UFRR

UNIVERSIDADE FEDERAL DE RORAIMA

PRÓ-REITORIA DE ENSINO E EXTENSÃO

DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

KEVIN COSTA AIRES OLIVEIRA

VERIFICAÇÃO FORMAL DE CIRCUITOS LÓGICOS BASEADOS EM TRANSFORMAÇÃO DE CÓDIGO COM *BOUNDED MODEL CHECKING*

Boa Vista - RR

2019

KEVIN COSTA AIRES OLIVEIRA

VERIFICAÇÃO FORMAL DE CIRCUITOS LÓGICOS BASEADOS EM TRANSFORMAÇÃO DE CÓDIGO COM *BOUNDED MODEL CHECKING*

Monografia de Graduação apresentada ao Departamento de Ciência da Computação da Universidade Federal de Roraima como requisito para a obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Dr. Herbert Oliveira Rocha

Boa Vista - RR

2019

KEVIN COSTA AIRES OLIVEIRA

VERIFICAÇÃO FORMAL DE CIRCUITOS LÓGICOS BASEADOS EM TRANSFORMAÇÃO DE CÓDIGO COM *BOUNDED MODEL CHECKING*

Monografia de Graduação apresentada ao Departamento de Ciência da Computação da Universidade Federal de Roraima como requisito para a obtenção do grau de Bacharel em Ciência da Computação.
Defendido em 10 de julho de 2019 e aprovado pela seguinte banca examinadora:

Prof. Dr. Herbert Oliveira Rocha
Orientador / Curso de Ciência da Computação - UFRR

Prof. Msc. Filipe Dwan Perreira
Curso de Ciência da Computação - UFRR

Prof. Dra. Marcelle Alencar Urzquiza
Curso de Ciência da Computação - UFRR

Dedico este trabalho a minha família e amigos e a todos aqueles que acreditaram no meu esforço.

AGRADECIMENTOS

Primeiramente, agradeço à minha família, por sempre estar ao meu lado nos momentos mais difíceis da faculdade, pelos conselhos e pelo o incentivo em todos estes anos de curso.

Agradeço ao meu orientador, professor Herbert Oliveira, por todo apoio desde a disciplina de arquitetura de computadores, por todas as orientações no desenvolvimento deste projeto, além dos incentivos para vida pós faculdade.

Agradeço também aos amigos que sempre ajudaram, seja lendo uma lauda por cada encontro, discutindo sobre seções no laboratório ou na sala, fazendo companhia em algum posto, incentivando e corrigindo os erros de português.

Agradeço aos professores do DCC que durantes estes anos de curso foram fonte de aprendizado e incentivos durante aulas e que cada uma pode contribuir de alguma forma no meu desenvolvimento como aluno e futuro profissional.

Não importa o que o mundo diz de mim, o que importa é que eu nunca fiz nada que contrariasse os meus princípios e nunca farei.

RESUMO

Objetiva-se neste trabalho a verificação de circuitos lógicos descritos em VHDL, adotando técnicas de verificação formal e transformações de código, buscando identificar funcionamentos incorretos de determinado circuito. A metodologia empregada realiza a inserção de assertivas contendo propriedades sobre um dado circuito analisado em VHDL, onde é aplicada a transformação e instrumentação de código VHDL para C visando a utilização da técnica de indução-k e exploração dos estados alcançáveis do circuito, e determinando a ocorrência ou não de violação de propriedade descrita na assertiva inserida ao código. Para a etapa de tradução foram desenvolvidos dois métodos de tradução, buscando a melhor performance, através de uma única ferramenta ou através de várias ferramentas. Para a análise foi utilizada a ferramenta ESBMC, (*Extended SMT-Based Bounded Model Checker*) que utilizou a técnica de indução-k para análise de todos os códigos. A ferramenta utilizou pré e pós condições inseridas em cada código para analisar se elas foram violadas ou não. Na elaboração dos resultados constatado um melhor desempenho do método de múltiplas traduções tendo um desempenho e 44% superior se comparado com o método anterior de tradução. Na etapa de análise também demonstrou uma melhoria através da técnica de indução-k se comparado apenas com a técnica de BMC utilizado anteriormente no desenvolvimento do trabalho.

Palavras-chaves: Transformação de código, Bounded Model Cheching, VHDL, Hardware, Assertivas.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

No contexto de sistemas compostos de hardware e software, tem-se os sistemas embarcados (SE) que são dispositivos semicondutores com software integrados, os quais se conectam a outros dispositivos. Usualmente, o principal propósito de um SE é o controle e provimento de informações para uma função específica (RAMESH et al., 2012).

Os SE têm se proliferado em partes consideráveis das atividades do nosso cotidiano, com a característica de possuir grande quantidade de complexidade e diversidade. Devido a esta complexidade, erros podem passar despercebidos, causando não apenas penalidades econômicas e/ou fracassos do produto no mercado, mas principalmente o risco de perda de vidas humanas (CABODI et al., 2016). Como, por exemplo, o acidente noticiado no (G1, 2012), no qual, de acordo com o secretário de transportes, uma falha na placa do circuito eletrônico responsável pelo controle de velocidade dos trens ocasionou a colisão entre duas composições na estação de metrô de São Paulo.

Segundo Rocha et al. (2015b), para se obter um alto nível de qualidade no desenvolvimento dos sistemas de hardware e software, a execução desses sistemas deve ser controlada e da mesma forma deve-se buscar meios de garantir que as propriedades definidas sejam atingidas. Por exemplo, a partir do conhecimento prévio do modo de implementação de determinado hardware, é possível utilizá-lo de forma mais eficiente. Neste sentido, diversas estratégias de verificação e de teste estão sendo pesquisadas e aplicadas para garantir a qualidade do software e hardware (HODER et al., 2010; ROCHA et al., 2010; BRAYTON; MISHCHENKO, 2010; CORDEIRO et al., 2012; CABODI et al., 2016).

Por este motivo, com o intuito de analisar e otimizar a descrição dos circuitos digitais, tem-se utilizado as linguagens de descrição de hardware (HDL). Estas se diferem das linguagens de programação por conseguirem gerar execuções não apenas sequenciais, como também concorrentes ou paralelas (CHU, 2006). Neste contexto, a *VHSIC Hardware Description Language* (VHDL) é uma das linguagens de descrição de hardware mais utilizadas atualmente. A descrição em VHDL pode ser em vários níveis de abstração, sendo o mais alto o nível comportamental que permite a descrição do circuito através de *loops* e processos, definindo-o na forma de algoritmo. Como características das linguagens de descrição, o VHDL permite declarações sequenciais ou concorrentes onde as declarações continuam ativas e sua ordem torna-se irrelevante (CAPPELATTI, 2010).

Aliada à linguagem de descrição de hardware, a verificação formal de sistemas computacionais tem desempenhado um papel importante para assegurar a previsibilidade e a confiabilidade na concepção de aplicações críticas. E, para isso, tem-se utilizado a técnica denominada *model checking*, que é baseada em formalismos matemáticos para provar propriedades de programas

reativos (BENSALEM; LAKHNECH, 1999). Esta técnica gera uma busca exaustiva no espaço de estados do modelo para determinar se uma dada propriedade é válida ou não (BAIER et al., 2008), tendo como principal razão para o seu sucesso o funcionamento completamente automático, ou seja, sem qualquer intervenção do usuário.

Visando contribuir com verificação de sistemas computacionais, principalmente, no âmbito de sistemas embarcados, o objetivo deste trabalho está situado no uso de metodologias e técnicas de verificação formal para programas escritos em VHDL (BIERE, 2016), focando principalmente no *Bounded Model Checking* (CORDEIRO et al., 2012; ROCHA et al., 2015a). Neste trabalho foi dada ênfase na parte de verificação de modelos de hardware descritos em níveis de circuitos de bits na linguagem VHDL, via transformações de código para gerar modelos, com assertivas, já suportados por *model checkers*, como o ESBMC (CORDEIRO et al., 2012). Dessa forma, objetivamos analisar as propriedades de alcançabilidade para a identificação de localizações de erro, bem como, assertivas contendo propriedades de segurança.

1.1 Definição do problema

O avanço da tecnologia, principalmente nas áreas de design e fabricação de eletrônicos, aumentou a importância do hardware nos dias atuais. Cada vez com mais funcionalidades integradas, aliada a velocidade e circuitos menores, faz com que a complexidade dos sistemas de hardware aumente. Neste sentido, a detecção tardia de erros no sistema pode resultar em perda de produção, mas também custos associados ao desenvolvimento do sistema (GUPTA, 1992). Sendo assim, a verificação de hardware visa assegurar que o circuito atinja as especificações para o qual foi projetado, através de técnicas formais ou dinâmicas (BOULE; ZILIC, 2007).

A utilização de métodos formais tornou-se uma abordagem atraente para superar as limitações inerentes à validação baseada em simulação. Portanto, a maioria das empresas de semicondutores têm investigado a sua aplicabilidade. Escolhas individuais de métodos, ferramentas e áreas de aplicação das empresas têm variado, assim como o seu nível de sucesso (CABODI et al., 2016). Com os recentes avanços na escalabilidade de automação dos *model checkers* e nas equivalências sequenciais de verificação, a maioria das empresas têm crescido ao contar com essas técnicas(CLARKE, 2008).

O problema considerado neste trabalho é expresso na seguinte questão: **Como complementar e aprimorar a verificação de propriedades de segurança em circuitos lógicos escritos em VHDL, de forma que uma propriedade possa ser mapeada em um problema de alcançabilidade simbólica? É possível alcançar um estado específico (para uma dada propriedade) a partir do estado inicial?**

1.2 Objetivos

O objetivo principal neste trabalho é desenvolver uma metodologia para efetuar a verificação de circuitos descritos em VHDL com portas lógicas em nível de bit por meio de técnicas de transformação de códigos combinado com a técnica *Bounded Model Checking* para exploração de estados alcançáveis no circuito, visando identificar erros ou comportamentos indevidos ou inesperados que podem resultar no funcionamento incorreto de um dado sistema.

Os objetivos específicos são:

1. Propor um método para especificar pré e pós-condições de circuitos digitais em nível de portas lógicas descritos em VHDL;
2. Especificar uma técnica de conversão de circuitos em linguagem de descrição de hardware VHDL para um modelo a ser verificado usando a técnica *Model Checking*;
3. Desenvolver um método para identificação de estados de erros ou ocorrências indevidas, de um dado circuito analisado, em modelos de hardware descritos em nível de bit na linguagem de descrição VHDL.
4. Validar a aplicação do método proposto sobre *benchmarks* públicos de programas em VHDL, a fim de examinar a sua eficácia e aplicabilidade.

1.3 Contribuições propostas

As contribuições propostas para este trabalho são:

- O desenvolvimento de uma metodologia para verificação de hardware com o intuito de facilitar os passos que evem ser seguidos e, ao mesmo tempo, reduzir substancialmente o tempo de verificação de projetos de hardware descritos em VHDL;
- O desenvolvimento e implementação de uma ferramenta de verificação de circuitos lógicos em VHDL com a integração da ferramenta ESBMC (*Efficient SMT-Based Context-Bounded Model Checker*)(CORDEIRO et al., 2012) na análise.

1.4 Organização do trabalho

A introdução deste trabalho apresentou: seus contexto, definição do problema e objetivos. Os próximos capítulos estão organizados da seguinte forma:

No Capítulo 2 **Conceitos e Definições**, são apresentados os conceitos abordados neste trabalho, especificamente: Linguagens de descrição de hardware; Verificação e validação de sistemas; e, Técnicas de compiladores.

No Capítulo 3, **Trabalhos Correlatos**, serão apresentados o método de pesquisa bibliográfica utilizado, sendo ele a revisão sistemática, seguido do resultado encontrado com esta pesquisa e, por fim, a contribuição dos artigos utilizados no desenvolvimento do projeto.

No Capítulo 4, **Metodologia Proposta**, são descritas as etapas de execução da metodologia proposta. Bem como, duas abordagens de modo a demonstrar sua execução, cada uma com usas particularidades.

No Capítulo 5 **Avaliação Experimental**, são apresentados os testes realizados com as duas abordagens em questão, demonstrando a metodologia usada para validar os experimentos e, também os pontos positivos e negativos de cada abordagem.

E, por fim, no Capítulo 6, **Considerações finais e trabalhos futuros**, são apresentas as considerações finais, principais pontos levantados no desenvolvimento da pesquisa, assim como, sugestões de trabalhos futuros.

1.5 Resumo do capitulo

Neste capítulo foi apresentado uma breve introdução, contendo a contextualização do assunto tratado na pesquisa. Tal como a definição de seu problema, que consiste na verificação de propriedades de circuitos lógicos. Em seguida, foram apresentados seus objetivos: geral e específicos. Sendo que, o geral consiste em desenvolver uma metodologia para identificação de erros em circuitos lógicos e, consequentemente, sua validação. Por fim, expõe a organização estrutural empregada na pesquisa.

2 CONCEITOS E DEFINIÇÕES

Este capítulo tem apresenta os principais conceitos e definições abordados neste trabalho, tais como: Linguagens de descrição de hardware, Verificação de sistemas e Técnicas de Compiladores.

2.1 Linguagens de descrição de hardware

As linguagens de descrição de hardware (HDL) foram desenvolvidas com o intuito de auxiliar a criação de circuitos lógicos com grande número de elementos e com uma gama de abstrações lógicas e eletrônicas (THOMAS; MOORBY, 2008). Entre os exemplos, podemos citar: *VDHL* (IEEE, 2011), *Verilog* (IEEE, 2006) e *SystemC* (IEEE, 2012).

Segundo (CHRISTEN; BAKALAR, 1999), linguagens de descrição de hardware são linguagens de programação utilizadas com o intuito de descrever o comportamento de um determinado circuito, técnica conhecida como modelagem. Os modelos descritos em HDL são utilizados como entrada para um simulador, onde o mesmo pode ter seu comportamento analisado.

As HDL's trazem consigo diversas vantagens, entre elas códigos independentes de tecnologia e fabricante, podendo ser portáteis e reutilizáveis (CAPPELATTI, 2010). As linguagens mais modernas, tais como VHDL e *Verilog*, possuem suporte tanto a descrição do circuito propriamente dito, quanto ao comportamento que o mesmo deve exercer (CHRISTEN; BAKALAR, 1999).

A *Verilog*, como já citado, é uma linguagem de descrição de *hardware* que fornece diversos níveis de abstração para desenvolvimento de sistemas digitais(THOMAS; MOORBY, 2008). A linguagem foi desenvolvida para ser simples e efetiva nos diversos níveis de abstração incluindo o suporte ao desenvolvimento, verificação, síntese e testes de *software* (IEEE, 2006).

Figura 1 – Exemplo de multiplexador descrito em Verilog.

```
1 primitive multiplexer (mux, control, dataA, dataB);
2 output mux;
3 input control, dataA, dataB;
4 endprimitive
```

Fonte: Adaptado de (IEEE, 2006).

Na Figura 1 apresenta um exemplo de multiplexador descrito em *Verilog* utilizando a

estrutura de modelagem, chamada *UDP* que permite a criação de novos primitivas para serem utilizada. Apresenta uma saída apenas(mux) e três entradas sendo uma de controle(control) e duas de dados(dataA e dataB).

A VHDL é uma linguagem de descrição de hardware amplamente utilizada, concebida na década de 80, devido uma necessidade do Departamento de Defesa dos Estados Unidos da América (CAPPELATTI, 2010). Assim como a Verilog, esta linguagem também suporta o desenvolvimento, a verificação, a síntese, e os testes de *hardware* (IEEE, 2011).

Figura 2 – Exemplo de multiplexador descrito em VHDL.

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3  use IEEE.std_logic_unsigned.all;
4  entity mux2x1 is
5  port (sel: in STD_LOGIC;
6        a,b: in STD_LOGIC;
7        y: out STD_LOGIC);
8  end mux2x1;
9
10 architecture dataflow of mux2x1 is begin
11 y <= (a AND NOT sel) OR (b AND sel);
12 end dataflow.
```

Fonte: (CAPPELATTI, 2010).

Na Figura 2 está descrito um multiplexador de duas entradas descrito em VHDL, apresentando na parte inicial as bibliotecas e na entidade(entity) a declaração das variáveis utilizadas. Na arquitetura(architecture) apresenta o funcionamento do multiplexador, relacionando entradas e saídas conforme tenham sido declarados. Estes conceitos serão melhor apresentados na subseção 2.1.1.

Conforme apresentado nas Figura 1 e Figura 2, ambas as linguagens apresentam diferenças nos modelos de declaração e utilização para descrição de hardware. Para este trabalho foi escolhido a VHDL devido a linguagem apresentar um melhor escopo para análise, facilitando o trabalho de análise proposto para este projeto.

2.1.1 VHSIC Hardware Description Language - VHDL

VHDL é uma linguagem de descrição de hardware destinada a ser utilizada em todas as etapas da criação de sistemas eletrônicos, sendo elas: desenvolvimento; verificação; síntese; e, teste de circuitos (IEEE, 2011). Segundo Cappelatti (2010), a VHDL apresenta três principais pilares: **abstração**, que consiste na descrição com diferentes níveis de detalhes; **modularidade**, que permite a divisão do projeto em vários blocos ou módulos para posterior interconexão; e **hierarquia**, permite que módulos possam ser compostos por submódulos e, os mesmos tenham

níveis de abstração diferentes, dependendo da necessidade do projeto. Devido a esta versatilidade que a linguagem VHDL foi selecionada para o trabalho proposto.

Segundo IEEE (2011), é necessário um padrão para formatação da descrição em VHDL: *Entity* (Pinos de entrada/saída) e *Architecture* (Arquitetura). A *Entity* ou entidade representa o bloco onde são declaradas as entradas e as saídas do circuito utilizadas em todo o sistema, conforme apresentado na Figura 3. A *Architecture* ou arquitetura, apresentado na Figura 4, define a relação entre entradas e saídas declaradas na entidade, podendo tais especificações serem completas ou parciais. Assim como em outras linguagens, bibliotecas podem ser adicionadas, para que novas funções e atributos possam ser utilizados no projeto.

Figura 3 – Exemplo de declaração de bibliotecas e entidade.

```
1 library ieee;
2 use IEEE.STD_LOGIC_1164.ALL
3
4 ENTITY full_adder IS PORT(
5   x1,x2,cin: in std_logic;
6    S,count:out std_logic);
7 END full_adder;
```

Fonte: Própria.

Figura 4 – Exemplo de declaração da arquitetura no VHDL.

```
1 ARCHITECTURE behavioral OF full_adder IS
2 BEGIN
3   s<=x1 XOR x2 XOR cin;
4     cout<=(x1 AND x2) OR (x1 AND cin) OR (b AND cin
        );
5 END behavioral;
```

Fonte: Própria.

Segundo Cappelatti (2010), uma descrição em VHDL pode conter diferentes níveis de abstração:

- **Comportamental:** Permite descrever o circuito através de laços e processos. O circuito é definido em forma de um algoritmo, utilizando construção similares às utilizadas em linguagem de programação.
- **Transferência de registradores:** Englobando a representação do dispositivo em nível de transferência entre registradores, que consiste na utilização de funções lógicas combinacionais e de registradores.

- **Estrutural:** O circuito é descrito mais próximo da implementação real, podendo ser definidas portas lógicas com atrasos unitários ou com atrasos detalhados.

2.1.2 Portas lógicas

Segundo Idoeta e Capuano (1982), o conceito de portas lógicas é baseado na conhecida álgebra de Boole ou álgebra booleana, desenvolvida pelo matemático inglês George Boole em 1854. A álgebra booleana é representada por apenas dois valores, sendo eles o 0 e 1 e, através disso, expressa a relação entre entrada e saída dentro de um circuito. As portas lógicas podem ser construídas a partir de diodos, transistores e resistores interconectados de modo que a saída seja o resultante de uma operação lógica básica realizada sobre as entradas (TOCCI et al., 2003). A Figura 5 apresenta um exemplo de álgebra booleana sem a utilização de diagramas.

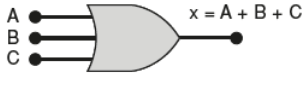
Figura 5 – Exemplo de algebra booleana, onde A=0, B=1, C=1 e D=1.

$$
\begin{aligned}
x &= \overline{A}BC(\overline{A+D}) \\
&= \overline{0} \cdot 1 \cdot 1 \cdot (\overline{0+1}) \\
&= 1 \cdot 1 \cdot 1 \cdot (\overline{0+1}) \\
&= 1 \cdot 1 \cdot 1 \cdot (\overline{1}) \\
&= 1 \cdot 1 \cdot 1 \cdot 0 \\
&= 0
\end{aligned}
$$

Fonte: (TOCCI et al., 2003)

Ainda segundo Tocci et al. (2003), devido a esta característica, a álgebra booleana possui três operações básicas, `OR` (ou), `AND` (e) e `NOT` (não). Tal conjunto de operações também é denominado de operações booleanas e, por meio da utilização de tabelas verdade é possível descrever as saídas baseando-se nas entradas e na operação aplicada. Cada operação booleana possui sua tabela verdade (Figura 6), bem como sua representação em forma de diagrama.

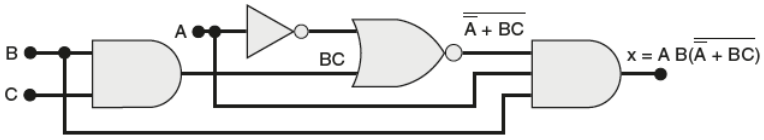
Figura 6 – Símbolo e tabela verdade para uma porta `OR` de três entradas.



A	B	C	x = A + B + C
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Fonte: (TOCCI et al., 2003)

Figura 7 – Representação de circuito lógico, utilizando portas lógicas.



Fonte: (TOCCI et al., 2003)

A Figura 7 apresenta um circuito lógico formado pelas portas básicas da álgebra booleana. O circuito apresenta três entradas representadas pelas letras A, B e C, mas também formada por duas portas `AND`, uma porta `NOT` e uma porta `OR`. Cada uma destas portas representando operações a serem realizadas, podendo receber como entrada um valor inicial ou resultante de outra porta, como no exemplo ocorre com a última porta `AND`.

Entretanto, segundo Kropf (2013), a análise de circuitos é extremamente complexa, resultando em um problema **NP-Completo**, algoritmos possuem tempo de execução exponencial. Contudo, este tempo de execução exponencial é uma complexidade exponencial para o pior caso, resultando crescimento exponencial da execução á medida que o tamanho do problema aumenta.

Ainda, segundo Kropf (2013) a avaliação das funções booleanas pode ser verdadeira (True) ou falso (False), com isso, para uma função com `n` variáveis, apresenta 2^n possibilidades de avaliação. Utilizando o exemplo da Figura 7 que apresenta 3 entradas, logo apresenta 8 possibilidades de avaliação ao final e este número aumenta à medida que o número de variáveis também aumenta.

2.1.3 Lógica proposicional

A lógica proposicional é uma linguagem formal onde é definida um alfabeto e conectivos proposicionais, e um conjunto de regras gramaticais, as quais serão utilizadas para construção das proposições (SOUZA, 2017). Porém apesar de importante, ela é limitada, não podendo expressar sentenças elementares importantes, tais como a da aritmética elementar. Por exemplo:

1. Todos são mortais.
2. Alguém é bondoso.

Na lógica preposicional, não poderiam ser analisadas, pois não teria como decompor ambas em sentença e assim não teria como analisar as diferenças entre ambas. Contudo, no exemplo: "Existem cavalos com patas verdes.", sentença determina que existem cavalos com a característica ou propriedade de terem as patas verde. A relação criada pela frase poderia a ser analisada pela lógica proposicional(ABE, 2002).

Segundo Souza (2017), alfabeto da lógica proposicional é formado por símbolo de pontuação, símbolos proposicionais e conectivos proposicionais, onde cada um apresenta uma função em específico, sendo:

- **Símbolos de pontuação:** Apenas dois símbolos de pontuação são utilizados o "("e o ")".
- **Símbolos proposicionais:** São utilizados para representar as proposições, onde um símbolo P pode ser utilizado para representar uma proposição qualquer, por exemplo: P="Está chovendo". O conjunto de símbolos proposicionais é infinito e enumerável, sendo possível representar infinitos e enumerável conjunto de proposições.
- **Conectivos proposicionais:** São os símbolos usando frequentemente na matemática. Os símbolos recebem a seguinte denominação:
  1. $\neg$: Representa a partícula de negação, ou seja, "Não".
  2. $\vee$: Representa a partícula "Ou"
  3. $\wedge$: Representa a partícula "E"
  4. $\rightarrow$: Representa a partícula "Se então ou implica".
  5. $\Leftrightarrow$: Representa a partícula "Se, e somente se".

Ainda segundo Souza (2017), existem as fórmulas que são constituídas de forma indutiva, a partir de símbolos do alfabeto conforme as regras apresentadas abaixo:

- Todo o símbolo preposicional é uma fórmula
- Se H é uma fórmula, então $(\neg H)$, a negação de H, é uma fórmula.
- Se H e G são fórmulas, então a disjunção de H e G, dada por $(H \vee G)$, é uma fórmula.
- Se H e G são fórmulas, então a conjunção de H e G, dada por $(H \wedge G)$, é uma fórmula.
- Se H e G são fórmulas, então a implicação de H e G, dada por $(H \rightarrow G)$, é uma fórmula. Neste caso, H é o antecedente e G consequente da fórmula.
- Se H e G são fórmulas, então a bi-implicação de H e G, dada por $(H \leftrightarrow G)$, é uma fórmula. Neste caso, H é o lado esquerdo e G o lado direito da fórmula.

2.2 Verificação de sistemas

No contexto de verificação de sistemas, faz-se necessário a diferenciação entre verificação e validação. Verificação e validação possuem o intuito de mostrar que determinado sistema funcione conforme o especificado, além de satisfazer as especificações do cliente (SOMMERVILLE, 2011).

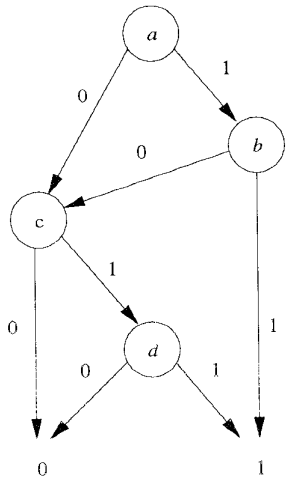
Segundo Sargent (2005), verificação é o processo de determinar se um modelo computacional obtido por discretização de um modelo matemático de um evento físico e o código que implementa o modelo computacional pode ser usado para representar o modelo matemático do evento com precisão suficiente. A validação é o processo de determinar se um modelo matemático de um evento físico representa o evento físico real com precisão suficiente.

A verificação consiste na identificação de erros e prováveis problemas que um componente pronto possa apresentar, enquanto a validação busca analisar se tal componente está seguindo os requisitos pré-definidos para sua construção (KOSCIANSKI; SOARES, 2007). O teste de programa, na qual o software é executado com dados de teste é a principal forma de validação, porém, técnicas de verificação, tais como, inspeção e revisões também podem integrar a etapa de validação (SOMMERVILLE, 2011).

2.2.1 Diagrama de decisão binária

Um diagrama de decisão binária (DDB) pode ser definido como um grafo acíclico para representação de funções booleanas. Existe uma ordem rigorosa na ocorrência de variáveis à medida que se percorre o grafo da raiz para a folha. Dado como exemplo a fórmula *f=(a∨b)∧(c∨d)* e utilizando a ordenação de variavél $a < b < c < d$, na Figura 8. Dada a atribuição de valores booleanos as variáveis a, b, c e d, pode-se decidir se a atribuição torna a fórmula verdadeira atravessando o início do gráfico na raiz e ramificação em cada nó. Definindo a, c e $d = 1$ e $c = 0$ leva a folha de rótulo 1, portanto, a fórmula é verdadeira para essa tarefa (CLARKE et al., 1994).

Figura 8 – Arvore de decisão binária



Fonte: (CLARKE et al., 1994)

O DDB permite o teste eficiente de satisfabilidade, validade e eficiência (KROPF, 2013). O uso de DDB em conjunto com *Model Checking* e técnicas de abstração permitiu o aumento na

capacidade de verificação, de modo que sistemas mais realistas pudessem ser verificados(BIERE et al., 2003)

2.2.2 Model Checking

Model checking é uma técnica de verificação formal que explora todos os possíveis estados do sistema, de modo a provar se um modelo satisfaz determinada propriedade. Pode ser aplicada em uma ampla gama de aplicações, tais como sistema embarcados, design de hardware e engenharia de software (BAIER et al., 2008).

Entre as vantagens da utilização de *Model Checking* estão:

- Possuem suporte à verificação parcial, em outras palavras, a verificação individual de propriedades, permitido foco primeiro as propriedades essenciais (BAIER et al., 2008).
- Não requer qualquer interação do usuário, nem um alto grau de especialização para o uso de *model checking* (BAIER et al., 2008).
- Possui como base a teoria de grafos, estruturas de dados e lógica no desenvolvimento da solução (BAIER et al., 2008).
- Útil para fins de depuração, pois fornece informações em caso de uma propriedade seja invalidada (BAIER et al., 2008).

Por outro lado, apresenta também algumas desvantagens:

- Sofre do problema de explosão de espaço de estados, ou seja, a quantidade de estados necessários pode ser maior que a memória disponível pelo sistema (BAIER et al., 2008).
- Faz a verificação apenas dos requisitos declarados, logo, não a garantia de integridade. Baseado nisso, a validade de propriedades não verificadas não pode ser julgada (BAIER et al., 2008).
- A verificação de modelos geralmente não é computável, isso deve-se a questões de decidibilidade utilizada (BAIER et al., 2008).

Outras técnicas foram implementadas em conjuntos com *Model checkings* com o intuito de aumentar a capacidade de verificação, como a técnica SAT na qual não sofre do problema de explosão de estados existente usando DDB. A técnica SAT em conjunto com *Model Checking* é conhecida como *Bounded Model Checking* (BIERE et al., 2003).

2.2.3 *Bounded Model Checking*

Segundo Rocha et al. (2015b), *Bounded Model Checking* (BMC) é um tipo especial de *Model Checking* que usualmente adota o método de satisfabilidade booleana, o qual tem sido introduzido como uma técnica complementar para diagrama de decisão binária para aliviar o problema da explosão de estados, visto que a técnica de *Model Checking* busca todos os estados possíveis para verificação.

Bounded Model Checking é uma técnica para a verificação de uma dada propriedade em uma determinada profundidade do sistema analisado. Logo, dado um sistema de transições M, uma propriedade ϕ, e um limite (*bound*) k, o BMC desenrola o sistema k vezes e traduz o sistema em uma condição de verificação(CV) ψ tal que ψ é satisfeito se e somente se ϕ tem um contraexemplo de profundidade menor ou igual a k (ROCHA et al., 2015b).

O *Extended SMT-Based Bounded Model Checker* (ESBMC) é um ferramenta que utiliza técnicas de *Bounded Model Checking* e solucionadores SMT(*Satisfiability Modulo Theories*) para verificação de programas em C/C++ (ROCHA et al., 2015a), permitindo a verificação aritmética de overflow e undeflow, segurança de ponteiros, limite de array e de gerar propriedades de segurança (CORDEIRO et al., 2012; ROCHA et al., 2015b).

No ESBMC, o programa analisado é modelado em um sistema de transição de estados, conforme a trupla: $M = (S, R, S_0)$, o qual é gerado um grafo de controle de fluxo (GFC), onde S representa o conjunto de estados, $R \subseteq SxS$ representa as transições e $S \subseteq S$ representa o conjunto de estados iniciais. Um estado $s \in S$ consiste no valor do contador de programa (PC) e os valores de todas as variáveis dos programas. O estado inicial S_0 atribui o inicio do programa GFC ao PC, desta forma o ESBMC identifica as transições, conforme a formula lógica, γ=(S_i,S_{i+1}) que captura as restrições sobre os valores correspondentes do PC e das variáveis do programa (CORDEIRO et al., 2012).

O motivo da escolha desta ferramenta, deve-se a mesma ter as funções necessárias para o desenvolvimento do método, como solucionadores SMT para verificação de assertiva e verificação em single-thread e multi-thread. Além de premiação em competições, como o SV-COMP, sendo premiado com 2 medalhas de ouro e duas de bronze em 2015 e uma medalha de ouro e uma de prata em 2016 (ESBMC, 2018).

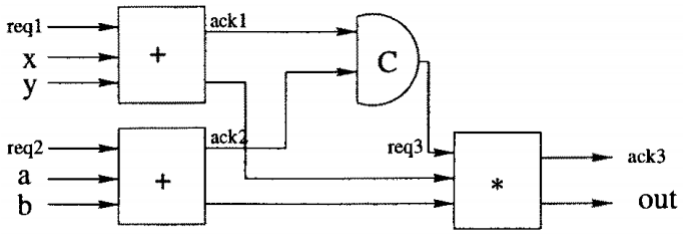
2.2.4 Redes de petri

Um rede de petri, N é um multigrafo direcionado, ponderado e bipartido, representado pela tupla $N = (P, T, I, O)$, onde P representa um conjunto finito de estados; T representa um conjunto finito de transições.; I representa a função de entrada; e O representa função de saída. Estas funções de entrada e saída determinam o fluxo dos token dentro da rede, sendo dos estados para as transições e das transições para os estados (HALDER; VENKATESWARLU, 2006).

Redes de petri são redes baseadas em abstração que pode ser usado como método de

modelagem, seja gráfica ou matemática (HALDER; VENKATESWARLU, 2006).Como ferramenta gráfica são utilizadas como comunicação visual, como fluxogramas, por exemplo, e como ferramenta matemática podem configurar modelos matemáticos que regem o comportamento dos sistemas (MURATA, 1989).

Figura 9 – Rede de petri

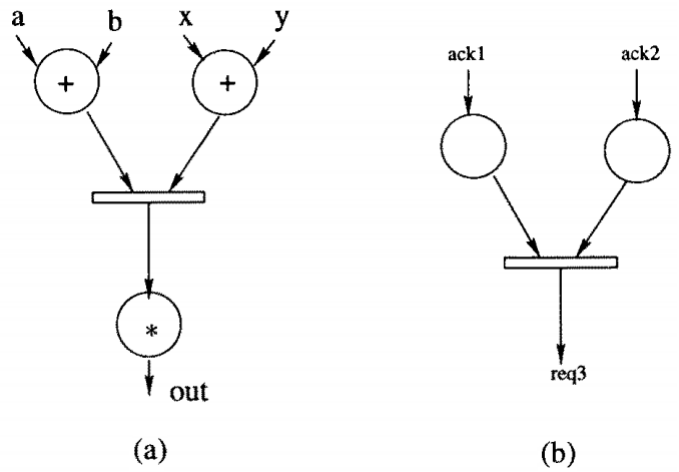


Fonte: (YAKOVLEV; KOELMANS, 1996)

A Figura 9 é a representação de um circuito que realiza a operação $out = (x+y)*(a+b)$, onde o *dathpath*, representação do elementos do circuito e interligações, é formado por dois somadores que recebem as entradas a, b, x e y e um multiplicador que recebe o resultado destas somatórias. Também possuem requisições e confirmações representados por req e ack, respectivamente, onde as operações somente são realizadas após a requisição das mesma e a porta AND C representa a confirmação das somatórias e requisição da multiplicação(YAKOVLEV; KOELMANS, 1996).

O funcionamento do circuito é realizado de modo a realiza as operações somente se houver a requisição das operações. Após a realização as somas, um sinal de confirmação é enviado para a porta and C e somente com ambas as confirmações é realizado a requisição da multiplicação. Para a multiplicação tem as entradas da requisição como o resultado das multiplicações e gera a confirmação e a resposta *out* (YAKOVLEV; KOELMANS, 1996).

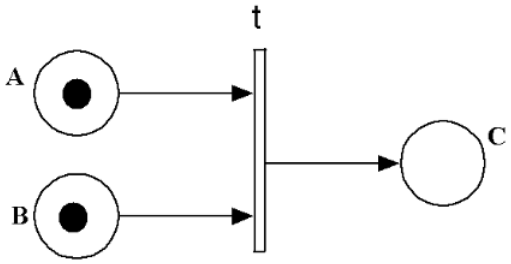
Figura 10 – Rede de petri



Fonte: (YAKOVLEV; KOELMANS, 1996)

A Figura 10(a) é uma rede de petri que representa as operações das operações que são realizadas, deste a soma até a saída e a Figura 10(b) representa o fluxo de controle as operações formado pelas confirmações dos somatórios e da requisição da multiplicação (YAKOVLEV; KOELMANS, 1996). Segundo Jain et al. (2010), umas das vantagens do uso de redes de petri é que um mesmo modelo pode ser usado pra análise de propriedades comportamentais e avaliação de desempenho, bem como para construção sistemática de simuladores e controladores de eventos discretos.

Figura 11 – Rede de petri da porta AND



Fonte: (JAIN et al., 2010)

Na Figura 11 é apresentado um exemplo de representação da porta `AND` em rede de petri. A porta `AND` apresenta, pelo menos, duas entradas e uma saída. Sendo que, está apenas será verdade, se ambas as entradas forem verdadeiras. Dessa forma, a transição `t` apenas será executada, se ambas as entradas `a` e `b` tiverem um *token* cada, simbolizando, assim, verdade e, a saída terá um *token* após a transição, mostrando-a como verdadeira.

2.2.5 Verificação de hardware

A verificação de hardware, por meio da simulação, visa assegurar que uma implementação, descrição do hardware em qualquer nível de abstração da hierarquia de hardware, atinja suas especificações, propriedades que devem ser respeitadas para que a corretude do mesmo seja comprovada (GUPTA, 1992). A Verificação formal de hardware consiste na utilização de técnicas para corretude de circuitos lógicos, ou seja, consiste na utilização de modelos matemáticos para descrição de propriedade e/ou de comportamento de um dado sistema (KROPF, 2013).

Entre as abordagens utilizadas na verificação formal de hardware consiste tanto na implementação quanto a especificação estarem descritas em lógica formal, neste caso a corretude será obtida através da comprovada relação entre a implementação e a especificação (SEGER, 1992). A especificação formal consiste na descrição do comportamento, bem como, das propriedades do sistema em linguagem matemática, tornando-se crucial para o processo de verificação. Na implementação formal, o nível de abstração, tais como, *Gate level* e RTL(Figura 12) são importantes informações para o desenvolvimento do formalismo, bem como as classes, por exemplo se o circuito é sequencial ou combinacional, se utiliza pipeline, etc (KROPF, 2013).

Figura 12 – Exemplo de multiplexador de duas entradas utilizando RTL em VHDL.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.all;
3
4  ENTITY mux2 is
5    PORT (opt, a, b: in STD_LOGIC;
6     f : out STD_LOGIC);
7  END mux2;
8
9  ARCHITECTURE behaviour OF mux2 IS
10 BEGIN
11   PROCESS(opt, a, b)
12     BEGIN
13          IF opt = '1' THEN
14               f <= a;
15          ELSE
16               f <= b;
17          END IF;
18     END PROCESS
19 END behaviour ;
```

Fonte: Própria.

A utilização de *Model Checking* também se faz presente nos modelos formais de verificação de hardware, onde apenas é utilizado o comportamento dos circuitos na verificação, de modo a verificar a se as propriedades presentes são satisfeitas. Para isso, são utilizados os conceitos da lógica proposicional, que por meio da utilização de fórmulas que representem as propriedades dos estados, é possível realizar as verificações necessárias (SEGER, 1992).

O BMC (BIERE et al., 1999) apresenta uma abordagem de verificação de hardware, além de testes comprovando a melhoria na utilização de solucionadores SAT para testes de satisfabilidade e mostrando uma performance significativamente melhor que o uso de DDB(BIERE et al., 1999). Baseado nisso, este trabalho fará uso de solucionadores SAT, sendo mostrado como melhor alternativa para análise.

2.2.6 Verificação formal de software

Observa-se que o uso de verificação de software tem sido feito aplicado em muitas áreas, tais como, infraestruturas industriais de missão crítica e de segurança. Logo, existe a necessidade de garantir a corretude destes sistemas. Devido a isso, a verificação formal tem sido utilizada em três principais abordagens, sendo elas (COUSOT; COUSOT, 2010; D'SILVA et al., 2008):

- **Métodos dedutivos:** Produzem provas matemáticas formais de corretude usando provadores de teoremas ou assistentes de prova para execução da prova e necessitam da interação humana para prover os argumentos (COUSOT; COUSOT, 2010);
- **Verificação de modelos:** Exploram exaustivamente modelos de execuções de programa, que podem ser sujeitos a explosão combinatória, necessário a intervenção humana para geração dos modelos, (ROCHA et al., 2015b);
- **Analise estática:** Engloba diversas técnicas para calcular automaticamente informações sobre o comportamento de um programa sem executá-lo, sendo utilizado em otimização de código e verificação em compiladores (D'SILVA et al., 2008).

Segundo Rocha et al. (2015b), na verificação formal de software apresentam-se dificuldades, tais como: quais as propriedades são de interesse na verificação em tempo de execução; e impossibilidade matemática de provar a corretude de propriedades não triviais no comportamento de programas(COUSOT; COUSOT, 2010).

2.2.6.1 Verificação usando Indução-K

Segundo Gadelha et al. (2017), o algoritmo de indução-k é uma técnica de verificação efetiva implementada em vários verificadores de modelos de software com o objetivo de comprovar a correção parcial de um grande número de programas e propriedades. O método de indução de k tem sido aplicado com sucesso em verificar projetos de hardware (representados como máquinas de estados finitos) usando um solucionador SAT (GADELHA et al., 2019).

Segundo Gadelha et al. (2017), a forma básica de indução-k consiste em duas etapas, sendo elas o caso base e o passo de indução. Para critério de explicação, dado um sistema de transição finito M, onde $I(s)$ sendo um conjunto de estado iniciais e $T(s, s_0)$ sendo a relação

de transição entre os estados. Seja $\phi(s)$ denotar estados que satisfaçam uma propriedade de segurança ϕ, este sistema pode ser definido pela seguintes equações:

$$Base_k = I(s_1) \wedge T(s_1, s_2) \wedge ... \wedge T(s_{k-1}, s_k) \wedge (\neg\phi(s_0) \vee ... \vee \neg\phi(s_k)) \tag{2.1}$$

$$Step_k = \phi(s_1) \wedge T(s_1, s_2) \wedge ... \wedge \phi(s_k) \wedge T(s_k, s_{k+1}) \wedge \neg\phi(s_{k+1}) \tag{2.2}$$

A interpretação intuitiva das fórmulas apresentadas acima é a seguinte: no caso base (Equação 2.1), será verificado se ϕ detém em todos os estados alcançáveis a partir de um estado inicial dentro de k passos, levando em consideração que $k \geq 0$; e na etapa de indução (Equação 2.2), o objetivo é verificar que sempre ϕ detém em k estados consecutivos $s_1, \ldots, s_k$, ϕ também mantém no próximo estado s_{k+1} do sistema. Um algoritmo pode então ser criado a partir dessas duas fórmulas, que desenrola o sistema de forma incremental e verifica se o caso base k é satisfatório ou se o etapa indutiva k é insatisfatório para determinar a terminação do sistema de transição de estado. Em particular, se o caso base k se tornar satisfatório no passo k, então encontramos uma violação da propriedade, pela demosntração de um contraexemplo. Se etapa indutiva k é insatisfatório no passo k, então a propriedade é válida (GADELHA et al., 2017).

2.2.7 Linguagem de especificação de propriedades

Linguagem de especificação de propriedades, do inglês *Property Specification Language* - PSL, é uma notação formal para especificação de comportamento em sistemas eletrônicos, compatível com as linguagens VHDL(IEEE, 2011), Verilog(IEEE, 2006) e SystemC(IEEE, 2012). Destina-se a utilização para a especificação funcional, mas também para entrada de ferramentas de verificação. A especificação do PSL é a utilização de assertivas sobre propriedades de um sistema, sendo estas propriedades constituídas de três elementos: expressões boolenas, expressões sequenciais e operadores temporais (IEEE, 2010).

Segundo (IEEE, 2010), o PSL foi totalmente desenvolvida com o intuito fornecer leitura e escrita de fácil entendimento, sintaxe concisa, semântica formal e rigorosamente bem definida e ser matematicamente precisa, visto a utilização da mesma em processo de verificação.

O PSL pode ser utilizado para capturar requisitos relativos ao comportamento total do projeto, bem como, sobre o modo que o mesmo deverá operar no ambiente, mas também para capturar requisitos comportamentais e os pressupostos que surgem durante o processo de design. Ambas podem ser utilizadas para verificação de sistemas.

O PSL possui 4 camadas, sendo:

- **Camada booleana:** É utilizada para criar as expressões que são usadas pelas outras camadas. As expressões de camada booleana são avaliadas em um único ciclo de avaliação e obedecem às características da linguagem na qual foi escrita. Na Figura 13 apresenta um expressão *if* usando PSL nas linguagens VHDL e Verilog(IEEE, 2010).

- **Camada temporal:** É usado para descrever as propriedades do circuito. Pode descrever propriedades envolvendo relações temporais e são avaliadas em uma serie de ciclos de avaliação. Na Figura 14 apresenta a estrutura `@clk`, permitindo que determinada operação seja executada em tempos específicos de execução ou em ciclos específicos.(IEEE, 2010).

- **Camada verificação:** É usada para informar de que modo as propriedades na camada temporal devem ser utilizadas pelas ferramentas de verificação. Na Figura 15 apresenta o uso de **assert** que busca provar a propriedade e **assume** que a ferramenta deve assumir como verdadeiro a propriedade (IEEE, 2010).

- **Camada de modelagem:** É usada para modelar comportamento de entradas, no caso de ferramentas que não utitizem casos de teste e para modelar circuitos adicionais para verificação, mas que não fazem parte do circuito principal(IEEE, 2010).

Figura 13 – Exemplo de PSL em VHDL e Verilog.

```
1  (a &   (a-1)) == 0          // Verilog
2  (a and (a-1)) =  0          -- VHDL
```

Fonte: (DUOLOS, 2018)

Figura 14 – Exemplo da camada temporal em PSL.

```
1  (always req -> next (ack until grant)) @clk
```

Fonte: (DUOLOS, 2018)

Figura 15 – Exemplo da camada de verificação em PSL.

```
1  vunit my_properties(myVerilog.instance.name){
2  assert (always req -> ack) @ clk;
3  assume (never req && reset)@ clk;
4  }
```

Fonte: (DUOLOS, 2018)

2.2.8 Verificação baseada em assertivas

Verificação baseada em assertivas, do inglês *Assertion-Based Verification* - ABV, é um paradigma de verificação igualmente adequado para verificação formal e abordagens baseadas em simulação (BOULE; ZILIC, 2005) e em conjunto com a técnica de PSL tem ganhado aceitação como um método essencial para verificação funcional do hardware (DAHAN et al., 2005).

Na ABV as assertivas são declaração adicionadas ao projeto com o intuito de especificar o comportamento do projeto (BOULE; ZILIC, 2005), podendo escritos em PSL ou em SVA (*System Verilog Assertions*). No caso deste projeto será utilizado a PSL, devido a mesma ser utilizada juntamente com a linguagem VHDL. Com a utilização de ABV o circuito pode ser verificado usando técnicas de simulação e/ou verificação, por exemplo, *model checking*, para garantir que o mesmo esteja de acordo com o pretendido projeto (DAHAN et al., 2005).

Na Figura 22 apresenta o modelo de assertiva utilizada no projeto, utilizando de um modelo próprio para assertivas, unindo as funcionalidades já existentes nas assertivas do VHDL, mas também novas (serão apresentadas nas próximas seções), fornecendo assim mais propriedades a serem analisadas. A assertiva apresentada na **linha 12** da Figura 16 faz a verificação se a porta AND, ao ter entradas de dois valores 1, a saída também será 1, caso não seja será relatado erro de verificação.

Figura 16 – Exemplo de declaração de assertiva no VHDL.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  entity AND_ent is
4  port(    x,y: in bit;
5           F: out bit
6  );
7  end AND_ent;
8
9  architecture behav1 of AND_ent is
10 begin
11   --@c2vhdl:ASSERT
12     --assert (F='0')
13     --report "O valor de F é diferente de 1"
14     --severity ERROR;
15     --@c2vhdl:END
16   process(x, y)
17       begin
18           if ((x='1') and (y='1')) then
19               F <= '1';
20           else
21               F <= '0';
22           end if;
23       end process;
24 end behav1;
```

Fonte: Própria.

2.2.9 Propriedades de segurança

Propriedades de segurança é um meio de validação do comportamento de um determinado sistema, de modo que se uma dada propriedade de segurança for violada, então através de uma execução finita, é possível verificar tal erro. Algumas propriedades de segurança, no entanto, podem impor requisitos em fragmentos de caminho finito e não podem ser verificadas considerando apenas os estados alcançáveis. (BAIER et al., 2008).

Segundo (CLARKE et al., 2003), podemos definir uma propriedade de segurança como: *dado um sistema de transições* $ST = (S, S_0, E)$*, seja um conjunto* $B \subset S$ *que especifica um conjunto de maus estados tais que* $S_0 \cap B = \emptyset$*, pode-se dizer que* ST *é seguro com relação a* B*, denotado por* $ST \models AG\neg B$ *se não existe um caminho no sistema de transição do estado inicial* S_0 *até o estado* B*, de outro modo é dito que* ST *não é seguro*.

2.3 Técnicas de compiladores

Compiladores são sistemas de software utilizados para tradução de uma linguagem de programação para outra, ou seja, o programa da linguagem fonte é lido e traduzido para um código equivalente na outra linguagem, ou linguagem alvo. Geralmente utilizada para transformar trechos de código em uma linguagem a ser executada pelo computador (AHO et al., 2007).

Diversas técnicas podem ser utilizadas nas diversas etapas, até que o código seja traduzido, tais como LL(1) e LR na análise sintática; eliminação de código morto; propagação de constante e *Peephole* na otimização de código (AHO et al., 2007). Neste projeto serão focadas as áreas de transformações de código e otimização de código.

2.3.1 Transformações de código

Transformações de código correspondem a uma complexa função que envolve analise de fluxo de dados e modificação completa do programa que são parte integral da alta performance e sistemas computacionais. Podem ser classificados em transformações escalares que reduzem o número de instruções a serem executadas no programa ou transformações paralelas que maximizam o paralelismo (SRIKANT; SHANKAR, 2002).

A Figura 17 representa o trecho de código apresentado na Figura 22. O trecho representa o comportamento das entradas, neste caso de um código da porta `AND`. E a figura Figura 18 representa a tradução deste trecho em código na linguagem C. Esta tradução foi realizada pela ferramenta V2C (GATTI; GHEZZI, 1995), utilizada neste projeto.

Segundo Cousot e Cousot (2002), a metodologia de transformação de código fornece ferramentas para o desenvolvimento de programas a partir da especificação e para verificação do programa. As técnicas de transformação código fornecem ferramentas para otimização, como, por exemplo, a otimização de cache(JIN et al., 2001) e a propagação de constantes(KILDALL, 1973);

Figura 17 – Trecho de código em VHDL representando porta `AND`.

```
process(x, y)
    begin
        if ((x='1') and (y='1')) then
            F <= '1';
        else
            F <= '0';
        end if;
    end process;
```

Fonte: Própria.

Figura 18 – Trecho de código traduzido para linguagem C.

```
/* Start of Translation */
/* p0: */
if (chg[x] || chg[y]) {
    if (((old[x]==1) && (old[y]==1))) {
      new[f]=1;
    }
    else {
      new[f]=0;
    }
}
/* End of Translation */
```

Fonte: Própria.

para customização de software e, para isso, podem utilizar-se da engenharia reversa(YANG; SUN, 1997) e da aplicação de políticas de segurança(ERLINGSSON; SCHNEIDER, 2000); e para compilação.

2.3.2 Otimização de código

A otimização de código consiste na melhoria do código intermediário gerado pelos algoritmos de compilação com o objetivo de obter um menor tempo de execução do programa, porém outros fatores podem ser introduzidos, como algoritmos menores. Este fato não implica em afirmar que o melhor código é gerado, visto que dada a complexidade, raramente pode ser obtido que o código gerado pode ser o melhor possível (AHO et al., 2007).

Segundo Aho et al. (2007), otimização local de código consiste na otimização de um bloco básico de código, ou seja, a otimização ocorre em um trecho especifico de código. Os blocos básicos são trecho onde não ocorrem saltos ou *loops*, para nenhuma outra parte do código. A otimização global de código é baseada na análise do fluxo de dados, de modo que ocorra as eliminação de instruções não necessárias ou substituição mais fácil. A otimização global consiste

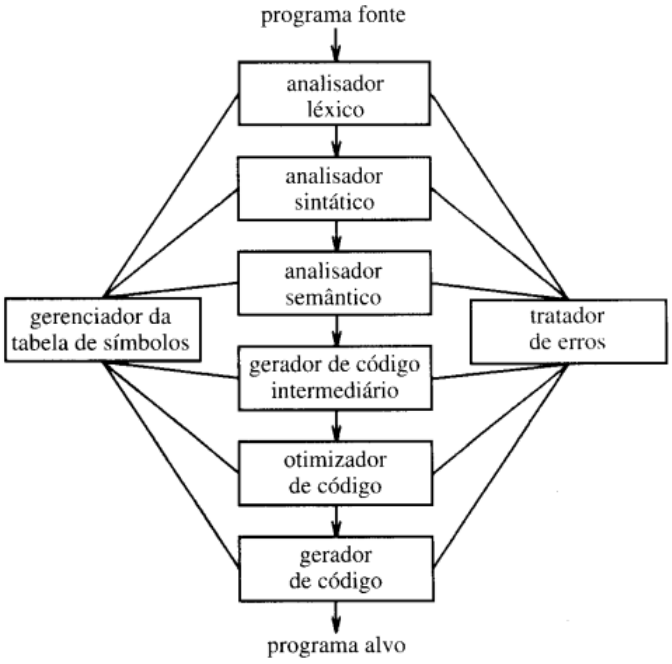
na análise completa do código.

2.3.2.1 Gerador de código

O gerador de código é responsável pela tradução final do programa-fonte para o programa-alvo, o mesmo recebe como entrada um código intermediário juntamente com a tabela de símbolos. Faz-se necessário que o programa-fonte tenha sido analisado sintática e lexicamente, assim evitando que erros possam ser passados para o programa-alvo, por outro lado a existência de um nível de otimização, faz-se opcional neste caso (AHO et al., 2007).

O gerador de código possui três tarefas primarias, sendo as mesmas seleção de instrução, alocação de registros e atribuições. A seleção de instruções envolve a escolha das instruções apropriadas da máquina-alvo para implementar as declarações do código intermediário. A atribuição e atribuição de registros envolve a decisão sobre os valores a serem registrados nos registros (AHO et al., 2007).

Figura 19 – Fases de um compilador



Fonte: (AHO et al., 2007)

Conforme apresentado na Figura 19 e, segundo Aho et al. (2007), a geração de código ocorre em dois momentos. Sendo que, inicialmente ocorre a geração de código intermediário, que tem como propriedades: buscar uma fácil produção e a tradução para o programa alvo. Posteriormente, ocorre a geração de código, onde as localizações de memória são selecionadas para cada variável utilizada no programa e, as instruções intermediárias são traduzidas numa sequência de instruções de máquina, que realizam a mesma tarefa.

2.4 Resumo do capitulo

Neste capítulo foi apresentado os principais conceitos utilizados e necessários para o desenvolvimento do método. Entre esses conceitos foi apresentado as linguagens de descrição de hardware que foram desenvolvidas para projetar circuitos lógicos e entre elas, a linguagem VHDL que foi a linguagem escolhida para este projeto. Também foi apresentado a verificação de sistemas, sendo estes sistemas de hardware e software. Entre as técnicas apresentadas como *bounded model checking* que gera a exploração de possíveis estados aliado a técnica de satisfabilidade booleana e diagrama de decisão binária. E por fim, foi apresentado técnicas de compiladores e otimização de códigos, extremamente necessárias para transformação de código, uma das principais técnicas utilizadas para o desenvolvimento do método.

3 TRABALHOS CORRELATOS

O capítulo a seguir apresenta o método de pesquisa utilizado para o levantamento bibliográfico, os resultados e os artigos e suas contribuições para o desenvolvimento do projeto descrito no método proposto.

3.1 Revisão sistemática

A revisão sistemática consiste em um método de identificação, análise e interpretação de pesquisas relevantes em determinada área ou questão de pesquisa(KITCHENHAM, 2004). A aplicação da revisão sistemática requer que seja seguido um conjunto bem definido e sequencial de passos e, por conta disso, é necessário um esforço considerável, se comparado a uma revisão informal a literatura(MAFRA; TRAVASSOS, 2006).

Vale ser observado que o objetivo do trabalho é realizar um estudo exploratório de caracterização de área, podendo assim dizer que esta revisão sistemática se caracteriza como uma quasi-sistemática(TRAVASSOS et al., 2008). Baseado nisso, as seguintes questões foram formuladas:

- **Q1:** Quais são os métodos para verificação de circuitos lógicos descritos na linguagem de programação VHDL?
  - **Q1.1:** Foi desenvolvido e está disponível alguma ferramenta para aplicação do método?
  - **Q1.2:** Qual a técnica de exploração de estados para circuitos lógicos?
  - **Q1.3:** O método proposto é baseado em técnicas de verificação de software?
  - **Q1.4:** Como o método proposto valida pré e pós condições no programa?
  - **Q1.3:** Foi utilizado algum *benchmark* de programas em VHDL para experimentação e o mesmo encontra-se disponível?
  - **Q1.4:** Quais as perspectivas futuras para melhorar a aplicação do método proposto?

A biblioteca digital utilizada para pesquisa foi a Scopus, acessível em http://www.scopus.com, que possui uma base de dados com mais de 22.800 títulos, abrangendo as áreas de tecnologia, medicina, ciências sociais, bem como, atualizações diárias em seus arquivos.

Devido ao tempo necessário para realização completa de todas as etapas, apenas o primeiro filtro da revisão sistemática foi realizado. Em contrapartida, cinco artigos foram selecionados e servirão como ferramentas de estudo para o desenvolvimento deste projeto.

Mais informações sobre como foi executada a revisão sistemática se encontra na Seção 7.1. Os cinco artigos selecionados serão apresentados nas seções a seguir. Estes trabalhos são relevantes para o desenvolvimento do método proposto.

3.2 V2c-A verilog to C translator

O artigo de Mukherjee et al. (2016b) apresenta uma ferramenta implementada em C++ chamada `v2c`, utilizada na transformação de código da linguagem de descrição Verilog para a linguagem de programação C. A ferramenta é executada a nível de palavra, visto que, esta abordagem garante um aumento na escalabilidade, mas também proporcionando que técnicas, como a interpolação(BEYER; KEREMOGLU, 2011), possam ser utilizadas para verificação, algo inviável, caso a tradução fosse executada a nível de bit. O sistema recebe como entrada um código em Verilog, onde são aplicadas regras semânticas e mapeados os bits de operação, é gerado um código em linguagem C a nível de palavra chamado software *netlist*.

A contribuição do trabalho de Mukherjee et al. (2016b) consiste na utilização de transformações de código como base principal e, partindo deste pressuposto, torna-se vantajoso devido a utilização de outras ferramentas que não apresente suporte a certas linguagens. Tais como a ferramenta ESBMC, utilizada neste trabalho, que não apresenta suporte a VHDL, porém apresenta suporte às linguagens C/C++, em outras palavras, a principal contribuição de tal artigo foi a utilização de traduções de códigos e utilização deste conceito como ponto de partida para a análise de circuitos desenvolvidos no método.

3.3 Unbounded safety verification for hardware using software analyzers

No artigo de Mukherjee et al. (2016a) foi apresentado uma metodologia que aborda a utilização de técnicas e analisadores de software, com o objetivo de abordar a análise de circuitos e, com isso, traçar um paralelo entre as abordagens. Para testes, foram utilizadas três metodologias de análise usando interpolação (BEYER; KEREMOGLU, 2011): interpolação abstrata(BLANCHET et al., 2003), *k-induction*(DONALDSON et al., 2011) e tecnologias híbridas.

Como resultado dos testes, foram observadas as principais causas de erros, como por exemplo, bits não precisos e, no caso das operações a *bit-level*, ocorria a perda de informações. Também foi observado que apesar de não serem otimizados para análises de hardware, alguns analisadores de software podem, dependendo da técnica utilizada no analisador, serem utilizados para análise de hardware(MUKHERJEE et al., 2016a).

O fato de analisadores de software possuírem a capacidade de utilizar hardware, deve-se

ao uso de técnicas de análise a nível de bit. Contudo, estas análises são menos precisas e menos testadas em software, além de usarem abstrações numéricas, o que gera perda de informações e, por isso, a grande quantidade de resultados errados gerados (MUKHERJEE et al., 2016a).

A utilização de técnicas de análise de software para o desenvolvimento de análise de *hardware* é o foco a ser abordado neste trabalho de maneira prática, utilizando os princípios analisados por (MUKHERJEE et al., 2016a). Neste contexto, utilizando um *Bounded model checking*, como o ESBMC, é possível utilizar técnicas como *k-induction* para a análise conforme apresentado no artigo.

3.4 Formal verification of timed VHDL programs

No trabalho apresentado por Bara et al. (2010) é proposta uma abordagem para a análise de tempo, relacionada a cada porta lógica dentro de um dado circuito analisado. A abordagem apresenta a tradução de um circuito lógico codificado em VHDL, para um formalismo baseado em autômato de tempo(ALUR; DILL, 1994). Tal formalismo é representado por um autômato de estados finitos, com relógios simbólicos que evoluem em taxa uniforme. A tradução é executada de modo automático, baseada na emulação da propagação de cada transação ao longo de cada sinal, ou seja, são autômatos programados e cronometrados do circuito. Após a tradução para autômato, a análise é feita pela ferramenta UPPAAL(LARSEN et al., 1997) que é um *model checking* de verificação de propriedades de tempo. Seguindo esta metodologia, a análise pode ser extraída de modo independente de cada bloco, desta forma, a análise é feita de modo mais preciso, podendo ser analisados inúmeros fatores, tais como limites de intervalo e sinal de correlação.

O artigo de Bara et al. (2010) apresenta uma abordagem interessante que pode ser adicionado ao método, pois além da utilização da linguagem VHDL, também apresenta um novo modelo de análise, baseado na utilização de autômato de tempo para verificação de circuitos e, desta forma, identificar a estabilidade dos sinais de entrada e saída baseados nos *delays* das portas do circuito.

3.5 On the use of assertions for embedded-software dynamic verification

Em Guglielmo et al. (2012) é apresentado uma metodologia para a integração dinâmica de *Assertion-Based Verification* para várias fases da análise da verificação de fluxo em sistemas embarcados, por exemplo, emulação, diagnósticos e *Debug*, mas também uma ferramenta chamada *RadCheck*. O método de aplicação, chamado *V-model* é dividido em fases de verificação em paralelo com as fases de *design* do circuito. Com base no *V-model*, o método é aplicado, iniciando com o nível de sistema e as especificações do sistema, neste caso especificado usando PSL. Neste nível é especificado todas as funcionalidades gerais da aplicação. No nível de

integração visa investigar problemas de interação que possam ocorrer, definindo propriedades que cobrem incrementalmente as unidades estruturais interativas de aplicação. No nível de unidade descrevem comportamentos internos e são definidas através de parâmetros de entrada/saída das unidades e estruturas de dados internas.

A contribuição deste projeto está relacionada a utilização de assertivas no contexto da verificação de software. As assertivas são o principal meio de análise proposto neste projeto, visto que estas mesmas assertivas serão analisadas pelo ESBMC. Aliado a isso em Guglielmo et al. (2012) é apresentado modelos de utilização de assertivas no processo de verificação de hardware, e tais assertivas foram adaptadas para um modelo proposto neste trabalho.

3.6 Incorporating efficient assertion checkers into hardware emulation

No trabalho de Boule e Zilic (2005) é apresentado uma ferramenta de geração de assertivas no contexto da emulação de circuitos, de modo que estas assertivas descrita em PSL possam transformadas para o modelo de linguagem de descrição de hardware. As instruções são transformadas percorrendo a *parse tree* na qual estão armazenadas. Cada subexpressão é conectada recursivamente pelo uso de pré-condição e resultado, desta forma se determinado nó na *parse tree* aceita um sinal de pré-condição de seu pai, se transforma recursivamente no circuito adequado e, em seguida, retorna o sinal de resultado para o pai.

O trabalho de Boule e Zilic (2005) consolida e apresenta aspectos importantes, tais como a utilização de uma ferramenta para geração de assertivas, o que torna-se, no contexto deste projeto, extremamente importante visto que o modelo adotado atualmente consiste tanto na geração automática das assertivas, bem como a inserção manual por parte do usuário.

3.7 Tabela comparativa

A tabela apresenta os artigos resultantes da revisão sistemática em comparação ao trabalho apresentado. Desta forma é possível apresentar a contribuição de cada um no desenvolvimento do método proposto. Na primeira coluna é apresentado os artigos para analise, na segunda, terceira e quarta coluna apresenta as técnicas utilizadas no trabalho, bem como se os artigos utilizaram ou não desta técnica.

Por meio da Tabela 1 é possível demonstrar que nem todos os elementos apresentados foram utilizados por todos os artigos, contudo, através de análise experimental foi possível comprovar que estes elementos podem relacionar-se entre si. Desta forma também comprova a contribuição dos mesmo para o desenvolvimento do método.

Com relação ao artigo **Formal verification of timed VHDL programs** o mesmo apre-

senta outra abordagem de análise para a ferramenta proposta, porém, ainda não foi adicionada. A utilização de autômatos de tempo permite a utilização de outras ferramentas que propicie novas abordagens de análise.

Tabela 1 – Tabela de apresentação

Artigos	**Transformação de código**	**Utilização de assertivas**	**Analise de código**
V2c - A verilog to C translator	Sim	Não	Não
Unbounded safety verification for hardware using software analyzers	Não	Não	Sim
Formal verification of timed VHDL programs	Não	Não	Não
On the use of assertions for embedded software dynamic verification	Não	Sim	Sim
Incorporating efficient assertion checkers into hardware emulation	Não	Sim	Sim
Ferramenta apresentada	Sim	Sim	Sim

Fonte: Própria

3.8 Resumo do capitulo

Neste capítulo foi apresentado de maneira resumida o método de levantamento bibliográfico utilizado, revisão sistemática, e algumas das etapas realizadas. Também foi apresentado 5 artigos que foram extraídos do resultado da revisão sistemática e a contribuição de cada artigo no desenvolvimento do método. Ao final foi apresentado uma tabela comparativa entre os artigos, explicitando a contribuição de cada artigo de modo mais visual.

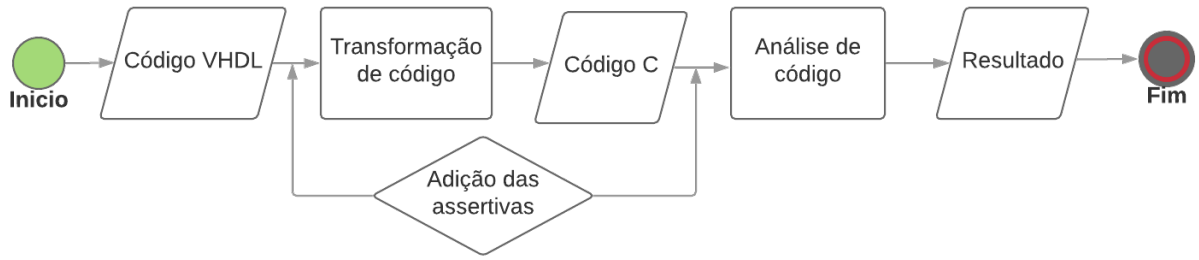
4 MÉTODO PROPOSTO

Neste capítulo descreve o método proposto utilizado no desenvolvimento deste projeto. Inicialmente será apresentado o modelo geral e posteriormente as duas abordagens utilizadas com base neste método, além dos pontos principais de cada uma delas. Ambos os métodos são baseados em transformações de códigos para análise de circuitos lógicos descritos em VHDL utilizando as técnicas de verificação *Bounded Model Checking*(ESBMC, 2018) e Indução-K(GADELHA et al., 2019), neste caso, implementadas pelo *model checker* ESBMC.

4.1 Visão geral

O método proposto consiste na análise de códigos VHDL através da inserção de assertivas ao código, de modo a validar ou mesmo gerar a ocorrência de erros que possam ocorrer durante a execução. Aliado a inserção de assertivas, também é utilizado a técnica de transformação de código, convertendo o mesmo de VHDL para linguagem C.

Figura 20 – Fluxograma do método proposto.



Fonte:Própria

Partindo do citado anteriormente e conforme mostrado na Figura 20, o método se inicia com o código em VHDL a ser analisado. Nenhuma análise (validação de propriedades) é realizada diretamente sobre o código VHDL, sendo apenas sobre o código já convertido para linguagem C, devido a existência de vários analisadores de código que são utilizados para identificação de erros nesta linguagem, por exemplo, a ferramenta ESBMC.

As assertivas podem ser inseridas antes ou depois da transformação de código. Independente do caso adotado, a mesma devem seguir o padrão aceito pela ferramenta de analise utilizada, caso contrário, a mesma não apresentará a eficácia necessária. A escolha do momento de inserção das assertivas depende principalmente do modo como a tradução ta sendo realizada e as capacidades da ferramenta de tradução em converter estas assertivas.

Sendo as assertivas inseridas antes da transformação, a mesma é traduzida para linguagem C, seja pela ferramenta, na forma de comentário ou por meio de alguma instrumentação que venha a ser realizada no código. Esta instrumentação não altera a lógica do código, pois desta forma a mesma alteraria o código que pretende ser analisado. Caso seja feita após a tradução, a mesma já pode ser inserida ao código diretamente em linguagem C, visto que o código a ser analisado já foi convertido.

4.2 Abordagens desenvolvidas

Inicialmente será apresentado dois métodos de tradução que foram desenvolvidos durante a elaboração do trabalho. Ambos os métodos foram baseados na estrutura geral do método, utilizando o sistema de assertivas, ferramentas de transformação. Para o total funcionamento das abordagens, são realizadas instrumentações para adaptação do códigos com as ferramentas e também na execucão das mesmas.

A parte referente a analise do código será apresentada na subseção 4.2.3, visto que ambas as abordagens apresenta a mesma ferramenta para analise e estruturas. Será demonstrado de que forma a ferramenta foi utilizada para verificação da corretude do código, através das assertivas.

Visando a explicação das atapas do método proposto, o código na Figura 21 será utilizado, este código é um conceito básico de ula, formado apenas pelas portas AND e OR, além de um inversor o qual inverte o sinal de entrada, caso seja necessário. A saída deste inversor é ligada as portas AND e OR, e a saída desta porta é ligada a um multiplexador de duas entradas. Ao final, a saída do multiplexador será um dos sinais de entrada, ou seja, ou o sinal enviado pela porta AND ou pela porta OR e este sinal de saída será determinado de acordo com o sinal de escolha enviado ao multiplexador.

Figura 21 – Exemplo de código VHDL de ULA com portas AND e OR.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  ENTITY Ula_tcc IS
5  PORT(A,B,Binvertido,Op1:IN std_logic;
6       Resultado:OUT std_logic);
7  END Ula_tcc;
8
9  ARCHITECTURE Ula_tcc_behavl OF Ula_tcc IS
10 SIGNAL and_port: std_logic;
11 SIGNAL or_port: std_logic;
12 SIGNAL mux2x1: std_logic;
13 BEGIN
14   PROCESS(A,B,Binvertido,Op1)
15   BEGIN
16     IF(Binvertido = '0') THEN
17       mux2x1 <= B;
18     ELSE
19       mux2x1 <= NOT B;
20     END IF;
21     and_port <= A AND mux2x1;
22     or_port <= A OR mux2x1;
23     IF(Op1 = '0') THEN
24       Resultado <= and_port;
25     ELSIF(Op1 = '1') THEN
26       Resultado <= or_port;
27     END IF;
28   END PROCESS;
29 END Ula_tcc_behavl;
```

Fonte: Própria.

4.2.1 Método de transformação direta

O método chamado de transformação direta apresenta uma unica ferramenta que realizada a conversão de VHDL para C. O circuito lógico descrito em VHDL, onde as assertivas serão inseridas pelo próprio usuário do método. O código VHDL é traduzido para linguagem C e recebe uma instrumentação de código para a posterior validação do mesmo. O Código em C automaticamente gerado será processado e analisado, e de acordo com as assertivas inseridas, apresentará falha, caso alguma delas seja violada.

4.2.1.1 Preprocessamento do VHDL e inserção de assertivas

A fase inicial do método consiste na adição das assertivas ao código VHDL. Vale ressaltar que mesmo a linguagem VHDL tendo um modelo de assertivas, tornou-se necessário a criação de um modelo próprio, baseado em comentários, para que as funções do model checker adotado fossem suportadas, como a geração de valores não determinísticos, ou seja, segundo Rocha et al.

(2015b), a variável pode assumir qualquer valor a partir de um tipo declarado.

As assertivas são adicionadas entre as tags `@c2vhdl:ASSERT` e `@c2vhdl:END` e todo trecho de código entre estas tags deve estar comentado, desta forma não apresentará erro na tradução do código em linguagem C, bem como na sintetização do código VHDL. Nas assertivas são incluídas as funções necessárias para a análise utilizando a ferramenta ESBMC. A assertiva apresenta três informações principais, sendo elas: condição, mensagem e gravidade.

Figura 22 – Exemplo de assertiva para verificação de porta `AND`.

```
1 --@c2vhdl:ASSERT
2 --assert (Resultado = '1')
3 --report "O resutado foi diferente a 0"
4 --severity ERROR;
5 --@c2vhdl:END
```

Fonte: Própria.

A condição, **linha** 2 da Figura 22, representa a assertiva propriamente dita e que será analisado pela aplicação. A condição será precedida de `--assert` e seguido ou não da palavra `not`, com isso a assertiva pode assumir valor negativo, conforme necessidade do usuário. Na Figura 22 a assertiva busca verificar se a variável `Resultado` terá valor igual a 1, com isso ao comparar o valor buscado pela assertiva e o gerado no final do código, caso ambos sejam iguais, ocorre falha na verificação.

A mensagem, **linha** 3 da Figura 22, é definida pelo usuário e será exibida caso a assertiva apresente falha, assim a mensagem pode ser utilizada como um meio de depuração das propriedades validadas. A severidade, **linha** 2 da Figura 22, pode ser definida como `error`, que representa um erro fatal e parada da verificação. A severidade do tipo `warning` que representa um erro não fatal, contabiliza as falhas das assertivas, porém não causa a parada da verificação.

Juntamente com as assertivas outra função que pode ser utilizada é a função `__ESBMC_assume()` suportada pelo model checker ESBMC. Esta função utilizada em conjunto com a ferramenta ESBMC permite que durante a verificação uma variável possa ter um valor definido durante o tempo de execução da verificação. A importância desta função é fazer verificações onde se conhece os valores de entrada juntamente com o valor resultante, por exemplo, na verificação de portas lógicas.

Figura 23 – Exemplo de utilização da função `__ESBMC_assume()`.

```
1    --__ESBMC_assume(A = '0');
2    --__ESBMC_assume(B = '1');
3    --__ESBMC_assume(Binvertido = '1');
4    --__ESBMC_assume(Op1 = '0');
```

Fonte: Própria.

Na Figura 24 apresenta o código exemplo da Figura 21. Nele já encontram inserido os *__ESBMC_assume()* anterior ao código para inicializar as variaveis necessárias e as assertivas após todo o código, de modo que a assertiva permaneça ao final do código.

Figura 24 – Exemplo da Figura 21 com assertiva

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  ENTITY Ula_tcc IS
5  PORT(A,B,Binvertido,Op1:IN std_logic;
6       Resultado:OUT std_logic);
7  END Ula_tcc;
8
9  ARCHITECTURE Ula_tcc_behavl OF Ula_tcc IS
10 SIGNAL and_port: std_logic;
11 SIGNAL or_port: std_logic;
12 SIGNAL mux2x1: std_logic;
13 BEGIN
14   PROCESS(A,B,Binvertido,Op1)
15   --__ESBMC_assume(A = '0');
16   --__ESBMC_assume(B = '1');
17   --__ESBMC_assume(Binvertido = '1');
18   --__ESBMC_assume(Op1 = '0');
19   BEGIN
20     IF(Binvertido = '0') THEN
21       mux2x1 <= B;
22     ELSE
23       mux2x1 <= NOT B;
24     END IF;
25     and_port <= A AND mux2x1;
26     or_port <= A OR mux2x1;
27     IF(Op1 = '0') THEN
28       Resultado <= and_port;
29     ELSIF(Op1 = '1') THEN
30       Resultado <= or_port;
31     END IF;
32       --@c2vhdl:ASSERT
33     --assert (Resultado = '1')
34     --report "O resutado foi diferente a 0"
35     --severity ERROR;
36     --@c2vhdl:END
37   END PROCESS;
38 END Ula_tcc_behavl;
```

Fonte: Própria.

4.2.1.2 Tradução de código VHDL para a linguagem C

Nesta etapa a tradução do código VHDL para linguagem C é executada. A partir deste ponto, a ferramenta é executada de maneira propriamente dita, tendo todo o processo automatizado até a apresentação do resultado. Para esta etapa do método foi selecionado a ferramenta V2C(GATTI; GHEZZI, 1995) que realiza a tradução de VHDL para Linguagem C.

A ferramenta V2C apresenta diversas vantagens em relação a equivalência de tradução de linguagens, contudo V2C apresenta certas limitações na tradução do código VHDL, em outras palavras, a mesma não reconhece algumas estruturas específicas do VHDL. A ferramenta aceita ape-

nas entradas e saídas do tipo: `bit`, `std_ulogic`, `qsim_state`, `std_ulogic_vector` e `interger`. Na parte de arquitetura, a ferramenta aceita uma gama maior de estruturas, trabalhando com expressões do tipo: `signal`, `variable`, `integers`, `strings` e caracteres. Em expressões condicionais os operandos são `AND`, `OR`, `NOT` <=, =>,=,<,> e <>. Aceita também a estrutura de `process`, além da estrutura `block`. A estrutura `process` é limitada apenas a: `if-else`, `case` e `loops`. Como trabalho futuro, neste trabalho tem-se buscado novas abordagens para ampliar o suporte a estruturas não suportadas pelo V2C.

Na tradução é necessário substituir os operadores originais por seus equivalentes em linguagem C. A exceção são os operadores específicos que gerenciam os valores de `bit`, tais como concatenação ou manipulação de partes vetores, para os quais são necessários construir procedimentos específicos em C.

Na tradução são gerados vários vetores para suporte a tradução dos sinais representados no VHDL, sendo eles `chg[]`, `old[]` e `new[]`. O vetor `old[]` contém o valor dos sinais do ciclo anterior e operação e a partir deste vetor, o valor a ser usado nos cálculos posteriores é obtido. O vetor `new[]` representa os novos valores que foram calculados durante a execução do ciclo e este novos são calculados pelos antigos existentes no vetor `old[]`. E o vetor `chg[]` contém um or exclusivo entre o valor novo e antigo e em caso de alteração entre os valores é copiado o valor de `new[]` para `old[]`.

No código C gerado, os vetores `in_data[]` e `out_data[]` contêm os sinais de entrada e saída, respectivamente. No caso de circuitos sequenciais, o valor do status é inserido no primeiro (índice 0). A ferramenta lerá os valores `in_data[]` e escreverá os resultados do processamento em `out_data[]`. Ao final da operação os valores de estado finais salvos no vetor `new[]` são passados para `out_data[]`.

Conforme especificado e seguindo os parâmetros da ferramenta, a mesma realiza a tradução, mantendo inalterado qualquer fragmento de código que esteja comentado, neste caso, as assertivas presentes no código VHDL permanecem inalteradas, sendo utilizadas na próxima etapa do método proposto.

Figura 25 – Exemplo da Figura 24 traduzido para linguagem C

```
1  [...]
2     /* Start of Translation */
3
4     /* p0: */
5     if (chg[A] || chg[B] || chg[Binvertido] || chg[
          Op1]) {
6        /*__ESBMC_assume(A = '0');
7         *__ESBMC_assume(B = '1');
8         *__ESBMC_assume(Binvertido = '1');
9         *__ESBMC_assume(Op1 = '0'); */
10              if ((old[Binvertido]==0)) {
11           if (chg[B]) {
12              new[mux2x1]=old[B];
13              }
14           }
15        else {
16           if (chg[B]) {
17              new[mux2x1]=~old[B];
18              }
19           }
20        if (chg[A] || chg[mux2x1]) {
21           new[and_port]=old[A] & old[mux2x1];
22           }
23        if (chg[A] || chg[mux2x1]) {
24           new[or_port]=old[A] | old[mux2x1];
25           }
26        if ((old[Op1]==0)) {
27           if (chg[and_port]) {
28              new[Resultado]=old[and_port];
29              }
30           }
31        else if ((old[Op1]==1)) {
32           if (chg[or_port]) {
33              new[Resultado]=old[or_port];
34              }
35           }
36        /*@c2vhdl:ASSERT
37         *assert (Resultado = '1')
38         *report "O resutado foi diferente a 0"
39         *severity ERROR;
40         *@c2vhdl:END */
41        }
42    /* End of Translation */
43    [...]
```

Fonte: Própria.

4.2.1.3 Instrumentação de código

As assertivas após a tradução permanecem comentadas, sendo necessário preparar-las, ou seja traduzi-las, para verificação posterior do código na linguagem em C. Com isso é necessário uma instrumentação do código para prover suporte as assertivas traduzidas para análise.

Todas as etapas da instrumentação são realizados sobre o código C já traduzido. O passo inicial da instrumentação é a adição da macros no início do código C com as definição das assertivas a serem utilizadas. Na Figura 26 são apresentados os macros utilizadas no código C. A Linha 1 da figura corresponde a mensagem de erro a ser apresentada e a Linha 2 corresponde ao modelo da assertiva e a chamada da macro definida na Linha 1 em caso de falha. Estas definições para as assertivas também podem ser utilizadas no código C traduzidos sem o uso do ESBMC.

Figura 26 – Macros das assertivas implementadas em linguagem C

```
1 #define log_error(M,...) fprintf(stderr,M,__FILE__,__LINE__,##__VA_ARGS__)
2 #define __MY_assert(A, M,...) if(!(A)) {log_error(M, ##__VA_ARGS__);
    assert(A); }
```

Fonte: Própria.

O passo seguinte é a identificação das assertivas comentadas ao longo do corpo do código traduzido. As assertivas são identificadas através das tags `@c2vhdl:ASSERT` e `@c2vhdl:END` e a busca é realizado através destas tags. Ao ser encontrado a tag `@c2vhdl:ASSERT`, é realizado um *loop* até que seja encontrada a tag **@c2vhdl:END** e com isso toda a assertiva inserida possa ser passada a função de busca das informações da assertiva contidas entre as tags apresentadas.

Com os dados das assertivas obtidos é realizado a busca da condição, mensagem e severidade através das tags `--assert`, `--report` e `--severity` respectivamente. Estas informações são encontradas através do uso e uma Regex no código e que são adicionadas ao código C seguindo o modelo definido na macro. Este processo é repetido até outra assertiva ser encontrada ou caso chegue ao final do código C.

Com a função `__ESBMC_assume()` ocorre processo semelhante ao das assertivas. Utilizando novamente uma regex que realiza a busca por esta função no código e é retirado o comentário da mesma, desta forma a mesma passa a esta acessível para o ESBMC, visto que a declaração da mesma já segue o modelo padrão a ser utilizado pelo ESBMC.

Durante a instrumentação de código também é realizado a entrada de sinais não determinísticos para variáveis não inicializadas ou argumentos de funções do código C gerado da tradução, utilizando a função `__VERIFIER_nondet_int()`. Esta função é utilizada em todas as variáveis de entrada e também nos sinais criados ao longo da arquitetura. Desta forma, todas as variáveis necessárias são inicializadas para verificação.

Segundo, Rocha et al. (2015b), a função `__VERIFIER_nondet_int()` tem a função de modelar valores inteiros não determinísticos e é importante no desenvolvimento do método, pois evita erros, onde dado estado do código não pode ser alcançado, devido a variável não inicializada.

Em outras palavras, na instrumentação de código é realizado a tradução das assertivas do modelo utilizado no código VHDL para para o modelo utilizado em linguagem C, além como de outras funções que possam ser utilizados pelo VHDL. Ao final da instrumentação o código C fica disponível para que possa ser dado como entrada para outras ferramentas de verificação de código e não apenas o ESBMC.

Figura 27 – Exemplo da Figura 24 após a instrumentação

```
1  #define log_error(M,...) fprintf(stderr ,M,__FILE__,
       __LINE__,##__VA_ARGS__)
2  #define __MY_assert(A, M,...) if(!(A)) {log_error(M
       , ##__VA_ARGS__); assert(A); }
3  [...]
4     /* Start of Translation */
5     /* p0: */
6     if (chg[A] || chg[B] || chg[Binvertido] || chg[
          Op1]) {
7        /*__ESBMC_assume(old[A] = 0);
8         *__ESBMC_assume(old[B] = 1);
9         *__ESBMC_assume(old[Binvertido] = 1);
10        *__ESBMC_assume(old[Op1] = 0);*/
11              if ((old[Binvertido]==0)) {
12           if (chg[B]) {
13              new[mux2x1]=old[B];
14              }
15           }
16        else {
17           if (chg[B]) {
18              new[mux2x1]=~old[B];
19              }
20           }
21        if (chg[A] || chg[mux2x1]) {
22           new[and_port]=old[A] & old[mux2x1];
23           }
24        if (chg[A] || chg[mux2x1]) {
25           new[or_port]=old[A] | old[mux2x1];
26           }
27        if ((old[Op1]==0)) {
28           if (chg[and_port]) {
29              new[Resultado]=old[and_port];
30              }
31           }
32        else if ((old[Op1]==1)) {
33           if (chg[or_port]) {
34              new[Resultado]=old[or_port];
35              }
36           }
37        __ESBMC_assert(new[Resultado] == 1,"Teste");
38        }
39    /* End of Translation */
40    [...]
```

Fonte: Própria.

4.2.2 Método multiplas transformações

O método, chamado de multiplas trasnformações, apresenta duas ferramentas de transformação, sendo a primeira de VHDL para Verilog e seguido de uma ferramenta de Verilog para C. Inicialmente o código em VHDL é passado para ferramenta juntamente com outro arquivo contendo as variáveis de entrada, saída, pré-condições e pós-condições. Seguidamente é passado para duas ferramenta de tradução e também duas etapas de instrumentação. Após as traduções, as pré-condições e pós-condições, conforme o arquivo, são adicionadas ao códigos C e o mesmo é analisado. Caso alguma condição seja violada, um contra exemplo é apresentado ao usuário.

4.2.2.1 Código VHDL e arquivo externo

A primeira etapa do método consiste inicialmente no código VHDL a ser analisado, juntamente com o arquivo contendo as informações de entradas e saídas do código e também as pré-condições, que são condições que devem ser verdadeira antes da execução de um algum código ou trecho de código, e pós-condições que são condições declaradas e devem ser verdadeiras após a execução de um código ou trecho de código.

O arquivo, Figura 28, deve conter todas as variáveis de entrada e saída do código VHDL, apenas o nome da variável em ambos os casos. As variáveis de entrada são declaradas na linha **INPUT** e as variáveis de saída são declaradas na linha **OUTPUT**. Nas linhas seguintes são informadas as pré-condições e pós-condições que serão introduzidas ao código em etapas posteriores dos processos.

Figura 28 – Exemplo de arquivo externo.

```
1 INPUT:A==0,B==1,Binvertido==1,Op1==0
2 OUTPUT:Resultado
3 PRECONDITION:
4 POSTCONDITION:Resultado== 0
```

Fonte: Própria.

Em comparação ao método anterior, as diferenças consistem principalmente em as assertivas não estarem mais presentes diretamente no código VHDL, desta forma, o mesmo é utilizado apenas nas etapas de tradução. A principal vantagem é a não necessidade de elementos comentados ao código VHDL e com isso o código fica mais legível e também sem a necessidade de modificação ou inserção de trecho para funcionamento da ferramenta em questão.

4.2.2.2 Tradução de VHDL para C e instrumentações de código

A primeira etapa é a tradução de código entre VHDL para C, contudo, diferente do demonstrado no primeiro método, onde apenas uma única ferramenta realizava a tradução, nesta

nova reformulação do método faz a utilização de duas ferramentas de tradução, inicialmente para Verilog e uma segunda ferramenta faz a tradução para C. Isso possibilita um aumento no potencial da tradução, contudo, devido ao uso de duas ferramentas, alguns erros ainda podem ser verificados na tradução.

Devido ao uso das ferramentas, também é necessário mais uma etapa de instrumentação, sendo uma após a tradução para Verilog e outra após a tradução para C. A primeira instrumentação modifica o código para que segunda ferramenta de tradução possa executar a tradução corretamente e a segunda instrumentação adiciona as assertivas, de acordo com o introduzido no arquivo citado na seção anterior. Nas próximas subseções serão explicados com mais detalhes as traduções e as instrumentações.

4.2.2.2.1 Tradução para Verilog e instrumentação de código

A primeira ferramenta de tradução utilizada, chamada VHD2Vl (DOOLITTLE, 2015), realiza a tradução do código VHDL para o código Verilog. O principal motivo de utilizar esta ferramenta deve-se a apresentar melhores resultados na tradução dos códigos, contudo, também apresenta algumas questões a serem observadas nas traduções.

Entre estas observações esta encontra-se a utilização das estruturas de `std_logic`, `std_logic_vector`, `integer` e `boolean`. Também permite a utilização `clock events` das estruturas de controle, como `if`, `elsif`, `else`, e `case`. A ferramenta também permite a instância de módulos no código VHDL, contudo o mesmo é ignorado na tradução o que causa erros de análise. Também é ignorado qualquer assertiva existente no código, tornando assim, mais necessário um arquivo externo com assertivas e condições a serem adicionadas ao texto (DOOLITTLE, 2015).

Figura 29 – Código da Figura 21 traduzido pela ferramenta Vhd2vl.

```
1  [...]
2  module Ula_tcc(
3  input wire A,
4  input wire B,
5  input wire Binvertido,
6  input wire Op1,
7  output reg Resultado
8  );
9  reg and_port;
10 reg or_port;
11 reg mux2x1;
12   always @(A, B, Binvertido, Op1) begin
13     if((Binvertido == 1'b0)) begin
14       mux2x1 <= B;
15     end
16     else begin
17       mux2x1 <=  ~B;
18     end
19     and_port <= A & mux2x1;
20     or_port <= A | mux2x1;
21     if((Op1 == 1'b0)) begin
22       Resultado <= and_port;
23     end
24     else if((Op1 == 1'b1)) begin
25       Resultado <= or_port;
26     end
27   end
28 endmodule
```

Fonte: Própria.

Contudo, o código conforme apresentado acima, não pode ser traduzido para C, pois está fora do padrão aceito pela ferramenta V2c, que realiza a tradução de verilog para C. Com isso é necessária uma instrumentação de código para que o mesmo possa ser traduzido. Esta instrumentação consiste na alteração dos modo de declaração das variáveis, na alteração de palavras chaves utilizadas no verilog e na busca de elementos que estejam dentro com escopo `always@()`, que contém a arquitetura do código.

Figura 30 – Diferença da declaração das variáveis após a intrumentação.

```
1 [...]
2 module Ula_tcc(
3 input wire A,
4 input wire B,
5 input wire Binvertido,
6 input wire Op1,
7 output reg Resultado
8 );
9 [...]
```

(a)

```
1 module Ula_tcc(A,B,Binvertido,Op1,Resultado);
2 input wire A;
3 input wire B;
4 input wire Binvertido;
5 input wire Op1;
6 output Resultado;
7 reg Resultado;
8 [...]
```

(b)

Fonte: Própria.

A primeira adaptação a ser feita é na declaração das variáveis. Conforme a Figura 30(a) apresenta a declaração após a tradução e a Figura 30(b) apresenta após a instrumentação. As mudanças consistem em o nome das variáveis serem declaradas como uma espécie de parâmetro e logo abaixo seus respectivos tipos, fora do modulo. Outra alteração é com relação ao **Output reg**, onde primeiro se declara como saída e depois como reg a mesma variável, como apresentado na linha 7 e 8 da Figura 30(b) em relação a linha 7 da Figura 30(a).

Outra alteração necessário é que nenhuma variável seja declarada dentro do modulo `always@()`, pois gera a não tradução do código por parte da ferramenta. Com isso toda e qualquer variável que venha a ser utilizado durante a execução do código deve ser declarado antes do modulo `always@()`, conforme Figura 31. E outra alteração consiste nas palavras reservadas, visto que verilog utiliza a palavra **define** para definição de constante. Esta alteração é importante, pois evita que o código seja traduzido de maneira errada.

Figura 31 – Declaração de variáveis anteriores ao modulo Always@()

```
1 [...]
2 reg and_port;
3 reg or_port;
4 reg mux2x1;
5
6 always @(A, B, Binvertido, Op1) begin
7     [...]
8 endmodule
```

Fonte: Própria.

4.2.2.2.2 Tradução para C e adição das pre-condições e pós-condições.

A segunda parte da tradução é realizada pela ferramenta V2C (MUKHERJEE et al., 2016b). Como citado anteriormente, após a instrumentação no código verilog o mesmo pode ser traduzido para C, conforme apresentado na Figura 32. Após a tradução o código recebe as assertivas e inicializações de variáveis.

Toda parte de arquitetura do código VHDL após a tradução para C é colocada em uma função de mesmo nome do arquivo. Também é definido uma `struct` que contém as variáveis que foram declaradas como `output` e também variáveis que tenham sido declaradas para serem utilizadas na execução da arquitetura, como os sinais por exemplo. As variáveis de entrada são declaradas na função `main()` e são repassadas como parâmetros da função.

A necessidade desta `struct` deve-se ao fato de que estas variáveis podem ter seus valores alterados em tempo de execução. Devido a este fato, variáveis são criadas para armazenar valores antigos e também receber novos valores, como ocorre nas linhas treze, catorze, quinze e dezeseis da Figura 32. Nota-se que estas variáveis possuem o mesmo nome das variáveis da *struct*, com a adição do sufixo `_old`.

Após a etapa de conversão para linguagem C ocorre uma nova etapa de instrumentação de código, onde as assertivas serão introduzidas ao código C. Para isso faz-se necessário a utilização do arquivo de pré-condições e pós-condições passado para ferramenta juntamente com o código VHDL e apresentado na Figura 28.

Figura 32 – Código C traduzido após a intrumentação no verilog.

```
1  #include <stdio.h>
2  #include <assert.h>
3  #define TRUE 1
4  #define FALSE 0
5  struct state_elements_Ula_tcc{
6  _Bool Resultado;
7  _Bool and_port;
8  _Bool or_port;
9  _Bool mux2x1;
10 };
11 void Ula_tcc(_Bool A, _Bool B, _Bool Binvertido,
       _Bool Op1, _Bool *Resultado){
12   struct state_elements_Ula_tcc  sUla_tcc;
13   _Bool Resultado_old;
14   _Bool and_port_old;
15   _Bool or_port_old;
16   _Bool mux2x1_old;
17   mux2x1_old = sUla_tcc.mux2x1;
18   and_port_old = sUla_tcc.and_port;
19   or_port_old = sUla_tcc.or_port;
20   Resultado_old = sUla_tcc.Resultado;
21   if((unsigned char)Binvertido == 0){
22     sUla_tcc.mux2x1 = B;
23   }
24   else{
25     sUla_tcc.mux2x1 = !B;
26   }
27   sUla_tcc.and_port = A && mux2x1_old;
28   sUla_tcc.or_port = A || mux2x1_old;
29   if((unsigned char)Op1 == 0){
30     sUla_tcc.Resultado = and_port_old;
31   }
32   else{
33     if((unsigned char)Op1 == 1){
34       sUla_tcc.Resultado = or_port_old;
35     }
36   }
37 }
38 void main() {
39 _Bool A;
40 _Bool B;
41 _Bool Binvertido;
42 _Bool Op1;
43 _Bool Resultado;
44 Ula_tcc(A, B, Binvertido, Op1, &Resultado);
45 }
```

Fonte: Própria.

Para a instrumentação é necessário a utilização de funções próprias do analisador, neste caso o ESBMC (ESBMC, 2018) e estas funções são: `ESBMC_assume()`, `__VERIFIER_nondet_int()` e `__VERIFIER_nondet_bool()`. A função `ESBMC_assume()` per-

mite que uma variável assuma um valor especifico. As funções **__VERIFIER_nondet_int()** e `__VERIFIER_nondet_bool()` modelam valores inteiros e booleanos não deterministicos respectivamente.

A função `ESBMC_assume()` é utilizada quando alguma variável foi inicializada no arquivo, desta forma a mesma é inicializada no código. Na linha 1 da Figura 28 é declarada variável inicializada e das linhas 21 a 24 da Figura 33 é utilizado as funções para declaração das variáveis. Desta forma, estes valores são definidos para aquela variável no tempo de execução da análise.

A importância das funções `__VERIFIER_nondet_int()` e `__VERIFIER_nondet_bool()` no desenvolvimento do método é evitar que um determinado estado do código não possa ser alcançado, devido a não inicialização de variáveis. Estas funções são utilizadas nas variáveis de entrada não inicializadas e as outras variáveis que não são inputs e nem outputs. Na Figura 33, linhas 48 a 51 apresenta o uso desta função.

Em outras palavras, na instrumentação de código é realizado a adição das assertivas do modelo utilizado no arquivo para o modelo utilizado em linguagem C, além como de outras funções que possam ser utilizados pelo VHDL. Ao final da instrumentação o código C fica disponível para que possa ser dado como entrada para outras ferramentas de verificação de código e não apenas o ESBMC.

Figura 33 – Código C adição das assertivas.

```
1  #include <stdio.h>
2  #include <assert.h>
3  #define TRUE 1
4  #define FALSE 0
5  struct state_elements_Ula_tcc{
6  _Bool Resultado;
7  _Bool and_port;
8  _Bool or_port;
9  _Bool mux2x1;
10 };
11 void Ula_tcc(_Bool A, _Bool B, _Bool Binvertido,
       _Bool Op1, _Bool *Resultado){
12   struct state_elements_Ula_tcc  sUla_tcc;
13   _Bool Resultado_old;
14   _Bool and_port_old;
15   _Bool or_port_old;
16   _Bool mux2x1_old;
17   mux2x1_old = sUla_tcc.mux2x1;
18   and_port_old = sUla_tcc.and_port;
19   or_port_old = sUla_tcc.or_port;
20   Resultado_old = sUla_tcc.Resultado;
21   __ESBMC_assume(A==0);
22   __ESBMC_assume(B==1);
23   __ESBMC_assume(Binvertido==1);
24   __ESBMC_assume(Op1==0);
25   if((unsigned char)Binvertido == 0){
26     sUla_tcc.mux2x1 = B;
27   }
28   else{
29     sUla_tcc.mux2x1 = !B;
30   }
31   sUla_tcc.and_port = A && mux2x1_old;
32   sUla_tcc.or_port = A || mux2x1_old;
33   if((unsigned char)Op1 == 0){
34     sUla_tcc.Resultado = and_port_old;
35   }
36   else
37     if((unsigned char)Op1 == 1){
38       sUla_tcc.Resultado = or_port_old;
39     }
40     __ESBMC_assert(sUla_tcc.Resultado == 0,"Teste")
         ;
41 }
42 void main() {
43 _Bool A;
44 _Bool B;
45 _Bool Binvertido;
46 _Bool Op1;
47 _Bool Resultado;
48 A=__VERIFIER_NONDET_BOOL();
49 B=__VERIFIER_NONDET_BOOL();
50 Binvertido=__VERIFIER_NONDET_BOOL();
51 Op1=__VERIFIER_NONDET_BOOL();
52 Ula_tcc(A, B, Binvertido, Op1, &Resultado);
53 }
```

Fonte: Própria.

4.2.3 Verificação de assertivas usando *Model Checker*

O *model checker* adotado no desenvolvimento de método é o ESBMC (CORDEIRO et al., 2012) na versão 6.0.0 e conforme explicado na Seção 2.2.3, esta ferramenta recebe como entrada um código C ou C++ e também utiliza solucionadores SMT para análise do programa.

Para verificação pode ser utilizado o métodos de análise, chamado indução-k. A indução-k é o meio principal de verificação, juntamente com o mesmo é adicionado as seguintes opções, estas que o analisador trabalhe diretamente e unicamente com as assertivas, evitando que outros fatores possam ser analisados no código:

- `--no-pointer-check`: para não realizar a checagem de ponteiros no código;
- `--no-div-by-zero-check`: para não realizar a checagem de divisões por zero no código; e
- `--no-bounds-check`: para não realizar a checagem de array bounds no código.

Desta forma o ESBMC realiza apenas a checagem necessária dentro da assertiva, evitando que outros parâmetros sejam verificados, sem a devida necessidade do mesmo. O ESBMC utilizando a técnica de indução-k realiza a análise das assertivas em cada código, considerando as pré condições e também os valores determinados para as variáveis dentro o código através do ESBMC_assume(). Ao final de todos os desdobramentos é apresentado o resultado, sendo positiva (sem erros) ou negativa (com violação de propriedades), dependendo da assertiva e do código analisado.

4.3 Resumo do capitulo

Neste capitulo foi apresentado os dois métodos de tradução composto pelo método de múltiplas traduções, na qual é utilizado duas ferramentas de tradução, e pelo método de tradução direta que utiliza apenas uma ferramenta e faz a conversão direta de VHDL para linguagem C. Também foi mostrado a ferramenta de análise chamada ESBMC e de que forma ela é utilizada para verificação dos códigos.

5 AVALIAÇÃO EXPERIMENTAL

Neste capitulo será apresentado o planejamento, execução e os resultados do método proposto neste trabalho. Serão apresentados dois experimentos neste capitulo, sendo inicialmente um para os métodos de tradução e outro para o método de verificação de propriedades. Desta forma abrangendo todo o método do trabalho. O objetivo destes experimentos é testar as ferramentas utilizadas, assim como descobrir os limites de utilização da mesma, na execução do método proposto.

5.1 Planejamento da avaliação experimental

Os testes apresentados neste capitulo tem o objetivo de apresentar o potencial da abordagem do método proposto em relação a tradução de código VHDL para C e análise de código apresentado Capítulo 4. O teste será dividido em duas metade etapas, inicialmente será testado os métodos de tradução, sendo os métodos de transformação direta e de múltiplas sucessões. Também será analisada o potencial de análise utilizada neste método, demonstrando a performance da mesma com a abordagem que apresentou o melhor resultado.

Os teste serão executados em ambiente Linux, precisamente o Ubuntu 16.04, utilizando a ferramenta WSL(Windows Subsystem fo Linux). A máquina utilizada para os teste corresponde a um Dell, processador I7 $5500U$ 2.4Ghz, 8GB de memória ram, contendo o sistema operacional Windows 10 64bit, na versão 1803.

Baseado no objetivo de testar o potencial das abordagens de tradução foram desenvolvidas as seguintes questões de pesquisa para os testes nas abordagens de tradução e nas ferramentas utilizadas:

1. As ferramentas de tradução de código são capazes de manipular as diferentes estruturas de código VHDL?
2. Existem melhorias a serem efetuadas na tradução de código?
3. Quais são os impactos de uso das ferramentas auxiliares na aplicação do método proposto?

Para os testes a serem realizados na ferramenta de análise ESBMC foram desenvolvidas as seguintes questões de pesquisa:

1. A ferramenta ESBMC foi capaz de analisar todos os códigos traduzidos de VHDL para C?
2. Quais as limitações apresentadas pela ferramenta de análise ESBMC?

Para os testes do método proposto foi utilizado um benchmark composto por 20 códigos escritos em VHDL. Estes códigos são oriundos das seguintes fontes:

- **Fonte própria do autor:** 10 códigos foram desenvolvidos pelo autor para utilização em testes das ferramenta. Estes códigos correspondem a portas lógicas, multiplexadores, entre outros códigos;
- **ISCAS99:** 6 códigos foram extraídos do benchmark externo ISCAS99. Apresentam um grau de complexidade maior que os códigos do autor, e desta forma buscar os limites das ferramentas. O benchmark está disponível em:

  `http://www.pld.ttu.ee/~maksim/benchmarks/iscas99/vhdl/`. Os códigos utilizados correspondem a códigos com switch-case de várias entradas e várias iformações de entrada que aumenta a complexidade do circuito.
- **Benchmark Vhd2vl:** 4 códigos foram extraídos dos exemplos existente junto com a ferramenta Vhd2vl. Estes códigos apresentam diversos graus de complexidade, buscando analisar não somente as limitações desta ferramenta, mas se é possível que estes códigos sejam analisados ao final. Os código são estruturas de memória, cadeias de `ifs` e um contador

Conforme citado anteriormente, os teste serão divididos em duas etapas, inicialmente o teste será realizado apenas com os métodos de tradução. Será passado para os métodos o benchmark de 20 códigos citados acima e é observado qual das abordagens demonstrou o melhor desempenho, ou seja, qual método foi capaz de traduzir mais códigos de VHDL para C. Cada método de tradução será executado conforme apresentado no Capítulo 4.

Para as traduções serão testadas 3 ferramentas, sendo duas ferramentas para o método de múltiplas transformações, sendo elas a ferramenta Vhd2vl, na versão 3.0 disponível no site <https://github.com/ldoolitt/vhd2vl>, que realiza a tradução de VHDL para Verilog. E a ferramenta V2c, na versão e disponível no site, que realiza a tradução de Verilog para C. No método de tradução direta apenas a ferramenta V2c que realiza a tradução de VHDL para C. É importante ressaltar que apesar do mesmo nome, V2c, os métodos apresentam ferramentas diferentes, com usos diferentes.

A segunda etapa de testes foi o teste da ferramenta de análise ESBMC. Os testes serão realizados em conjunto com o método de tradução que obteve o melhor resultado, ou seja, o método que apresentou o maior conjunto de códigos traduzidos será utilizado em conjunto com a ferramenta ESBMC para o teste da ferramenta. A ferramenta ESBMC está disponível no site <https://github.com/esbmc/esbmc> e será utilizado esta na versão 6.0.0.

O motivo desta abordagem de teste ter sido utilizada deve-se ao fato de uma quantidade maior de códigos ser utilizada para os testes, além de que os melhores resultados, tecnicamente

deve apresentar os melhores códigos. Para os testes os códigos traduzidos para linguagem C terão assertivas inseridas aos códigos e a ferramenta deve ser capaz de encontrar estas assertivas, mas também verificar se alguma propriedade destas assertivas foi violada ou não.

5.2 Execução e análise dos resultados

Nesta sessão será apresentado os resultados proveniente dos testes explanados na sessão anterior. Primeiramente será apresentado os resultados dos métodos de tradução e posteriormente será apresentado os resultados relacionados a ferramenta de análise ESBMC.

5.2.1 Testes realizados nas abordagens de tradução

Após a execução dos benchmarks, no teste dos métodos de tradução, os resultado obtidos são apresentados na Tabela 2, formada pelas colunas: **Id** que é uma identificação para cada código; **Nome do código**; **Abordagem 01** que representa a abordagem de multiplas traduções(subseção 4.2.2); **vhd2vl** que apresenta o status para tradução da ferramenta vhd2vl; **v2c** para o status da ferramenta de tradução; **Abordagem 02** que representa a abordagem de transformação direta(subseção 4.2.1); **v2c** que representa o status de tradução da ferramenta v2c. Para cada operação positiva de tradução ou análise foi atribuído o valor sim na tabela e para cada operação negativa de tradução ou análise de código realizada foi atribuído o valor não.

A ferramenta ESBMC também foi adicionada como forma de parâmetro para testar as traduções realizadas, de modo a saber se algum estrutura em C gerada pelas traduções não seria reconhecida, visto que o ESBMC será utilizado no método proposto como o verificador de pós-condições. Vale lembrar que a ferramenta V2C utilizada na **Abordagem 01** é diferente da ferramenta utilizada na **Abordagem 02**. A primeira realiza traduções de Verilog para C , enquanto a segunda realiza de VHDL para C.

Tabela 2 – Resultados das abordagens de tradução

Benchmark Oficial						
ID	**Nome arquivo**	**Abordagem 01**			**Abordagem 02**	
		VHD2VL	**V2C**	**ESBMC**	**V2C**	**ESBMC**
1	AND_ent.vhd	SIM	SIM	SIM	SIM	SIM
2	XOR_ent.vhd	SIM	SIM	SIM	SIM	SIM
3	ifchain.vhd	SIM	SIM	SIM	NÃO	-
4	B01.vhd	SIM	SIM	SIM	SIM	NÃO
5	B06.vhd	SIM	SIM	SIM	SIM	NÃO
6	B10.vhd	SIM	SIM	SIM	SIM	NÃO
7	Comb1.vhd	SIM	SIM	SIM	SIM	SIM
8	FlipFlop_D.vhd	SIM	SIM	SIM	SIM	SIM
9	seq1.vhd	SIM	SIM	SIM	SIM	SIM
10	Ula_tcc.vhd	SIM	SIM	SIM	SIM	SIM
11	dff.vhd	SIM	SIM	SIM	SIM	SIM
12	mux_2to1_top.vhd	SIM	SIM	SIM	SIM	SIM
13	b02.vhd	SIM	SIM	SIM	SIM	NÃO
14	b03.vhd	SIM	SIM	SIM	SIM	NÃO
15	b09.vhd	SIM	SIM	SIM	SIM	NÃO
16	counters.vhd	SIM	NÃO	-	NÃO	-
17	ifchain2.vhd	SIM	SIM	NÃO	NÃO	-
18	mem.vhd	SIM	NÃO	-	NÃO	-
19	Nand_ent.vhd	SIM	SIM	SIM	SIM	SIM
20	Nor_ent.vhd	SIM	SIM	SIM	SIM	SIM

Entre os códigos utilizado no benchmark, os seguintes resultados da **Abordagem 01** foram obtidos:

- 20 códigos passaram pela ferramenta Vhd2vl. Contudo, os códigos com ID 16 e 18 não foram inteiramente traduzidos devido a ferramenta não ser capaz de fazer a link entre as variáveis declaradas;
- Dos 20 códigos foram traduzidos para Verilog e instrumentados 18 que foram traduzidos para linguagem C. Além dos códigos citado anteriormente com problemas de link entre as variáveis, temos que o código com ID 17 apresentou uma especie de "lixo", como se o tradutor não reconhecesse a estrutura no verilog durante a tradução para C.
- Dos 18 códigos, 17 códigos foram reconhecidos pelo ESBMC. O código com ID 17 apresentou erros, como citado acima o que ocasionou operação abortada pelo analisador ESBMC.

Entre os códigos utilizado no benchmark, os seguintes resultados da **Abordagem 02** foram obtidos:

- Dos 20 códigos, apenas 16 foram traduzidos pela ferramenta v2c pela ferramenta. Isso deve-se a alguma declaração de variável inteira nos códigos e/ou devido a estrutura `downto` no VHDL apresentar erro na tradução; e
- Dos 16 códigos traduzidos e instrumentados, apenas 10 foram aceitos pela ferramenta de analise do ESBMC.

Com base nestes resultados apresentados é possível destacar que o método de múltiplas traduções (Abordagem 01) apresentou um desempenho melhor na tradução dos códigos. Vale ressaltar que é uma melhoria na tradução, visto que o método de tradução simples foi utilizado no desenvolvimento do TCC 1. Uma ressalva deve ser apresentada na abordagem de múltiplas traduções, uma maior quantidade de ferramentas para a tradução, como no caso de Vhdl-Verilog e Verilog-C, pode aumentar a complexidade da tradução e gerar mais erros de sintaxe.

5.2.2 Análise da verificação de código

Como citado no inicio do capítulo, a apresentação dos resultados da ferramenta ESBMC seria realizado com a abordagem que tivesse a melhor desempenho no teste de tradução, sendo o caso, o método de múltiplas transformações. Foram utilizados 17 códigos para o teste da ferramenta de verificação ESBMC, todos os códigos apresentam as pré (usando a função `__VERIFIER_assume` do ESBMC) e pós condições em `asserts`. Os resultados esperados na verificação com o ESBMC são `TRUE`, a assertiva não foi violada, ou *FALSE* caso contrário. Nas tabelas Tabela 3 e Tabela 4 apresentam os resultados dos teste e também o tempo de cada operação.

A Tabela 3 apresenta **ID**, **Nome do código**, **Tradução verilog**, se o código foi capaz de ser traduzido, **Tempo de tradução** desde o VHDL até a linguagem C, **Pre-condição** e **Assume**. A tabela ainda apresenta dois termos, sendo VNB e EAE. O primeiro é a sigla para __VERIFIER_NONDET_BOOL() que gerar valores não determinísticos para ser utilizado durante o experimento. O segundo é a sigla para __ESBMC_assume() que é uma função da ferramenta de análise utilizada para aplicar valores a determinadas variáveis.

A Tabela 4 apresenta **ID**, **Pós-condição**, **Resultado esperado** que é o resultado a ser esperado pela ferramenta de análise, de acordo com as pre e pós condições, e **Tempo de execução**. Assim como a tabela anterior, aqui é apresentado a sigla EAT para __ESBMC_assert(), a qual é utilizada para adicionar as assertivas ao código VHDL.

Tabela 3 – Resultado da ferramenta de análise

BENCHMARK OFICIAL					
ID	Nome do Código	Tradução verilog	Tempo de tradução	Pre-condição	Assume
1	AND_ent.vhd	SIM	0m0.231s	X=VNB; Y=VNB	EAE(x==TRUE); EAE(y==TRUE);
2	XOR_ent.vhd	SIM	0m0.161s	X=VNB; Y=VNB	EAE(x==FALSE); EAE(y==TRUE);

3	B01.vhd	SIM	0m7.008s	line1=VNB;
				line2=VNB;
				reset=VNB;
				clock=VNB;
					EAE(reset == 0);
					EAE(line1 == 0);
					EAE(line2 == 0);
					EAE(stato_old == 1);

4	B06.vhd	SIM	0m3.431s	eql=VNB;
				clock=VNB;
				reset=VNB;
				cont_eql=VNB;
					EAE(reset == 0);
					EAE(cont_eql == 1);
					EAE(eql == 1);
					EAE(state_old == 0);

5	Comb1.vhd	SIM	0m1.292s	a=VNB;
				b=VNB;
				c=VNB;
				d=VNB;
				e=VNB;
					-

6	FlipFlop_D.vhd	SIM	0m0.677s	D=VNB; CLK=VNB; RST=VNB;	EAE(RST==1);

7	seq1.vhd	SIM	0m0.958s	in1=VNB;
				in2=VNB;
				reset=VNB;
					EAE(reset == 1);
					EAE(in1 == 0);
					EAE(in2 == 0);

8	Ula_tcc.vhd	SIM	0m0.905s	A=VNB; B=VNB; Binvertido=VNB; Op1=VNB;	EAE(A==0); EAE(B==1); EAE(Binvertido==0); EAE(Op1==0);
9	dff.vhd	SIM	0m0.479s	data_in=VNB;	EAE(data_in==FALSE);

10	mux_2to1_top.vhd	SIM	0m0.585s	w0=VNB;
				w1=VNB;
				s=VNB;
					EAE(w0 == TRUE);
					EAE(w1 == FALSE);
					EAE(s == FALSE);

11	b02.vhd	SIM	0m3.007s	reset=VNB clock=VNB linea=VNB;	EAE(reset == 0); EAE(linea == 0); EAE(stato_old == 1);
12	Nand_ent.vhd	SIM	0m0.365s	X=VNB; Y=VNB;	EAE(x==TRUE); EAE(y==TRUE);
13	Nor_ent.vhd	SIM	0m0.386s	X=VNB; Y=VNB;	EAE(x==TRUE); EAE(y==TRUE);

Tabela 4 – Resultado da ferramenta de análise

BENCHMARK OFICIAL			
ID	Pós-condição	Resultado esperado	Tempo de execução
1	EAT(sAND_ent.f==TRUE);	TRUE	0m1.571s
2	EAT(sAND_ent.f==TRUE);	TRUE	0m0.996s
3	EAT(sb01.outp==0&&sb01.overflw==FALSE);	TRUE	0m2.127s
4	EAT(sb06.ackout==FALSE&&sb06.enable_count==FALSE);	TRUE	0m2.017s
5	EAT(h==a^!(b && c));	FALSE	0m1.367s
6	EAT(sFlipFlop_D.Q==D);	FALSE	0m1.050s
7	EAT(sseq1.out==TRUE);	FALSE	0m1.637s
8	EAT(sUla_tcc.Resultado==0);	TRUE	0m3.284s
9	EAT(sdff.data_out==FALSE);	TRUE	0m2.417s
10	EAT(smux_2to1_top.f==TRUE);	TRUE	0m2.334s
11	EAT(sb02.u==TRUE);	FALSE	0m2.086s
12	EAT(sNAND_ent.f==TRUE);	FALSE	0m1.106s
13	EAT(sAND_ent.f==TRUE);	FALSE	0m0.774s

Conforme apresentados a ferramenta ESBMC foi capaz de localizar as assertivas e encontrar o resultado conforme o esperado, inclusive nos resultados na qual a mesma teria que apresentar a falha, ou seja, a assetiva foi violada gerando o erro. A mesma também apresentou tempos baixos tanto de tradução quanto na análise, mesmo nos códigos mais simples. O que configura uma boa atuação da ferramenta no contexto ao qual a mesma foi aplicada.

Na verificação dos programas foi utilizado no ESBMC a técnica de indução-k e as opções `--no-pointer-check` na qual não realiza a checagem de ponteiros, `--no-div-by-zero-check` na qual não realiza a checagem de divisão por zero e `--no-bounds-check` na qual não realiza a checagem de array bounds, conforme apresentado na subseção 4.2.3.

Quanto melhor a tradução realizada, melhor será a verificação do código. Novas técnicas podem ser implementadas ao métodos visando a melhorias da verificação, por exemplo, inicialmente foi utilizado a técnica de *Bounded Model Checking* no método, contudo a utilização da indução-k, que apresenta a técnica de BMC juntamente com técnica de desdobramento de loop(GADELHA et al., 2019), aumentou, devido a utilização do desdobramento de loop para encontrar os erros e do BMC para validação do erro, fato este que contribuiu para a qualidade na verificação dos códigos.

5.3 Resumo do capitulo

Neste capitulo foi apresentado os resultados das duas abordagens de tradução mostrando que o método de múltiplas traduções teve um desempenho melhor que o método de tradução direta, apresentando uma quantidade maior de códigos traduzidos e reconhecidos pela ferramenta

de análise. Também foi mostrado o resultado da análise com alguns códigos resultando da transformação de múltiplas transformações para demonstrar o desempenho da ferramenta.

6 CONSIDERAÇÕES FINAIS E TRABALHOS FUTUROS

O trabalho aborda o desenvolvimento de um método para análise de circuitos lógicos descrito em VHDL através da aplicação de técnica de transformação de código e a técnica *Model Cheking*. Este método tem como objetivo de explorar os estados alcançáveis no circuito, e dessa forma identificar erros que possam resultar em mal funcionamento do sistema.

O método apresentou um novo meio de realizar as transformações de código de VHDL para C e também um novo modelo de utilização de assertivas para validação de propriedades. O método de tradução de múltiplas abordagens que apresentou significativa melhoria, em torno de 44% de aumento na quantidade de códigos traduzidos e aceitos pela ferramenta de análise, mesmo que duas ferramentas tenham sido utilizadas para realização do processo.

O processo de inserção das assertivas através do método de pré-condições e pós-condições aumentou a capacidade de análise por parte do desenvolvedor, promovendo mais capacidades a ferramenta de tradução. A utilização de mais um método, a indução-k também possibilitou que o aumento da capacidade de análise de propriedades em programas.

Para trabalhos futuros, é necessário refinar a ferramenta e buscar melhorar a interação entre as ferramentas utilizadas no método de múltiplas transformações, com o intuito de aumenta a interação entre as ferramentas e evitar que erros na tradução possam ser gerados. Desta forma, mas também explorando cada vez mais as ferramentas auxiliares, é possível traçar os reais limites do método. Também é possível aumentar o nível da ferramenta através de análise de código com o objetivo de realizar a inserção automática das assertivas, promovendo ao desenvolvedor um teste totalmente automático pela ferramenta.

7 APÊNDICE A

7.1 Revisão sistemática

A revisão sistemática consiste é um método de identificação, análise e interpretação de pesquisas relevantes em determinada área ou questão de pesquisa. Para execução da revisão sistemática se requer um esforço maior, em comparação às pesquisas tradicionais, sendo necessário seguir uma sequência de passos metodológicos sobre a área ou questão de pesquisa ao qual deseja ser feita a pesquisa (KITCHENHAM, 2004).

Para a execução da revisão sistemática, é necessário um esforço considerável, se comparado a uma revisão informal a literatura. Enquanto que a revisão de literatura informal é conduzida de forma *ad-hoc*, sem planejamento e critérios de seleção estabelecidos a priori, a revisão sistemática requer um protocolo formal bem definido, com uma sequência de passos metodológicos, para conduzir uma pesquisa sobre o tema ao qual deseja-se realizar a pesquisa (MAFRA; TRAVASSOS, 2006).

A aplicação da revisão sistemática requer que seja seguido um conjunto bem definido e sequencial de passos, seguindo um protocolo de pesquisa desenvolvido previamente. Através deste método é possível realizar uma extensa pesquisa, contemplando uma grande quantidade de informações sobre o assunto pesquisado (MAFRA; TRAVASSOS, 2006). Este protocolo é construído considerando um tema específico que representa o elemento central da investigação, onde os passos da pesquisa, as estratégias definidas para coletar as evidências e o foco das questões de pesquisa são definidas explicitamente, de tal forma que outros pesquisadores sejam capazes de reproduzir o mesmo protocolo de pesquisa (BIOLCHINI et al., 2005).

Segundo Mafra e Travassos (2006), o processo para a condução de revisões sistemáticas envolve três etapas:

1. **Planejamento da Revisão:** os objetivos da pesquisa são listados e o protocolo da revisão é definido.

2. **Condução da Revisão:**nesta atividade, as fontes para a revisão sistemática são selecionadas, os estudos primários são identificados, selecionados e avaliados de acordo com os critérios de inclusão, exclusão, e de qualidade estabelecidos durante o protocolo da revisão.

3. **Análise dos Resultados:** os dados dos estudos são extraídos e sintetizados para análise e apresentação dos resultados.

Vale ressaltar que como o objetivo deste trabalho é realizar um estudo exploratório de caracterização de área podemos dizer que esta revisão sistemática se caracteriza como uma quasi-sistemática (TRAVASSOS et al., 2008), pois segue o mesmo processo da revisão sistemática e preserva o rigor e mesmo formalismo para as fases metodológicas de elaboração de protocolo e execução da revisão, mas sem a aplicação de uma meta-análise a princípio, que pode ser aplicada posteriormente.

7.1.1 Planejamento da Revisão Sistemática

Objetivo: Este estudo tem o objetivo esquematizado a partir da estrutura do paradigma GQM (do ingles *Goal,Question and Metric*)(BASILI et al., 1994).

Analisar	Publicações científicas através de um estudo baseado em revisão sistemática
Com propósito de	Identifica-las
Com relação as	Vantagens e desvantagens da utilização de assertiva e transformações de código na verificação do código na linguagem de descrição VHDL
Do ponto de vista do	Pesquisador
No contexto	Acadêmico ou industrial para verificação de assertivas na linguagem de descrição VHDL

Tabela 5 – Objetivo do estudo utilizando o paradigma GQM

Formulação da Pergunta: Buscamos respostas para as seguintes perguntas:

- **Q1:** Quais são os métodos para verificação de circuitos lógicos descritos na linguagem de programação VHDL?
  - **Q1.1:** Foi desenvolvido e está disponível alguma ferramenta para aplicação do método?
  - **Q1.2:** Qual a técnica de exploração de estados para circuitos lógicos?
  - **Q1.3:** O método proposto é baseado em técnicas de verificação de software?
  - **Q1.4:** Como o método proposto valida pré e pós condições no programa?
  - **Q1.3:** Foi utilizado algum *benchmark* de programas em VHDL para experimentação e o mesmo encontra-se disponível?
  - **Q1.4:** Quais as perspectivas futuras para melhorar da aplicação do método proposto?

Escopo da pesquisa: Na delimitação do escopo da pesquisa foram estabelecidos critérios, buscando garantir a viabilidade da execução, acessibilidade dos dados e abrangência do estudo realizado. A pesquisa dar-se-á a partir de bibliotecas digitais através das suas respectivas máquinas de busca e, quando os dados não estiverem disponíveis eletronicamente, através de consultas manuais.

Critérios de Seleção de Fontes. Para as bibliotecas digitais é desejado:

- Possuir uma máquina de busca que permita o uso de expressões lógicas ou mecanismo equivalente.
- Incluir em sua base, publicações de exatas ou correlatas que possuam relação direta com o tema a ser pesquisado
- As máquinas de busca deverão permitir a busca no texto completo das publicações.

Os mecanismos de busca utilizados devem garantir resultados únicos através da busca de um mesmo conjunto de palavras-chaves (*string* de busca). Quando isto não for possível, deve-se estudar e documentar uma forma de minimizar os potenciais efeitos colaterais desta limitação.

Métodos de Busca das Publicações. As fontes digitais foram acessadas via Web, através de expressões de busca pré-estabelecidas. A biblioteca digital consultada foi a Scopus, acessível em http://www.scopus.com. Segundo Elsevier (2017), a Scopus é uma das maiores bases de dados de resumos e citações da literatura de pesquisa *peer-reviewed* com mais de $22,800$ títulos de mais de $5,000$ editoras internacionais abrangendo as áreas de tecnologia, medicina, artes, ciências sociais e com atualizações diárias. Entre as editoras podemos citar: Springer (SPRINGER, 2018); IEEE Xplore Digital Library (IEEE, 2018);ACM Digital Library (ACM, 2018) ; ScienceDirect/Elsevier (B.V, 2018); Wiley Online Library (SONS, 2018); dentre outras.

***String* de Busca.** A *string* de busca foi definida segundo o padrão PICO (do inglês *Population, Intervention, Comparison, Outcomes*) (KITCHENHAM et al., 2009), conforme a estrutura abaixo:

- População: Trabalhos publicados em conferências e periódicos que relacionam verificação de propriedades de circuitos lógicos em códigos para a descrição de hardware.
- Intervenção: Verificação de propriedades relacionadas a verificação de circuitos para as diferentes estruturas das linguagens de descrição de hardware;
- Comparação: análise de cobertura e suporte das abordagens identificadas para a verificação de propriedades das linguagens de descrição de hardware;
- Resultados: a partir da descrição das abordagens pretende-se verificar a cobertura que cada abordagem apresenta na manipulação das diferentes estruturas da linguagem de descrição de hardware para a verificação de propriedades baseada em assertivas.

Este estudo representa um estudo de mapeamento/caracterização, a string de busca foi definida de acordo com dois aspectos: População e Intervenção, como é apresentado na estrutura abaixo. Posteriormente esta mesma *string* de busca será executada na biblioteca Scopus para busca de artigos e publicações de modo a gerar uma interseção entre população e intervenção.

- População: publicações que fazem referência a verificação de propriedades de circuitos lógicos e sinônimos:
  - **Palavras-chaves:** "circuit checker"OR "circuit verification"OR "contract based verification"OR "code analysis"OR "static analysis"OR "dynamic analysis"OR "safety verification"OR "RTL analysis"OR "program analysis"OR "property verification"OR "formal verification"OR "model checking"OR "model checker"OR "hardware checker"OR "hardware verification"OR "validity checker"OR "hardware assertion"OR "assertion checker"OR "assert verification"OR "assertion-based verification"OR "assertion based verification"OR "assertion-based design"OR "bit level verification"
- Intervenção: Verificação de circuitos e sinônimos:
  - **Palavras-chaves:**"hardware statement"OR "hardware code"OR "hardware source code"OR "hardware semantics"OR "hardware property"OR "logical gates"OR "sequential circuit"OR "parallel circuit"OR "real circuits"OR "complex circuits"OR "control circuitry"OR ”software netlist” OR “hardware emulation” OR “silicon debug”

7.1.1.1 Procedimentos de Seleção e Critérios

A estratégia de busca será aplicada por um pesquisador para identificar as publicações em potencial. A seleção das publicações dar-se-á em 2 etapas, conforme apresentado a seguir.

1. **Seleção e catalogação preliminar dos dados coletados:** A seleção preliminar das publicações será feita a partir da aplicação da string de busca na biblioteca Scopus, o resultado desta busca corresponde a seleção preliminar. Todas as publicações serão armazenadas para análise posterior;
2. **Seleção dos dados relevantes - [1 filtro]:** A seleção preliminar com o uso da expressão de busca não garante que todo o material coletado seja útil no contexto da pesquisa, pois a aplicação das expressões de busca é restrita ao aspecto sintático. Por isso, é necessário a criação de filtros de exclusão e inclusão, de modo a classificar os artigos e publicações que se enquadram no contexto da pesquisa realizada. Nesta etapa apenas serão lidos o título, abstracts, e palavras-chaves e classificar de acordo com os filtros, ou seja, se será aceito ou excluído. Devem ser excluídas as publicações contidas no conjunto preliminar que:
   - **CE1-01:** Não serão selecionadas publicações em que as palavras-chave da busca não apareçam no título, resumo e/ou texto da publicação (excluem-se os seguintes campos: as seções de agradecimentos, biografia dos autores, referências bibliográficas e anexas).
   - **CE1-02:** Não serão selecionadas publicações em que descrevam e/ou apresentam ‘keynote speeches’, tutoriais, cursos e similares.

- **CE1-03:** Não serão selecionadas publicações em que o contexto das palavras-chave utilizadas no artigo leve a crer que a publicação não cita uma abordagem para verificação/validação de códigos para descrição de hardware.
- **CE1-04:** Não serão selecionadas publicações em que o contexto das palavras-chave utilizadas no artigo leve a crer que a publicação não cita uma abordagem para verificação de código de descrição de hardware baseado em assertivas ou propriedades de verificação de hardware.

Podem ser incluídas apenas as publicações contidas no conjunto preliminar que:

- **CI1-01:**Podem ser selecionadas publicações em que o contexto das palavras-chave utilizadas no artigo leve a crer que a publicação cita uma abordagem para verificação de código de descrição de hardware baseado em assertivas ou propriedades de verificação de hardware.
- **CI1-02:**Podem ser selecionadas publicações em que o contexto das palavras-chave utilizadas no artigo leve a crer que a publicação cita recomendações de melhoria na utilização de abordagens para verificação de código de descrição de hardware baseado em assertivas ou propriedades de verificação de hardware.

3. **Seleção dos dados relevantes - [2 filtro]:** Apesar do 1º filtro limitar o universo de busca, infelizmente não há garantias de que todo material selecionado no filtro anterior seja útil no contexto da pesquisa. Neste caso, novos filtros são gerados, buscando, da mesma forma que o primeiro filtro, classificar os artigos e publicações que se enquadram no contexto da pesquisa. Para este filtro é necessário a leitura na íntegra dos artigos selecionados anteriormente. O objetivo é identificar que relacionam assertivas e/ou propriedade de verificação de hardware.

   - **CS2 -ASS -VER_HARD:** Não devem ser selecionadas publicações que não contextualizam verificação de hardware e não citam assertivas
   - **CS2 +ASS -VER_HARD:** Não devem ser selecionadas publicações que não contextualizam verificação de hardware, mas citam assertivas
   - **CS2 -ASS +VER_HARD:** Não devem ser selecionadas publicações que contextualizam verificação de hardware, mas citam assertivas.

   Dessa forma, todas as publicações devem respeitar o critério abaixo:

   - **CS3 +ASS +VER_HARD:** Serão selecionadas publicações que contextualizam verificação de hardware e citam assertivas em seu contexto.

Devido ao tempo necessário para o desenvolvimento execução dos filtros da revisão sistemática serem extensos, apenas o primeiro filtro foi executado neste projeto, mesmo que

os parâmetros para o segundo filtro já tenho sido desenvolvidos. Em contrapartida foram selecionados cinco artigos que serviram como ferramentas de estudo para o desenvolvimento da metodologia.

REFERÊNCIAS

ABE, J. **Introducao a Logica Para a Ciencia Da Computacao**. Arte & Ciência, 2002. ISBN 9788574730455. Disponível em: <https://books.google.com.br/books?id=AgBJdjoNPwUC>.

ACM. 2018. Acessado em: 01 de março de 2018. Disponível em: <http://dl.acm.org/>.

AHO, A. V.; SETHI, R.; ULLMAN, J. D. **Compilers: principles, techniques, and tools**. [S.l.]: Addison-wesley Reading, 2007. v. 2.

ALUR, R.; DILL, D. L. A theory of timed automata. **Theoretical computer science**, Elsevier, v. 126, n. 2, p. 183–235, 1994.

BAIER, C.; KATOEN, J.-P.; LARSEN, K. G. **Principles of model checking**. [S.l.]: MIT press, 2008.

BARA, A.; BAZARGAN-SABET, P.; CHEVALLIER, R.; LEDU, D.; ENCRENAZ, E.; RENAULT, P. Formal verification of timed vhdl programs. In: IET. **FDL**. [S.l.], 2010. p. 80–85.

BASILI, V. R.; CALDIERA, G.; ROMBACH, H. D. Experience factory. **Encyclopedia of software engineering**, Wiley Online Library, 1994.

BENSALEM, S.; LAKHNECH, Y. Automatic generation of invariants. **Formal Methods in System Design**, Springer, v. 15, n. 1, p. 75–92, 1999.

BEYER, D.; KEREMOGLU, M. E. Cpachecker: A tool for configurable software verification. In: SPRINGER. **International Conference on Computer Aided Verification**. [S.l.], 2011. p. 184–190.

BIERE, A. The aiger and-inverter graph (aig) format. **Available at http://fmv.jku.at/aiger**, 2016.

BIERE, A.; CIMATTI, A.; CLARKE, E. M.; FUJITA, M.; ZHU, Y. Symbolic model checking using sat procedures instead of bdds. In: ACM. **Proceedings of the 36th annual ACM/IEEE Design Automation Conference**. [S.l.], 1999. p. 317–320.

BIERE, A.; CIMATTI, A.; CLARKE, E. M.; STRICHMAN, O.; ZHU, Y. et al. Bounded model checking. **Advances in computers**, v. 58, n. 11, p. 117–148, 2003.

BIOLCHINI, J.; MIAN, P. G.; NATALI, A. C. C.; TRAVASSOS, G. H. Systematic review in software engineering. **System Engineering and Computer Science Department COPPE/UFRJ, Technical Report ES**, v. 679, n. 05, p. 45, 2005.

BLANCHET, B.; COUSOT, P.; COUSOT, R.; FERET, J.; MAUBORGNE, L.; MINÉ, A.; MONNIAUX, D.; RIVAL, X. A static analyzer for large safety-critical software. In: ACM. **ACM SIGPLAN Notices**. [S.l.], 2003. v. 38, n. 5, p. 196–207.

BOULE, M.; ZILIC, Z. Incorporating efficient assertion checkers into hardware emulation. In: IEEE. **Computer Design: VLSI in Computers and Processors, 2005. ICCD 2005. Proceedings. 2005 IEEE International Conference on**. [S.l.], 2005. p. 221–228.

BOULE, M.; ZILIC, Z. Efficient automata-based assertion-checker synthesis of seres for hardware emulation. In: IEEE COMPUTER SOCIETY. **Proceedings of the 2007 Asia and South Pacific Design Automation Conference**. [S.l.], 2007. p. 324–329.

BRAYTON, R.; MISHCHENKO, A. Abc: An academic industrial-strength verification tool. In: SPRINGER. **Computer Aided Verification**. [S.l.], 2010. p. 24–40.

B.V. 2018. Acessado em 1 de março de 2018. Disponível em: <http://www.sciencedirect.com/>.

CABODI, G.; LOIACONO, C.; PALENA, M.; PASINI, P.; PATTI, D.; QUER, S.; VENDRAMINETTO, D.; BIERE, A.; HELJANKO, K. Hardware model checking competition 2014: an analysis and comparison of solvers and benchmarks. **Journal on Satisfiability, Boolean Modeling and Computation**, v. 9, p. 135–172, 2016.

CAPPELATTI, D. **Praticando VHDL**. Editora Feevale, 2010. ISBN 9788577171200. Disponível em: <https://books.google.com.br/books?id=z4_CYog_UskC>.

CHRISTEN, E.; BAKALAR, K. Vhdl-ams-a hardware description language for analog and mixed-signal applications. **IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing**, IEEE, v. 46, n. 10, p. 1263–1272, 1999.

CHU, P. P. **RTL hardware design using VHDL: coding for efficiency, portability, and scalability**. [S.l.]: John Wiley & Sons, 2006.

CLARKE, E.; FEHNKER, A.; HAN, Z.; KROGH, B.; STURSBERG, O.; THEOBALD, M. Verification of hybrid systems based on counterexample-guided abstraction refinement. In: SPRINGER. **TACAS**. [S.l.], 2003. v. 3, p. 192–207.

CLARKE, E. M. The birth of model checking. In: **25 Years of Model Checking**. [S.l.]: Springer, 2008. p. 1–26.

CLARKE, E. M.; GRUMBERG, O.; LONG, D. E. Model checking and abstraction. **ACM transactions on Programming Languages and Systems (TOPLAS)**, ACM, v. 16, n. 5, p. 1512–1542, 1994.

CORDEIRO, L.; FISCHER, B.; MARQUES-SILVA, J. Smt-based bounded model checking for embedded ansi-c software. **IEEE Transactions on Software Engineering**, IEEE, v. 38, n. 4, p. 957–974, 2012.

COUSOT, P.; COUSOT, R. Systematic design of program transformation frameworks by abstract interpretation. In: ACM. **ACM SIGPLAN Notices**. [S.l.], 2002. v. 37, n. 1, p. 178–190.

COUSOT, P.; COUSOT, R. **A gentle introduction to formal verification of computer systems by abstract interpretation**. [S.l.]: IOS Press, 2010.

DAHAN, A.; GEIST, D.; GLUHOVSKY, L.; PIDAN, D.; SHAPIR, G.; WOLFSTHAL, Y.; BENALYCHERIF, L.; KAMIDEM, R.; LAHBIB, Y. Combining system level modeling with assertion based verification. In: **Sixth international symposium on quality electronic design (isqed'05)**. [S.l.: s.n.], 2005. p. 310–315. ISSN 1948-3287.

DONALDSON, A. F.; HALLER, L.; KROENING, D.; RÜMMER, P. Software verification using k-induction. In: SPRINGER. **International Static Analysis Symposium**. [S.l.], 2011. p. 351–368.

DOOLITTLE, L. 2015. Acessado em: 13 de Outubro de 2018. Disponível em: <http://doolittle.icarus.com/~larry/vhd2vl/>.

D'SILVA, V.; KROENING, D.; WEISSENBACHER, G. A survey of automated techniques for formal software verification. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, IEEE, v. 27, n. 7, p. 1165–1178, 2008.

DUOLOS, D. . D. K. 2018. Acessado em: 27 de junho de 2018. Disponível em: <https://www.doulos.com/knowhow/psl/structure_psl/>.

ELSEVIER. 2017. Acessado em: 01 março 2018. Disponível em: <https://www.elsevier.com/__data/assets/pdf_file/0007/69451/0597-Scopus-Content-Coverage-Guide-US-LETTER-v4-HI-singles-no-ticks.pdf>.

ERLINGSSON, U.; SCHNEIDER, F. B. Sasi enforcement of security policies: A retrospective. In: IEEE. **DARPA Information Survivability Conference and Exposition, 2000. DISCEX'00. Proceedings**. [S.l.], 2000. v. 2, p. 287–295.

ESBMC. **ESBMC (Efficient SMT-Based Context-Bounded Model Checker)**. 2018. Acessado em: 16 de janeiro 2018. Disponível em: <http://esbmc.org/>.

G1. **Acidente foi provocado por falha em circuito eletrônico, diz secretário**. 2012. Disponível em: <http://g1.globo.com/sao-paulo/noticia/2012/05/acidente-foi-provocado-por-falha-em-circuito-eletronico-diz-secretario.html>.

GADELHA, M. R.; MONTEIRO, F.; CORDEIRO, L.; NICOLE, D. Esbmc v6. 0: Verifying c programs using k-induction and invariant inference. 2019.

GADELHA, M. Y.; ISMAIL, H. I.; CORDEIRO, L. C. Handling loops in bounded model checking of c programs via k-induction. **International Journal on Software Tools for Technology Transfer**, Springer, v. 19, n. 1, p. 97–114, 2017.

GATTI, A.; GHEZZI, C. **VHDL 2 C: Studio e realizzazione di una tecnica di traduzione automatica del VHDL in linguagio C**. 1995. Http://web.tiscali.it/sito01/pro/v2c/main.htm.

GUGLIELMO, G. D.; GUGLIELMO, L. D.; FUMMI, F.; PRAVADELLI, G. On the use of assertions for embedded-software dynamic verification. In: IEEE. **Design and Diagnostics of Electronic Circuits & Systems (DDECS), 2012 IEEE 15th International Symposium on**. [S.l.], 2012. p. 330–335.

GUPTA, A. Formal hardware verification methods: A survey. In: SPRINGER. **Computer-Aided Verification**. [S.l.], 1992. p. 5–92.

HALDER, A.; VENKATESWARLU, A. A study of petri nets modeling analysis and simulation. **Department of Aerospace Engineering Indian Institute of Technology Kharagpur, India**, 2006.

HODER, K.; KOVÁCS, L.; VORONKOV, A. Interpolation and symbol elimination in vampire. **Automated Reasoning**, Springer, p. 188–195, 2010.

IDOETA, I. V.; CAPUANO, F. G. **Elementos de eletrônica digital**. [S.l.]: Livros Erica, 1982.

IEEE. Ieee standard for verilog hardware description language. **IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)**, p. 1–560, 2006.

IEEE. Ieee standard for property specification language (psl). **IEEE Std 1850-2010 (Revision of IEEE Std 1850-2005)**, p. 1–182, April 2010.

IEEE. Iec/ieee international standard - behavioural languages - part 1-1: Vhdl language reference manual. **IEC 61691-1-1:2011(E) IEEE Std 1076-2008**, p. 1–648, May 2011.

IEEE. Ieee standard for standard systemc language reference manual. **IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)**, p. 1–638, Jan 2012.

IEEE. 2018. Acessado em: 1 de março de 2018. Disponível em: <http://ieeexplore.ieee.org/Xplore/home.jsp>.

JAIN, S.; NAIK, P. K.; BHOOSHAN, S. V. Petri nets: an application in digital circuits. In: **ICWET**. [S.l.: s.n.], 2010. p. 1005.

JIN, G.; LI, Z.; CHEN, F. A theoretical foundation for program transformations to reduce cache thrashing due to true data sharing. **Theoretical computer science**, Elsevier, v. 255, n. 1-2, p. 449–481, 2001.

KILDALL, G. A. A unified approach to global program optimization. In: ACM. **Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages**. [S.l.], 1973. p. 194–206.

KITCHENHAM, B. Procedures for performing systematic reviews. **Keele, UK, Keele University**, v. 33, n. 2004, p. 1–26, 2004.

KITCHENHAM, B.; BRERETON, O. P.; BUDGEN, D.; TURNER, M.; BAILEY, J.; LINKMAN, S. Systematic literature reviews in software engineering–a systematic literature review. **Information and software technology**, Elsevier, v. 51, n. 1, p. 7–15, 2009.

KOSCIANSKI, A.; SOARES, M. dos S. **Qualidade de Software - 2ª Edição: Aprenda as metodologias e técnicas mais modernas para o desenvolvimento de software**. Novatec, 2007. ISBN 9788575221129. Disponível em: <https://books.google.com.br/books?id=O9aWoUq6L88C>.

KROPF, T. **Introduction to Formal Hardware Verification**. Springer Berlin Heidelberg, 2013. ISBN 9783662038093. Disponível em: <https://books.google.com.br/books?id=7ImrCAAAQBAJ>.

LARSEN, K. G.; PETTERSSON, P.; YI, W. Uppaal in a nutshell. **International journal on software tools for technology transfer**, Springer, v. 1, n. 1-2, p. 134–152, 1997.

MAFRA, S. N.; TRAVASSOS, G. H. Estudos primarios e secundarios apoiando a busca por evidencia em engenharia de software. **Relatorio Tecnico, RT-ES**, v. 687, n. 06, 2006.

MUKHERJEE, R.; SCHRAMMEL, P.; KROENING, D.; MELHAM, T. Unbounded safety verification for hardware using software analyzers. In: EDA CONSORTIUM. **Proceedings of the 2016 Conference on Design, Automation & Test in Europe**. [S.l.], 2016. p. 1152–1155.

MUKHERJEE, R.; TAUTSCHNIG, M.; KROENING, D. v2c–a verilog to c translator. In: SPRINGER. **International Conference on Tools and Algorithms for the Construction and Analysis of Systems**. [S.l.], 2016. p. 580–586.

MURATA, T. Petri nets: Properties, analysis and applications. **Proceedings of the IEEE**, IEEE, v. 77, n. 4, p. 541–580, 1989.

RAMESH, U. B. K.; SENTILLES, S.; CRNKOVIC, I. Energy management in embedded systems: Towards a taxonomy. In: IEEE PRESS. **Proceedings of the First International Workshop on Green and Sustainable Software**. [S.l.], 2012. p. 41–44.

ROCHA, H.; CORDEIRO, L.; BARRETO, R.; NETTO, J. Exploiting safety properties in bounded model checking for test cases generation of c programs. 2010.

ROCHA, H.; ISMAIL, H.; CORDEIRO, L.; BARRETO, R. Model checking embedded c software using k-induction and invariants (extended version). **arXiv preprint arXiv:1509.02471**, 2015.

ROCHA, H. O. et al. Verificacao de sistemas de software baseada em transformacoes de codigo usando bounded model checking. Universidade Federal do Amazonas, 2015.

SARGENT, R. G. Verification and validation of simulation models. In: WINTER SIMULATION CONFERENCE. **Proceedings of the 37th conference on Winter simulation**. [S.l.], 2005. p. 130–143.

SEGER, C.-J. **An introduction to formal hardware verification**. [S.l.]: University of British Columbia, Department of Computer Science, 1992.

SOMMERVILLE, I. **Engenharia de software**. PEARSON BRASIL, 2011. ISBN 9788579361081. Disponível em: <https://books.google.com.br/books?id=H4u5ygAACAAJ>.

SONS, J. W. . 2018. Acessado em: 1 de março de 2018. Disponível em: <http://onlinelibrary.wiley.com/>.

SOUZA, J. **Lógica para Ciência da Computação**. Elsevier Brasil, 2017. ISBN 9788535278255. Disponível em: <https://books.google.com.br/books?id=Ds2sCQAAQBAJ>.

SPRINGER. 2018. Acessado em: 1 de março de 2018. Disponível em: <https://link.springer.com/>.

SRIKANT, Y.; SHANKAR, P. **The Compiler Design Handbook: Optimizations and Machine Code Generation**. CRC Press, 2002. ISBN 9781420040579. Disponível em: <https://books.google.com.br/books?id=0K_jIsgyNpoC>.

THOMAS, D.; MOORBY, P. **The Verilog® Hardware Description Language**. Springer US, 2008. ISBN 9780387853444. Disponível em: <https://books.google.com.br/books?id=DxQGrz7q-SwC>.

TOCCI, R. J.; WIDMER, N. S.; MOSS, G. L. **Sistemas digitais: princípios e aplicações**. [S.l.]: Prentice Hall, 2003. v. 8.

TRAVASSOS, G. H.; SANTOS, P. S. M. dos; MIAN, P. G.; NETO, A. C. D.; BIOLCHINI, J. An environment to support large scale experimentation in software engineering. In: IEEE. **Engineering of Complex Computer Systems, 2008. ICECCS 2008. 13th IEEE International Conference on**. [S.l.], 2008. p. 193–202.

YAKOVLEV, A. V.; KOELMANS, A. M. Petri nets and digital hardware design. In: SPRINGER. **Advanced Course on Petri Nets**. [S.l.], 1996. p. 154–236.

YANG, H.; SUN, Y. Reverse engineering and reusing cobol programs: A program transformation approach. In: BCS LEARNING & DEVELOPMENT LTD. **Proceedings of the 1st Irish conference on Formal Methods**. [S.l.], 1997. p. 181–200.