Framework VHDL para LingPON 2.0

Edmar A. Lanes Junior

***Resumo*—O Paradigma Orientado a Notificações (PON) foi concebido como alternativa aos paradigmas Declarativo (PD) e Imperativo (PI). A linguagem LingPON, já em sua segunda versão, foi concebida para concretizar o PON. O compilador da linguagem LingPON 2.0 trouxe uma estrutura centralizada para a representação do código a ser compilado (GrafiPON). Este trabalho teve como objetivo desenvolver um framework para a conversão de LingPON para VHDL, linguagem de descrição para hardware. Este framework foi desenvolvido em C++ e é capaz de converter código com uma única instância de fbe com a ressalva de ter algumas falhas na interpretação das conjunções entre condições a ser solucionada em trabalhos futuros.**

***Index Terms*—Paradigma Orientado a Notificações, LingPON 2.0, GrafiPON, VHDL, Linguagens e Compiladores**

I. INTRODUÇÃO

ATUALMENTE, as principais técnicas de programação aplicadas na criação de algoritmos podem ser qualificadas como implementações dos consagrados Paradigma Imperativo (PI) e Paradigma Declarativo (PD). Estas técnicas geram, na maioria das vezes, aplicações com processamento desnecessário, gerados por motivos como redundâncias em avaliações causais e utilizações de estruturas de dados computacionalmente custosas. [1]

Para trazer uma alternativa a estes paradigmas, foi proposto o Paradigma Orientado a Notificações (PON). Este paradigma trabalha com atributos (Attributes) que notificam premissas (Premisses) relacionadas logicamente por condições e sub-condições (Conditions e Sub-conditions) que são organizadas por meio de regras (Rules) e ações (Actions) a serem executadas no caso das regras serem satisfeitas. [2]

De modo a concretizar o PON, foi elaborada a linguagem de programação LingPON. Em sua segunda versão (LingPON 2.0) ela apresenta a possibilidade de tratar instâncias hierárquicas.

As análises léxica e sintática do código LingPON 2.0 é realizada pelo compilador PON por meio dos softwares flex e bison. Este software gera uma estrutura de grafo que representa o programa a ser compilado (GrafiPON).

A ideia deste trabalho foi desenvolver um interpretador do GrafiPON capaz de transcrever seu código na linguagem de descrição de circuitos VHDL. Foi utilizado como base, o trabalho de Kershbaumer que possibilitou a conversão de código LingPON 1.0 para VHDL.[3]

A seção II deste documento trata sobre a contextualização com relação a linguagens, compiladores e o PON. A seção III traz informações sobre o estado da arte e detalhes da implementação do framework. A seção IV apresenta o código teste utilizado e o resultado da compilação. Na seção V, é relatada a experiência com o LingPON 2.0 e na seção final são apresentadas as considerações gerais sobre o trabalho.

II. CONTEXTUALIZAÇÃO

Uma linguagem de programação é um conjunto de regras sintáticas e semânticas para definir um programa de computador.

O código de um programa de computador, para ser compilado, deve passar pelos estágios de análise léxica e sintática.

Como mostra a figura 2.1, a análise léxica consiste em transformar um fluxo de símbolos em um fluxo de tokens. Um notável software analisador léxico é o Lex.

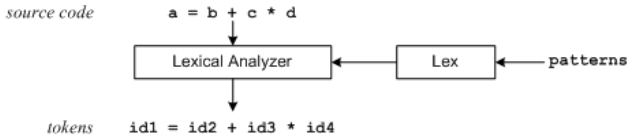

Figura 2.1 - Analise Léxica [4]

Na análise sintática, como ilustra a figura 2.2, os tokens são estruturados em árvore. O Bison é uma opção de software para este fim.

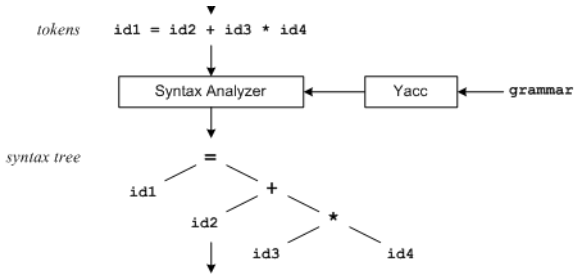

Figura 2.2 – Análise Sintática [4]

A árvore obtida, ao termino da análise sintática é a base para a geração de código na linguagem alvo.

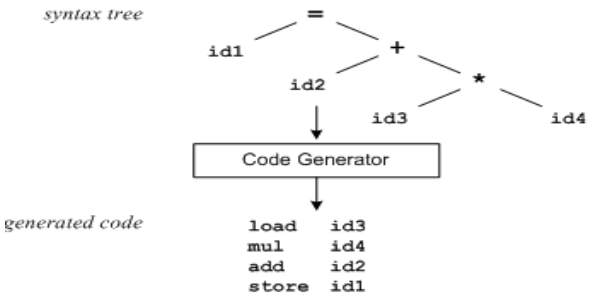

Figura 2.3 – Geração de Código [4]

Proposto pelo professor Dr J. M, Simão, o Paradigma Orientado a Notificações (PON) é uma alternativa aos consagrados Paradigma Declarativo (PD) e Paradigma Imperativo (PI).[5]

O PON possui um fluxo orientado a notificações em eventos de alteração de valor. É composto pelas estruturas:

- Atributos: armazenam dados sensíveis e notificam premissas
- Premissas: comparam atributos a constantes ou a outros atributos e notificam condições e/ou subcondições
- Condições e Subcondições: condições relacionam premissas, podem estar subdivididas em subcondições e são organizadas em regras
- Regras: organizam condições e notificam ações
- Ações: são subdivididas em Instigações que executam Chamadas
- Chamadas: executam métodos
- Métodos: Modificam o valor de atributos

A figura 2.4 apresenta o fluxo padrão das notificações no PON.

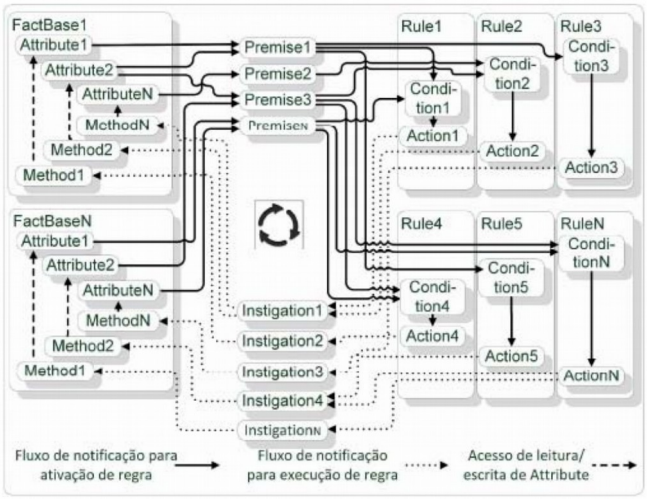

Figura 2.4 – Fluxo Padrão das Informações no PON [6]

A linguagem de programação do PON é o LingPON. Atualmente na versão 2.0, melhorias podem ser observadas como a utilização de instâncias hierarquicas, estruturação centralizada dos tokens (GrafiPON), opções de execução de instigations e calls sequencial ou paralelamente e blocos code para escrita de métodos. [6]

III. ESTADO DA ARTE E IMPLEMENTAÇÃO

O Framework 2.0 para VHDL é capaz de gerar código VHDL a partir de programas em PON que possuem apenas a instância main com a ressalva de ainda haver um erro na interpretação das condições a ser discutido mais a frente. Esta implementação foi projetada para também realizar testes em programas que se utilizam da hierarquia de instâncias, ou seja, com mais de uma instância. Contudo, esta funcionalidade carece de testes.

Algumas restrições foram inseridas, para uma melhor aderência à linguagem VHDL. A primeira delas, se refere aos atributos. O compilador aceita atributos do tipo bit, integer, char e string. Este último deve ser utilizado apenas para descrever números binários, por exemplo “01010”.

Uma segunda restrição importante se refere à execução paralela chamadas (calls) e instigações (instigations). Embora o LingPON 2.0 permita a utilização de calls e instigations sequenciais, as mesmas não são adequadas para uma linguagem tipicamente paralela como VHDL. Incluir blocos sequenciais, pode provocar um considerável atraso de propagação, limitando o clock de entrada.

Outras restrições dependem de mudanças na linguagem LingPON 2.0. Não foi possível implementar a atribuição aritmética, pois o GrafiPON não estava preparado para ela durante o desenvolvimento deste framework. A geração de métodos foi adaptada para implementar o bloco code, mas esta implementação se tornou inviável devido a falta de uma estrutura que guarde um tipo de retorno para o método ao invés de uma atribuição.

Foram aproveitados os blocos de premissas e atributos usados por Kerschbaumer. [3] Os métodos são implementados como entidades (entities) geradas em tempo de execução para permitir uma futura implementação do bloco de código (code).

Nas subseções a diante são descorridas alguns detalhes de implementação do framework para geração de código VHDL a partir do GrafiPON

A. Metodos que Percorrem Instâncias

Para receber referências a todos os atributos, métodos e regrasde toda a hierarquia, são utilizados métodos recursivos que varrem todo o grafo e retornam mapas para estes componentes. Estes métodos são:

- **map<string, Attribute*>* getAllMyAttributes (Instance* instance)**:
  - Mapa de todos os atributos com chave “<nomeDaInstancia>_<nomeDoAtributo>”
- **map<string, Method*>* getAllMyMethods (Instance* instance):**
  - Mapa de todos os métodos com chave “<nomeDaInstancia>_<nomeDoMétodo>”
- **map<string, Rule*>* getAllMyRules(Instance* instance):**
  - Mapa de todas as regras com chave “<nomeDaInstancia>_<nomeDaRegra>”

B. Estruturas de Impressão de Código

O código VHDL segue uma estrutura com blocos bastante definidos. A sequência de análise dos componentes de PON não é exatamente compatível para uma impressão sequencial. Assim, o código é organizado em blocos que são inseridos em variáveis do tipo std::string, para depois ser impresso. Estes blocos são:

- **fbe:** sinais de entrada e saída do bloco main

- **component:** componentes utilizados em main (atributos, métodos e premissas)
- **methods_cl:** definições das entities que representam os métodos
- **signals:** ligações entre componentes internos
- **attributes:** instanciação de componentes atributos
- **premisesSt:** instanciação de componentes premissas
- **methods:** instanciação de componentes métodos (a partir de calls)
- **att_in:** definição de entradas para os atributos
- **att_out:** definição de saídas para os atributos
- **pre_in:** definição de entradas para as premissas
- **met_in:** definição de entradas para os métodos (a partir das rules)

Para imprimir estes blocos na estrutura do VHDL, é usado método void print (std::string fbeName, <blocos de código>). Este método é apresentado na figura 2.1. Os blocos de código que são parâmetros do método foram, em parte, omitidos para fins de simplificação.

```
void Framework_2_VHDL::print(std::string fbeName, std::string fbe, std::string filename, std::strin
{
    std::ofstream filestream;
    filestream.open(filename, std::fstream::out);
    filestream<<"LIBRARY ieee;\nUSE ieee.std_logic_1164.all;\nUSE work.data_type_pkg.all;\n\n";
    filestream<<methods_cl<<std::endl;
    filestream<<"\n\nENTITY "<<fbeName<<" IS\nPORT(\n"<<fbe<<"\n);\nEND "<<fbeName<<";\n\n";

    filestream<<"ARCHITECTURE "<<fbeName<<"_arq OF "<<fbeName<<" IS\n";
    filestream<<component<<std::endl;
    filestream<<signals<<std::endl;
    filestream<<"BEGIN"<<std::endl;
    filestream<<attributes<<"\n\n";
    filestream<<premisses<<"\n\n";
    filestream<<methods<<"\n\n";

    filestream<<att_in<<"\n\n";
    filestream<<att_out<<"\n\n";
    filestream<<pre_in<<"\n\n";
    filestream<<met_in<<"\nEND "<<fbeName<<";";

    filestream.close();
}
```

Figura 2.1 – Código do método print. A maior parte dos parâmetros representada por blocos de código de tipo std::string foi omitida para fins de simplicidade.

C. Atributos e Premissas

Tanto para atributos quanto premissas foram implementados os blocos propostos por Kerschbaumer. [3]

Os atributos possuem número de bits parametrizável e um número múltiplo de barramentos de entrada (newValue), pinos de set (setValue) e um barramento de saída de valor atual (Value). Cada barramento atende a uma referência a este atributo seja de uma atribuição de método, ou de uma ligação a um barramento externo. Portanto, um atributo deve ter um número de barramentos newValue e pinos setValue igual ao número de chamadas e/ou pinos externos que atribuem valor a ele. Quando um destes métodos modifica o atributo, envia um valor e aciona o pino set. O código do atributo, utiliza o clock para iterar pelos pinos de set. Quando encontra um pino set acionado, modifica Value. A figura 2.2 apresenta um diagrama de blocos deste componente.

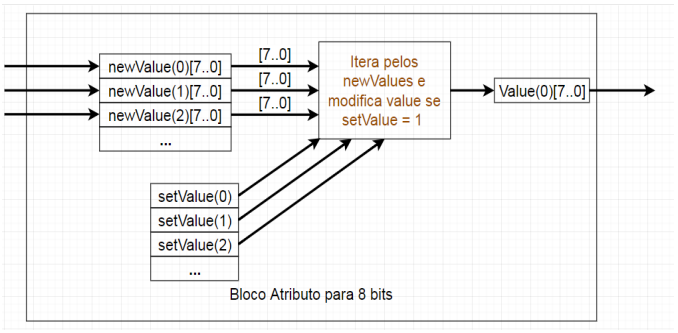

Figura 2.2 – Representação em Diagrama de Blocos de um Componente Attribute.

Já o bloco de premissa possui dois barramentos de entrada de tamanho parametrizável e uma operação de comparação destas duas entrada que também é parametrizavel. O componente faz a comparação requisitada entre as duas entradas e modifica seu bit de saída para true ou false de acordo com o resultado. A figura 2.3 apresenta o diagrama de blocos deste componente.

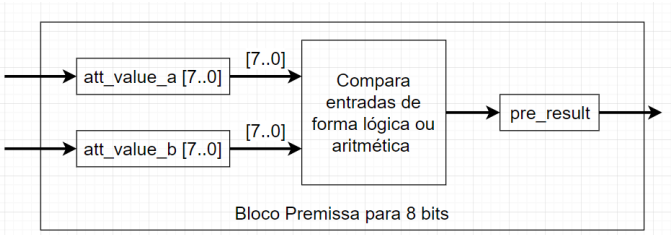

Figura 2.3 – Diagrama de Blocos de um Componente Premise

D. Métodos e Chamadas

Os métodos são modelados como entidades (entities) geradas em tempo de execução. Esta arquitetura confere duas características interessantes de customização. A primeira diz respeito à possibilidade de haver um número variado de parâmetros de entrada. A segunda é o suporte ao bloco de código (code) que pode ser inserido na ARCHITECTURE da entidade que representa o método. Mas, para que a segunda se torne utilzável, é necessária uma forma de vinculação do método a um atributo. Uma forma proposta, seria o método ter uma estrutura que representasse um tipo de retorno. Desta forma, poderia a chamada vincular o atributo ao método, ou seja, a chamada poderia ser algo como **<atributo> = <metodo>( )**.

Métodos contam com as entradas oppAttA e oppAttB que são utilizadas em atribuições. O barramento de saída(output), o pino execute que serve de gatilho para execução do método e o pino de saída que notifica atributos modificados, completam a entidade.

Chamadas, por sua vez, são implementadas como instâncias da entidade que representa o método que referenciam. Assim, cada chamada é tratada como um componente devidamente instanciado.

E. *Regras, Condições e Subcondições*

A estrutura de árvore representada pelas regras condições e subcondições é representada por uma expressão lógica combinacional.

O framework navega pela estrutura de regras, condições e subcondições e, conforme encontra estes elementos, cria esta expressão lógica, cujos termos são a saída dos blocos de premissas.

A saída desta expressão lógica alimenta todas as calls que estão dentro da estrutura ação (action) vinculada a esta regra.

Aqui há um erro que ainda não foi solucionado. No framework atual, todas as conjunções que estruturam esta expressão lógica estão sendo interpretadas como do tipo SINGLE.

F. *Fluxo de Funcionamento*

O fluxo de funcionamento se incia, percorrendo todos os atributos do código para montar o mapa de referências (attRef). Este mapa servirá para definir o número de barramentos de entrada que cada atributo deve possuir.

Se o atributo pertence a instância main e é público, um pino set , um barramentos de entrada e um barramento de saída para este atributo são adicionados à entity que representa main. Alemdisso, como uma entrada externa também pode modificar o atributo, é contada a primeira referencia no mapa, ou seja, **attRef[“this_<nomeDoAtributo>”] = 1**. Se, por outro lado, a visibilidade do atributo é privada, ele é inserido no mapa com valor igual a 0, ou seja, **attRef [“<instanciaDoAtributo>_<nomeDoAtributo>”] = 0;**

Em seguida, todos os métodos do grafo são percorridos e suas respectivas entities são criadas. Também são criados components em main relativos a estas entities.

Numa próxima fase, é percorrida toda a estrutura de regras, condições, subcondições e premissas. Enquanto navega por esta estrutura, o framework monta a expressão lógica que define cada regra. Ao chegar no nível das premissas, o framework cria os blocos para as mesmas e relaciona suas saídas como termos expressão lógica. Dentro de cada regra, também é percorrida a estrutura de ações, instigações e chamadas. No nível das chamadas, o bloco referente a cada uma é criado como uma instância da entidade que represesenta o metodo chamado. A saída da expressão lógica que define a regra é ligada ao pino de entrada execute do bloco da chamada. São criadas as ligações (signals) deste bloco com os argumentos referenciados e o atributo a ser modificado. Este atributo possui seu contador de referencias (attRef) incrementado. As entradas de attOpA e attOpB são definidas de acordo com os dados contidos no método referenciado pela chamada, permitindo a atribuição.

Após isto, os atributos são percorridos e criados com o número de barramentos de entrada equivalentes ao número de referencias contido em attRef. Alem disso, para cada atributo, é definido um valor inicial, de acordo com o número de bits que tem.

A última fase do fluxo é a chamada do método print() que recebe como entrada as variáveis do tipo std::string que contém os blocos de código e, a partir deles, imprime o código estruturado em VHDL.

IV. Código de Teste e Resultado

A partir do exemplo do contador em PON, foi criado um código adaptado para verificar funcionalidades do compilador. Este código não possui compromisso com a semântica correta, e é apenas um teste de compilação. A figura 4.1 apresenta este código.

```
fbe Main

    public Integer<6> count = 0

    private method mtInc
    params
            Integer<6> newCount
        end_params
        attribution
            this.count = 1
        end_attribution
    end_method

    private method mtRst
        attribution
            this.count = 0
        end_attribution
    end_method

    rule RlCounterEquals5
        condition
            subcondition CdCounterEquals5or4
                premise PrCounterEquals5
                    this.count == 5
                end_premise
        or
        premise PrCounterEquals4
            this.count == 4
        end_premise
            end_subcondition
        and
        subcondition CdCounterEquals3or2
                premise PrCounterEquals2
                    this.count == 2
                end_premise
        or
        premise PrCounterEquals3
            this.count == 3
        end_premise
            end_subcondition
        end_condition
        action parallel
            instigation parallel
                call this.mtRst()
            end_instigation
        end_action
    end_rule
    rule RlCounterLess5
        condition
            subcondition CdCounterLess5
                premise PrCounterLess5
                    this.count < 5
                end_premise
            end_subcondition
        end_condition
        action parallel
            instigation parallel
                call this.mtInc(1)
            end_instigation
        end_action
    end_rule

end fbe
```

Figura 4.1 Código PON de teste.

A tradução deste código, como já informado, acabou por encontrar falhas no tange às conjunctions que estruturam as expressões lógicas que representam as regras. As figuras 4.2 a 4.10 apresentam este código compilado.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.data_type_pkg.all;



ENTITY this_mtInc_method IS
    PORT(
        newCount :IN STD_LOGIC_VECTOR(5 downto 0);
        attOpA :STD_LOGIC_VECTOR(5 downto 0);
        attOpB :STD_LOGIC_VECTOR(5 downto 0);
        execute :IN STD_LOGIC;
        output :OUT STD_LOGIC_VECTOR(5 downto 0);
        notify :OUT STD_LOGIC
    );
END this_mtInc_method

ARCHITECTURE this_mtInc_method_arq OF this_mtInc_method IS
BEGIN
    PROCESS(execute)
    BEGIN
        result<=attOpA;
    END PROCESS;
END this_mtInc_method_arq;

ENTITY this_mtRst_method IS
    PORT(
        attOpA :STD_LOGIC_VECTOR(5 downto 0);
        attOpB :STD_LOGIC_VECTOR(5 downto 0);
        execute :IN STD_LOGIC;
        output :OUT STD_LOGIC_VECTOR(5 downto 0);
        notify :OUT STD_LOGIC
    );
END this_mtRst_method

ARCHITECTURE this_mtRst_method_arq OF this_mtRst_method IS
BEGIN
    PROCESS(execute)
    BEGIN
        result<=attOpA;
    END PROCESS;
END this_mtRst_method_arq;
```

Figura 4.2 – Definição das Entities dos Métodos do código gerado

```
ENTITY Main IS
PORT(

    clock :IN STD_LOGIC;
    in_this_count :IN STD_LOGIC_VECTOR(5 downto 0);
    in_set_this_count :IN STD_LOGIC;
    out_this_count :OUT STD_LOGIC_VECTOR(5 downto 0)
);
END Main;
ARCHITECTURE Main_arq OF Main IS
COMPONENT NOP_attribute
    GENERIC (
        N_bits: INTEGER;
        N_new_values: INTEGER;
        initial_value: STD_LOGIC_VECTOR
        );
    PORT(
        att_clock :IN STD_LOGIC;
        att_new_value :IN data(0 TO N_new_values-1)(N_bits-1 downto 0);
        att_set_value :IN STD_LOGIC_VECTOR(N_new_values-1 downto 0);
        att_value :OUT STD_LOGIC_VECTOR(N_bits-1 downto 0)
        );
END COMPONENT;
COMPONENT Nop_premise
    GENERIC (
        N_bits: INTEGER;
        N_new_values: INTEGER;
        pre_result :OUT STD_LOGIC
        );
    PORT(
        att_value_a :IN STD_LOGIC_VECTOR(N_bits-1 downto 0);
        att_value_b :IN STD_LOGIC_VECTOR(N_bits-1 downto 0);
        pre_result :OUT STD_LOGIC)
        );
END COMPONENT;
```

Figura 4.3 Definição da Entity Main e dos Componentes Básicos

```
COMPONENT this_mtInc_method
    PORT(
        newCount :IN STD_LOGIC_VECTOR(5 downto 0);
        attOpA :STD_LOGIC_VECTOR(5 downto 0);
        attOpB :STD_LOGIC_VECTOR(5 downto 0);
        execute :IN STD_LOGIC;
        output :OUT STD_LOGIC_VECTOR(5 downto 0);
        notify :OUT STD_LOGIC
    );
END COMPONENT
COMPONENT this_mtRst_method
    PORT(
        attOpA :STD_LOGIC_VECTOR(5 downto 0);
        attOpB :STD_LOGIC_VECTOR(5 downto 0);
        execute :IN STD_LOGIC;
        output :OUT STD_LOGIC_VECTOR(5 downto 0);
        notify :OUT STD_LOGIC
    );
END COMPONENT
```

Figura 4.4 - Components Utilizados na Criação de Calls

```
SIGNAL this_R1CounterEquals5_PrCounterEquals2_result: STD_LOGIC;
SIGNAL this_R1CounterEquals5_PrCounterEquals2_input_a: STD_LOGIC_VECTOR(5 downto 0);
SIGNAL this_R1CounterEquals5_PrCounterEquals2_input_b: STD_LOGIC_VECTOR(7 downto 0);
SIGNAL this_R1CounterEquals5_PrCounterEquals3_result: STD_LOGIC;
SIGNAL this_R1CounterEquals5_PrCounterEquals3_input_a: STD_LOGIC_VECTOR(5 downto 0);
SIGNAL this_R1CounterEquals5_PrCounterEquals3_input_b: STD_LOGIC_VECTOR(7 downto 0);
SIGNAL this_R1CounterEquals5_PrCounterEquals4_result: STD_LOGIC;
SIGNAL this_R1CounterEquals5_PrCounterEquals4_input_a: STD_LOGIC_VECTOR(5 downto 0);
SIGNAL this_R1CounterEquals5_PrCounterEquals4_input_b: STD_LOGIC_VECTOR(7 downto 0);
SIGNAL this_R1CounterEquals5_PrCounterEquals5_result: STD_LOGIC;
SIGNAL this_R1CounterEquals5_PrCounterEquals5_input_a: STD_LOGIC_VECTOR(5 downto 0);
SIGNAL this_R1CounterEquals5_PrCounterEquals5_input_b: STD_LOGIC_VECTOR(7 downto 0);
SIGNAL this_mtRst_0_0_output: STD_LOGIC_VECTOR(6 downto 0);
SIGNAL this_mtRst_0_0_execute: STD_LOGIC;
SIGNAL this_mtRst_0_0_notify: STD_LOGIC;
SIGNAL this_mtRst_0_0_oppAttA: STD_LOGIC_VECTOR(6 downto 0)
SIGNAL this_mtRst_0_0_oppAttB: STD_LOGIC_VECTOR(6 downto 0)
SIGNAL this_R1CounterLess5_PrCounterLess5_result: STD_LOGIC;
SIGNAL this_R1CounterLess5_PrCounterLess5_input_a: STD_LOGIC_VECTOR(5 downto 0);
SIGNAL this_R1CounterLess5_PrCounterLess5_input_b: STD_LOGIC_VECTOR(7 downto 0);
SIGNAL this_mtInc_0_0_0_newCount: STD_LOGIC_VECTOR(6 downto 0);
SIGNAL this_mtInc_0_0_output: STD_LOGIC_VECTOR(6 downto 0);
SIGNAL this_mtInc_0_0_execute: STD_LOGIC;
SIGNAL this_mtInc_0_0_notify: STD_LOGIC;
SIGNAL this_mtInc_0_0_oppAttA: STD_LOGIC_VECTOR(6 downto 0)
SIGNAL this_mtInc_0_0_oppAttB: STD_LOGIC_VECTOR(6 downto 0)
SIGNAL this_count_new_value: data(0 TO 0)(5 downto 0);
SIGNAL this_count_value: STD_LOGIC_VECTOR(5 downto 0);
SIGNAL this_count_set: STD_LOGIC_VECTOR(0 downto 0);
```

Figura 4.5 – Signals Gerados

```
BEGIN

    this_count: NOP_attribute GENERIC MAP(N_bits => 6, N_new_values => 1, initial_value => 000000)
    PORT MAP(att_clock => clock, att_new_value => this_count_new_value,
    att_set_value => this_count_set, att_value =>this_count_value);


    this_R1CounterEquals5_PrCounterEquals2_EQUAL: NOP_premise
    GENERIC MAP(N_bits => 6, dataType => 0, operation => 1)
    PORT MAP(att_value_a => this_R1CounterEquals5_PrCounterEquals2_input_a,
    att_value_b => this_R1CounterEquals5_PrCounterEquals2_input_b,
    pre_result => this_R1CounterEquals5_PrCounterEquals2_result);

    this_R1CounterEquals5_PrCounterEquals3_EQUAL: NOP_premise
    GENERIC MAP(N_bits => 6, dataType => 0, operation => 1)
    PORT MAP(att_value_a => this_R1CounterEquals5_PrCounterEquals3_input_a,
    att_value_b => this_R1CounterEquals5_PrCounterEquals3_input_b,
    pre_result => this_R1CounterEquals5_PrCounterEquals3_result);

    this_R1CounterEquals5_PrCounterEquals4_EQUAL: NOP_premise
    GENERIC MAP(N_bits => 6, dataType => 0, operation => 1)
    PORT MAP(att_value_a => this_R1CounterEquals5_PrCounterEquals4_input_a,
    att_value_b => this_R1CounterEquals5_PrCounterEquals4_input_b,
    pre_result => this_R1CounterEquals5_PrCounterEquals4_result);

    this_R1CounterEquals5_PrCounterEquals5_EQUAL: NOP_premise
    GENERIC MAP(N_bits => 6, dataType => 0, operation => 1)
    PORT MAP(att_value_a => this_R1CounterEquals5_PrCounterEquals5_input_a,
    att_value_b => this_R1CounterEquals5_PrCounterEquals5_input_b,
    pre_result => this_R1CounterEquals5_PrCounterEquals5_result);

    this_R1CounterLess5_PrCounterLess5_LESSER_THAN: NOP_premise
    GENERIC MAP(N_bits => 6, dataType => 0, operation => 3)
    PORT MAP(att_value_a => this_R1CounterLess5_PrCounterLess5_input_a,
    att_value_b => this_R1CounterLess5_PrCounterLess5_input_b,
    pre_result => this_R1CounterLess5_PrCounterLess5_result);
```

Figura 4.6 - Instanciação de Atributos e Premissas

```
this_mtRst_0_0: this_mtRst PORT MAP(output => this_mtRst_0_0_output,
execute => this_mtRst_0_0_execute, notify => this_mtRst_0_0_notify,
attOpA => this_mtRst_0_0_oppAttA, attOpB => this_mtRst_0_0_oppAttB);

this_mtInc_0_0: this_mtInc PORT MAP(newCount => this_mtInc_0_0_0_newCount,
output => this_mtInc_0_0_output, execute => this_mtInc_0_0_execute,
notify => this_mtInc_0_0_notify, attOpA => this_mtInc_0_0_oppAttA,
attOpB => this_mtInc_0_0_oppAttB);


this_count_new_value(0) <= in_this_count;
this_count_set(0) <= in_set_this_count;
this_count_new_value(1) <= this_mtRst_0_0_output;
this_count_set(1) <= this_mtRst_0_0_notify;
this_count_new_value(2) <= this_mtInc_0_0_output;
this_count_set(2) <= this_mtInc_0_0_notify;


out_this_count <= this_count_value;
```

Figura 4.7 – Instanciação de Calls e Atribuição de Sinais à Entrada/Saída de Atributos

```
  this_RlCounterEquals5_PrCounterEquals2_input_a <= this_count_value;
  this_RlCounterEquals5_PrCounterEquals2_input_b <=
  std_logic_vector(to_unsigned(2,this_RlCounterEquals5_PrCounterEquals2_input_b'length));
  this_RlCounterEquals5_PrCounterEquals3_input_a <= this_count_value;
  this_RlCounterEquals5_PrCounterEquals3_input_b <=
  std_logic_vector(to_unsigned(3,this_RlCounterEquals5_PrCounterEquals3_input_b'length));
  this_RlCounterEquals5_PrCounterEquals4_input_a <= this_count_value;
  this_RlCounterEquals5_PrCounterEquals4_input_b <=
  std_logic_vector(to_unsigned(4,this_RlCounterEquals5_PrCounterEquals4_input_b'length));
  this_RlCounterEquals5_PrCounterEquals5_input_a <= this_count_value;
  this_RlCounterEquals5_PrCounterEquals5_input_b <=
  std_logic_vector(to_unsigned(5,this_RlCounterEquals5_PrCounterEquals5_input_b'length));
  this_RlCounterLess5_PrCounterLess5_input_a <= this_count_value;
  this_RlCounterLess5_PrCounterLess5_input_b <=
  std_logic_vector(to_unsigned(5,this_RlCounterLess5_PrCounterLess5_input_b'length));


  this_mtRst_0_0_execute <=
  ( ( this_RlCounterEquals5_PrCounterEquals2_result
  SINGLE this_RlCounterEquals5_PrCounterEquals3_result ) SINGLE
  ( this_RlCounterEquals5_PrCounterEquals4_result
  SINGLE this_RlCounterEquals5_PrCounterEquals5_result ) );

  this_mtRst_0_0_oppAttA <= std_logic_vector(to_unsigned(0,this_mtRst_0_0_oppAttA'length));
  this_mtInc_0_0_0_newCount <= std_logic_vector(to_unsigned(1,this_mtInc_0_0_0_newCount'length));
  this_mtInc_0_0_execute <= ( ( this_RlCounterLess5_PrCounterLess5_result ) );
  this_mtInc_0_0_oppAttA <= std_logic_vector(to_unsigned(1,this_mtInc_0_0_oppAttA'length));
END Main;
```

Figura 4.8 - Definição de Entradas de Premissas, Atribuição de Regras a Chamadas e Fim do Código.

V. Experiência de Implementação

A experiência de desenvolvimento deste compilador contou com algumas dificuldades que podem ser pontuadas. A primeira barreira foram as peculiaridades na linguagem VHDL. A forma como se referenciam blocos, sinais, entidades e componentes tornou difícil a estruturação do código compilado em múltiplos arquivos e, assim, foi necessária uma mudança de estratégia, para escrita em um arquivo só. Pode-se citar também como dificuldade, as diferenças na estrutura de grafo trabalhada por Kerschbaumer e o Grafipon no LingPON 2.0. Estas diferenças exigiram bastante cuidado na análise do código do compilador antigo para entender o que realmente poderia ser aproveitado. Também houve dificuldades para lidar com arquivos múltiplos de PON num mesmo programa, não ficando muito claro como o PON faz o link entre estes arquivos.

Quanto à utilização do LingPON, pode-se ressaltar que, após o correto entendimento da estrutura, foi possível operar com ela de maneira bastante prática. Seria interessante se todos os componentes (atributos, métodos, regras) tivessem referência para a instância. Alem disso, seria interessante a elaboração de uma estrutura que informe o tipo de retorno para metodos que contam com o bloco code.

VI. Considerações Finais

Apesar de não ser possível completar este framework para a geração de código VHDL a partir de código PON, é possível dizer que esta experiência possibilitou um grande aprendizado de conceitos de Linguagens e Compiladores e do Paradigma Orientado a Notificações.

Os próximos passos para a conclusão deste trabalho devem se iniciar pela resolção dos bugs relativos às conjunções que organizam as expressões lógicas que implementam as regras.

Outro passo importante, é o teste deste frame work para códigos que se utilizem de uma hierarquia de instâncias, visando verificar se é capaz de atender este recurso.

VII. Referencias

[1] SANTOS, L. A., **Linguagem e Compilador para o Paradigma Orientado a Notificações: Avanços para Facilitar a Codificação e sua Validação em uma Aplicação de Controle de Futebol de Robôs**, dissertação, CPGEI, Universidade Tecnológica Federal do Paraná, Curitiba, PR, Brasil, 2017.

[2] BANASZEWSKI, R. F. **Paradigma Orientado a Notificações: Avanços e Comparações**. Dissertação de Mestrado, CPGEI, Universidade Tecnológica Federal do Paraná, Curitiba, PR, Brasil, 2009.

[3] KERSCHBAUMER R. **Proposição do Paradigma Orientado a Notificações no Desenvolvimento de Circuitos Lógico Digitais Reconfiguráveis**. Dissertação de Doutorado, CPGEI, Universidade Tenológica Federal do Paraná, Curitiba, PR, Brasil, 2018.

[4] **https://www.epaperpress.com/lexandyacc/intro.html, acessado em: 01/10/2018**

[5] SIMÃO, J. M., **"A contribution to the development of a hms simulation tool and proposition of a meta-model for holonic control,"** Ph.D. dissertation, School in Electrical Engineering and Industrial Computer Science (CPGEI) at Federal University of Technology - Paraná (UTFPR, Brazil) and Research Center For Automatic Control of Nancy (CRAN)- Henry Poincaré University (UHP, France), 2005.

[6] **RONSZCKA, A. F. et all, "Notification-Oriented Paradigm Framework 2.0:An Implementation Based On Design Patterns",** publicado na revista IEEE LA, vol 15, 11/11/2017