações de dependências de dados e controle em separado, sua implementação eficiente em um sistema de verificação funcional resulta notavelmente vantajosa. O trabalho elaborado em (ZHU; BIAN; WU, 2006) demonstra a possibilidade de se obter o CDFG do DUV a partir do código em linguagem VHDL, e, a partir de tal grafo, gerar a FSM correspondente para checagem formal de propriedades, e módulos de processamento a serem simulados para ajudar na cobertura de estados do sistema com maior velocidade. Adicionalmente, é possível realizar uma análise estática diretamente no CDFG para verificar a validade de propriedades relacionadas com sinais entre processos.

2.3.1.5 Árvores Sintáticas Abstratas

Uma outra forma de representar a funcionalidade do DUV, a ser tratada nesta seção, são as árvores sintáticas abstratas (AST, do inglês *Abstract Syntax Tree*). As ASTs são estruturas de dados hierárquicas ramificadas, cujos nós correspondem a variáveis, operadores, sentenças e funções do código do projeto. No contexto da verificação formal de hardware, tais árvores são usadas por uma grande variedade de metodologias e ferramentas que executam seus algoritmos sobre elas. Como mencionado acima, a ferramenta CheckSyC é capaz de gerar FSMs a partir do código em SystemC de um determinado circuito, ainda que, para chegar em uma FSM final, a ferramenta cria inicialmente uma AST como forma de representação intermediária. Outra ferramenta, chamada PINAPA (MOY; MARANINCHI; MAILLET-CONTOZ, 2005), utiliza um compilador genérico para criar uma representação da arquitetura do circuito, que liga cada processo com sua AST correspondente; a partir de tal ponto a representação pode ser usada para verificar qualquer aspecto do projeto, com a assistência de ferramentas complementares.

2.3.2 Checagem de Modelos

A técnica de checagem de modelos, conhecida também como checagem de propriedades, é empregada, mormente, como um meio para se determinar a exatidão de um conjunto de propriedades específicas do DUV, coerentes com as características funcionais contidas no documento de especificação funcional. Mais formalmente, a checagem de modelos é o termo estabelecido academicamente para a verificação de determinadas propriedades no percurso da FSM do sistema (KERN; GREENSTREET, 1999). Tais propriedades podem ser de tipo

estático, as quais devem ser válidas para todos os estados (caso em que costumam ser chamadas de “invariantes”), e de tipo temporal, avaliadas para um subconjunto bem definido de estados em que devem ser válidas.

2.3.2.1 Especificação de Propriedades

A forma de se especificar as propriedades varia segundo o tipo de projeto sob verificação, mas, de forma geral, são descritas na forma de asserções sob padrões desenvolvidos para tal propósito. Há vários padrões de especificação de propriedades como LTL (*Linear Time Logic*), CTL (*Computation Tree Logic*), OVL (*Open Verification Library*), e, os mais popularizados, PSL (*Property Specification Language*) e SVA (*System Verilog Assertions*). Usualmente, as linguagens de descrição de hardware permitem a definição de asserções para a checagem de propriedades, porém, se comparadas com as anteriores, são bastante limitadas.

Além de inconvenientes como aspectos ou limitações próprias das linguagens para propriedades, e a possível existência de erros, inclusive, no código das asserções, escolher um conjunto adequado de propriedades ótimo para sua posterior checagem é um problema nada trivial. Habitualmente, as propriedades são sintetizadas manualmente a partir da especificação funcional (que está em uma linguagem natural, ou seja, informal), e sua subsequente formalização requer a intervenção direta do engenheiro de verificação. Esta tarefa se baseia na perspectiva e experiência da equipe de verificação, e portanto, não há garantia de produzir um conjunto completo, e ao mesmo tempo mínimo, de propriedades do DUV (FUMMI; PRAVADELLI, 2007).

2.3.2.2 Algoritmos de Checagem de Modelos

A abordagem mais intuitiva e simples para realizar a checagem formal das propriedades do DUV é chamada de checagem de modelos explícita. Tal estratégia, como seu nome o indica, usa uma representação direta e explícita de todos os estados do sistema. O processo de checagem começa nos estados válidos, e prossegue, aplicando a cada instante todas as possíveis entradas válidas ao DUV. Desta forma, continua ciclo após ciclo, enumerando todos os estados alcançados, e checando a validade das propriedades em cada um deles (WILE; GOSS; ROESNER, 2005). Tal abordagem leva ao problema de explosão de estados (GOLDREICH, 2008), o qual, por sua complexidade, tem sido a principal motivação para o aperfeiçoamento de tecnologias alternativas de verificação formal.

A checagem simbólica de modelos adota representações como BDDs para representar o espaço de estados do modelo. Esta é uma solução que, desde suas pesquisas iniciais, provou aumentar a eficiência da checagem proeminentemente (KERN; GREENSTREET, 1999). De fato, na atualidade, grande parte das ferramentas de verificação formal de propriedades inclui algoritmos de checagem simbólica implicitamente. A principal desvantagem da checagem simbólica provém, contraditoriamente, da aplicação de BDDs para representar os estados; a explicação é que, como já mencionado, a descrição de funções Booleanas com BDDs requer algoritmos e otimizações especiais para evitar o crescimento desmesurado do diagrama. Isto limita ainda o número de estados que é possível checar de forma simbólica.

A técnica BMC busca tratar do problema de explosão de estados através da restrição a um inteiro K , o comprimento máximo das sequências de estados durante a execução do algoritmo de checagem. Se durante tal checagem for achado um erro no DUV, produzir-se-á o respectivo contraexemplo (sequência de entradas que produz o erro); caso contrário, o processo começará de novo com um incremento no valor de K que, por sua vez, poderá atingir um valor máximo ditado pela quantidade de recursos físicos disponíveis para a verificação (GAJSKI et al., 2009). Em contrapartida, o BMC não pode ser mais considerado um método de verificação exaustivo, já que não fornece exploração completa do sistema, de forma que, cabe ao engenheiro de verificação estabelecer a região específica de estados a ser coberta pelo algoritmo.

Em geral, o BMC se destaca pela sua escalabilidade e aplicabilidade a projetos de grande escala, fato que é evidenciado pelo número de ferramentas de verificação que contêm implementações do algoritmo, como por exemplo, SymC (RUF et al., 2003) e NuSMV (CAVADA, 2010).

2.3.3 Checagem de Equivalência

A checagem de equivalência é definida como o problema de estabelecer se dois sistemas descrevem o mesmo comportamento. É comum se adotar uma interpretação de que, dados dois circuitos A e B , sua equivalência é determinada quando se assegura que ambos, A e B , implementam a mesma função Booleana. Neste contexto, o problema pode ser definido formalmente como segue (MOLITOR; MOHNKE, 2004).

“Dadas duas representações d_F e d_G de duas funções Booleanas $f, g: \{0,1\}^n \rightarrow \{0,1\}^m$, objetiva-se decidir se as funções Booleanas f e g são iguais, ou seja, se $f(\alpha) = g(\alpha)$ se mantém para todo $\alpha \in \{0,1\}^n$.”

A expressão anterior traduz-se, basicamente, em que a resposta à saída dos dois circuitos deve ser igual, para qualquer entrada permitida, segundo sua especificação funcional. Vale notar ainda que a definição anterior refere-se unicamente ao caso da checagem de equivalência combinatória; para a checagem de equivalência sequencial considera-se a mesma definição, porém, fazendo a respectiva análise para todos os estados da FSM que descreve o circuito.

2.3.3.1 Checagem de Equivalência Combinatória (CEC)

O algoritmo a aplicar para a solução do problema da checagem de equivalência depende, particularmente, do nível de abstração no que se faz tal checagem. O tamanho do circuito e o tipo de funções que o caracterizam também são fatores determinantes na escolha de um determinado algoritmo. A forma mais trivial de se checar a equivalência entre duas descrições é através da obtenção das respectivas formas canônicas. Se os dois resultados forem idênticos, então, pode-se concluir que os circuitos são equivalentes; no entanto, a aplicação de tal técnica se limita a modelos de tamanho pequeno (operações aritméticas, funções lógicas simples, etc).

Duas formas canônicas, amplamente usadas pelas ferramentas de checagem de equivalência, são o ROBDD para o caso de checagem no nível de bit, e o diagrama de momento binário multiplicativo (*BMD, do inglês: *Multiplicative Binary Moment Diagram*) no nível de palavra. Um *BMD é, em geral, um diagrama que representa uma função pseudo-Booleana (aquela cujas saídas representam um inteiro, através de algum tipo de codificação como, por exemplo, binário sem sinal, sinal-magnitude, ou complemento de dois), o qual tem pesos associados a seus ramos que são combinados de forma multiplicativa. A checagem no nível de bit consiste em considerar que uma função Booleana, com n entradas e m saídas, pode ser representada por meio de m ROBDDs, cada um com n entradas. Já para o caso de funções pseudo-Booleanas (isto é, funções cuja saída representa um dado inteiro), é mais eficiente o uso de *BMDs, que são diagramas que fornecem um nível superior de abstração e que permitem modelar uma função inteira em um único diagrama completo (KERN; GREENSTREET, 1999).

Outra abordagem para verificar a equivalência de dois circuitos consiste da observação à saída de um *miter* (rede utilizada para comparar as respostas de vários circuitos em paralelo) sobre o qual se executam algoritmos SAT ou ATPG. O funcionamento da rede composta pelo *miter* é ilustrado na Figura 5. Basicamente, a equivalência entre os dois circuitos, *A* e *B*, será provada se a saída mantiver o valor ‘0’ para quaisquer sinais de entrada (MOLITOR; MOHNKE, 2004). Na prática, é mais interessante buscar um caso de discrepância entre os circuitos, quando a saída do miter apresenta valor ‘1’. Para solucionar o problema de checagem esquematizado na Figura 5, empregando, por exemplo, o algoritmo de satisfatibilidade DPLL, calcula-se a função Booleana do *miter* completo, em CNF. A seguir, o algoritmo escolhe uma das variáveis, a substitui pelo valor ‘0’ ou ‘1’, encontra o valor de outras variáveis através de regras de implicação (para manter o valor ‘1’ à saída), e faz uma chamada recursiva para continuar com variáveis que não tenham nenhum valor designado ainda. Se, ao final do algoritmo, todas as variáveis tiverem um valor designado, ou estiverem explicitamente indefinidas, significa que a função foi satisfeita (isto é, os dois circuitos não são equivalentes).

Figura 5. *Miter* utilizado para checagem de equivalência de dois circuitos.

2.3.3.2 Checagem de Equivalência Sequencial (SEC)

A forma básica de se verificar se duas FSMs são equivalentes consiste da exploração do espaço completo de estados, resolvendo, para cada um deles, um problema de checagem de equivalência combinatória. Porém, tal estratégia de checagem é impraticável para projetos de tamanho real, pois a exploração exaustiva, como já mencionado anteriormente, resulta em um problema de explosão de estados. Em tal situação se faz necessário simplificar o problema de exploração dos estados das FSMs, de forma a possibilitar a execução dos algoritmos de checagem. A Figura 6 ilustra uma possível simplificação, onde, para duas FSMs, assume-se

que o estado atual de cada uma delas é o mesmo. Na figura, as duas FSMs recebem um mesmo vetor de sinais de entrada, A , como também, um mesmo vetor de estados, B . Assim, os vetores de saída, Z_1 e Z_2 , e os vetores de estados futuros, B_1^+ e B_2^+ , são comparados através de duas portas XOR, obtendo duas saídas, E_1 e E_2 , respectivamente; as duas máquinas de estados serão equivalentes se os vetores E_1 e E_2 forem '0' em todo momento (WILE; GOSS; ROESNER, 2005).

Figura 6. Simplificação do problema de checagem de equivalência entre duas FSMs.

De forma geral, a equivalência entre dois circuitos representados por duas FSMs, poderá ser garantida sempre que a respectiva equivalência entre as máquinas for demonstrada. Convencionalmente, duas FSMs, M_A e M_B , são equivalentes se seus estados iniciais, s_{0A} e s_{0B} , forem funcionalmente equivalentes; por sua vez, dois estados, s_{0A} e s_{0B} , pertencentes a duas FSMs, M_A e M_B , respectivamente, são equivalentes se, a partir deles, toda sequência de entradas produzir a mesma sequência de saídas, nas duas máquinas. A formalização dos conceitos e definições para determinar a equivalência entre duas FSMs, emprestada de (MOLITOR; MOHNKE, 2004) e adaptada, quando necessário, é apresentada a seguir.

2.3.3.3 Checagem de Equivalência Sequencial entre Duas FSMs

Esta seção apresenta a definição tradicional de SEC entre máquinas de estados, cuja aplicação real limita-se à comparação de descrições de circuitos com o conjunto de entradas e saídas iguais, e correspondência, um a um, entre elementos internos de memória; tal

correspondência traduz-se em um mapeamento bijectivo, ou no mínimo sobrejectivo, entre os estados das máquinas que representam as duas descrições. Para começar, define-se, inicialmente, o conceito de FSM ao que se fará referência ao longo do presente trabalho.

Definição 1. Máquina de estados finitos: uma máquina de estados finitos (FSM) é definida como o conjunto $M = \{S, I, O, \delta, \lambda, s_0\}$, de forma que:

- $S = \{s_0, \dots, s_r\}$ é o conjunto finito de $r+1$ estados, e $r \in \mathbb{N}$;
- I é o alfabeto de entrada, que engloba um conjunto de $n+1$ variáveis $\{v_{i0}, \dots, v_{in}\}$;
- O é o alfabeto de saída, que engloba um conjunto de $m+1$ variáveis $\{v_{o0}, \dots, v_{om}\}$;
- $\delta: S \times I \rightarrow S$ é a função de transição entre estados;
- $\lambda: S \times I \rightarrow O$ é a função de atualização de valores das variáveis em O ;
- s_0 é o estado inicial.

A rigor, as descrições em RTL devem conter máquinas de estados completamente especificadas, mas esta condição não sempre é verdadeira no caso de descrições em níveis superiores de abstração. Levando isto em consideração, a metodologia presente foi desenvolvida de forma que o fato de as FSMs serem completamente especificadas, ou não, é transparente.

Definição 2. Condição de entrada: uma condição de entrada, i , corresponde a uma expressão de avaliação com resultado Booleano que contém um par de elementos $\{v_i \in I, \text{valor de comparação}\}$.

Para fins de simplicidade, poderá ser usado, indistintamente, o termo “entrada”. O elemento ‘ v_i ’ será referido como “variável de entrada”, ou, quando possível, simplesmente como “variável”.

Definição 3. Atribuição de saída: a atribuição de saída, o , corresponde a uma expressão de designação que contém um par $\{v_o \in O, \text{valor a designar}\}$.

Para fins de simplicidade, poderá ser usado, indistintamente, o termo de “saída”. O elemento ‘ v_o ’ será referido como “variável de saída”, ou, quando possível, simplesmente como “variável”.

Definição 4. Estados equivalentes: Dadas duas máquinas, M_1 e M_2 , com $I_1 = I_2 = I$, e $O_1 = O_2 = O$, definem-se como equivalentes dois estados, $s_1 \in S_1$ e $s_2 \in S_2$, se, e somente se,

qualquer sequência de condições de entrada $\{i_0, i_1, \dots, i_n\}$, quando aplicada a s_1 e a s_2 , resulta na mesma sequência de atribuições de saída $\{o_0, o_1, \dots, o_n\}$, tanto em M_1 , quanto em M_2 . Caso a condição seja falsa, chamam-se de estados distinguíveis.

Definição 5. Máquinas equivalentes.

Dadas duas máquinas, M_1 e M_2 , com $I_1 = I_2$, e $O_1 = O_2$, é possível definir a equivalência entre elas a partir da equivalência entre seus estados. Formalmente, duas FSM, M_1 e M_2 , podem ser consideradas equivalentes se seus estados iniciais forem equivalentes. Caso contrário, chamam-se de máquinas distinguíveis (MOLITOR; MOHNKE, 2004).

No exemplo da Figura 7, somente as máquinas M_1 e M_2 são equivalentes entre elas ao produzirem sequências de saídas iguais, para sequências de entradas iguais. No sentido estrito da Definição 4, no entanto, M_3 não é equivalente com as anteriores, pois existe uma transição onde sua saída é diferente (o símbolo ' $\emptyset$ ' representa um elemento de entrada, ou de saída, indefinido para a transição onde aparecer).

Figura 7. Exemplo de FSMs equivalentes e distinguíveis, de acordo com a definição tradicional.

3 TRABALHOS CORRELATOS

O refinamento gradual de CIs digitais resulta na produção de inúmeras versões diferentes para a sua descrição, cada vez com um nível menor de abstração, que devem ser verificadas com o maior grau de exaustão possível. No contexto da aplicação de SEC para as tarefas de depuração, toda nova versão do projeto pode ser verificada através da comparação do seu comportamento com aquele de um modelo de referência global, ou com o de uma versão anterior do circuito previamente verificada e validada. De acordo com isto, o problema se reduz a como fazer tal comparação, sendo que, para tal fim, é necessário recorrer a formas de representação específicas, além de heurísticas e algoritmos bem definidos que se adequem ao nível de abstração próprio de cada etapa do projeto.

Representações intermediárias de baixo nível são comuns quando a verificação se limita, similarmente, a descrições RTL, ou ainda de maior especificidade. Dado que a checagem de equivalência formal no nível de bit, e ainda de palavra, apresenta problemas de complexidade até hoje não resolvidas, é possível encontrar trabalhos relativamente recentes nesta área. Exemplos são os apresentados em (CABODI et al., 2010) e (BAUMGARTNER et al., 2009). O primeiro visa descobrir transformações que preservam a equivalência em circuitos combinatórios no nível de porta lógica, analisando-os com algoritmos SAT sobre fórmulas CNF. No segundo, aplica-se um algoritmo de procura de invariantes (nós equivalentes) entre dois projetos sequenciais, representados por *netlists*, para determinar regiões condicionalmente equivalentes.

Para metodologias de projeto de alto nível, a escalabilidade de modelagens como as anteriores é reduzida e, portanto, é necessário adotar técnicas e ferramentas mais abstratas. Ferramentas comerciais podem ser incorporadas diretamente na verificação com SEC em alto nível, especialmente no âmbito de projeto com HLS. Em (MATHUR et al., 2009), relata-se o uso da ferramenta SLEC para checar a equivalência de módulos RTL obtidos a partir de programas escritos em C através da ferramenta Catapult; ambas as ferramentas são produtos da Calypto Design Systems (CALYPTO, 2013). Segundo os autores, vários erros resultantes da incapacidade da Catapult para interpretar ambiguidades no código C, são detectados, efetivamente, com o uso de SLEC. No entanto, esta ferramenta necessita de informações temporais adicionais, relativas à latência e vazão da implementação RTL, para conseguir realizar a comparação, ou, de outra forma, como reportado em (MAALEJ; MARTINEZ, 2008), os resultados da checagem podem ser inexatos. Outra técnica, publicada em

(SAGLAMDEMIR; SEN; DUNDAR, 2012), propõe um quadro de verificação baseado nas ferramentas Simulink, Real-Time Workshop (RTW), e HDL Coder, da Mathworks (MATHWORKS, 2013). Como mostrado na Figura 8, um modelo descrito em Simulink é traduzido para Verilog, utilizando HDL Coder para blocos predefinidos, e RTW e Catapult para blocos definidos pelo usuário, para sua posterior comparação com uma implementação RTL (escrita manualmente), através de Formality da Synopsys (SYNOPTSYS, 2013). Nesta técnica, no entanto, o fato de transformar o modelo de referência, mesmo através das ferramentas mencionadas, implica a necessidade de verificar, por sua vez, que a transformação foi feita de forma coerente com a funcionalidade pretendida.

Figura 8. Técnica de checagem com ferramentas de HLS e SEC no nível RTL.

Outras alternativas consistem da adoção de representações intermediárias e algoritmos personalizados. O autor de (HU, 2007), por exemplo, modela uma especificação em software na forma de um grafo de circuito, o qual representa instruções de C com células (como somadores, multiplexadores, e comparadores) para, assim, trazê-la ao mesmo nível de abstração do DUV. Esta ideia é ilustrada na Figura 9, a qual representa uma função escrita em C, cujos parâmetros de entrada são um inteiro *key* e um arranjo de inteiros *data*, e cuja saída é um inteiro *count*, calculado iterativamente, com valor igual ao número de elementos em *data* iguais a *key*. Após tal transformação, ambos os modelos são simulados com estímulos aleatórios para estabelecer um conjunto candidato de *cutpoints*, ou pontos de corte, definidos como pontos do modelo em software que são possivelmente equivalentes com estados do modelo em hardware. Finalmente, uma checagem de equivalência combinatória é realizada sobre tais pontos. Além de certo grau de incerteza introduzido pela estimulação aleatória, o autor relata que, dado o número de caminhos possíveis do modelo em software poder crescer exponencialmente, o cálculo dos *cutpoints* pode requerer uma quantidade impraticável de execuções da simulação.

```

int count_matches(int key, int data[7]){
    int i, count = 0;
    for (i=0; i<7; i++) {
        if (key==data[i]) count++;
    } return count; }

```

(b)

Figura 9. Parte do grafo de circuito (a) que representa uma função em C (b).

Em (HAO; RAY; XIE, 2012), grafos chamados de CCDFGs (*Clocked CDFGs*) são usados para representar o modelo de referência, em ESL, e que descreve o comportamento do DUV em RTL, o qual foi obtido com HLS. Após o mapeamento de correspondências de elementos de memória e sinais entre o CCDFG e o código RTL do DUV, efetua-se uma simulação simbólica no nível de palavra com ajuda de um solucionador de teoremas de módulo de satisfatibilidade (SMT, do inglês *Satisfiability Modulo Theory*). Similarmente, Koelbl et al. apresentam, em (KOELBL et al., 2009), uma forma especial de DFG, levantada a partir da combinação de ambos os grafos do DUV e do modelo de referência, à qual, em adição, tem se integrado um conjunto de restrições e condições secundárias, que possibilitam a checagem entre os diferentes níveis de abstração. Similar ao método em (HU, 2007), realiza-se simulação aleatória para se obter uma série de pares potencialmente equivalentes, sobre os quais, aplicam-se solucionadores no nível de bit e de palavra, determinando realmente, assim, a equivalência entre os dois grafos originais. Estes últimos três trabalhos, embora viabilizem a verificação com SEC entre ESL e RTL, possuem, como desvantagem comum, o acréscimo de dados, estruturas, e em geral, informações ao modelo de referência, o qual é uma intervenção

que aumenta a complexidade e, além disso, implica certa probabilidade de inserção de erros ao ambiente de verificação.

Presente nas propostas anteriores, a procura de pontos de comparação direta demonstra ser uma prática comum em metodologias de SEC em alto nível. Um exemplo disto é apresentado em (VASUDEVAN et al., 2008), onde grafos de transição de estados são usados para modelar ambas as descrições, ESL e RTL, e simulados simbolicamente em uma estratégia passo a passo. Para isto, são reunidas em um conjunto de pontos de comparação sequencial, todas as variáveis, cujo valor, em cada modelo, for igual para o mesmo instante de tempo. Depois, aplica-se um solucionador SAT Booleano a cada ponto de comparação para provar a equivalência global. Uma forte limitação de tal método é que a informação sobre o ciclo de comparação é extraída da implementação RTL, e não se providenciam meios para detectar um erro que fizesse com que algum valor fosse calculado no ciclo incorreto. Semelhantemente, o trabalho introduzido em (MATSUMOTO et al., 2009) faz uso de uma versão estendida de grafo de dependências de sistema (ExSDG, do inglês *Extended System Dependence Graph*), através do qual se comparam duas descrições de sistemas na linguagem SpecC. Os ExSDGs de cada sistema são simulados simbolicamente para construir, progressivamente, classes com objetos de equivalência que contêm as variáveis que se presumem equivalentes, assim como, as respectivas linhas de código onde elas foram achadas. Caso não seja possível provar a equivalência entre duas classes, então cada uma delas é expandida, de forma incremental, acrescentando a elas as linhas de código que as precedem ou sucedem. Contudo, os autores comentam que a técnica é bem sucedida, unicamente, para sistemas de tamanho relativamente pequeno. Ademais, o trabalho não demonstra ser aplicável na checagem de descrições em RTL.

Em trabalhos recentes têm-se encontrado alternativas de comparar sequências de eventos ou estados, ao invés de pontos ou estados individuais. Tal abordagem aparece em (NISHIHARA; MATSUMOTO; FUJITA, 2006), na qual o modelo de referência em alto nível, e o DUV em RTL, são representados a partir de FSM com dados (FSMD), aplicando um conjunto de regras específicas para checar a equivalência entre sequências executáveis de estados de ambas as máquinas. O método é efetivo para projetos obtidos com HLS, mas apresenta alguns aspectos limitantes. Primeiro, as duas FSMDs devem ter conjuntos de entradas e saídas idênticos; segundo, as regras para comparação entre sequências são específicas do método, e abrangem só um subconjunto de HDL consideravelmente restrito se comparado aos possíveis

comportamentos de projetos reais; terceiro, e último, o tamanho das sequências é restrito a três estados.

Bombieri et al. (BOMBIERI et al., 2007) usam sequências de eventos observadas às entradas e saídas principais do projeto. Aqui, propõe-se dividir os estados do DUV em três tipos, isto é, de entrada, computação, e saída; percorre-se a FSM que representa o DUV, seguindo as sequências que a conformam, e, dependendo do tipo de estado alcançado, operações específicas são realizadas. De um lado, no caso de estados de entrada ou saída (Figura 10a), estruturas de alto nível, como FIFOs e eventos, são a eles acrescentadas, e, desta forma, o nível de abstração do DUV é elevado de RTL a TLM. Por outro lado, estados de computação que envolvem tomadas de decisão (por exemplo, estados que formam bifurcações e laços na FSM) são agrupados e substituídos por instruções de código de alto nível como *if*, *while*, e *for* (Figura 10b); isto também permite levar o DUV ao mesmo nível do seu respectivo modelo comportamental. No entanto, o acréscimo de elementos significa um aumento de complexidade no processo de verificação. Ademais, os autores não documentam como solucionar o problema de equivalência para o código elaborado nos estados de computação.

Figura 10. Transformações realizadas em cima dos estados para elevar o nível de abstração desde RTL até TLM.

O uso de *cutpoints* pode ser aproveitado, também, para realizar SEC baseada em sequências de estados, como em (LEE et al., 2011), onde tais pontos correspondem a todos os estados que envolvem tomadas de decisão (ou seja, com mais de uma transição de saída). Em tal trabalho, um algoritmo é criado para percorrer o espaço de sequências das FSMs do DUV, e do modelo de referência, comparando-as e estabelecendo áreas comportamentais equivalentes entre as duas máquinas de forma progressiva. Porém, segundo os autores, este trabalho foi

desenvolvido para verificação após um conjunto específico de operações de HLS, e sua escalabilidade para outros tipos de refinamento é indeterminada. Adicionalmente, a quantidade de sequências extraídas a partir de tais *cutpoints* é consideravelmente grande, e dado que o algoritmo utiliza mecanismos de checagem baseados em tentativa e erro, assim como em iteração recursiva, a complexidade e eficiência da metodologia geral são questionáveis.

Os trabalhos acima focalizam na corroboração das descrições do DUV e os comportamentos especificados no modelo em alto nível. Todavia, é igualmente importante verificar que todos os aspectos exclusivos da implementação em RTL (ausentes no modelo em alto nível) estejam projetados coerentemente com relação à especificação funcional. Tais aspectos, geralmente, possuem um forte vínculo comportamental com a temporalização, sincronismo, e sequenciamento de sinais, e interfaces de entrada e saída do sistema.

Técnicas formais têm sido aplicadas na verificação destes elementos, como em (CHAKRABARTI et al., 2002), onde os diagramas temporais que descrevem, tipicamente, os protocolos do projeto, são transformados em propriedades formais descritas na linguagem RTCTL (do inglês *Real-Time Computation Tree Logic*), cuja validade, subsequentemente, é testada sobre a FSM do projeto utilizando-se um checador de propriedades customizado baseado em BDDs. A obtenção de propriedades formais a partir da especificação funcional, para sua posterior checagem com respeito ao comportamento descrito por uma FSM que representa o projeto, pode ser considerada uma técnica relativamente popular, e utilizada em trabalhos como (PENG; TAHAR, 2000) e (D'SILVA; RAMESH; SOWMYA, 2005). Ao invés de propriedades, é possível gerar, a partir da especificação, uma FSM que descreva o funcionamento do protocolo e interfaces de baixo nível de blocos de hardware, o que permite verificá-los através de SEC como em (EIRIKSSON, 1996) e (YANG et al., 2005).

Na revisão do estado da arte realizada, não foram encontrados trabalhos que tratassem a verificação integral do projeto com técnicas de SEC, considerando tanto os aspectos comportamentais de alto nível, quanto os de interfaces, sinais, e protocolos de baixo nível. A tabela 1 apresenta uma síntese do estado da arte, resumindo as principais características de cada trabalho. A primeira coluna mostra as referências aos trabalhos citados. As segunda e terceira colunas mostram, respectivamente, o nível de abstração das descrições que se objetiva verificar, e a forma interna de representação usada para seu tratamento formal. A terceira coluna corresponde ao tipo de ferramenta utilizada no trabalho, e a quarta relaciona o

elemento de comparação, se houver, utilizado na ferramenta para se realizar a verificação. A quinta coluna indica se o método faz uso de simulação com estimulação aleatória. As sexta e sétima colunas explicam a estratégia e desvantagens do trabalho, respectivamente.

Tabela 1. Relação comparativa de trabalhos correlatos com a metodologia proposta.

Trabalho	Nível de abstração	Forma de representação	Elemento de comparação	Ferramenta	Estratégia	Desvantagem
(CABODI et al., 2010)	porta lógica	CNF	vértices e arestas de grafos acíclicos dirigidos	própria	transformações que preservam a equivalência	baixo nível
(BAUMGARTNER et al., 2009)	RTL	netlist	nós equivalentes	Própria	procura de invariantes	baixo nível
(MATHUR et al., 2009)	ESL, RTL (C, Verilog)	--	--	Calypto SLEC	uso de informações de HLS	dependência forte das informações de HLS
(SAGLAMDEMIR; SEN; DUNDAR, 2012)	ESL, RTL (Simulink, C, Verilog)	--	--	Synopsys Formality	SEC	alteração do modelo HL
(HU, 2007)	ESL, RTL (C, Verilog)	grafo de circuito	cutpoints	própria	SEC	simulação, explosão
(HAO; RAY; XIE, 2012)	ESL, RTL (C, Verilog)	CCDFG	cutpoints	própria	SEC	alteração do modelo HL
(KOELBL et al., 2009)	ESL, RTL (C, Verilog)	DFG	cutpoints	própria	SAT, BDDs, SEC	simulação, alteração do modelo HL
(VASUDEVAN et al., 2008)	ESL, RTL (SystemC)	grafos de transição de estados	variáveis com mesmo valor	própria baseada em ZChaff	SAT, SEC	precisa informação temporal do RTL
(MATSUMOTO et al., 2009)	ESL (SpecC)	ExSDG	objetos possivelmente equivalentes	própria baseada em CVC	procura iterativa, SEC	somente alto nível
(NISHIHARA ; MATSUMOTO; FUJITA, 2006)	ESL, RTL (C, Verilog)	FSMD	sequências de transições entre estados	própria	SAT, BDDs, checagem de equivalência	interfaces idênticas, sequências máximas de 3 estados
(BOMBIERI et al., 2007)	TLM, RTL (SystemC)	EFSM (igual a FSMD)	sequências de eventos às interfaces	própria	transformação de estados	acréscimo de elementos ao modelo HL
(LEE et al., 2011)	ESL, RTL	FSMD	sequências de estados de decisão	própria	cobertura de caminhos de estados	restrito a operações específicas HLS
Metodologia aqui proposta	ESL, RTL (SystemC, VHDL)	FSMD	sequências de estados	própria, CVC3 para CEC de datapath	redução e comparação de sequências de estados	precisa de similaridade algorítmica entre modelo HL e DUV

O capítulo seguinte apresenta uma metodologia que visa atender tal problemática, onde o DUV é comparado com um modelo de referência em ESL, e, posteriormente, elementos obtidos em tal comparação são aproveitados para checar a validade dos comportamentos exclusivamente nele existentes.

4 VERIFICAÇÃO FUNCIONAL DE UMA IMPLEMENTAÇÃO NO NÍVEL RTL ATRAVÉS DE MÉTODOS FORMAIS DE SEC

Os capítulos anteriores ilustraram, de forma introdutória, o problema de verificar se duas FSMs que representam circuitos, descritos em níveis diferentes de abstração, correspondem ao mesmo comportamento. Como já foi sugerido, todo projeto de hardware digital em RTL pode ser dividido em duas regiões comportamentais básicas, isto é:

1. a computacional, que descreve as funções de operação e comunicação de dados contidas no respectivo modelo de referência em ESL;
2. a de protocolo de comunicação, mais ligada à síntese do circuito, detalhando com exatidão as propriedades relacionadas aos sinais de controle às interfaces, domínios de relógio, sincronismo e escalonamento de tarefas.

Consideradas estas duas regiões, então, a verificação funcional formal com SEC, para um projeto em RTL, pode ser efetuada em duas etapas, dentro de um arcabouço teórico unificado como o aqui proposto. O presente capítulo apresenta tal abordagem, primeiramente apresentando as máquinas de estados com dados envolvidas na SEC com foco na descrição RTL. Em seguida, na sub-seção seguinte, os conceitos formais para a checagem da região computacional do DUV são introduzidos. Finalmente, mostra-se como aproveitar o mesmo quadro teórico para a verificação da região temporal.

4.1 As Máquinas de Estados

Como mencionado acima, pode-se separar a descrição funcional RTL em parte computação e parte comunicação. Como indicado na Figura 11, isto pode estar contemplado no fluxo de projeto por dois movimentos: a partir de uma descrição funcional em alto nível (HL, do

Figura 11. Inclusão de comportamentos de comunicação, e de computação, no projeto em RTL.

inglês: *High-Level*), onde um processo de refinamento é realizado; e, a partir de um documento de descrição do protocolo em termos de sinais de interfaceamento efetua-se um processo de inclusão e adaptação. Assim, a descrição em RTL é mais complexa que as descrições HL e de protocolo individualmente, podendo conter sinais extras; porém, basicamente, é a união delas.

Um exemplo da diferença entre tais descrições pode ser vista na Figura 12, onde um circuito de cálculo de GCD (do inglês: *Greatest Common Divisor*) é ilustrado. Na Figura 12^a temos o pseudocódigo C do modelo em alto nível, enquanto na Figura 12b aparece a descrição do modelo de protocolo de comunicação em pseudocódigo SystemC e, na Figura 12c, a descrição RTL também em pseudocódigo SystemC. Pode-se observar neste caso que a variável *aux*, do código RTL, aparece apenas no modelo HL, e que os sinais *reset* e *load* aparecem apenas no modelo de protocolo. A variável *swap*, que não ocorre em nenhum deles, corresponde a um elemento intermediário exclusivo da implementação.

<pre> void gcd() { int x=in1, y=in2; while (y!=0) { while (x>=y) { x=x-y; aux=x; x=y; y=aux; } out=x; } } </pre> <p>(a)</p>	<pre> SC_MODULE (gcd) { void read() { if (~reset && load) x=in1; y=in2; } void write() { if (reset) out=0; else out=x; } } </pre> <p>(b)</p>	<pre> SC_MODULE (gcd) void start() { if (reset) x=0; y=0; else if (load) x=in1; y=in2; else if (swap) x=y; y=aux; else x=aux; } void calculate() { if (y!=0) { out=0; if (x>=y) swap=0; aux=x-y; else swap=1; aux=x; } else out=x; } } </pre> <p>(c)</p>
--	--	---

Figura 12. Pseudocódigo SystemC para um exemplo de implementação de calculador de GCD: (a) Modelo HL, (b) Especificação de protocolo, (c) Descrição RTL.

Especificamente, o objetivo do formalismo apresentado neste capítulo é permitir determinar se uma descrição RTL é coerente com um dado modelo de referência em ESL e um modelo de protocolo. Desta forma, daqui em diante, trabalha-se sobre três máquinas de estados finitos, M_{RTL} , M_{HL} e M_E , correspondentes a cada descrição.

Assim, como na checagem de equivalência tradicional, deseja-se observar se as duas máquinas produzem sequências de saídas iguais, para sequências de entradas iguais. Contudo, é necessário considerar os seguintes problemas específicos:

- M_{RTL} pode conter um número maior de variáveis, o que se traduz no fato de os alfabetos de entrada e saída serem diferentes das outras duas máquinas;
- M_{HL} e M_E podem conter estados que não têm um par equivalente em M_{RTL} ;
- Entradas e saídas podem aparecer na mesma ordem em ambas M_{HL} e M_{RTL} (ou em M_E e M_{RTL}), mas distribuídas ao longo de um número maior ou menor de transições entre estados.

Para se solucionar os itens acima, propõe-se, aqui, trabalhar na base de sequências de estados, ao invés de comparação de estados individuais como na checagem de equivalência tradicional. Porém, como mencionado, o princípio fundamental de equivalência, expressado nas Definições 4 e 5, deverá ser mantido, isto é, a partir do estado inicial, as saídas resultantes de uma sequência de entradas em M_{HL} (ou M_E), deverão ser as mesmas em M_{RTL} , para a mesma sequência de entradas. As próximas definições fornecem uma formalização do problema, assim como, a solução correspondente ao quadro de trabalho aqui desenvolvido.

Definição 6. Sequência de estados. Dada uma FSM, $M = \{S, I, O, \delta, \lambda, s_0\}$, chama-se sequência de estados a todo conjunto $k = \{s_0, s_1, \dots, s_{n-1}, s_n\}$, $k \in S$, de forma que $\delta(s_0, i_0) = s_1, \dots, \delta(s_{n-1}, i_{n-1}) = s_n$, com $n \geq 1$. Isto, para quaisquer condições de entrada $\{i_0, \dots, i_{n-1}\}$, com variáveis $\{v_0, \dots, v_{n-1}\} \in I$, que ativarem as correspondentes transições (assume-se que tais condições de entrada existem).

Desta forma, toda máquina M pode ser decomposta em um conjunto de sequências $K = \{k_0, \dots, k_{m-1}\}$, com $m \geq 1$. Sob o presente quadro de trabalho, toda sequência de estados, $k \in K$, deverá respeitar as seguintes condições:

- 1) a sequência k começa a partir do estado inicial de M , s_0 , e possui um número finito de estados ($n < \infty$);
- 2) seja $s_i \in S$ um estado com duas ou mais transições de saída, de forma que o fluxo a partir de tal estado depende do valor de uma variável de entrada $v_{fi} \in I$. O estado s_i pode aparecer mais de uma vez em k se as duas condições seguintes forem verdadeiras:
 - a. existe um caminho de estados e transições que faz um laço de retorno a s_i ;

- b. ao longo de tal caminho, designa-se a v_{li} um novo valor, de forma que, ao retornar a s_i , o fluxo de controle de M pode seguir através de uma transição diferente;
- 3) se a condição anterior não for satisfeita, cada transição de saída de s_i se deriva em uma sequência de estados diferente.

Na Figura 13, mostra-se um exemplo simples de FSM que ajuda a ilustrar como as sequências de estados são interpretadas de acordo com o acima estipulado. A máquina da figura contém os estados $s_0, s_i, s_x, s_y, s_w, s_n$; é possível observar que há um laço de retorno a s_i . Neste caso, ‘ v ’ corresponde a um elemento interno do código de descrição original, e, portanto, pode funcionar como variável de entrada e saída na FSM. O formato das entradas e saídas nas transições é “entrada / saída”, e o símbolo ‘ $\emptyset$ ’ representa uma entrada ou saída não especificada.

Inicialmente, v recebe o valor ‘1’, ativando a transição de s_i para s_x . Depois, v recebe o valor ‘0’, mas desta vez, ao retornar a s_i , o fluxo da FSM segue em direção a s_w . Assim, a máquina da Figura 13 estará caracterizada por uma única sequência $k=\{s_0; s_i; s_x; s_y; s_i; s_w; s_n\}$. Agora, como contraexemplo, suponha-se que a atribuição de valor “ $v_{li}=0$ ”, de s_x para s_y , não existe. Em tal caso, a máquina poderia ser decomposta nas sequências $k_0=\{s_0; s_i; s_x; s_y\}$ e $k_1=\{s_0; s_i; s_w; s_n\}$.

Figura 13. Exemplo de FSM composta por uma única sequência de estados.

A partir de uma sequência de estados, é possível identificar o comportamento descrito por ela; assim, a presente metodologia se baseia na comparação de comportamentos entre sequências de estados. Como será demonstrado nas seções seguintes, se houver comportamentos equivalentes entre as sequências de duas máquinas, então se conclui que

estas são funcionalmente equivalentes, fato que pode ser aproveitado para estabelecer se M_{RTL} representa uma implementação correta.

Como ponto de partida, definem-se, a seguir, sequências de variáveis de entrada e de saída, e sequências retraídas de variáveis, indispensáveis no presente quadro de trabalho, pois são estas sequências as que carregam o significado comportamental das sequências de estados. Mais à frente, as definições seguintes estabelecem a base para determinar se duas sequências de estados descrevem comportamentos equivalentes, o que, por sua vez, permite determinar se duas FSMs descrevem comportamentos equivalentes.

Definição 7. Sequências de variáveis de entrada. Uma sequência de variáveis de entrada, w_I , consiste de uma lista ordenada de conjuntos $\{g_0, \dots, g_i, \dots, g_p\}$, de forma que $|w_I|=p+1$, $g_i=\{v_{i0}, \dots, v_{ii}, \dots, v_{im}\}$, $|g_i|=m+1$, e $\forall v_{ii} \in I$.

Definição 8. Sequências de variáveis de saída. Uma sequência de variáveis de saída, w_O , consiste de uma lista ordenada de conjuntos $\{h_0, \dots, h_i, \dots, h_q\}$, de forma que $|w_O|=q+1$, $h_i=\{v_{0i}, \dots, v_{oi}, \dots, v_{on}\}$, $|h_i|=n+1$, e $\forall v_{oi} \in O$.

Considere-se uma FSM, M , com $I=\{a, b, c, d, e, f\}$ e $O=\{u, v, w, x, y, z\}$. Exemplos de w_I incluiriam $\{(a, b); c; d; (e, f)\}$ com $g_0=\{a, b\}$, $g_1=\{c\}$, $g_2=\{d\}$, $g_3=\{e, f\}$, ou $\{(a, b, c, d, e, f)\}$, com $g_0=\{a, b, c, d, e, f\}$. Similarmente, exemplos de w_O incluiriam $\{u; v; w; x; y; z\}$ com $h_0=\{u\}$, $h_1=\{v\}$, $h_2=\{w\}$, $h_3=\{x\}$, $h_4=\{y\}$, e $h_5=\{z\}$, ou $\{(u, v, w); (x, y, z)\}$ com $h_0=\{u, v, w\}$, e $h_1=\{x, y, z\}$.

Definição 9. Sequência retraída de variáveis. Seja w uma sequência de variáveis, indiferentemente de serem variáveis de entrada ou de saída. Define-se w' como sequência retraída de w , se as seguintes condições se cumprirem:

- 1) w' tem o mesmo conjunto de variáveis de w ;
- 2) se em w houver uma variável, v_a , que está na mesma posição ou que antecede a outra variável, v_b , sendo que $v_b \neq v_a$, então, v_b não antecede a v_a em w' ;

Como exemplo, considere-se uma sequência de entradas $w_I=\{(a, b); c; d; (e, f)\}$. De acordo com a última definição, uma sequência $w_I'=\{(a, b, c); (d, e); f\}$ seria uma sequência retraída de w_I . Pelo contrário, outra sequência $w_I''=\{(a, b, c); e; (d, f)\}$ não seria uma sequência retraída de w_I , uma vez que 'e' e 'd' estão em conjuntos diferentes, e 'e' antecede a 'd'.

Lema 1. Toda sequência w é uma sequência retraída de w .

Premissa 1. Toda máquina de estados finitos, M , é uma máquina determinística. Isto significa que M não contém nenhum estado com mais de uma saída diferente para uma mesma entrada. Assim, também não existirão em M mais de uma sequência de estados, com sequências de saídas diferentes, para uma mesma sequência de entradas.

Por conseguinte, a partir da Premissa 1, cada sequência, k , é única, e descreve um único comportamento de entrada e saída da máquina M .

4.2 Verificação da Região Computacional

A partir da Premissa 1 admite-se que o comportamento de M_{HL} está descrito na íntegra pelo seu conjunto de sequências K_{HL} e, similarmente, o de M_{RTL} pelo seu conjunto K_{RTL} . Desta forma, e sob a hipótese de para toda sequência $k_{HL} \in K_{HL}$ existir uma sequência $k_{RTL} \in K_{RTL}$ com o mesmo comportamento, é possível, então, afirmar que M_{RTL} representa uma implementação correta da especificação representada por M_{HL} . Para demonstrar que a hipótese anterior é suficiente para, de fato, se garantir a validade de M_{RTL} , apresenta-se, primeiramente, o conceito de comportamento equivalente entre duas sequências, k_{HL} e k_{RTL} .

Daqui em diante, assume-se a existência de I_{HL} e O_{HL} , e I_{RTL} e O_{RTL} , que são os conjuntos de variáveis de entrada e de saída de M_{HL} , e de M_{RTL} , respectivamente. Também, designam-se como $I_{\cap HL}$, e $O_{\cap HL}$, aos subconjuntos de variáveis de entrada e de saída, respectivamente, comuns a ambas M_{HL} e M_{RTL} .

Dado que o presente trabalho trata implementações RTL refinadas a partir de um modelo em alto nível, é necessário assumir formalmente que o conjunto de variáveis de M_{RTL} contém, ou é igual, ao de M_{HL} . Isto tem coerência com o exposto na Seção 4.1 (Figura 11).

Premissa 2. Os conjuntos de variáveis de entrada e saída de M_{RTL} contém ou são iguais aos de M_{HL} . Ou seja, $I_{RTL} \supseteq I_{HL}$ e $O_{RTL} \supseteq O_{HL}$. A partir disto, assume-se a existência dos conjuntos $I'_{HL} = I_{RTL} \setminus I_{HL}$, e $O'_{HL} = O_{RTL} \setminus O_{HL}$, de variáveis de entrada e saída, respectivamente, que pertencem exclusivamente a M_{RTL} (o operador ‘ $\setminus$ ’ simboliza exclusão). Quando possível, referir-se-á a V como o conjunto $I \cup O$. Assim, de forma a simplificar a notação, far-se-á referência aos conjuntos V_{HL} , V_{RTL} , $V_{\cap HL}$, e V'_{HL} . Assim, finalmente, estabelece-se que $V_{HL} = V_{\cap HL}$. Esta configuração é ilustrada na Figura 14, onde o círculo maior representa o conjunto V_{RTL} , o qual engloba o conjunto $V_{\cap HL}$ (que é o mesmo V_{HL}). O conjunto V'_{HL} é o espaço por fora do círculo listrado ($V_{\cap HL}$) e por dentro de V_{RTL} .

Figura 14. Representação gráfica dos conjuntos de variáveis adotados.

Definição 10. Função de redução de sequências de estados em relação a $V_{\cap HL}$. Considere-se, inicialmente, um conjunto $Q=\{S,V\}$, composto por um grupo de estados S , e um correspondente grupo de variáveis, V , que conformam as entradas e saídas dos estados em S . Assim, definem-se os conjuntos $Q_{RTL}=\{S_{RTL},V_{RTL}\}$, onde S_{RTL} são os estados de M_{RTL} , com seu correspondente grupo de variáveis V_{RTL} , e $Q_{\cap}=\{S_{\cap},V_{\cap}\}$, onde $S_{\cap}$ é um grupo de estados cujas entradas e saídas estão formadas por variáveis confinadas a $V_{\cap}$. Então, dada uma sequência k_{RTL} , define-se $R_{HL}(k_{RTL}) : Q_{RTL} \rightarrow Q_{\cap}$ como a função que reduz os estados e variáveis de entrada e saída de k_{RTL} àqueles pertencentes, unicamente, ao conjunto $Q_{\cap}$. Isto significa que $R_{HL}(k_{RTL})$ resulta em uma sequência reduzida k_{RTL}^R , e, no que diz respeito esta, $V'_{HL} = \{\emptyset\}$. As operações realizadas por R_{HL} sobre uma sequência k_{RTL} , para se obter uma sequência k_{RTL}^R , são descritas a seguir:

- 1) propagação: os valores de variáveis pertencentes a V_{RTL} , que possuem interdependências de dados com variáveis pertencentes a $V_{\cap HL}$, são propagados até estas. A sequência resultante denomina-se k_{RTL}^1 ;
- 2) remoção: remove-se de k_{RTL}^1 todo elemento pertencente a V'_{HL} , junto com seu correspondente valor, substituindo-os pelo símbolo ' $\emptyset$ '. A sequência resultante denomina-se k_{RTL}^2 ;
- 3) agrupamento: sejam s_i e $q=\{s_{i+1}, \dots, s_n\}$ um estado e uma subsequência de estados, respectivamente, englobados por k_{RTL}^2 , tal que s_i precede imediatamente a q . Sendo verdadeiras as condições seguintes:
 - a. a entrada i_i , para o estado s_i , está especificada (isto é, $i_i \neq \emptyset$);
 - b. as entradas dos estados em q não estão especificadas (isto é, $\{i_{i+1}, \dots, i_n\} = \emptyset$);

então, s_i e q são agrupados e, assim, passam a conformar um único estado, $s_{(i\dots n)}$, cuja transição de saída reúne a entrada especificada em s_i , junto com as saídas especificadas nos

estados em q , e aponta ao mesmo estado apontado por s_n . Estados ociosos (ou seja, sem entrada nem saída especificadas) são incorporados, indistintamente, ao estado que os precedem, ou sucedem. A sequência resultante é k_{RTL}^R .

A Figura 15 mostra um exemplo de aplicação da função $R_{HL}(k_{RTL}) = k_{RTL}^R$. Para o caso, tomam-se os conjuntos $V_{\cap HL} = \{a, b, c, x, y, z\}$ e $V'_{HL} = \{d, e, f, u, v, w\}$. No primeiro passo, é possível observar que os valores de 'u' e 'v' são propagados às variáveis que dependem delas, ou seja, 'a', 'b', 'x', 'z', e 'c', respectivamente. No segundo passo, todos os elementos de V'_{HL} são removidos, e, finalmente, os estados são agrupados para se obter k_{RTL}^R .

Figura 15. Exemplo de redução de uma sequência k_{RTL} até k_{RTL}^2 .

Postulado 1. Duas sequências de estados, k_1 e k_2 , apresentam o mesmo comportamento em relação a um conjunto específico de variáveis, $V=I \dot{\cup} O$, sempre que as três condições seguintes forem satisfeitas:

- 1) k_1 e k_2 têm o mesmo conjunto de variáveis de entrada, com o mesmo ordenamento, ou seja, $w_{Ik1} = w_{Ik2}$;
- 2) a sequência de variáveis de saída descrita por k_2 , w_{Ok2} , é uma sequência retraída da sequência de variáveis de saída de k_1 , w_{Ok1} ;
- 3) sejam duas entradas, i_a e i_b , e uma saída, o , com variáveis $\{v_{ia}, v_{ib}\} \in I$, e $v_o \in O$, respectivamente. Tanto em k_1 , quanto em k_2 , i_a resulta, direta ou indiretamente, na mesma atribuição de saída, o . Note-se que uma saída, o , é resultado direto de uma entrada, i_a , se

ocorrer no mesmo estado que i_a , e indireto se ocorrer em algum estado posterior, mas antes de qualquer outro estado onde aparecer uma nova entrada, i_b .

O postulado acima nos permite, então, certificar se o comportamento de uma sequência é mantida após a aplicação da função de redução.

Teorema 1. A operação de propagação da função de redução, R_{HL} , não altera o comportamento entre k_{RTL} e k^1_{RTL} , em relação a $V_{\cap HL}$.

--Prova--

Com relação à primeira condição do Postulado 1, a demonstração é automática, pois a propagação só atua na transferência de valores de variáveis, e não há mudança no sequenciamento destas. Portanto o ordenamento das sequências de variáveis de entrada é mantido.

Com relação à segunda condição do Postulado 1, a demonstração é também automática, pois a propagação só atua na transferência de valores de variáveis, e não há mudança em sequenciamento de variáveis. Portanto o ordenamento das palavras de saída é mantido, e devido ao Lema 1, $w_{Ok}^1_{RTL}$ é uma sequência retraída de w_{OkRTL} .

Com relação à terceira condição do Postulado 1, é possível distinguir dois casos, ilustrados na Figura 16, que fariam com que k^1_{RTL} tivesse um comportamento diferente de k_{RTL} . No primeiro existiria, no mínimo, uma entrada i , com $v_i \in I_{\cap HL}$, que, em k_{RTL} , ativaria uma saída o_1 , com $v_{o1} \in O_{\cap HL}$, mas que, em k^1_{RTL} , ativaria uma saída o_2 , com $v_{o2} \in O_{\cap HL}$, onde $o_1 \neq o_2$. No segundo existiria, no mínimo, uma saída o , com $v_o \in O_{\cap HL}$, que, em k_{RTL} , seria ativada por uma entrada i_1 , com $v_{i1} \in I_{\cap HL}$, mas que, em k^1_{RTL} , seria ativada por uma entrada i_2 , com $v_{i2} \in I_{\cap HL}$, diferente de i_1 . Fazendo uma análise por passos, tem-se que:

- como já foi provado que a sequência de variáveis de entrada não se altera, se o primeiro caso for causado pela propagação, significa que o valor da saída o_2 , em k^1_{RTL} , é diferente do valor de o_1 , em k_{RTL} , enquanto suas variáveis são iguais, ou seja $v_{o1} = v_{o2}$;
- como somente houve mudanças de valor para as variáveis de $V_{\cap HL}$ que são dependentes de valores de V_{RTL} , a afirmação anterior significa que, sendo o valor de o_2 diferente do valor de o_1 , e o valor de o_2 propagado a partir de alguma designação anterior “ $u = valor_u$ ”, então o valor de o_1 é, obrigatoriamente, diferente de “ $valor_u$ ”. Como $o_1 = u$, ter-se-ia $u \neq u$, demonstrando a impossibilidade de comportamentos diferentes;

Similarmente, se o segundo caso for causado pela propagação, significa que o valor da entrada i_2 , em k^1_{RTL} , é diferente do valor de i_1 , em k_{RTL} . Portanto, é possível concluir que a operação de propagação, da função de redução R_{HL} , não altera o comportamento entre k_{RTL} e k^1_{RTL} , em relação a $V_{\cap HL}$.

--Fim da prova do Teorema 1--

Figura 16. Casos hipotéticos que fariam com que a operação de propagação violasse a terceira condição do Postulado 1.

Teorema 2. A operação de remoção da função de redução, R_{HL} , não altera o comportamento entre k^1_{RTL} e k^2_{RTL} , em relação a $V_{\cap HL}$.

--Prova--

Com relação à primeira condição do Postulado 1, assume-se inicialmente que, devido à remoção, $w_{ik}^1_{RTL} \neq w_{ik}^2_{RTL}$, em relação ao conjunto $I_{\cap}$. Isto significaria que alguma entrada, i , com $v_i \in I_{\cap HL}$, foi removida ou deslocada, o que pode acontecer através de dois casos diferentes:

- caso 1: i foi removida ou deslocada na própria etapa de remoção, o que implica que esta etapa atua sobre variáveis que não são elementos de V'_{HL} , o que, por sua vez, contradiz a definição da operação de remoção;
- caso 2: i já teria sido removida ou deslocada na propagação, mas isto é incoerente com o Teorema 1.

Com relação à segunda condição do Postulado 1, seguindo o mesmo raciocínio da prova para a primeira condição, e aplicando-o às sequências de variáveis de saída, conclui-se, também, que não é possível que a operação de propagação produza uma sequência k^2_{RTL} , de forma que $w_{ok}^1_{RTL} \neq w_{ok}^2_{RTL}$. Portanto o ordenamento é mantido, e devido ao Lema 1, $w_{ok}^2_{RTL}$ é uma sequência contraída de $w_{ok}^1_{RTL}$.

Com relação à terceira condição do Postulado 1, já foi provado acima que as sequências de variáveis não se alteram. Como base para a presente prova, mostrar-se-á, primeiramente, que tanto a quantidade de estados, assim como o posicionamento das variáveis, se mantêm em k^2_{RTL} . Por absurdo, considera-se que qualquer uma das, ou ambas as situações, não ocorreu para k^2_{RTL} . Isto significa que, ao transformar k^1_{RTL} a k^2_{RTL} pode ter acontecido um dos casos seguintes: o processo de remoção eliminou algum estado, ou, a remoção modificou a posição de alguma variável de $V_{\cap HL}$. A análise de tais situações é feita, a seguir:

- caso 1: como a remoção só substitui condições e atribuições de V'_{HL} , a situação é impossível;
- caso 2: novamente, como a remoção só substitui condições e atribuições de V'_{HL} , a situação é impossível;
- Com as premissas acima, e sabendo que as sequências de variáveis se mantêm, novamente, faz-se referência aos dois casos mencionados na prova da operação de propagação, para a terceira condição, como as possíveis situações que implicariam um comportamento de k^2_{RTL} diferente do de k^1_{RTL} ;
- se o primeiro caso for causado pela remoção, isto é, os valores das variáveis de saída de $O_{\cap HL}$ forem alterados, significa que a tarefa de remoção realizou a alteração dos valores; como a remoção só substitui condições e atribuições de V'_{HL} , a situação é impossível;
- a impossibilidade do segundo caso é demonstrada de forma análoga.

Conclui-se, assim, que a operação de remoção, da função de redução R_{HL} , não altera o comportamento entre k^1_{RTL} e k^2_{RTL} , em relação a $V_{\cap HL}$.

--Fim da prova do Teorema 2--

Teorema 3. A operação de agrupamento da função de redução, R_{HL} , não altera o comportamento entre k^2_{RTL} e k^R_{RTL} , em relação a $V_{\cap HL}$.

--Prova--

Com relação à primeira condição do Postulado 1, e levando em conta a Definição 10, considere-se, sem perda de generalidade, que existe um estado s_{n+1} , em k^2_{RTL} , que segue imediatamente a subsequência q , e que contém a condição de entrada $i_{n+1} \neq \emptyset$. Suponha-se, também, que existe outra entrada $i_j \neq \{\emptyset, i_i, i_{n+1}\}$ que ocorre antes de i_i ou após i_{n+1} . Tal configuração é ilustrada na Figura 17. Com base nisto, é possível fazer uma análise por contradição, como segue:

- assume-se inicialmente que, devido ao agrupamento, $w_{\text{Ik}}^2_{\text{RTL}} \neq w_{\text{Ik}}^{\text{R}}_{\text{RTL}}$, em relação ao conjunto V_{HL} ;
- considera-se, como causa disso, que $k^{\text{R}}_{\text{RTL}}$ seja a sequência agrupada de k^2_{RTL} , onde i_j ocorre imediatamente depois de i_i , devido a qualquer um de dois casos específicos (Figura 17);

Figura 17. Configuração inicial de uma sequência k^2_{RTL} , junto aos dois casos hipotéticos que fariam com que a operação de agrupamento infringisse a primeira condição do Postulado 1.

- caso 1: houve uma inversão (entre i_j e i_i , ou entre i_{n+1} e i_j) realizada pela tarefa de agrupamento em $k^{\text{R}}_{\text{RTL}}$;
- caso 2: i_i foi agrupada com i_{n+1} em um único estado;
- em relação ao primeiro caso, porém, o agrupamento não pode inverter por definição, mostrando sua impossibilidade;
- no segundo caso, segundo a Definição 10, a função R agrupa uma única entrada com a sequência de saídas imediatamente posterior a ela. Assim, entende-se que todo elemento de entrada, imediatamente posterior a uma sequência de saídas, será unicamente agrupado, se existir, com a sequência de saídas imediatamente posterior a ele.

Portanto, não é possível que R altere a sequência de entradas de k^2_{RTL} , pois o agrupamento de estados ocorre, de forma individual, para cada elemento de entrada.

Com relação à segunda condição do Postulado 1, assume-se inicialmente que, devido ao agrupamento, $w_{\text{Ok}}^{\text{R}}_{\text{RTL}}$ não é uma sequência retraída de $w_{\text{Ok}}^2_{\text{RTL}}$, em relação ao conjunto V_{HL} . A partir disto, é possível realizar o seguinte raciocínio:

- considere-se, sem perda de generalidade, que k^2_{RTL} tem um estado s_i , com entrada i_i e saída o_i , e que, também, existe uma saída o_j no mesmo estado que o_i ou na subsequência q ;

- considere-se, adicionalmente, que k_{RTL}^R seja a sequência agrupada de k_{RTL}^2 com uma saída o_i antecedida de o_j ;
- para tal situação ocorrer em k_{RTL}^R , significaria que o_j pertenceria a uma primeira subsequência de saída, enquanto o_i pertenceria a uma segunda;
- dado que, por definição, o agrupamento não pode mudar a posição entre qualquer par de variáveis, o_j deveria ocorrer antes de o_i em k_{RTL}^2 , contradizendo, assim, a suposição inicial.

Com relação à terceira condição do Postulado 1, já foi comprovado que a sequência k_{RTL}^R mantém o mesmo ordenamento para as entradas e que w_{OK}^R é uma sequência retraída de w_{OK}^2 . Fazendo uma análise similar à das provas anteriores, é possível chegar a uma contradição, como segue:

- assumindo que k_{RTL}^R seja a sequência agrupada de k_{RTL}^2 , a qual tem um estado $s_{(in)}$ com entrada i_i e saídas $\{o_{i+1}, o_{i+2}, \dots, o_n\}$, poderiam, então, ocorrer dois casos segundo os quais a terceira condição seria infringida;
- caso 1: existe uma saída $o_{ik} \in \{o_{i1}, o_{i2}, \dots, o_{in}\}$, em k_{RTL}^R , que não pertença a q em k_{RTL}^2 ;
- caso 2: existe uma saída $o_{ik} \in q$, em k_{RTL}^2 , que não pertence a $\{o_{i1}, o_{i2}, \dots, o_{in}\}$ em k_{RTL}^R ;
- segundo a Definição 10, se $o_{ik} \notin q$, o agrupamento resulta em $\{o_{i1}, o_{i2}, \dots, o_{in}\} \setminus o_{ik}$, o que contradiz o caso 1;
- segundo a Definição 10, se $o_{ik} \in q$, o agrupamento resulta em $\{o_{i1}, o_{i2}, \dots, o_{ik}, \dots, o_{in}\}$ o que contradiz o caso 2.

Assim, é possível concluir que a operação de agrupamento, da função de redução R_{HL} , não possui condições para violar nenhuma das condições estabelecidas no Postulado 1, e por conseguinte, em relação ao conjunto $V_{\cap HL}$, k_{RTL}^2 e k_{RTL}^R descrevem o mesmo comportamento.

--Fim da prova do Teorema 3--

Lema 2. Se uma sequência k_A apresenta o mesmo comportamento que uma sequência k_B em relação a suas variáveis V_{AB} , e esta sequência k_B apresenta o mesmo comportamento que uma sequência k_C em relação a suas variáveis V_{BC} , então a sequência k_A apresenta o mesmo comportamento que uma sequência k_C em relação a suas variáveis V_{AC} , sempre que $V_{AB} = V_{BC} = V_{AC}$.

A partir dos teoremas anteriores, e do Lema 2 acima, é possível fazer uma primeira conclusão formal, exposta através do Corolário 1, a seguir.

Corolário 1. A sequência k_{RTL} apresenta o mesmo comportamento que sua correspondente sequência reduzida, k_{RTL}^R , em relação ao conjunto $V_{\cap HL}$.

Ao se realizar qualquer das operações da função de redução, chegamos a uma descrição com menos elementos que a anterior, enquanto o comportamento em relação ao conjunto $V_{\cap HL}$ é mantido. Como alternativa de formalização da condição acima, pode-se definir o conceito de cobertura funcional de uma sequência para outra em relação a $V_{\cap HL}$ da maneira descrita em (CASTRO MARQUEZ; STRUM; CHAU, 2013).

O Corolário 1 indica que para a verificação do comportamento de um modelo RTL, quando o foco é no seu comportamento em relação às variáveis de $V_{\cap HL}$, basta verificar o comportamento simplificado do modelo reduzido. Somando-se isto ao fato de que o modelo reduzido contém as mesmas variáveis do modelo HL, então fica clara a possibilidade de se comparar sequências correspondentes como forma de checagem direta de equivalência. Entretanto, as sequências devem apresentar os mesmos símbolos e estados (Seção 2.3.3.3) para que o conceito de equivalência possa ser aplicado.

Adicionalmente, neste trabalho, entende-se que o código da descrição em alto nível pode incluir um conjunto de expressões de inicialização que não dependem de lógica nenhuma. Em termos de máquinas de estados, isto se traduz para um estado inicial, s_0 , sem entrada especificada. Considerando isto, junto ao acima mencionado, fica estabelecida a premissa seguinte.

Premissa 3. As sequências no conjunto K_{HL} contêm um estado s_n , sem entrada especificada, só se s_n é o estado inicial. Também, as sequências no conjunto K_{HL} contêm um estado s_n , sem saída especificada, só se o estado seguinte, s_{n+1} , tiver sua entrada especificada. Finalmente, assume-se que as sequências em K_{HL} não contêm estados ociosos.

Isto é ilustrado graficamente na Figura 18, a partir de quatro estados, s_{HL0} , s_{HL1} , s_{HL2} , e s_{HL3} ; s_{HL0} é o estado inicial. À esquerda, mostram-se dois exemplos de sequências que podem aparecer em qualquer conjunto K_{HL} . À direita, mostram-se dois exemplos de sequências, sobre os quais, assume-se que não são contidas por nenhum conjunto K_{HL} .

Figura 18. Exemplos de sequências de estados que podem existir (à esquerda), ou não (à direita), em qualquer conjunto K_{HL} .

A Premissa 3 é necessária para facilitar o gerenciamento da verificação. No presente trabalho, assume-se que o código fonte das descrições HL pode conter um conjunto inicial de expressões de inicialização, que se traduzem em um estado inicial, sem condição de entrada, mas que contém um conjunto de atribuições de saída. Aceita-se, também, que as sequências $k_{HL} \in K_{HL}$ são mínimas, e somente as sequências $k_{RTL} \in K_{RTL}$ precisam ser reduzidas (Entende-se, por sequência mínima, aquela que resultaria inalterada ao aplicar a função de redução sobre esta).

Esta premissa, apesar de se apresentar como uma restrição, de fato não afeta a generalidade do modelo, uma vez que, para qualquer sequência k_{HL} que não atenda a premissa acima, um processo semelhante ao do agrupamento pode ser aplicado como exposto no teorema seguinte, sem modificar o comportamento. Desta forma, a premissa é estabelecida.

Agora, dado que o objetivo é comparar k_{HL} e k_{RTL} , e assim, verificar se seus comportamentos são equivalentes segundo a Definição 5 (note-se que as sequências de estados são um tipo especial de FSM), torna-se necessário provar que todo k_{RTL}^R apresenta as mesmas características que qualquer sequência, k_{HL} , definidas na Premissa 3.

Teorema 4. A operação de agrupamento, da função de redução R_{HL} , implica que na sequência resultante k_{RTL}^R , para cada estado, haverá um elemento de entrada especificado, a menos que seja o estado inicial.

--Prova--

Assume-se, inicialmente, que existe uma sequência k_{RTL}^R com, no mínimo, um estado, s_{RTL} , cuja condição de entrada não está especificada. A partir disto é possível fazer a análise seguinte:

- a suposição anterior significaria que s_{RTL} era um estado em k_{RTL}^2 que não foi agrupado com nenhum outro estado com entrada especificada, para, assim, se obter um novo estado com entrada e saída especificadas;

- se s_{RTL} pertencer a uma subsequência de estados que, em k^2_{RTL} , engloba ao estado inicial, a suposição acima faria sentido, pois não existiria nenhum outro estado com entrada especificada precedendo-o, o qual, de fato, é coerente com a Premissa 3;
- caso contrário, chega-se a um absurdo, pois, segundo o definido para a função R_{HL} , a operação de agrupamento é realizada entre todo estado com entrada especificada, e a subsequência completa de estados, cuja entrada não esteja especificada, que lhe proceda imediatamente.

--Fim da prova do Teorema 4--

De acordo com o Teorema 4, poder-se-ia chegar à conclusão que, ao serem equivalentes, duas sequências k_{HL} e k^R_{RTL} descrevem o mesmo comportamento em relação ao conjunto de variáveis $V_{\cap HL}$. Isto, a princípio, poderia servir de base para, eventualmente, demonstrar a coerência de uma máquina, M_{RTL} , com respeito a outra, M_{HL} . Contudo, tal estratégia poderia apenas ser considerada um tipo fraco de verificação, pois caso estas duas sequências não forem equivalentes, nada se pode afirmar sobre a coerência de comportamentos entre a sequência original, k_{RTL} , e a da especificação, k_{HL} .

O ideal seria ter garantia de equivalência funcional entre duas descrições em ESL e RTL quaisquer, desde que a segunda seja corretamente projetada, isto é, se o processo de checagem de equivalência resultar negativo, ter-se a certeza de que o comportamento RTL contenha erros. Dadas as muitas nuances existentes em um projeto RTL, tal checagem de equivalência é de alta complexidade.

No presente quadro de trabalho, será adotada uma perspectiva que permita um tipo mais robusto de verificação, isto é, uma técnica com maior fortaleza que o tipo fraco de verificação acima descrito. Desta forma, procura-se estabelecer uma classe de sequências k_{RTL} para as quais pode-se determinar se possui, ou não, um comportamento igual ao de uma sequência k_{HL} , em relação ao conjunto $V_{\cap HL}$. Para realizar tal demonstração, o conjunto de todas as possíveis sequências será simplificado, ou seja, somente aquelas k_{RTL} , que sejam sequências estendidas de k_{HL} , serão consideradas. O conceito de sequência estendida de estados define-se, a seguir.

Definição 11. Sequência estendida de estados entre M_{HL} e M_{RTL} . Seja $k_{HL} \in K_{HL}$ uma sequência com n estados, e $k_{RTL} \in K_{RTL}$ uma sequência com m estados, de forma que $m \geq n$. Assumindo verdadeiras as condições seguintes, chama-se a k_{RTL} uma sequência estendida de estados de k_{HL} :

- além dos elementos pertencentes a V'_{HL} , k_{RTL} contém, mantendo ordem e quantidade, todas as entradas e saídas também contidas em k_{HL} ;
- para todo estado $s_{HL} \in k_{HL}$, e toda entrada $i_{HL} \in I_{\cap HL}$, existe um estado $s_{RTL} \in k_{RTL}$, e uma entrada $i_{RTL} \in I_{\cap HL}$, de forma que $i_{HL} = i_{RTL}$, e o resultado de $\lambda(s_{HL}, i_{HL})$, sempre que definido, é consistente com o resultado de $\lambda(s_{RTL}, i_{RTL})$, isto, seja de forma direta, ou indireta. Dito de forma intuitiva, a função de saída, para os elementos pertencentes a $V_{\cap HL}$, mantém a causalidade com as respectivas entradas em ambas k_{HL} e k_{RTL} .

No exemplo da Figura 19, mostram-se três sequências k_{HL} , k_{RTL1} , e k_{RTL2} , de forma que $I_{\cap HL} = \{i_1, i_2\}$, $O_{\cap HL} = \{o_1, o_2\}$, $I'_{HL} = \{i_3\}$ e $O'_{HL} = \{o_3, o_4\}$. Assim, k_{RTL1} é uma sequência estendida de k_{HL} , pois todas as saídas ocorrem como resultado das mesmas entradas em ambas sequências. Note-se que, em k_{RTL1} , a segunda ocorrência de ‘ o_2 ’ é o resultado indireto de $\lambda(s_{22}, i_2)$, uma vez que a entrada do estado ‘ s_{23} ’ é $i_3 \notin I_{\cap HL}$, o que, mesmo assim, não contradiz a relação de causalidade de k_{HL} . A sequência k_{RTL2} , pelo contrário, não pode ser considerada uma sequência estendida de k_{HL} , pois a segunda ocorrência de ‘ o_2 ’ é o resultado direto de $i_1 \in I_{\cap HL}$.

As condições para que qualquer sequência k_{RTL} seja uma estendida de k_{HL} pode parecer um tanto restritiva, porém é bem razoável, se considerarmos que estamos tratando de casos de refinamento sucessivo a partir da descrição HL. Sugere-se ao projetista do modelo RTL utilizar tal abordagem como um guia, para que em uma etapa posterior, a verificação formal seja facilitada.

Figura 19. Exemplos seqüências que satisfazem (k_{RTL1}), ou não (k_{RTL2}), a definição de seqüência estendida.

Teorema 5. Sejam duas seqüências, k_{HL} e k_{RTL} , de forma que a última é uma seqüência estendida da primeira. Então, afirma-se que as duas seqüências apresentam o mesmo comportamento em relação ao conjunto de variáveis $V_{\cap HL}$ (ou seja, as duas satisfazem as três condições do Postulado 1).

--Prova--

Inicialmente, far-se-á uma hipótese que contradiga o enunciado do teorema, ou seja, se k_{RTL} é uma seqüência estendida de k_{HL} , isto não implica que seu comportamento, em relação ao conjunto de variáveis $V_{\cap HL}$, seja o mesmo, afirmação que significaria que:

- k_{RTL} não satisfaz, obrigatoriamente, uma ou várias das condições expressadas no Postulado 1;
- isto, no entanto, é incoerente em relação à definição disponibilizada para seqüências estendidas;
- de um lado, a primeira condição de tal definição implica, necessariamente, que toda seqüência estendida, k_{RTL} , possui todos os elementos, tanto de entrada, quanto de saída, e na mesma ordem e quantidade que a respectiva seqüência k_{HL} , o que, de fato, significa que k_{RTL} possui as mesmas seqüências de entradas e de saídas descritas por k_{HL} ;
- isto satisfaz, invariavelmente, as duas primeiras condições do Postulado 1;

- de outro lado, como expressado na segunda condição da definição de sequências estendidas, se k_{RTL} for uma sequência estendida de k_{HL} , as duas resultam na mesma saída, se a mesma entrada for aplicada;
- isto satisfaz, invariavelmente, a terceira condição do Postulado 1.

Portanto, todo par de sequências, k_{RTL} e k_{HL} , sendo k_{RTL} uma sequência estendida de k_{HL} , possui o mesmo comportamento em relação ao conjunto $V_{\cap HL}$.

--Fim da Prova do Teorema 5--

Baseando-se no Teorema 5, e considerando o Postulado 1, assim como o Lema 1, é possível fazer uma segunda conclusão formal, expressada através do Corolário 2, a seguir.

Corolário 2. Seja k_{RTL} uma sequência estendida de outra sequência k_{HL} , e seja k_{RTL}^R uma sequência reduzida a partir de k_{RTL} . Então, k_{HL} e k_{RTL}^R apresentam o mesmo comportamento em relação ao conjunto de variáveis $V_{\cap HL}$.

A partir do Corolário 2, e Levando em conta o Teorema 4, junto às Premissas 1, 2 e 3, é possível, finalmente, estabelecer formalmente a equivalência entre as sequências reduzidas de M_{RTL} e as sequências de M_{HL} . O teorema seguinte fornece tal formalização.

Teorema 6. Afirma-se que uma sequência k_{HL} é equivalente a toda sequência k_{RTL}^R , se esta for reduzida a partir de uma k_{RTL} que, por sua vez, seja uma sequência estendida de k_{HL} .

--Prova--

Sob a hipótese de k_{HL} e k_{RTL}^R não serem equivalentes, seria necessário aceitar, pelo menos, uma das seguintes duas condições:

- k_{RTL}^R e k_{RTL} não têm o mesmo comportamento em relação ao conjunto $V_{\cap HL}$;
- k_{RTL} e k_{HL} não têm o mesmo comportamento em relação ao conjunto $V_{\cap HL}$;
- a primeira condição seria uma contradição dos Teoremas 1, 2, e 3 (Corolário 1), onde se estabelece que a função de redução R sempre criará uma sequência k_{RTL}^R que tem o mesmo comportamento que a k_{RTL} correspondente;
- a segunda condição seria uma contradição do Teorema 5, onde se estabelece que toda sequência k_{RTL} tem o mesmo comportamento que uma sequência k_{HL} , sempre que a primeira seja uma sequência estendida da segunda.

Assim, fica demonstrado que não há condições para que, dada uma sequência estendida, k_{RTL} , de uma outra sequência, k_{HL} , a correspondente sequência reduzida, k_{RTL}^R , não seja equivalente com k_{HL} .

--Fim da prova do Teorema 6--

Este teorema formaliza, então, as bases para que a SEC possa ser aplicada ao modelo RTL, bastando que se faça a checagem entre as sequências originais k_{HL} e as sequências reduzidas k_{RTL}^R . A noção de equivalência entre as sequências de estados pode ser estendida para máquinas inteiras, como demonstrada no seguinte teorema.

Teorema 7. Duas FSM, M_{HL} e M_{RTL} , com respectivos conjuntos de sequências de estados, K_{HL} e K_{RTL} , possuem o mesmo comportamento, em relação ao conjunto $V_{\cap HL}$, se as duas condições seguintes forem verdadeiras:

- 1) para toda $k_{HL} \in K_{HL}$, existe uma $k_{RTL1} \in K_{RTL}$, de forma que k_{RTL1} é uma sequência estendida de k_{HL} , e, por conseguinte, a correspondente sequência reduzida, k_{RTL1}^R , é equivalente a k_{HL} ;
- 2) não existe nenhuma sequência $k_{RTL2} \in K_{RTL}$, com a mesma sequência de entradas que k_{RTL1} (i.e. $w_{IRTL1} = w_{IRTL2}$), mas que não seja uma sequência estendida de k_{HL} .

--Prova--

Considere-se inicialmente que, em relação ao conjunto $V_{\cap HL}$, M_{HL} e M_{RTL} descrevem comportamentos equivalentes apesar do fato de uma das duas condições do teorema não ser verdadeira. Assim, tem-se que:

- se unicamente a primeira condição for verdadeira, chegar-se-ia imediatamente a uma contradição, pois M_{RTL} não seria determinística, o que se opõe à Premissa 1;
- se unicamente a segunda condição for verdadeira, significaria que existe, no mínimo, uma sequência, k_{HL} , para a qual não existe nenhuma sequência a ela equivalente no conjunto de sequências reduzidas K_{RTL}^R ;
- a partir do anterior, é possível identificar dois casos diferentes que fariam com que o estabelecido no teorema não se mantivesse:
 - caso 1: existe uma $k_{RTL1} \in K_{RTL}$ que é uma sequência estendida de k_{HL} , mas a correspondente sequência reduzida, k_{RTL1}^R , não é equivalente a k_{HL} ;

- caso 2: não existe nenhuma $k_{RTL} \in K_{RTL}$ que seja uma sequência estendida de k_{HL} ;
- o primeiro caso é impossível, pois contradiz claramente o Teorema 6;
- o segundo caso implicaria, na realidade, que as duas máquinas não possuem um comportamento equivalente, em relação a $V_{\cap HL}$;

--Fim da prova do Teorema 7--

Assim, restringindo-se ao conjunto $V_{\cap HL}$, fica demonstrado que as duas condições do teorema da cobertura funcional formal são necessárias, e ao mesmo tempo válidas, para garantir que M_{RTL} possua o mesmo comportamento que M_{HL} . Em tal caso, pode-se dizer que M_{RTL} representa uma implementação correta de um modelo em alto nível, representado por M_{HL} , uma vez que M_{RTL} reconhece todas as sequências de entrada, reconhecidas por M_{HL} , e, também, produz as mesmas sequências de saída que M_{HL} , para as mesmas sequências de entrada.

4.3 Verificação do Protocolo de Comunicação

No formalismo anterior, é possível observar que, ao se fazer uso da função R , o subconjunto V'_{HL} foi omitido, visando verificar, unicamente, os aspectos do sistema mais ligados à computação de dados, descritos no modelo de referência, em alto nível. Este subconjunto consiste de variáveis que expressam o comportamento de aspectos exclusivos da implementação RTL, tais como, temporização (por exemplo, *clock*, latência, vazão), protocolo (por exemplo, portas de *reset*, *ready*, *done*, *rw*), e sinais ou elementos de memória internos adicionais, resultantes do processo de refinamento, seja manual, ou com ferramentas de HLS (COUSSY; MORAWIEC, 2008).

De forma a possibilitar a verificação de tais comportamentos, ausentes no modelo ESL, estabelece-se aqui, como ponto de partida, o mesmo conjunto de sequências de estados da seção anterior, K_{RTL} . Neste caso, como referência, será utilizado um conjunto de sequências, K_E , extraído de uma FSM, M_E , a qual, por sua vez, terá sido obtida diretamente a partir do documento de especificação funcional do DUV. Vale a pena notar que, então, a metodologia aqui apresentada visa atender o problema da verificação da região temporal em ambientes de desenvolvimento onde tal documento de especificação existe, e no qual, todos os aspectos específicos de uma eventual implementação RTL têm sido detalhados. Dado este requisito, garante-se que tais aspectos podem ser organizados, e reunidos em uma representação formal, ou seja, na máquina M_E .

Para se realizar a comparação, e assim, a checagem entre M_E e M_{RTL} , todo o formalismo já introduzido pode ser aplicado realizando as revisões e adaptações oportunas, como segue:

- neste caso, trabalha-se com os novos conjuntos de sequências K_E e K_{RTL} , e de variáveis V_E , V_{RTL} , $V_{\cap E} = V_E \cap V_{RTL}$, e $V_E = V_{RTL} \setminus V_{\cap}$. Note-se que, analogamente, $V_{RTL} \supseteq V_{\cap E}$, e $V_{\cap E} = V_E$;
- a função de redução pode operar diretamente sobre os conjuntos anteriores, mas, para fins de clareza, adota-se a notação k^1_{RTL} , k^2_{RTL} , e, finalmente, k^R_{RTL} , para as diferentes sequências por ela geradas (as demonstrações e provas formais são igualmente válidas para o presente caso);
- o Teorema 7 é válido, da mesma maneira, para as máquinas M_E e M_{RTL} , e, portanto, se as duas condições nele estabelecidas forem verdadeiras, é possível afirmar que M_{RTL} descreve o mesmo comportamento que M_E , em relação ao conjunto $V_{\cap E}$.

A seguir, descrevemos o conjunto de formalismos relevantes para a SEC da descrição de M_E , a partir da teoria exposta na seção anterior. Provas formais dos teoremas serão omitidas uma vez que são semelhantes às aquelas já apresentadas, e as conclusões são análogas.

Definição 12. Função de redução de sequências de estados em relação a $V_{\cap E}$. Dada uma sequência k_{RTL} , define-se $R_E(k_{RTL})$: $\{ V_{RTL} \} \rightarrow \{ V_{\cap E} \}$ como a função que reduz as variáveis de entrada e saída de k_{RTL} a variáveis pertencentes, unicamente, ao conjunto $V_{\cap E}$. Isto significa que $R_E(k_{RTL})$ resulta em uma sequência reduzida k^R_{RTL} , e, no que diz respeito esta, $V'_E = \{\emptyset\}$.

Corolário 3. A sequência k_{RTL} apresenta o mesmo comportamento que sua correspondente sequência reduzida, k^R_{RTL} , em relação ao conjunto $V_{\cap E}$.

Premissa 4. As sequências no conjunto K_E contêm um estado s_n , sem entrada especificada, só se s_n é o estado inicial. Também, as sequências no conjunto K_E contêm um estado s_n , sem saída especificada, só se o estado seguinte, s_{n+1} , tiver sua entrada especificada. Finalmente, assume-se que as sequências em K_E não contêm estados ociosos.

Esta premissa indica que a descrição do protocolo de comunicação restringe-se à dependência e sequenciamento da sinalização na interface do modelo RTL, enquanto aspectos temporais como a quantidade de ciclos decorridos entre sinais não são considerados.

Definição 13. Sequência estendida de estados entre M_{HL} e M_E . Seja $k_E \in K_E$ uma sequência com n estados, e $k_{RTL} \in K_{RTL}$ uma sequência com m estados, de forma que $m \geq n$. Assumindo verdadeiras as mesmas condições seguintes, chama-se a k_{RTL} uma sequência estendida de estados de k_E :

- além dos elementos pertencentes a V'_E , k_{RTL} contém, mantendo ordem e quantidade, todas as entradas e saídas também contidas em k_{HL} ;
- para todo estado $s_E \in k_E$, e toda entrada $i_E \in I_{\cap E}$, existe um estado $s_{RTL} \in k_E$, e uma entrada $i_{RTL} \in I_{\cap E}$, de forma que $i_E = i_{RTL}$, e o resultado de $\lambda(s_E, i_E)$, sempre que definido, é consistente com o resultado de $\lambda(s_{RTL}, i_{RTL})$, isto, seja de forma direta, ou indireta. Dito de forma intuitiva, a função de saída, para os elementos pertencentes a $V_{\cap E}$, mantém a causalidade com as respectivas entradas em ambas k_E e k_{RTL} .

Teorema 8. Uma sequência k_E é equivalente a toda sequência k^R_{RTL} , se esta for reduzida a partir de uma k_{RTL} que, por sua vez, seja uma sequência estendida de k_E .

Teorema 9. Duas FSM, M_E e M_{RTL} , com respectivos conjuntos de sequências de estados, K_E e K_{RTL} , possuem o mesmo comportamento, em relação ao conjunto $V_{\cap E}$, se as duas condições seguintes forem verdadeiras:

- 1) para toda $k_E \in K_E$, existe uma $k_{RTL1} \in K_{RTL}$, de forma que k_{RTL1} é uma sequência estendida de k_E , e, por conseguinte, a correspondente sequência reduzida, k^R_{RTL1} , é equivalente a k_E ;
- 2) não existe nenhuma sequência $k_{RTL2} \in K_{RTL}$, com a mesma sequência de entradas que k_{RTL1} (i.e. $w_{IRTL1} = w_{IRTL2}$), mas que não seja uma sequência estendida de k_E .

Fica estabelecido então, pelos dois teoremas anteriores, o arcabouço pelo qual a SEC entre máquinas M_E e M_{RTL} pode ser aplicada. Para tal a equivalência entre sequências correspondentes k_E e k^R_{RTL} é verificada.

5 METODOLOGIA, ALGORITMOS E IMPLEMENTAÇÃO

De acordo como o quadro formal do capítulo anterior, é possível concluir que, se as condições dos Teorema 7 e Teorema 9, da Seção 4.3, forem satisfeitas para M_{RTL} , quando comparada tanto com M_{HL} , quanto com M_E , então M_{RTL} corresponde à representação de um circuito correto. O correspondente arcabouço metodológico, para a verificação funcional integral de uma descrição em RTL, é mostrado, de forma global, na Figura 20.

Figura 20. Diagrama de fluxo da metodologia global de verificação funcional formal baseada em SEC entre sequências de estados.

Como ilustrado, o primeiro passo consiste na obtenção de formas de representação intermediárias para o DUV, o modelo de referência, e para o comportamento temporal da especificação funcional. Para este trabalho, adotou-se uma modelagem especificamente na forma de FSMD, em virtude dos seguintes motivos:

- a representação de sistemas digitais através de FSMD possui menor complexidade que outros tipos menos abstratos de máquinas de estados, no que diz respeito tanto à sua construção, quanto à análise;

- o formalismo do Capítulo 4 é adaptável a todo tipo de FSM, sempre que suas entradas e saídas possam ser tratadas de forma atômica para checar diretamente a equivalência entre elas, quando necessário, e isto inclui FSMDs;
- fazendo uso das ferramentas apropriadas, modelos descritos em ESL, ou em RTL, podem ser transformados e manipulados eficientemente na forma de FSMDs.

Posterior à geração das máquinas (assunto a ser detalhado na Seção 5.5), as respectivas sequências de estados são obtidas, e comparadas, considerando as condições dos Teorema 7 e Teorema 9. Para se realizar a comparação, computam-se os dois conjuntos de sequências reduzidas, K_{RTL}^R e $K_{RTL}^{R'}$, e checa-se sua equivalência aos conjuntos K_{HL} e K_E , respectivamente. As seções seguintes detalham tais procedimentos, e os ilustram com o exemplo de um calculador de GCD, tanto para os aspectos computacionais, quanto para os de protocolo.

5.1 Decomposição de FSMDs em Sequências de Estados

Os princípios para a obtenção do conjunto $K=\{k_0, k_1, \dots, k_n\}$, para qualquer M , foram introduzidos na Seção 0, através da Definição 6. Visando sistematizar a construção de K , e com fundamento nas condições estabelecidas em tal definição, estabelecem-se, a seguir, as regras específicas para a exploração e síntese de sequências de estados:

- regra 1: toda sequência começa a partir do estado inicial, s_0 , e possui um número finito de estados;
- regra 2: estados com m transições de saída (chamados, aqui, de estados de controle), permitem que sequências sigam por m bifurcações, gerando m sequências de estados (a menos que a regra 4 indique o contrário);
- regra 3: uma sequência não pode conter estados repetidos, a menos que seja seu último estado (ou a menos que a regra 4 indique o contrário);

As regras 1 a 3 permitem, a princípio, explorar o espaço completo de estados alcançáveis de qualquer FSMD. O objetivo da regra 4 seguinte é, ao substituir as regras 2 e 3, o de aperfeiçoar o mecanismo de comparação entre sequências, reduzindo o número de estados a comparar.

- regra 4: a regra 2 é descartada se, para um estado de controle, s_c , existir uma variável de entrada, $v_{ic} \in I$, cujo valor, quando s_c é alcançado, ativa uma das suas transições de saída, sendo esta uma transição não percorrida ainda, continuando a sequência. A regra 3 é

descartada se existir um caminho de estados e transições, a partir do estado que será adicionado de novo à sequência, até um estado de controle, s_c , sendo este um estado para o qual a regra 2 já foi descartada.

A utilização das regras é ilustrada na Figura 21. À esquerda, somente as regras 1 a 3 são consideradas para decompor a máquina mostrada nas sequências k_0 , k_1 , k_2 , e k_3 . À direita, a regra 4 já é considerada, e, em vista das designações e condições no valor das variáveis ‘ c ’ e ‘ x ’, o resultado é uma única sequência, fazendo com que a comparação possa se reduzir a menos da metade dos estados.

O algoritmo global que constrói o conjunto de sequências, indiferentemente de se tratar de M_{HL} , M_E , ou M_{RTL} , é apresentado no diagrama de fluxo da Figura 22^a. O algoritmo funciona de maneira recursiva, realizando uma varredura dos estados e acrescentando-os, quando

Figura 21. Sequências construídas em desconsideração (a), ou em consideração (b) da regra 4 para exploração de estados.

corresponder, a cada $k \in K$. Suas operações básicas, enumeradas na figura, são detalhadas a seguir:

- 1) a cada recursão, toma-se como parâmetro um conjunto K que, inicialmente, é composto por uma única sequência, que por sua vez, é composta pelo estado s_0 ;
- 2) laço iterativo para examinar e estender cada $k \in K$;
- 3) inserem-se na lista PROXIMOS todos os próximos estados ao último estado em k ;

Figura 22. Algoritmos desenvolvidos para construção dos conjuntos de seqüências de estados (a), e para checagem de equivalência entre estas (b).

- 4) caso o último estado em k for um nó terminal, prossegue-se com a seqüência seguinte;
- 5) laço iterativo para cada estado s em $PROXIMOS$;
- 6) se o estado s tiver uma transição de saída, continua-se a partir do item 10;

- 7) caso s tiver mais de uma transição de saída, avalia-se a regra 4 para se determinar se a regra 2 deve ser descartada. Devido ao fato de as FSMDs serem percorridas de forma simbólica, é possível que várias transições de saída possam ser ativadas no mesmo momento, caso em que o algoritmo considera uma possibilidade de cada vez (isto será ilustrado na Seção 5.3);
- 8) se a anterior avaliação resultar falsa (a regra 2 será considerada), o estado s é anexado à sequência k ;
- 9) cria-se um número de cópias de k igual ao número de transições de saída em s , de forma que cada nova sequência aponta a um, e somente um próximo estado diferente; prossegue-se com o item 13;
- 10) avalia-se se o estado s já foi adicionado previamente à sequência k atual;
- 11) caso verdadeiro, avalia-se se a regra 3 deve ser descartada, e, se esta decisão for negativa, a iteração do item 2 continua com a sequência seguinte;
- 12) o estado s é anexado à sequência k ;
- 13) inicia-se uma nova recursão tomando como parâmetro o novo conjunto K .

O algoritmo termina quando o último estado de todas as sequências for um estado final, ou, basicamente, de acordo com as regras 1 a 4, se não houver mais critérios para aumentar as sequências. Coerentemente com o presente arcabouço metodológico, após a obtenção dos conjuntos de sequências para duas FSMDs, prosseguir-se-á com a comparação de tais conjuntos, e assim, estabelecer sua equivalência funcional.

No pior dos casos, o algoritmo não encontrará condições para ignorar as regras 2 e 3, de forma que o maior número de sequências será produzido. Neste caso, a complexidade corresponde àquela da busca por profundidade para um grafo direcionado, ou seja, $O(\text{MAX}(n, e))$, onde n é o número de estados, e e o número de transições (AHO; HOPCROFT; ULLMAN, 1974).

5.2 Checagem de Equivalência entre Sequências de Estados

O algoritmo que reúne a função de redução (Definição 10), junto com a checagem das regras de comportamento equivalente do Postulado 1, é apresentado na Figura 22b; o diagrama se baseia nos conjuntos K_{HL} e K_{RTL} , mas aplica-se, sem modificações, entre K_E e K_{RTL} . A intenção geral deste algoritmo é a de obter dois conjuntos de sequências, com alfabetos iguais, para, essencialmente, verificar se a máquina M_{RTL} descreve o comportamento especificado por M_{HL} (ou por M_E).

Os procedimentos do algoritmo de SEC da Figura 22b são descritos, a seguir:

- 1) laço iterativo para cada sequência $k_{HL} \in K_{HL}$;
- 2) laço iterativo para cada sequência $k_{RTL} \in K_{RTL}$;
- 3) geração da sequência k_{RTL}^R , reduzida a partir de k_{RTL} ;
- 4) laço iterativo para cada entrada $i_{HL} \in I_{HL}$, de acordo com o ordenamento estabelecido pela sequência de variáveis, $w_{I_{HL}}$, da atual sequência de estados k_{HL} ;
- 5) laço iterativo para cada entrada $i_{RTL}^R \in I_{RTL}$, de acordo com o ordenamento estabelecido pela sequência de variáveis, $w_{I_{RTL}^R}$, da atual sequência de estados k_{RTL}^R ;
- 6) checa-se se, para o ciclo atual de iteração, $i_{HL} = i_{RTL}^R$;
- 7) se a avaliação anterior resultar verdadeira, checa-se se i_{HL} é a entrada do último estado em k_{HL} ; se, por sua vez, o resultado desta última avaliação resultar verdadeiro, procede-se ao item 10, caso contrário, o algoritmo avança para a próxima entrada, i_{HL} (item 4);
- 8) se a avaliação do item 6 resultar falsa, checa-se se i_{RTL}^R é a entrada do último estado em k_{RTL}^R ; se, por sua vez, o resultado desta última avaliação resultar falso, o algoritmo avança para a próxima entrada, i_{RTL}^R (item 5);
- 9) caso a avaliação anterior seja verdadeira, determina-se que k_{HL} e k_{RTL}^R não são equivalentes, pois suas sequências de entradas são diferentes; neste caso, o algoritmo procede ao próximo k_{RTL}^R , no item 2 (a seta correspondente não se mostra na figura, para fins de claridade do diagrama);
- 10) laço iterativo para cada saída $o_{HL} \in O_{HL}$, de acordo com o ordenamento estabelecido pela sequência de variáveis, $w_{O_{HL}}$, da atual sequência de estados k_{HL} ;
- 11) laço iterativo para cada saída $o_{RTL}^R \in O_{RTL}$, de acordo com o ordenamento estabelecido pela sequência de variáveis, $w_{O_{RTL}^R}$, da atual sequência de estados k_{RTL}^R ;
- 12) checa-se se, para o ciclo atual de iteração, $o_{HL} = o_{RTL}^R$;
- 13) se a avaliação anterior resultar verdadeira, checa-se se tanto o_{HL} , quanto o_{RTL}^R , são o resultado da aplicação de entradas equivalentes, tanto em k_{HL} , quanto em k_{RTL}^R , para o ciclo atual de iteração;
- 14) se a avaliação anterior seja verdadeira, checa-se se o_{HL} é a saída do último estado em k_{RTL}^R ; se, por sua vez, o resultado desta última avaliação resultar falso, o algoritmo avança para a próxima saída, o_{HL} (item 10);
- 15) se alguma das avaliações dos itens 12 ou 13 resultar em falso, checa-se se o_{RTL}^R é a saída do último estado em k_{RTL}^R ; se, por sua vez, o resultado desta última avaliação resultar falso, o algoritmo avança para a próxima saída, o_{RTL}^R (item 11), ou, caso resultar

verdadeiro, determina-se que k_{HL} e k_{RTL}^R não são equivalentes, pois suas sequências de saída são diferentes, ou suas saídas são produto de entradas não equivalentes;

- 16) finalmente, se o resultado da avaliação do item 14 resultar verdadeira, significa que k_{HL} e k_{RTL}^R são equivalentes, e o algoritmo avança um ciclo de iteração para a próxima sequência k_{HL} , no item 1.

Se, no final do algoritmo, uma sequência k_{RTL}^R equivalente a toda k_{HL} tiver sido achada, significa que a máquina M_{RTL} , de fato, descreve o mesmo comportamento que M_{HL} , em relação ao conjunto de variáveis $V_{\cap}$. Em tal caso, então, é possível afirmar que o DUV representado por M_{RTL} implementa corretamente o comportamento do modelo de referência representado por M_{HL} .

A complexidade deste algoritmo, no pior caso, é $O(\text{MAX}(n_{HL}n_{RTL}p_{HL}p_{RTL}, n_{HL}n_{RTL}q_{HL}q_{RTL}))$, onde n_{HL} é o número de sequências em K_{HL} , n_{RTL} é o número de sequências em K_{RTL} , p_{HL} é o número de variáveis de entrada em w_{IHL} , p_{RTL} é o número de variáveis de entrada em w_{IRTL} , q_{HL} é o número de variáveis de saída em w_{OHL} , e q_{RTL} é o número de variáveis de saída em w_{ORTL} . Esta complexidade é estabelecida a partir da maior quantidade de vezes que é necessário executar os itens 5 e 11, no algoritmo da Figura 22b.

5.3 Verificação Funcional de um Projeto de Calculador de Máximo Divisor Comum

Com fins ilustrativos, o arcabouço conceitual e metodológico acima introduzido é aplicado, a seguir, a um exemplo de circuito que calcula o máximo divisor comum (GCD, do inglês *Greatest Common Divisor*) entre dois inteiros. Primeiro, apresenta-se a checagem da região computacional, e em seguida, a da região de protocolo de comunicação.

5.3.1 Checagem do Projeto de GCD com Respeito a um Modelo em Alto Nível

O código VHDL que descreve os processos do GCD é mostrado na Figura 23. Uma versão puramente comportamental, em alto nível, é apresentada na Figura 23^a, enquanto a correspondente implementação em RTL, na Figura 23b. Nesta última, é possível identificar, além das portas, condições e operações de cômputo, outros sinais adicionais de temporização e inicialização (*clk* e *rst*), assim como, variáveis internas de dados e controle (*temp_x*, *state*).

As representações na forma de FSMd para o calculador de GCD em alto nível (M_{HL}), e para o em RTL (M_{RTL}) são mostradas nas Figuras 24^a e 24b, respectivamente. Ao examinar e comparar visualmente as duas figuras, percebe-se que seus alfabetos e comportamento temporal são diferentes, e que não existe um mapeamento de equivalência claro dos estados

de M_{HL} para os de M_{RTL} . Após a geração dos modelos, os algoritmos de geração e comparação de sequências podem ser aplicados sobre eles diretamente. Os passos seguidos na exploração de estados de M_{HL} são descritos, a seguir, indicando em cada etapa o estado das sequências:

<pre> process(in1, in2) begin x := in1; y := in2; out <= 0; if (x /= y) then if (x < y) then y := y - x; else x := x - y; end if; else out <= x; end if; end process; </pre> <p style="text-align: center;">(a)</p>	<pre> process(clk, rst) begin if (clk'event and clk='1') then if (rst='1') then out <= "0000"; state := '0'; else case state is when '0' => if (go_i='1') then x <= in1; y <= in2; state := '1'; else state := '0'; end if; when '1' => temp_x := x - y; state := '2'; when '2' => if (x /= y) then if (x < y) then y <= -temp_x; else x <= temp_x; end if; state := '1'; else out <= x; state := '0'; end if; end case; end if; end if; end process; </pre> <p style="text-align: center;">(b)</p>
--	---

Figura 23. Código VHDL dos processos de um calculador de GCD em alto nível (a) e em RTL (b)

- o algoritmo inicia a partir do estado inicial, a_0 ($K_{HL}=\{k_{HL0}\}$, $k_{HL0}=\{a_0\}$);
- o próximo estado é a_1 , que possui duas transições de saída, e, uma vez que as variáveis que compõem as entradas de ambas as transições receberam valores novos, a regra 2 é descartada. Note-se que pode-se optar por qualquer uma das transições; assume-se que o algoritmo escolhe ativar a transição que volta para a_0 ($K_{HL}=\{k_{HL0}\}$, $k_{HL0}=\{a_0, a_1\}$);
- o próximo estado é a_0 , que é adicionado à sequência atual. A seguir, o próximo estado é a_1 novamente, mas, dado que não há novos valores para as variáveis de entrada, a sequência é terminada, e o estado não é anexado. Isto faz com que a regra 2 não possa ser

descartada, e uma nova sequência será criada a partir da anterior ($K_{HL} = \{k_{HL0}, k_{HL1}\}$, $k_{HL0} = \{a_0, a_1, a_0\}$, $k_{HL1} = \{a_0, a_1\}$);

- o próximo estado, para k_{HL1} , é a_2 , o qual é adicionado à sequência. De novo, qualquer uma das transições de saída de a_2 pode ser ativada, mas ainda não houve valores novos designados às variáveis das entradas, o que faz com que mais uma sequência seja criada a partir de k_{HL1} . Assume-se que o algoritmo percorre a transição com etiqueta (1) na Figura 24^a ($K_{HL} = \{k_{HL0}, k_{HL1}, k_{HL2}\}$, $k_{HL0} = \{a_0, a_1, a_0\}$, $k_{HL1} = \{a_0, a_1, a_2\}$, $k_{HL2} = \{a_0, a_1, a_2\}$);

Figura 24. Representações FSMD para os códigos do GCD em alto nível (a), e RTL (b).

- as variáveis das entradas do estado a_2 receberam valores novos, portanto a regra 3 é descartada, e o estado a_1 é adicionado, de novo, a k_{HL1} ($K_{HL}=\{k_{HL0}, k_{HL1}, k_{HL2}\}$, $k_{HL0}=\{a_0, a_1, a_0\}$, $k_{HL1}=\{a_0, a_1, a_2, a_1\}$, $k_{HL2}=\{a_0, a_1, a_2\}$);
- os próximos estados a partir de a_1 são a_0 e a_2 . A transição para a_2 já foi considerada em k_{HL1} , e dado que não haverá novas mudanças nas variáveis de entrada, adota-se a transição para a_0 , o que finaliza a sequência ($K_{HL}=\{k_{HL0}, k_{HL1}, k_{HL2}\}$, $k_{HL0}=\{a_0, a_1, a_0\}$, $k_{HL1}=\{a_0, a_1, a_2, a_1, a_0\}$, $k_{HL2}=\{a_0, a_1, a_2\}$);
- a sequência k_{HL2} é completada de forma análoga a k_{HL1} , mas seguindo a transição com etiqueta (2) ($K_{HL}=\{k_{HL0}, k_{HL1}, k_{HL2}\}$, $k_{HL0}=\{a_0, a_1, a_0\}$, $k_{HL1}=\{a_0, a_1, a_2, a_1, a_0\}$, $k_{HL2}=\{a_0, a_1, a_2, a_1, a_0\}$);

Assim, obtêm-se, para cada máquina, as sequências ilustradas na Figura 25, com as etiquetas numéricas, provenientes da Figura 24, sinalizando alternativas tomadas quando houver mais de uma transição entre dois estados. Observa-se que M_{HL} acaba sendo descrita por três sequências, k_{HL0} , k_{HL1} , e k_{HL2} , e M_{RTL} pelas sequências k_{RTL0} a k_{RTL3} . Desta forma, para se determinar se M_{RTL} descreve o mesmo comportamento que M_{HL} , é necessário achar, na

Figura 25. Sequências resultantes para a FSM do calculador de GCD em alto nível (a), e em RTL (b).

primeira, uma sequência estendida (Figura 10) de cada sequência obtida para a segunda.

Aplicando o algoritmo de comparação, é possível verificar que k_{RTL1} , k_{RTL2} , e k_{RTL3} , são sequências estendidas de k_{HL0} , k_{HL1} , e k_{HL2} , respectivamente. A seguir, será mostrado o raciocínio do algoritmo para o caso das sequências k_{HL1} e k_{RTL2} :

- para começar, as sequências de entradas e saídas, para ambas k_{HL1} e k_{RTL2} , são ordenadas de maneira explícita, em sentido vertical, na Figura 26; neste caso, têm-se $V_{\cap HL} = \{in1, in2, x, y, out\}$, e $V'_{HL} = \{rst, go, state, temp_x\}$, e, em tal figura, é possível observar em destaque as expressões formadas por elementos pertencentes a este último conjunto;
- como mostrado na Figura 27, a primeira operação da função de redução realiza a propagação de valores, de forma que a variável ‘y’ recebe o valor de ‘temp_x’, o que resulta na sequência k^1_{RTL2} ;
- a segunda operação remove todos os elementos pertencentes a V'_{HL} , e, portanto, na sequência k^2_{RTL3} , na Figura 27, tais elementos são removidos, sendo substituídos pelo símbolo ‘ $\emptyset$ ’ quando não estiverem acompanhados de elementos pertencentes a $V_{\cap HL}$;
- por último, todas as entradas são agrupadas com as sequências de saídas que lhes sucedem imediatamente, obtendo-se, assim, a sequência k^R_{RTL3} que é equivalente com k_{HL3} .

k_{HL1}		
	i_{HL0}	o_{HL0}
1.	$\emptyset$	$x = in1, y = in2, out = 0$
2.	$x \neq y$	$\emptyset$
3.	$x < y$	$y = y - x$
4.	$x == y$	$out = x$

k_{RTL2}		
	i_{RTL2}	o_{RTL2}
1.	$rst == 1$	$state=0$, $out=0$
2.	$rst == 0$	$\emptyset$
3.	$state == 0$	$\emptyset$
4.	$go == 1$	$x=in1, y=in2$, $state=1$
5.	$rst == 0$	$\emptyset$
6.	$state == 1$	$temp_x=x-y$, $state=2$
7.	$rst == 0$	$\emptyset$
8.	$state == 2$	$\emptyset$

k_{RTL2}		
	i_{RTL2}	o_{RTL2}
9.	$x \neq y$	$state=1$
10.	$x < y$	$y = -temp_x$
11.	$rst == 0$	$\emptyset$
12.	$state == 1$	$temp_x=x-y$, $state=2$
13.	$rst == 0$	$\emptyset$
14.	$state == 2$	$\emptyset$
15.	$x == y$	$out=x$, $state=0$

Figura 26. Sequências de entradas e saídas para as sequências k_{HL1} e k_{RTL2} .

	i_{RTL2}	o_{RTL2}
1.	rst == 1	state=0, out=0
2.	rst == 0	$\emptyset$
3.	state == 0	$\emptyset$
4.	go == 1	x=in1, y=in2, state=1
5.	rst == 0	$\emptyset$
6.	state == 1	temp_x=x-y, state=2
7.	rst == 0	$\emptyset$
8.	state == 2	$\emptyset$

	i_{RTL2}	o_{RTL2}
9.	x /= y	state=1
10.	x < y	y= -(x-y)
11.	rst == 0	$\emptyset$
12.	state == 1	temp_x=x-y, state=2
13.	rst == 0	$\emptyset$
14.	state == 2	$\emptyset$
15.	x == y	out=x, state=0

	i_{RTL2}	o_{RTL2}
1.	$\emptyset$	out=0
2.	$\emptyset$	$\emptyset$
3.	$\emptyset$	$\emptyset$
4.	$\emptyset$	x=in1, y=in2
5.	$\emptyset$	$\emptyset$
6.	$\emptyset$	$\emptyset$
7.	$\emptyset$	$\emptyset$
8.	$\emptyset$	$\emptyset$

	i_{RTL2}	o_{RTL2}
9.	x /= y	$\emptyset$
10.	x < y	y= -(x-y)
11.	$\emptyset$	$\emptyset$
12.	$\emptyset$	$\emptyset$
13.	$\emptyset$	$\emptyset$
14.	$\emptyset$	$\emptyset$
15.	x == y	out=x

	i_{HL0}	o_{HL0}
1.	$\emptyset$	x = in1, y = in2, out=0
2.	x /= y	$\emptyset$
3.	x < y	y = y-x
4.	x == y	out = x

Figura 27. Operações de propagação, remoção, e agrupamento, realizadas pela função R_{HL} .

5.3.2 Checagem do Comportamento do Protocolo de Comunicação do Projeto de GCD

Como exposto no Capítulo 4, os aspectos referentes a comportamentos de temporização, sincronismo, e, de forma geral, do protocolo de comunicação da implementação em RTL, devem ser verificados por comparação com uma máquina de estados que represente tal funcionalidade. No fluxo do projeto de verificação apresentado neste trabalho, a máquina utilizada como fonte de referência para esta tarefa deve ser sintetizada manualmente a partir da documentação que especifica todos os detalhes da região temporal da implementação.

Esta seção ilustra a checagem de equivalência baseada em sequências de estados entre M_E , e M_{RTL} , para a mesma implementação em RTL do GCD anterior. Suponha-se, assim, que o documento de especificação funcional de tal projeto descreve a configuração, e correlação entre os sinais de entrada e saída, da forma ilustrada no diagrama de formas de onda que aparece na Figura 28.

Figura 28. Formas de onda da especificação do protocolo para o calculador de GCD.

A partir do diagrama da Figura 28 é possível estabelecer os seguintes requisitos para garantir o correto funcionamento do circuito:

- o sinal *go* somente terá efeito se o sinal *rst* for igual a zero;
- as portas de entrada somente podem ser lidas se o sinal *go* for igual a ‘1’;
- a porta de saída, *out*, somente pode ser escrita se o sinal *go* for igual a ‘1’;
- quando o sinal *rst* for igual a ‘1’, o valor de *out* deve equivaler a zero, e, generalizando, o sinal *rst* deve equivaler a zero em todo ciclo ao longo da computação.

Desta forma, o diagrama e as condições anteriores podem ser sintetizados na FSMd da Figura 29^a, a qual pode ser decomposta nas sequências de estados da Figura 29b. Neste caso, têm-se os conjuntos $V_{\cap E} = \{rst, go, in1, in2, out\}$, e $V'_E = \{state, x, y, temp_x\}$. Nota-se que para este exemplo, as variáveis *x* e *y* não são consideradas dentro do espaço $V_{\cap E}$, pois elas não fazem parte da interface de protocolo da implementação. A Figura 30 mostra em detalhe as

Figura 29. Máquina de estados, M_E , que especifica o comportamento temporal do circuito de cálculo de GCD.

seqüências de entradas e saídas correspondentes às duas seqüências de estados obtidas anteriormente. Obedecendo os passos do algoritmo de checagem, e em concordância com a função de redução, R_E , a Figura 31 ilustra as diferentes seqüência geradas por esta função, até se chegar à seqüência k^R_{RTL1} , que é equivalente à seqüência k_{E1} . Na seqüência k_{RTL1} , aparecem destacados todos aqueles elementos pertencentes a V'_E . Por fim, uma vez que não há variáveis a propagar, neste exemplo, as seqüências k_{RTL1} e k^1_{RTL1} são idênticas.

k_{E0}			k_{E1}		
	i_{E0}	o_{E0}		i_{E0}	o_{E0}
1.	rst==1	out = 0	1.	rst==1	out = 0
2.	rst==0	$\emptyset$	2.	rst==0	$\emptyset$
3.	go==0	$\emptyset$	3.	go==1	x=in1, y=in2
			4.	rst==0	out = x

Figura 30. Seqüências de entradas e saídas para M_E .

k_{RTL1}			k^1_{RTL1}		
	i_{RTL1}	o_{RTL1}		i_{RTL1}	o_{RTL1}
1.	rst == 1	state=0, out=0	1.	rst == 1	state=0, out=0
2.	rst == 0	$\emptyset$	2.	rst == 0	$\emptyset$
3.	state == 0	$\emptyset$	3.	state == 0	$\emptyset$
4.	go == 1	x=in1, y=in2, state=1	4.	go == 1	x=in1, y=in2, state=1
5.	rst == 0	$\emptyset$	5.	rst == 0	$\emptyset$
6.	state == 1	temp_x=x-y, state=2	6.	state == 1	temp_x=x-y, state=2
7.	rst == 0	$\emptyset$	7.	rst == 0	$\emptyset$
8.	state == 2	$\emptyset$	8.	state == 2	$\emptyset$
9.	x == y	out=x, state=0	9.	x == y	out=x, state=0

k^2_{RTL1}			k^R_{RTL1}		
	i_{RTL1}	o_{RTL1}		i_{RTL1}	o_{RTL1}
1.	rst == 1	out=0	1.	rst == 1	out=0
2.	rst == 0	$\emptyset$	2.	rst == 0	$\emptyset$
3.	$\emptyset$	$\emptyset$	3.	go == 1	x=in1, y=in2
4.	go == 1	x=in1, y=in2	4.	rst == 0	out=x
5.	rst == 0	$\emptyset$			
6.	$\emptyset$	$\emptyset$			
7.	rst == 0	$\emptyset$			
8.	$\emptyset$	$\emptyset$			
9.	$\emptyset$	out=x			

Figura 31. Obtenção da seqüência k^R_{RTL1} , equivalente a k_{E1} , através da função R_E .

5.4 Extensão a Máquinas de Estados que Representam Processos Concorrentes

Dentro do presente quadro metodológico, considera-se que as máquinas de estados podem apresentar uma topologia concorrente e intrínseca de, praticamente, todo projeto digital. De acordo com isto, entende-se que uma FSMd estará composta por uma ou várias submáquinas concorrentes que se comunicam entre si.

De forma geral, os conceitos e métodos das sessões anteriores são aplicáveis a este caso, sempre que as considerações necessárias forem tomadas. Especificamente, este trabalho trata os aspectos de escalonamento, sensibilidade, comunicação, e indeterminismo entre processos, a partir dos quais os seguintes critérios foram estabelecidos:

- tanto a sensibilidade estática (isto é, com lista de sensibilidade), quanto a dinâmica (através de eventos), são consideradas. Isto permite tratar comportamentos concorrentes que podem ocorrer, também, nos modelos de referência;
- ao se percorrer as máquinas para obter as sequências de estados, submáquinas que receberem notificação de ativação, seja através de sensibilidade estática, ou dinâmica, são inseridas em uma lista chamada EXECUTAVEIS, enquanto aquelas que alcançarem uma instrução de espera (*wait*), ou que chegarem ao seu estado final, são inseridas na lista PENDENTES (note-se que, apesar de os nomes das listas sugerirem a ocorrência de simulação, as FSMs são, na realidade, percorridas utilizando métodos simbólicos);
- embora um circuito descrito de forma apropriada não deva conter comportamentos não determinísticos entre processos, a presente metodologia considera a possibilidade de casos quando, no mesmo instante, apareçam dois ou mais submáquinas para entrarem em execução. Em tal caso, todas as possíveis transições entre tais máquinas são consideradas para a construção das sequências de estados;

Um exemplo de decomposição em sequências de estados para uma FSM que representa três processos concorrentes é apresentado na Figura 32, de forma que cada processo é modelado por uma submáquina diferente; o exemplo possui uma sintaxe em estilo VHDL, e, para fins de simplicidade, não se incluem condições de entrada. É possível observar que os processos representados pelas máquinas M_0 e M_1 possuem sensibilidade estática, respectivamente, aos sinais 'clk' e 'a'. Adicionalmente, M_0 é sensível a mudanças em 'c', e M_2 a mudanças em 'b'. Inicialmente, o percurso é feito a partir do estado s_{00} , em M_0 , e, ao alcançar s_{02} , a instrução '*wait*' implica que a máquina entre na lista PENDENTES; M_1 e M_2 são colocadas em EXECUTAVEIS, previamente, dadas as designações às variáveis 'a' e 'b'. Tal situação dá origem às sequências k_0 e k_1 , pois, em vista do indeterminismo, a exploração de estados pode continuar, tanto através de s_{10} , em M_1 , como através de s_{20} , em M_2 . Um novo indeterminismo aparece ao se adicionar s_{22} a k_1 , uma vez que, nesse instante, ambas M_0 e M_1 estão em EXECUTAVEIS.

Figura 32. Três sequências de estados, k_0 , k_1 , e k_2 , obtidas a partir de três submáquinas concorrentes, M_0 , M_1 , e M_2 .

Pode-se perceber, então, que a única diferença, relacionada com o tratamento de máquinas que modelam processos concorrentes, encontra-se no mecanismo de decomposição destas em suas respectivas sequências de estados. Após a obtenção de tais sequências, o algoritmo de comparação pode ser aplicado diretamente, sem modificações.

5.5 Implementação e Ferramentas

Visando sistematizar a metodologia proposta, e assim permitir seu aproveitamento em descrições RTL reais, foi desenvolvida uma ferramenta que reúne os algoritmos das seções anteriores, e cuja arquitetura geral é mostrada na Figura 33. A ferramenta destina-se a descrições de hardware escritas em VHDL, ou em SystemC, e integra várias aplicações preexistentes dentro de um *script* global escrito em linguagem Python (PYTHON, 2013). Os blocos na figura, de bordas arredondadas, sinalizam arquivos ou dados, enquanto os retangulares, com linha dupla, representam operações realizadas sobre os anteriores.

Figura 33. Arquitetura representada através de blocos correspondente à ferramenta desenvolvida.

O *script* recebe, inicialmente, as árvores sintáticas abstratas (ASTs, do inglês: *Abstract Syntactic Trees*) de ambos o DUV e o modelo de referência. No caso de descrições em VHDL, o analisador gramatical VAUL (VAUL, 2013) foi utilizado, e, no caso de SystemC, aproveitou-se a ferramenta desenvolvida em (PATIÑO ALVAREZ, 2005), que permite gerar uma base de dados, a partir da qual, por sua vez, é possível se obter a AST para sistemas descritos na base de tais bibliotecas. Na presente implementação, o DUV e o modelo de referência precisam estar descritos na mesma linguagem (por exemplo, não é permitido um projeto RTL em VHDL, cujo respectivo modelo em alto nível esteja descrito em SystemC).

Após a geração das ASTs, o *script* obtém, a partir destas, as correspondentes representações na forma de FSMs; como exceção, M_E deve ser escrita manualmente a partir do documento de especificação funcional. Em seguida, o algoritmo de extração de sequências constrói os conjuntos K_{HL} , K_{RTL} , e K_E , os quais são processados pela função da comparação. Esta contém um analisador de dependências entre as variáveis ao longo de cada sequência; dado que as

sequências de estados apresentam um encadeamento linear, a análise pôde ser implementada com relativa simplicidade, através de um algoritmo de varredura e comparação que executa, no pior dos casos, $n \times (n+1)/2$ iterações, onde n é o número de estados da sequência. O objetivo de tal análise é o de determinar possíveis discordâncias entre as sequências de duas máquinas diferentes, quanto ao ordenamento das variáveis, porém que não impliquem a não-equivalência comportamental total destas. Desta forma, a função de comparação consegue estabelecer cenários de equivalência parcial entre sequências, como mostrado na Figura 34, onde a sequência k_{RTL1} , apesar de desrespeitar a definição de sequência estendida (pelo fato de a saída “ $x=1$ ” ser resultado indireto da condição “ $a=1$ ”), realiza uma designação final correta à variável ‘ b ’. Já no caso da sequência k_{RTL2} , o comportamento é evidentemente diferente, pois a variável ‘ b ’ terá um valor diferente daquele em k_{HL} . É necessário notar que a equivalência comportamental parcial não é um critério suficiente para considerar a validade de uma máquina M_{RTL} ; esta é uma característica incorporada à implementação para permitir, ao final da verificação, se fazer uma avaliação e tomar as medidas de depuração necessárias.

Figura 34. Dois casos em que um ordenamento diferente produz comportamentos parcialmente (k_{RTL1}) e totalmente inequivalentes (k_{RTL2}).

O passo final consiste em percorrer e comparar os conjuntos de sequências usando o solucionador CVC3 (CVC3, 2013). Esta ferramenta consiste de um conjunto de bibliotecas em C++ para prova de teoremas do tipo SMT, que possibilita realizar uma checagem de equivalência combinatória em um nível de abstração superior a outras técnicas como SAT ou ATPG. Utilizando a interface da linguagem Python, foi possível programar métodos para comprovar diretamente a equivalência entre as expressões que compõem as entradas e saídas das FSMDs. A Figura 35 ilustra um exemplo de checagem de equivalência entre entradas e saídas de descrições em alto nível, e em RTL. O exemplo mostra a sintaxe básica de CVC3, com alguns dos tipos de operadores, tipos de dados e expressões que integram a linguagem da

ferramenta; nos casos considerados, o resultado da checagem, feita com a instrução “*query*”, seria verdadeiro.

```

I_HL:  BOOLEAN = FORALL (x:INT): x <= 0 OR x > y ;
I_RTL: BOOLEAN = FORALL (x:INT): x < 0 OR x = 0 OR x > y ;
QUERY I_HL = I_RTL;

```

```

z: BITVECTOR(8);
ASSERT z = 0hexff;
O_HL:  z/4;
O_RTL: z<<2;
QUERY I_HL = I_RTL;

```

Figura 35. Exemplo do uso de CVC3 para checagem da igualdade de duas expressões.

5.6 Aplicabilidade, Observações, e Restrições da Metodologia de Checagem de Equivalência baseada em Sequências

Algumas considerações são feitas a seguir, que precisam ser levadas em conta na eventual implantação desta metodologia em um ambiente de verificação funcional. Um primeiro aspecto é a necessidade de o DUV ser algoritmicamente concordante com o modelo de referência. Assim, por exemplo, não é possível verificar, com os algoritmos apresentados, a validade de uma implementação RTL de calculador de GCD que se baseia em subtração, se o respectivo modelo de referência realizar o cálculo de GCD por divisão e módulo. Esta é uma limitação relacionada com o nível de integração alcançado entre as ferramentas, mas que é passível de se contornar incorporando sofisticções em versões futuras.

Também vale a pena notar que a checagem de sintaxe das descrições em HDL depende exclusivamente dos analisadores gramaticais adotados, e, portanto, não é possível garantir, além da abrangência de tais analisadores, que o código verificado esteja livre de erros (não funcionais) que impeçam sua eventual compilação em ferramentas de síntese RTL.

Finalmente, assume-se que as correspondências de variáveis, sinais, e elementos de memória entre o modelo de referência e o DUV já são conhecidas. De forma geral, espera-se

que tais correspondências sejam registradas e documentadas, por um lado, no caso de projetos produzidos por refinamento manual, e do outro, informadas automaticamente no caso da aplicação de ferramentas de síntese lógica. Existem trabalhos que relatam técnicas de mapeamento que possibilitam checar a equivalência entre dois circuitos sequenciais, como em (YOSHIDA; FUJITA, 2009), onde variáveis potencialmente equivalentes são detectadas por simulação, ou em (MOLITOR; MOHNKE, 2004), onde algoritmos formais estabelecem as correspondências entre *latches* de circuitos sequenciais. O problema de correspondência entre elementos de descrições de circuitos diferentes não é tratado nesta tese por ser considerado além do escopo do presente trabalho.

6 RESULTADOS

Os capítulos anteriores introduziram o embasamento teórico e prático para verificar a funcionalidade de projetos em RTL, com respeito a um modelo em ESL, baseando-se na comparação das suas sequências de estados.

A seguir, é apresentada uma avaliação qualitativa e quantitativa da aplicação da teoria do Capítulo 4, e as técnicas do Capítulo 5, a módulos de hardware utilizados no desenvolvimento de CIs digitais reais. Primeiramente, os blocos utilizados como caso de estudo são apresentados, e, a seguir, os dados resultantes do uso das ferramentas com tais blocos são detalhados, e analisados.

6.1 Casos de Estudo

Nas experiências realizadas para avaliar a metodologia proposta, e dado que as ferramentas desenvolvidas aceitam as duas HDLs, foram utilizados exemplos de projetos escritos tanto em VHDL, quanto em SystemC. Estes aparecem listados abaixo, junto a descrições de suas principais funções e características. A rigor, não há diferenças conceituais no uso de uma linguagem e outra, portanto este consistiu apenas em uma questão de testar a ferramenta neste aspecto.

6.1.1 Blocos de Hardware descritos em VHDL

Objetivou-se, inicialmente, avaliar o desempenho das ferramentas e, adquirir uma noção da escalabilidade da metodologia de verificação. Para isto, as ferramentas foram testadas em diferentes tipos de circuitos descritos em RTL, que foram obtidos da biblioteca de núcleos de hardware disponível em (OPENCORES, 2013):

- **CRC** (*Cyclic Redundancy Checker*). Os checadores de redundância cíclica são amplamente utilizados na comprovação da existência de erros em sistemas de comunicações. Na transmissão, os dados são embaralhados, ao longo de um registrador de deslocamento, que implementa um polinômio específico através de uma interface lógica com portas XOR. Na recepção, os dados são desembaralhados, e se não houver erros, o mesmo registrador conterá zeros, unicamente. Este CRC utiliza o polinômio $x^{16}+x^{15}+x^2+1$, portanto, o registrador é inicializado com o valor “1000000000000101”.
- **CORDIC** (*Coordinate Rotation Digital Computer*). Implementa um método para calcular funções elementares usando hardware mínimo, como, por exemplo, registradores de

deslocamento, somadores, e comparadores. Especificamente, esta implementação contempla o cálculo do seno e cosseno, e realiza a conversão de coordenadas polares a retangulares.

- MDCT (*Modified Discrete Cosine Transform*). Este bloco realiza uma transformada bidimensional para sistemas de compressão de imagem como o JPEG, e está baseado no uso de decomposição da imagem em filas e colunas, e dois correspondentes blocos de DCT unidimensionais conectados através de uma memória RAM como matriz de transposição.
- AES (*Advanced Encryption Standard*). Esta é uma implementação do algoritmo AES-128, para criptografia de dados e chaves de encriptação em blocos de 128 bits. O processamento dos dados é feito sobre matrizes de 4x4 bytes, em dez laços que efetuam, a cada vez, as operações *SubBytes*, *ShiftRows*, *MixColumns*, e *AddRoundKey*. A primeira insere não-linearidade à encriptação, substituindo cada byte por seu correspondente em uma tabela predefinida, e é implementada com uma RAM de 4 kilobytes. A segunda roteia as filas das matrizes, enquanto a terceira mistura as suas colunas de acordo a uma ordem predefinida. A última operação combina (através de uma porta XOR) uma chave a cada matriz que é obtida iterando-se a chave de encriptação geral.

Adicionalmente, o exemplo de GCD da Seção 5.3 foi incorporado neste conjunto de experiências.

De posse apenas dos modelos RTL, descritos acima, foi definido um método para se obter as versões comportamentais em alto nível correspondentes. Consistiu de se realizar a abstração dos primeiros de forma manual. Assim, um subconjunto das expressões e instruções, típicas dos projetos em RTL, foram substituídas por outras que possuem um estilo mais abstrato. Por exemplo, quando possível, sinais foram trocadas por variáveis, e designações do tipo “ $a \leq b(4 \text{ to } 8)$ ” foram substituídas por “ $a := b \text{ srl } 4$ ”.

As FSMs de especificação de protocolo foram criadas seguindo os princípios ilustrados para o caso do calculador de GCD na Seção 5.3. Deste modo, as correspondentes especificações, dos blocos acima enumerados, foram examinadas, e analisadas, para extrair o comportamento relevante à construção de cada M_E .

6.1.2 Blocos de Hardware descritos em SystemC

A intenção deste segundo conjunto de testes foi, principalmente, avaliar os resultados de aplicação das ferramentas em descrições, com inserção intencional de erros, e com uma quantidade razoável de tomadas de decisão (isto é, estados de controle), visando analisar a dependência dos algoritmos implementados sobre tais estruturas. Os casos de estudo descritos com o suporte destas bibliotecas SystemC correspondem a um conjunto de módulos descritos em alto nível e que realizam operações sobre cadeias de bits, para o processamento de pacotes de uma implementação da camada de banda base do protocolo Bluetooth; a implementação foi elaborada, pelo grupo GSEIS, como parte do projeto Brazil-IP (BRAZILIP, 2013).

Os módulos utilizados enumeram-se, a seguir:

- CRC. De forma similar ao CRC em VHDL, este módulo calcula um código de 16 bits a partir da carga útil, utilizando um registrador de deslocamento que, neste caso, usa o polinômio $x^{16}+x^{12}+x^5+1$ (BLUETOOTH, 2007).
- FEC. Módulo para correção de erros nos bits da carga útil. Durante a transmissão, códigos de redundância são acrescentados aos bits da carga útil neste submódulo. Assim, para cada 10 bits de carga útil, o submódulo FEC acrescenta um código de 5 bits utilizando o esquema FEC 2/3. O cálculo do código de redundância é feito com um registrador de deslocamento implementando um codificador *Hamming* 10/5.
- FHEC. Operações de detecção e correção de erros nos bits do cabeçalho dos pacotes.
- WHITE. O módulo WHITE gera a sequência de bits utilizada para cifrar os pacotes do protocolo *Bluetooth* objetivando reduzir o nível DC.
- CIPHER. Este módulo realiza a geração da sequência de bits utilizada para cifrar os dados no caso do estabelecimento de enlaces seguros. O módulo gera uma sequência de bits a partir de certos parâmetros: a chave de enlace (128 bits), o endereço *Bluetooth* completo do dispositivo mestre (48 bits) e o valor do relógio do enlace (26 bits). Após estes valores serem obtidos, o CIPHER gera uma sequência de bits que é fundida aos bits da carga útil do pacote utilizando-se uma função XOR.

Para o caso dos exemplos em SystemC, o procedimento foi realizado em sentido inverso àquele dos blocos em VHDL. Ou seja, os modelos de referência foram refinados, introduzindo sinais adicionais para seu respectivo controle e comunicação às interfaces externas, seguindo as premissas previstas para sequências expandidas, de acordo com a Definição 11 da Seção 4.2. Também, expressões com estilo mais próximo ao do software

foram alteradas para outras com atributos próprios das descrições em RTL. No desenvolvimento da presente pesquisa, os projetos em SystemC foram utilizados em experiências que verificam, unicamente, a região computacional das descrições. Isto foi devido, principalmente, ao tempo e esforço necessários para se obter, tanto os modelos VHDL em alto nível, quanto as versões refinadas em RTL dos projetos em SystemC. Além do mais, como o objetivo primordial era avaliar a metodologia frente a erros de projeto, a ausência de testes da região temporal, para estes exemplos, foi de pouca relevância.

6.2 Aplicação das Ferramentas de Extração e Comparação de Sequências

A ferramenta apresentada na Seção 5.5 foi executada para todos os exemplos de descrições de hardware acima mencionados. Para as experiências, foi utilizado um sistema com processador Intel Core i5 a 3,1 GHz, 4 GB de RAM, e sistema operacional Linux Fedora 18 de 64 bits.

6.2.1 Resultados para os Modelos em VHDL

No caso dos blocos em VHDL, foi checada a equivalência, primeiramente, entre cada par de máquinas M_{HL} e M_{RTL} . Estes resultados foram apresentados em (CASTRO MARQUEZ; STRUM; CHAU, 2013), e são mostrados na Tabela 2. A primeira coluna contém os nomes de cada exemplo, enquanto a segunda e a terceira mostram o número de estados resultante para cada M_{HL} e M_{RTL} , respectivamente. A quarta e quinta colunas correspondem ao número total de sequências para cada M_{HL} e M_{RTL} , respectivamente, enquanto a sexta e sétima colunas mostram o número total de estados considerando todas as sequências de estados extraídas, em M_{HL} e M_{RTL} , respectivamente. Na última coluna mostra-se o tempo de execução de cada

Tabela 2. Resultados de checagem entre M_{HL} e M_{RTL} para blocos de hardware em VHDL.

DUV	Estados em M_{HL}	Estados em M_{RTL}	Sequências em M_{HL}	Sequências em M_{RTL}	Estados em K_{HL}	Estados em K_{RTL}	Tempo (ms)
CRC	2	5	2	7	4	33	23.0
GCD	3	5	3	4	14	41	99.1
CORDIC: polar a ret.	20	28	65	193	941	1,527	230.2
CORDIC: ret. a polar	35	49	69	251	814	2,766	536.7
MDCT	14	36	42	167	22	85	87.5
AES	260	265	256	256	1,118	1,316	140.6

modelo (inclui a geração e a comparação de sequências), que foi sempre inferior a um segundo; apesar de se tratar de exemplos relativamente pequenos, os resultados indicam que a ferramenta possui uma eficiência razoável. É possível observar que os tempos são possuem um grau de proporcionalidade com o número total de estados nos conjuntos K_{HL} e K_{RTL} . Este é um resultado esperado, pois cada estado produz uma iteração no algoritmo da Figura 22b.

Os resultados da Tabela 2 são comparáveis com os de outros autores. Por exemplo, os trabalhos apresentados em (HAO; RAY; XIE, 2012) e (LEE et al., 2011) também usam o exemplo de GCD; para este circuito, o tempo de execução da ferramenta foi aproximadamente três vezes maior que o de (LEE et al., 2011), mas vinte vezes menor que o de (HAO; RAY; XIE, 2012)¹. Ainda que os outros casos de estudo nestes trabalhos são diferentes, seu tamanho e características são similares aos aqui utilizados. De forma geral, a eficiência da metodologia e ferramenta presentes é comparável com (LEE et al., 2011), e consideravelmente maior que no caso de (HAO; RAY; XIE, 2012). Os números da Tabela 2 também sinalizam um ganho de eficiência se comparados com os resultados apresentados em (HU, 2007) e (NISHIHARA; MATSUMOTO; FUJITA, 2006).

O próximo conjunto de experiências consistiu na checagem dos comportamentos de protocolo de cada modelo RTL. Para tal, a especificação funcional dos projetos foi analisada, e a funcionalidade requerida para os sinais de entrada e saída foi extraída e sintetizada na forma de FSMs. Os dados resultantes de aplicação da ferramenta de checagem sobre cada par de máquinas, M_E e M_{RTL} , estão documentados na Tabela 3. As colunas correspondentes a aspectos de M_{RTL} contêm a mesma informação o da Tabela 2, enquanto as colunas 2, 4, e 8, mostram o número de estados, sequências, e bifurcações de estados de controle, respectivamente, para cada M_E . Na última coluna, que apresenta os tempos de execução, pode-se apreciar que a ferramenta realizou a checagem de forma mais rápida que no caso anterior, o que se deve ao tamanho consideravelmente menor das máquinas M_E .

Nesta primeira fase de testes, a metodologia e a ferramenta que conformam o presente quadro de trabalho foram aplicadas a diferentes tipos de descrições de circuitos, em VHDL, que incluem checagem de erros de dados, operações matemáticas, transformadas discretas

¹ Em (LEE et al., 2011) é usado um equipamento com processador Intel Core 2 de 1.86 GHz e 2 GB de memória RAM. Em (HAO; RAY; XIE, 2012), o equipamento possui processador Intel Xeon de 3 Ghz e 2 GB de memória RAM.

para sistemas de imagem, e encriptação de dados para segurança. Para estes exemplos, não foram detectados erros em M_{RTL} , e os resultados mostram que todo tipo de implementação foi verificado dentro de tempos de execução aceitáveis.

6.2.2 Resultados para Modelos em SystemC

Como mencionado anteriormente, as descrições RTL em SystemC foram obtidas por refinamento manual dos seus correspondentes modelos de referência em alto nível. Especificamente, os sinais *clk*, *reset*, *read_ready*, *read_done*, *write_ready*, e *write_done*, foram acrescentados aos modelos comportamentais, seguindo as premissas de expansão previstas na Definição 11 da Seção 4.2. Os sinais de entrada *read_done* e *write_ready* servem para indicar que um novo dado foi lido à saída, ou que será escrito à entrada, respectivamente. Analogamente, os sinais de saída *read_ready* e *write_done*, indicam que um dado está pronto para ser lido à saída, ou que foi escrito à entrada, respectivamente. Estes sinais, específicos das versões em RTL, foram inseridos utilizando as mesmas condições, e estruturas, para todos os exemplos.

As experiências realizadas com o blocos em SystemC serviram para avaliar o desempenho real da ferramenta para identificar erros no DUV. Tendo isto como objetivo, todas as versões em RTL foram alteradas, injetando-se nelas 50 erros intencionais. Tais erros foram tanto de tipo combinatório, quanto sequencial, e são detalhados, a seguir:

- 1) Combinatórios. Mudanças pontuais em expressões de avaliação ou designação, que consistiram das seguintes alterações:

Tabela 3. Resultados de checagem entre M_E e M_{RTL} para blocos de hardware em VHDL.

DUV	Estados em M_E	Estados em M_{RTL}	Sequências em M_E	Sequências em M_{RTL}	Bifurcações de estados de controle em M_E	Bifurcações de estados de controle em M_{RTL}	Tempo (ms)
CRC	5	5	2	7	2	23	9.8
GCD	4	5	2	4	2	11	14.5
CORDIC: polar a ret.	6	28	1	193	2	25	45.9
CORDIC: ret. a polar	6	49	1	251	2	37	38.1
MDCT	15	36	2	167	8	76	15.6
AES	11	265	3	256	5	124	36.2

- Troca de operadores como, por exemplo, ‘<’ por ‘<=’, ou ‘^’ por ‘*’;
- Troca das variáveis que compõem expressões de designação ou de avaliação. Exemplos incluem “a=b+c” por “b=a+c”, ou “x<y” por “y<x”;

2) Sequenciais. Mudança do ponto onde ocorrem eventos dentro do código das descrições. Dois erros sequenciais introduzidos: primeiro, a troca das condições entre duas expressões de controle (*if*, *for*, etc.) consecutivas, e, segundo, a mudança da localidade no código de uma expressão de designação.

A Tabela 4 apresenta os resultados obtidos para todos os blocos descritos em SystemC. A primeira coluna mostra o nome de cada projeto sob verificação. A segunda e terceira colunas mostram o número de estados para as máquinas M_{HL} e M_{RTL} de cada bloco, respectivamente. A quarta e quinta colunas mostram o número total de bifurcações de todos os estados de controle, para cada M_{HL} e M_{RTL} , de forma respectiva. A sexta e sétima colunas correspondem ao número de erros injetados e detectados, respectivamente, em M_{RTL} , pela ferramenta implementada.

Dado que o script em *Python* não consegue detectar todos os erros injetados dentro uma única execução, tais erros foram detectados de forma progressiva, através de várias execuções consecutivas. As últimas duas colunas da Tabela 4 apresentam o número de execuções, e o tempo médio destes, respectivamente, requeridos para achar os erros inseridos. Ao se avaliar a razão de não detecção de erros no caso do CRC, FEC, e FHEC, fato não esperado, verificou-se que o problema foi a troca do operador ‘==’ por ‘=’; esta deficiência foi ocasionada por um uso incorreto do analisador de SystemC, e do solucionador CVC3, pois o operador de comparação ‘==’ estava sendo transformado no de designação ‘=’ de forma inadvertida.

Tabela 4. Resultados de aplicação da ferramenta aos blocos descritos em SystemC.

DUV	Estados em M_{HL}	Estados em M_{RTL}	Bifurcações de estados de controle em M_{HL}	Bifurcações de estados de controle em M_{RTL}	Erros injetados em M_{RTL}	Erros detectados em M_{RTL}	Número de testes	Tempo médio (ms)
CRC	25	29	49	57	50	45	50	89.3
FEC	43	47	70	78	50	48	50	65.4
FHEC	34	42	65	73	50	45	50	81.0
WHITE	36	44	44	52	50	50	26	56.9
CIPHER	21	29	24	32	50	50	33	25.3

Exceto por este problema, todos os demais erros combinatórios e os de tipo sequencial foram detectados na íntegra. Fica evidente que a metodologia proposta, tendo o modelo RTL sido obtido como um modelo expandido da descrição de alto-nível, realiza a checagem forte de equivalência como definido na Seção 4.2.

7 CONCLUSÕES

O presente trabalho apresentou uma metodologia, desenvolvida ao longo deste projeto de doutorado, para efetuar a verificação formal de hardware implementado em RTL através de técnicas de checagem de equivalência. A metodologia utiliza técnicas formais de modelagem e exploração de estados, para estabelecer a validade do DUV, em relação a um modelo de referência em alto nível, assim como, ao comportamento especificado para seu correspondente protocolo de comunicação.

De forma a estabelecer tal estratégia de SEC, uma caracterização da checagem de equivalência tradicional foi realizada, o que permitiu visualizar os requisitos e carências relacionados com o quadro conceitual deste trabalho, no qual níveis de abstração e interfaces desiguais tiveram que ser considerados.

Após a análise anterior, foi apresentado um quadro teórico formalizado, específico para resolver o principal problema do presente trabalho, que é o de verificar que o DUV é funcionalmente coerente a partir de um modelo de referência em alto nível, e do comportamento especificado para as interfaces de comunicação. Para tal teoria, definiu-se, como fundamento, realizar a checagem de equivalência baseando-se em sequências inteiras de estados, ao invés de estados individuais (que é o caso da SEC tradicional). Esta base teórica considerou todos os aspectos relevantes à checagem de equivalência entre níveis diferentes de abstração, como conjuntos diferentes de portas, sinais e variáveis, e comportamentos de sincronismo e temporização dessemelhantes.

De acordo com o formalismo desenvolvido, foram adotadas FSMs como forma de representação interna para o DUV, o modelo de referência, e o protocolo. Assim, visando realizar o tratamento de tais FSMs, e fazendo uso das definições e conceitos da teoria apresentada, algoritmos específicos foram desenvolvidos para permitir a sistematização da verificação de sistemas digitais, descritos em RTL. Foi demonstrado que estes algoritmos conseguem percorrer o espaço de todos os estados alcançáveis das FSMs, gerando as correspondentes sequências de estados, e comparando-as como necessário.

Os algoritmos anteriores foram implementados em uma ferramenta que integra as linguagens *Python* e *C++*, e as ferramentas VAUL, um analisador gramatical de SystemC, e CVC3. Esta ferramenta conforma um ambiente de verificação que aceita projetos descritos tanto em VHDL, quanto em SystemC, e os transforma em FSMs, que, por sua vez, são

decompostas em sequências de estados que serão comparadas para estabelecer, no final, se os projetos inseridos à ferramenta são funcionalmente coerentes.

A ferramenta anterior foi aplicada em experiências com diferentes casos estudo descritos em ambas as linguagens VHDL e SystemC. Os correspondentes resultados demonstram que a metodologia é efetiva, e eficiente, na verificação formal de tipos diversos de circuitos digitais (englobando correção de erros, encriptação, processamento de imagens, e funções matemáticas). Também se evidenciou que a ferramenta consegue encontrar erros, de tipo combinatório e sequencial, dentro de tempos e quantidades de iterações praticáveis em casos reais.

Os resultados apresentados neste trabalho foram publicados nos artigos:

- “Formal equivalence checking between high-level and RTL hardware designs”, no evento 14th Latin American Test Workshop (LATW2013);
- “Functional Verification of Complete Sequential Behaviors: A Formal Treatment of Discrepancies between System-Level and RTL Descriptions”, no evento 8th IEEE International Design & Test Symposium 2013, IDT’13;
- “A Unified Sequential Equivalence Checking Approach to Verify High-Level Functionality and Protocol Specification Implementations in RTL Designs”, no evento 15th Latin American Test Workshop (LATW2014).

8 TRABALHOS FUTUROS

A metodologia de verificação formal apresentada nesta tese de doutorado contém, ainda, restrições que limitam sua aplicação a um conjunto específico de projetos digitais. Em virtude disto, é possível pensar em melhoras e aperfeiçoamentos no futuro, como exposto a seguir:

- como foi visto, as descrições sujeitas a verificação através das técnicas introduzidas devem estar enquadradas dentro de um ambiente de projeto com refinamentos sucessivos. Isto também limita o universo possível de projetos a verificar, pois quaisquer duas descrições, a serem processadas pela ferramenta desenvolvida, deverão basear-se no mesmo princípio algorítmico. Portanto, espera-se, para uma futura versão da metodologia e da ferramenta, que o DUV possa ter um distanciamento comportamental maior (no que diz respeito ao seu nível de abstração) ao modelo de referência, assim como, ao protocolo de comunicação;
- de certa forma, derivada do problema anterior, o conceito de sequência estendida de estados foi introduzido nesta tese para se conseguir uma técnica forte de verificação. O espaço de possíveis sequências resulta ser limitado a este tipo de sequências, e, portanto, a equivalência funcional entre duas FSMDs só poderá ser garantida se uma delas contiver uma sequência estendida para cada sequência da outra. Assim, aumentar a teoria apresentada para um universo de sequências mais amplo, resultará em uma metodologia de verificação mais produtiva, dado que um número maior de casos de aplicação poderá ser considerado;
- a verificação da funcionalidade do protocolo impõe um esforço considerável, o que se explica a partir da necessidade de sintetizar a correspondente FSMD de forma manual, a partir de uma especificação que, em muitos casos, possui um formato informal. Como trabalho futuro, propõe-se adotar, ou desenvolver, um método que torne mais eficiente esta tarefa. Possíveis soluções incluem a geração automática das FSMDs de protocolo a partir de propriedades formais predefinidas, ou por tradução de transatores provenientes de ambientes de simulação.

9 REFERÊNCIAS

ACCELLERA. **SystemC – Accellera Systems Initiative**. Disponível em: <<http://www.accellera.org/downloads/standards/systemc>>. Acesso em Setembro de 2013.

AHO, A.; HOPCROFT, J.; ULLMAN, J. **The Design and analysis of computer algorithms**. Addison-Wesley. 1974.

BAUMGARTNER, J. et al. **Scalable conditional equivalence checking: An automated invariant-generation based approach**. Em: Formal Methods in Computer-Aided Design, 2009. FMCAD 2009. p. 120-127.

BENJAMIN, M. et al. **A study in coverage-driven test generation**. Em: Design Automation Conference, 1999. p. 970-975.

BERGERON, J. Writing testbenches: functional verification of HDL models. Kluwer Academic Publishers, 2003. v. 2.

BERKMIN. **Berkmin**. Disponível em: <<http://eigold.tripod.com/BerkMin.html>>. Acesso em Setembro de 2013.

BLUETOOTH, S. Specification of the Bluetooth system: core package version 2.1+ EDR. 2007.

BOMBIERI, N. et al. **Towards equivalence checking between TLM and RTL models**. Em: Formal Methods and Models for Codesign, 2007. MEMOCODE 2007. 5th IEEE/ACM International Conference on. p. 113-122.

BRAZILIP. **Brazil-IP**. Disponível em: <<http://www.brazilip.org.br/projetos/home>>. Acesso em Setembro de 2013.

CABODI, G. et al. Finding Multiple Equivalence-Preserving Transformations in Combinational Circuits through Incremental-SAT. Journal of Electronic Testing. Springer, 2010.

CALYPTO. **Calypto Design Systems**. Disponível em: <<http://calypto.com>>. Acesso em Setembro de 2013.

CASTRO MARQUEZ, C.; STRUM, M.; CHAU, W. J. **Formal equivalence checking between high-level and RTL hardware designs**. Em: Test Workshop (LATW), 2013 14th Latin American. p. 1-6.

CAVADA, R. Nusmv 2.5 user manual. 2010.

CHAKRABARTI, A. et al. **Formal verification of module interfaces against real time specifications**. Em: Proceedings-Design Automation Conference. p. 141-145.

CIBRASIL. **CI Brasil**. Disponível em: <<http://www.ci-brasil.gov.br/>>. Acesso em Setembro de 2013.

COUSSY, P.; MORAWIEC, A. High-level synthesis: from algorithm to digital circuit. Springer, 2008.

CVC3. **CVC3**. Disponível em: <<http://www.cs.nyu.edu/acsys/cvc3/>>. Acesso em Agosto de 2013.

D'SILVA, V.; RAMESH, S.; SOWMYA, A. Synchronous protocol automata: a framework for modelling and verification of SoC communication architectures. IEEE Proceedings-Computers and Digital Techniques IET, 2005.

DA SILVA, K. R. et al. A methodology aimed at better integration of functional verification and RTL design. Design Automation for Embedded Systems. Springer, 2005.

DRECHSLER, R. **Advanced formal verification**. Kluwer. 2004.

DRECHSLER, R.; FEY, G. **Formal verification meets robustness checking—Techniques and challenges**. Em: Design and Diagnostics of Electronic Circuits and Systems (DDECS), 2010 IEEE 13th International Symposium on. p. 4-4.

EIRIKSSON, Á. T. **Integrating formal verification methods with a conventional project design flow**. Em: Design Automation Conference Proceedings 1996, p. 666-671.

FUJITA, M.; GHOSH, I.; PRASAD, M. **Verification techniques for system-level design**. Morgan Kaufmann, 2008.

FUMMI, F. et al. **A CLP-Based Functional ATPG for Extended FSMs**. Em: Microprocessor Test and Verification, 2007. MTV'07. Eighth International Workshop on. p. 98-105.

FUMMI, F.; PRAVADELLI, G. **Too Few or Too Many Properties? Measure it by ATPG!**. Journal of Electronic Testing. Springer, 2007.

GAJSKI, D. D. et al. **Embedded System Design: Modeling, Synthesis and Verification**. [s.l.] Springer Publishing Company, Incorporated, 2009.

GOLDREICH, O. **Computational Complexity: A Conceptual Perspective**. Cambridge University Press, 2008.

GROßE, D.; DRECHSLER, R. **CheckSyC: An efficient property checker for RTL SystemC designs**. Em: Circuits and Systems, 2005. ISCAS 2005. IEEE International Symposium on. p. 4167-4170.

GSEIS. **Grupo GSEIS**. Disponível em: <<http://www.psi.poli.usp.br/index.php?p=grupos&s=9&k=apresentacao>>. Acesso em Setembro de 2013.

HAO, K.; RAY, S.; XIE, F. **Equivalence checking for behaviorally synthesized pipelines**. Em: Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE. p. 344-349.

HU, A. J. **High-level vs. RTL combinational equivalence: An introduction**. Em: Computer Design, 2006. ICCD 2006. International Conference on. p. 274-279.

KAMKIN, A. **The UniTESK Approach to Specification-Based Validation of Hardware Designs**. Em: Leveraging Applications of Formal Methods, Verification and Validation, 2006. ISoLA 2006. Second International Symposium on. p. 60-66.

KERN, C.; GREENSTREET, M. R. **Formal verification in hardware design: a survey**. ACM Transactions on Design Automation of Electronic Systems (TODAES). ACM, 1999.

KOELBL, A. et al. **Solver technology for system-level to RTL equivalence checking**. Em: Design, Automation & Test in Europe Conference & Exhibition, 2009. DATE'09. p. 196-201.

KROPF, T. **Introduction to Formal Hardware Verification**. Springer, 1999.

LEE, C.-H. et al. **Equivalence checking of scheduling with speculative code transformations in high-level synthesis**. Em: Design Automation Conference (ASP-DAC), 2011 16th Asia and South Pacific. p. 497-502.

MAALEJ, A.; MARTINEZ, P.-Y. **Formal proof between two designs having different level of abstraction**. Em: Design and Technology of Integrated Systems in Nanoscale Era, 2008. DTIS 2008. 3rd International Conference on. p. 1-4.

MARQUES-SILVA, J. P.; SAKALLAH, K. A. **GRASP: A search algorithm for propositional satisfiability**. Computers, IEEE Transactions on. IEEE, 1999.

MATHUR, A. et al. Functional equivalence verification tools in high-level synthesis flows. Design & Test of Computers, IEEE. IEEE, 2009.

MATHWORKS. **MatLab**. Disponível em: <www.mathworks.com/products/matlab/>. Acesso em Setembro de 2013.

MATSUMOTO, T. et al. **Equivalence checking of high-level designs based on symbolic simulation**. Em: Communications, Circuits and Systems, 2009. ICCAS 2009. International Conference on. p. 1129-1133.

MOLITOR, P.; MOHNKE, J. Equivalence checking of digital circuits: fundamentals, principles, methods. Kluwer. 2004.

MOY, M.; MARANINCHI, F.; MAILLET-CONTOZ, L. **Pinapa: an extraction tool for SystemC descriptions of systems-on-a-chip**. Em: Proceedings of the 5th ACM international conference on Embedded software. p. 317-324.

NAVEH, Y. et al. Constraint-based random stimuli generation for hardware verification. AI magazine, 2007.

NISHIHARA, T.; MATSUMOTO, T.; FUJITA, M. **Equivalence Checking with Rule-Based Equivalence Propagation and High-Level Synthesis**. Em: High-Level Design Validation and Test Workshop, 2006. Eleventh Annual IEEE International. p. 162-169.

OPENCORES. **OpenCores**. Disponível em: <<http://opencores.org/>>. Acesso em Agosto de 2013.

PATÍÑO ALVAREZ, G. A. Projeto e Criação Automática de uma Representação Intermediária de Sistemas Digitais Descritos em SystemC. Dissertação de Mestrado. Universidade de São Paulo. 2005.

PENG, H.; TAHAR, S. **Hardware modeling and verification of an ATM ring MAC protocol**. Em: Microelectronics, 2000. ICM 2000. Proceedings of the 12th International Conference on. p. 21-24.

PIZIALI, A. Functional verification coverage measurement and analysis. Springer, 2004.

PYTHON. **Python Programming Language**. Disponível em: <<http://www.python.org/>>. Acesso em Agosto de 2013.

ROMERO, E. L. Técnicas para a verificação funcional eficiente de uma implementação RTL da camada banda base do protocolo Bluetooth. Dissertação de Mestrado. Universidade de São Paulo. 2005.

RUF, J. et al. **Bounded property checking with symbolic simulation.** Em: Forum on Specification and Design Languages. v. 2003.

SAGLAMDEMIR, M. O.; SEN, A.; DUNDAR, G. **A formal equivalence checking methodology for Simulink and Register Transfer Level designs.** Em: Synthesis, Modeling, Analysis and Simulation Methods and Applications to Circuit Design (SMACD), 2012 International Conference on. p. 221-224.

SCHUEGRAF, K. et al. **Semiconductor Logic Technology Innovation to Achieve Sub-10 nm Manufacturing.** Electron Devices Society, IEEE Journal of the. IEEE, 2013.

SYNOPSYS. **Synopsys.** Disponível em: <<http://www.synopsys.com/home.aspx>>. Acesso em Julho de 2013.

VAHID, F.; GIVARGIS, T. Embedded System Design, A Unified Hardware/Software Introduction. John Wiley & Sons, 2013.

VASUDEVAN, S. et al. **Sequential equivalence checking between system level and rtl descriptions.** Design Automation for Embedded Systems. Springer, 2008.

VAUL. **The Free HDL Project.** Disponível em: <<http://www.freehdl.seul.org/>>. Acesso em Julho de 2013.

WILE, B.; GOSS, J. C.; ROESNER, W. **Comprehensive Funcional Verification.** Kluwer. 2005.

WILLIAMS, P. F.; ANDERSEN, H. R.; HULGAARD, H. **Satisfiability checking using boolean expression diagrams.** Em: Tools and Algorithms for the Construction and Analysis of Systems. Springer, 2001. p. 39-51.

YANG, Y.-C. et al. **Formal compliance verification of interface protocols.** Em: VLSI Design, Automation and Test, 2005.(VLSI-TSA-DAT). 2005 IEEE VLSI-TSA International Symposium on. p. 12-15.

YOSHIDA, H.; FUJITA, M. **Rule-based equivalence checking of system-level design descriptions.** Em: Communications, Circuits and Systems, 2009. ICCCAS 2009. International Conference on. p. 1139-1143.

ZCHAFF. **zChaff**. Disponível em: <<http://www.princeton.edu/~chaff/zchaff.html>>. Acesso em: 9–19 2013.

ZHU, M.; BIAN, J.; WU, W. A novel collaborative scheme of simulation and model checking for system properties verification. Computers in industry. Elsevier, 2006.