

Universidade Federal de Minas Gerais  
Escola de Engenharia  
Curso de Graduação em Engenharia de Controle e Automação

**Aplicação de verificação baseada em Asserções  
automáticas em um simulador de linguagem  
Ladder**

Gabriel Guedes de Azevedo Barbosa

Orientador: Prof. Ricardo de Oliveira Duarte, Dr.  
Supervisor: Eng. Lucas Ferreira de Melo Diniz

Belo Horizonte, Novembro de 2014

## **Monografia**

### **Aplicação de verificação baseada em Asserções automáticas em um simulador de linguagem Ladder**

Monografia submetida à banca examinadora designada pelo Colegiado Didático do Curso de Graduação em Engenharia de Controle e Automação da Universidade Federal de Minas Gerais, como parte dos requisitos para aprovação na disciplina Projeto Final de Curso II.

Belo Horizonte, Novembro de 2014

# Resumo

Controladores Lógicos Programáveis (CLPs) são equipamentos amplamente utilizados em diferentes tipos de tarefas na indústria. Eles possibilitam a automação de processos críticos, fato que cria a necessidade de verificar a confiabilidade de seus programas.

Na indústria de desenvolvimento de hardware, existe um conceito denominado verificação formal que permite a depuração do funcionamento dos chips desenvolvidos. Dessa forma é possível garantir que os mesmos funcionam de acordo com um conjunto de especificações.

Com o objetivo de verificar as lógicas desenvolvidas em diagramas Ladder, este trabalho propõe técnicas que possibilitam a conversão desses diagramas em linguagens de descrição de hardware, para que seja possível utilizar uma ferramenta de verificação já consolidada no mercado, o JasperGold®.

Os resultados apresentados mostram que a conversão realizada é capaz de representar fielmente o sistema real, fato que possibilita a verificação formal do mesmo, resultando na garantia de que o controle ocorre como especificado.

**Palavras-Chave:** Verificação formal de hardware, diagramas Ladder, linguagens de descrição de hardware.

# Agradecimentos

Ao universo, por sua existência plena, incerta e paradoxal.

A esta universidade, por fornecer conhecimentos e desafios que auxiliaram no meu desenvolvimento intelectual.

Aos professores, pois sem eles a universidade seria apenas uma casca vazia e sem vida.

Ao orientador Prof. Dr. Ricardo de Oliveira Duarte, pela oportunidade e apoio na elaboração deste trabalho.

Ao supervisor Eng. Lucas Ferreira de Melo Diniz da Jasper Design Automation, pelo apoio técnico e emocional.

Ao Eng. Mauro Antonio da Costa Junior, por ser um terceiro revisor, mesmo sem a obrigação de fazê-lo.

Aos colegas da Jasper Design Automation, pois são as pessoas mais fantásticas com quem trabalhei até hoje.

Ao Felipe de Freitas Castro, por seguir o caminho da graduação comigo, me incentivando até o final.

Aos amigos, por fazerem meus dias mais felizes.

À Amanda, pelo suporte incondicional nas poucas horas fáceis e nas muitas difíceis. Pela compreensão nas horas ausentes.

Aos meus pais, pelo grande esforço que dedicaram para minha criação.

Aos meus irmãos, Júlia e Gustavo, pelo carinho fraterno.

E a todos que direta ou indiretamente fizeram parte da minha formação.

# Sumário

<b>Resumo</b>	<b>i</b>
<b>Agradecimentos</b>	<b>ii</b>
<b>Lista de Figuras</b>	<b>vii</b>
<b>Lista de Tabelas</b>	<b>viii</b>
<b>1 Introdução</b>	<b>1</b>
1.1 Motivação . . . . .	1
1.2 Objetivos do Projeto . . . . .	2
1.3 Local de Realização . . . . .	2
1.4 Estrutura da Monografia . . . . .	3
<b>2 Conceitos Preliminares</b>	<b>4</b>
2.1 Máquina de Estados Finitos . . . . .	4
2.2 Linguagens de Descrição de Hardware . . . . .	4
2.2.1 Verilog . . . . .	6
2.3 Diagrama Ladder . . . . .	6
2.4 Verificação de Hardware . . . . .	8
2.4.1 Verificação por Simulação . . . . .	9
2.4.2 Verificação Formal . . . . .	9
2.4.3 Propriedades em Verificação . . . . .	9
2.4.4 Trace . . . . .	11
<b>3 Revisão Bibliográfica</b>	<b>15</b>
3.1 O uso de Verificação Formal . . . . .	15
3.2 Modelos formais de diagramas Ladder . . . . .	16
<b>4 Materiais e Métodos</b>	<b>18</b>
4.1 Materiais . . . . .	18
4.1.1 JasperGold® . . . . .	18

4.1.2	i-TRiLOGI . . . . .	19
4.1.3	C++ . . . . .	20
4.1.4	TCL . . . . .	20
4.2	Métodos . . . . .	20
4.2.1	Realizando o setup no JasperGold® . . . . .	21
<b>5</b>	<b>Desenvolvimento</b>	<b>23</b>
5.1	Modelo lógico do diagrama Ladder . . . . .	23
5.2	Conversão Automática para Verilog . . . . .	25
5.2.1	Exemplo . . . . .	27
5.3	Modelo Comportamental . . . . .	28
5.3.1	Formalizando as propriedades . . . . .	30
5.3.2	Comportamento de Equipamentos . . . . .	31
5.4	Modelo de Especificações . . . . .	34
<b>6</b>	<b>Resultados</b>	<b>36</b>
6.1	Motor de Duas Rotações . . . . .	36
6.1.1	Teste 1 . . . . .	37
6.1.2	Teste 2 . . . . .	38
6.1.3	Teste 3 . . . . .	39
6.2	Esteira Transportadora . . . . .	40
6.2.1	Comportamento da Esteira . . . . .	42
6.2.2	Comportamento da Caixa . . . . .	42
6.2.3	Validando o modelo de Comportamento . . . . .	42
6.3	Deteção de Deadlock . . . . .	44
6.3.1	Comportamento das Esteiras . . . . .	46
6.3.2	Comportamento da Caixa . . . . .	47
6.3.3	Verificando a ocorrência de Deadlock . . . . .	47
<b>7</b>	<b>Conclusões</b>	<b>51</b>
7.1	Considerações Finais . . . . .	51
7.2	Propostas de Continuidade . . . . .	51
7.2.1	Biblioteca de Comportamento de Equipamentos . . . . .	51
7.2.2	Estudo sobre Especificações de Tempo . . . . .	52
7.2.3	Modelagem das Linguagens da IEC-61131-3 . . . . .	52
7.2.4	Desenvolvimento de Interface Gráfica . . . . .	52
<b>8</b>	<b>Anexos</b>	<b>53</b>
8.1	Código do Parser . . . . .	53
8.1.1	Cabeçalho . . . . .	53
8.1.2	Implementação . . . . .	53

8.2	Códigos dos Modelos de Comportamento . . . . .	53
8.2.1	Container . . . . .	53
8.2.2	Esteira . . . . .	54
8.3	Exemplo completo de transformação . . . . .	54
8.3.1	Diagrama Ladder . . . . .	54
8.3.2	Formato i-TRiLOGI . . . . .	54
8.3.3	Verilog - Modelo do Diagrama Ladder . . . . .	54
8.3.4	Verilog - Modelo de Comportamentos . . . . .	54
8.3.5	Verilog - Especificações . . . . .	54

<b>Referências Bibliográficas</b>	<b>66</b>
-----------------------------------	-----------

# Lista de Figuras

2.1	Máquina de estados finitos - contador de 2 bits . . . . .	5
2.2	Circuito escrito em HDL (Verilog) - Fonte [5] . . . . .	5
2.3	Fluxo de energia em um diagrama Ladder - Adaptado de [14] .	7
2.4	Exemplo de diagrama Ladder - Fonte [3] . . . . .	7
2.5	Relação entre diagrama Ladder e diagrama lógico - Fonte [15]	8
2.6	Teste formal da funcionalidade módulo - Adaptado de [21] . .	10
2.7	Exemplo de trace . . . . .	11
3.1	Falhas Detectadas em Simulação - Adaptado de [13] . . . . .	16
3.2	Comparação entre Verificação e Simulação - Adaptado de [13]	16
3.3	Obtendo modelo formal de diagramas Ladder - Fonte [18] . . .	17
4.1	Interface gráfica do software JasperGold® - Fonte [2] . . . . .	19
4.2	Fluxo de verificação do App FPV - Fonte [2] . . . . .	19
4.3	Software de edição de diagramas Ladder i-Trilogi - Fonte [9] .	19
4.4	Verificação de diagramas Ladder utilizando JasperGold® . . .	21
4.5	Exemplo de setup da Ferramenta JasperGold® . . . . .	22
5.1	Diagrama Ladder com intertravamento . . . . .	23
5.2	Loop combinacional obtido com a transformação direta do di- agrama Ladder descrito na Figura 5.1 . . . . .	24
5.3	Loop combinacional instável - Adaptado de [16] . . . . .	24
5.4	Quebra do loop combinacional pela adição de Flop-Flop D . .	25
5.5	Modelagem da quebra de um loop genérico . . . . .	25
5.6	Diagrama Ladder de exemplo . . . . .	27
5.7	Arquivo referente ao diagrama Ladder da Figura 5.6 . . . . .	27
5.8	Netlist referente ao diagrama Ladder da Figura 5.6 . . . . .	28
5.9	Diagrama de portas lógicas referente ao diagrama Ladder da Figura 5.6 . . . . .	29
5.10	Exemplo de diagrama Ladder . . . . .	29
5.11	Esquemático do funcionamento da esteira . . . . .	32
5.12	Análise de tempo em duas fases - Adaptado de [7] . . . . .	33



5.13	Esquemático do funcionamento do container . . . . .	33
5.14	Ligação errada de motor de duas rotações . . . . .	34
6.1	Primeira implementação do acionamento do motor de duas rotações . . . . .	37
6.2	Tabela com o resultado da prova das propriedades de especi- ficação, de acordo com o diagrama Ladder da Figura 6.1 . . .	38
6.3	Trace da condição que viola a Propriedade 1 . . . . .	38
6.4	Implementação com Trava nos acionamentos de entrada . . . .	38
6.5	Tabela com o resultado da prova das propriedades de especi- ficação, de acordo com o diagrama Ladder da Figura 6.4 . . .	39
6.6	Implementação com linhas mutualmente exclusivas . . . . .	39
6.7	Tabela com o resultado da prova das Propriedades de Especi- ficação, de acordo com o diagrama Ladder da Figura 6.6 . . .	40
6.8	Planta referente à esteira transportadora . . . . .	40
6.9	Diagrama de estados da esteira transportadora . . . . .	43
6.10	Trace obtido a partir da prova formal da Propriedade 5 . . . .	43
6.11	Trace obtido a partir da prova formal da Propriedade 6 . . . .	44
6.12	Processo de pintura e verniz . . . . .	44
6.13	Diagrama Ladder para o controle do sistema . . . . .	46
6.14	Trace de deadlock no sistema representado pela Figura 6.12 .	48
6.15	Um evento de retrabalho ocorrendo de acordo com o Trace . .	49
6.16	Um evento de expulsão de peça ocorrendo de acordo com o Trace . . . . .	49
6.17	O deadlock ocorrendo de acordo com o Trace . . . . .	50
8.1	Cabeçalho do Parser em C++ . . . . .	55
8.2	Implementação Parser - parte 1 . . . . .	56
8.3	Implementação Parser - parte 2 . . . . .	57
8.4	Implementação Parser - parte 3 . . . . .	58
8.5	Implementação Parser - parte 4 . . . . .	59
8.6	Implementação da Caixa . . . . .	59
8.7	Implementação da Esteira . . . . .	60
8.8	Diagrama Ladder para controle do pátio de pintura . . . . .	61
8.9	Diagrama Ladder da Figura 8.8 em formato texto . . . . .	62
8.10	Modelo do Diagrama Ladder em Verilog . . . . .	63
8.11	Flip-Flop D utilizado na quebra de loop combinacional . . . .	63
8.12	Ligação do Modelo de Comportamentos . . . . .	64
8.13	Propriedade de Deadlock . . . . .	65

# Lista de Tabelas

2.1	Estruturas básicas de circuitos lógicos . . . . .	12
2.2	Tipos de contatos de um diagrama Ladder - Fonte [17] . . . . .	12
2.3	Tipos de bobinas de um diagrama Ladder - Fonte [17] . . . . .	13
2.4	Funções úteis para descrição de propriedades em SVA . . . . .	14
5.1	Conversão das funções básicas de diagramas Ladder para Verilog	26

# Capítulo 1

## Introdução

Controladores Lógicos Programáveis são equipamentos amplamente utilizados em diferentes tipos de tarefas na indústria. Eles possibilitam a automação de processos críticos, por exemplo, extração de petróleo e produção de aço [14].

Os programas desenvolvidos para Controladores Lógicos Programáveis, por sua vez, devem ser confiáveis, de modo a garantir que os processos controlados sejam tolerantes a falhas, melhorando a segurança e diminuindo prejuízos financeiros na linha de produção.

A norma IEC 61131 define padrões de programação para Controladores Lógicos Programáveis (CLPs). Em sua terceira seção, a IEC 61131-3 sugere regras para as linguagens de programação, resultando em melhor qualidade do software[11].

Estas regras, porém, se limitam a descrever normas estruturais, ou seja, da arquitetura de programação. Entretanto, não há normas definidas para testar a funcionalidade de um programa desenvolvido para operar em um CLP.

### 1.1 Motivação

Na indústria de desenvolvimento de hardware, existe um conceito denominado "*Electronic Design Automation*" (EDA), sendo também um ramo da ciência da computação que visa criar ferramentas e abstrações que auxiliem engenheiros e programadores durante a concepção, projeto, desenvolvimento e produção de componentes de hardware.

Durante o projeto, quando uma nova lógica é criada, ou novos blocos são desenvolvidos, uma verificação deve ser realizada nesses circuitos, com o objetivo de testar se inconsistências foram inseridas [13].

Existem dois grandes grupos de verificação:

- Verificação por simulação: Consiste em gerar um conjunto aleatório, ou semi-aleatório, de padrões de entradas que testarão a(s) funcionalidade(s) do sistema. Essas entradas têm a probabilidade de gerar estados inconsistentes. É um método não determinístico para verificação [19].
- Verificação formal: Utiliza propriedades matemáticas para encontrar estados inconsistentes. Essa abordagem, porém, necessita de um modelo preciso do sistema. Em termos práticos, ela mostra quais valores das entradas geram estados distintos dos esperados durante o correto funcionamento do sistema [19].

As ferramentas utilizadas na indústria de EDA verificam circuitos projetados em Linguagens de Descrição de Hardware (HDL). Estas são linguagens baseadas em texto e especificamente orientadas à descrição das estruturas e comportamento do hardware. Existem 3 principais HDLs: VHDL, Verilog e SystemC [20].

Portanto, para a realização deste Projeto Final de Curso, foi realizada a tradução de diagramas Ladder para Linguagens de Descrição de Hardware, com o objetivo de possibilitar a verificação de tais diagramas em ferramentas de EDA.

## 1.2 Objetivos do Projeto

Tendo em vista os pontos expostos, este projeto tem por objetivos:

- Criar um modelo para a representação de diagramas Ladder na linguagem de descrição de hardware conhecida como Verilog;
- Aplicar técnicas de verificação formal de hardware, no modelo criado, utilizando exemplos teóricos e práticos para validação do mesmo;

## 1.3 Local de Realização

Este projeto de fim de curso foi desenvolvido na empresa Jasper Design Automation, no Departamento de R&D (Research and Development, em português Pesquisa e Desenvolvimento), responsável pelo desenvolvimento de novas aplicações para o Software de EDA vendido pela empresa, o JasperGold®.

A Jasper Design Automation, é uma empresa privada, que desenvolve um software de alto retorno de investimento para projetos e verificação de sistemas eletrônicos. A empresa realiza projetos de pesquisa, desenvolvimento e treinamento nas áreas de desenvolvimento de hardware e software.

## 1.4 Estrutura da Monografia

Este trabalho foi dividido em 7 capítulos.

- Capítulo 1: Descreve as motivações e objetivos do projeto e uma pequena introdução ao assunto;
- Capítulo 2: Introduz conceitos de desenvolvimento e verificação de hardware, propriedades formais, especifica as características dos diagramas Ladder importantes para o modelo desenvolvido;
- Capítulo 3: Contém a revisão bibliográfica, descrevendo os trabalhos atuais que possuem contexto técnico-científico com este projeto;
- Capítulo 4: Descreve as Ferramentas utilizadas e os métodos implementados para possibilitar a verificação formal de diagramas Ladder;
- Capítulo 5: Descreve a implementação do modelo desenvolvido em linguagem de descrição de hardware;
- Capítulo 6: Apresenta a aplicação de técnicas de verificação formal e seu resultado sobre o modelo proposto;
- Capítulo 7: Descreve a conclusão do trabalho, destacando os principais pontos do Projeto e trabalhos futuros;
- Capítulo 8: Apresenta os anexos do projeto.

# Capítulo 2

## Conceitos Preliminares

### 2.1 Máquina de Estados Finitos

Máquina de Estados Finitos, ou em inglês, *Finite State Machine* (FSM), é um modelo matemático utilizado para representar o conjunto de estados possíveis de um circuito lógico [20]. A FSM é composta de estados e transições, em que estados possuem os valores dos elementos sequenciais (memória) e as transições são as mudanças de estados a partir de determinadas condições. A Figura 2.1 representa uma FSM simples, referente a um contador de 2 bits e possui as seguintes características:

- O conjunto de estados:  $\{0, 1, 2, 3\}$ , sendo  $\{0\}$  o estado inicial;
- Entrada de 1 bit: *crescente*;
- Saída de 2 bits: *contador*;
- Se *crescente* == 1, o valor de *contador* será incrementado em 1, fazendo com que as transições possíveis sejam:  $\{0,1\}\{1,2\}\{2,3\}\{3,0\}$ ;
- Se *crescente* == 0, o valor de *contador* será decrementado em 1, fazendo com que as transições possíveis sejam:  $\{0,3\}\{1,0\}\{2,1\}\{3,2\}$ ;

FSMs são amplamente utilizadas para descrever o funcionamento de circuitos lógicos digitais [20]

### 2.2 Linguagens de Descrição de Hardware

Linguagens de Descrição de Hardware, em inglês Hardware Description Languages (HDLs), são linguagens textuais utilizadas para descrever circuitos

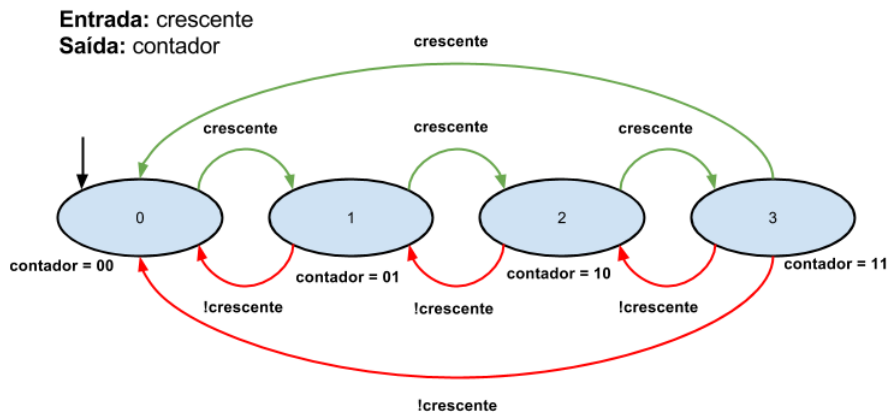


Figura 2.1: Máquina de estados finitos - contador de 2 bits

lógicos. Elas se fizeram necessárias à medida que os projetos lógicos se tornaram maiores e mais complexos [20].

As HDLs possibilitam, de maneira organizada [20]:

- Descrever as interconexões estruturais entre componentes;
- Incluir métodos que permitem descrever o comportamento dos componentes;

A Figura 2.2 possui a representação em HDL de um circuito lógico. É possível observar a descrição das conexões lógicas, entradas e saídas.

```

module binaryToESeg;
  wire   eSeg, p1, p2, p3, p4;
  reg    A, B, C, D;

  nand #1
    g1(p1, C, ~D),
    g2(p2, A, B),
    g3(p3, ~B, ~D),
    g4(p4, A, C),
    g5(eSeg, p1, p2, p3, p4);
endmodule

```

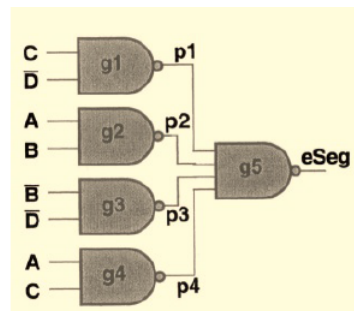


Figura 2.2: Circuito escrito em HDL (Verilog) - Fonte [5]

As HDLs fornecem uma grande quantidade de estruturas e funções lógicas para descrever circuitos. Visando criar um modelo que seja simples, mas possa representar o comportamento de um diagrama Ladder, apenas um subconjunto das abstrações fornecidas pela linguagem serão utilizadas. Elas

são as funções AND, OR, NOT, flip-flops tipo D e contadores. A Tabela 2.1 apresenta estes circuitos em diagramas esquemáticos e em HDL.

As funções AND, OR e NOT serão usadas para modelar as conexões físicas elementares dos diagramas Ladder, o flip-flop D será aplicado para representar operações envolvendo memória e contadores servirão para replicar FIFOs (do inglês First In First Out, estrutura de dados que permite armazenar e retirar dados, seguindo a ordem de entrada).

Quanto à HDL escolhida para a implementação deste trabalho, dentre as mais populares: SystemC, VHDL e Verilog, a última foi a escolhida.

### 2.2.1 Verilog

O Verilog é uma HDL criada pela Cadence Design Systems e atualmente regulamentada pela IEEE 1364. Historicamente, foi projetada para ter uma sintaxe similar à linguagem de programação C. Uma evidência disso é que elas compartilham algumas palavras reservadas (if/else, for, while, case) [5].

O motivo da escolha desta linguagem se deve ao fato de que, diferentemente do VHDL, o Verilog não é fortemente tipado e pela experiência já adquirida nesta linguagem pelo autor deste projeto.

## 2.3 Diagrama Ladder

O Diagrama Ladder é uma linguagem gráfica de programação em CLPs. É análoga aos diagramas de circuitos de relés utilizados na automação. Por permitir fácil manipulação, interpretação e representação de conexões físicas, é utilizada em larga escala [14].

Uma linha vertical à esquerda representa um barramento energizado. Uma outra linha paralela à direita representa uma barra de aterramento. Elementos constituídos por contatos e bobinas são dispostos na horizontal formando malhas seriais ou paralelas [17]. A energia flui do barramento energizado para o terra, como demonstrado na Figura 2.3.

Um típico Diagrama Ladder pode ser observado na Figura 2.4.

Cada contato é associado a uma variável lógica. Se é um contato normalmente aberto (NA) e a variável associada está em TRUE, o contato estará ativo e haverá fluxo de corrente. Para este mesmo contato, se a variável associada está em FALSE, não há fluxo de corrente [17].

Já em contatos normalmente fechados (NF), se a variável associada está em TRUE, não há fluxo de corrente, caso contrário, haverá condução [17].

Quando todos os contatos de uma linha estão fechados, existirá fluxo de corrente até as bobinas. Essas são os últimos elementos de cada linha do



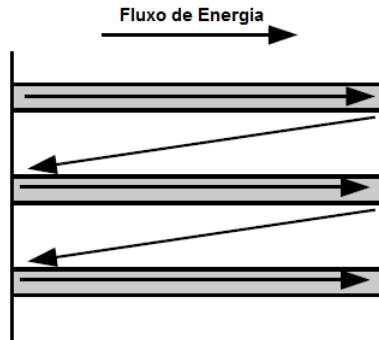


Figura 2.3: Fluxo de energia em um diagrama Ladder - Adaptado de [14]

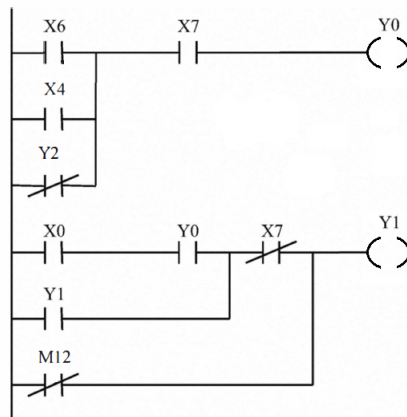


Figura 2.4: Exemplo de diagrama Ladder - Fonte [3]

Ladder. Quando a bobina é energizada, seus contatos associados passarão a ter seus estados ligados ou desligados, dependendo da natureza dos mesmos (NA ou NF) [17].

As Tabelas 2.2 e 2.3 mostram, respectivamente, os tipos de contatos e bobinas presentes em Diagramas Ladder e que serão utilizados no desenvolvimento deste projeto.

Portanto, é possível estabelecer relações lógicas entre contatos e bobinas, de modo a criar lógicas de acionamento. Estas relações podem ser traduzidas em lógicas booleanas simples, como demonstrado na Figura 2.5. Ligações em paralelo representam portas lógicas OR, enquanto ligações em série são portas AND. No caso da realimentação da Bobina M, cria-se um loop combinacional no circuito, fato que é prejudicial à Verificação Formal. A razão disso, e sua solução, serão explicados no Capítulo 5.

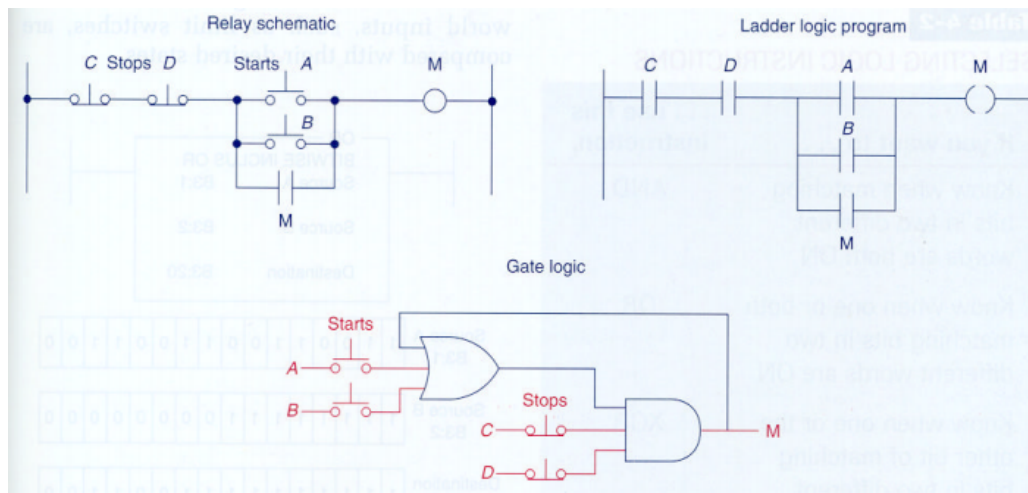


Figura 2.5: Relação entre diagrama Ladder e diagrama lógico - Fonte [15]

## 2.4 Verificação de Hardware

A verificação de hardware é um processo utilizado para garantir que projetos de circuitos lógicos estão de acordo com suas especificações. Devido à organização incremental do projeto de circuitos integrados, bem como ao alto custo de fabricação das primeiras amostras destes componentes, achar erros nas primeiras etapas do processo não é apenas desejável, é essencial. Assim, a aplicação de técnicas de verificação é um ponto crucial para qualquer projeto [8].

O processo de verificação acontece da seguinte maneira:

Um System on Chip (SOC) é criado a partir da conexão de vários subsistemas proprietários (IPs).

A funcionalidade de cada IP é verificada parcialmente por seu projetista, com o intuito de encontrar defeitos óbvios, e posteriormente, por um time de engenheiros de verificação, os quais produzem testes mais elaborados [13].

Após qualidade de cada IP ser verificada, o time de engenheiros de verificação deve testar as interconexões do SOC, para garantir que os IPs se comunicam de acordo com as especificações [13].

Fazendo um paralelo com o cotidiano: Os processadores de smartphones são SOCs. Eles possuem diversos circuitos internos, responsáveis por funções como: vídeo, áudio, GPS, touch-screen, etc. Estes são os IPs. As especificações do projeto determinam como as funções devem se comunicar com o processador central, de modo que o sistema por completo funcione corretamente, mesmo que os IPs sejam projetados por diferentes empresas.

Existem dois tipos de verificação: dinâmica (simulação) e estática (for-

mal) [8].

### 2.4.1 Verificação por Simulação

Na Verificação por Simulação, o engenheiro de verificação fica encarregado de criar um conjunto de entradas para testar as funcionalidades do sistema. Estas entradas, porém, são consideradas incompletas, uma vez que são geradas de maneira aleatória, ou semi-aleatória [13].

Portanto, o processo de simulação tem uma forte base teórica na estatística. Possui a característica de melhoria monotônica de qualidade, ou seja, a qualidade melhora sempre que um erro é consertado, porém nunca atinge um nível absoluto [13].

### 2.4.2 Verificação Formal

A Verificação Formal consiste no uso de técnicas baseadas em lógica matemática, para demonstrar formalmente, que um circuito implementa corretamente as funcionalidades para que foi projetado. Seus modelos são descrições das FSMs dos sistemas sob verificação. A verificação Formal nada mais é que uma busca por estados específicos em uma FSM. Para se realizar este tipo de verificação, existe a necessidade de se estabelecer pré-condições (condições de funcionamento) das funcionalidades e determinar as pós condições (se o sistema se comporta como deveria, devido às pré-condições) [21].

Estas condições são as propriedades do circuito.

### 2.4.3 Propriedades em Verificação

Uma propriedade nada mais é que uma preposição de um comportamento do circuito [8]. Existem 3 tipos de propriedades que podem ser utilizadas para descrever comportamentos [18]:

- Propriedade Estática: Propriedade  $p$  é sempre TRUE;
- Propriedade Safety: Propriedade  $p$  é TRUE enquanto outra Propriedade  $q$  é TRUE;
- Propriedade Liveness: Propriedade  $p$  será TRUE eventualmente;

A Figura 2.6 demonstra propriedades aplicadas a uma funcionalidade que calcula o módulo negativo de um número de -10 a 10. Para este sistema, deseja-se verificar se a saída será limitada ao intervalo  $[-10, 0]$ . Caso contrário, o sistema está fora de suas especificações.

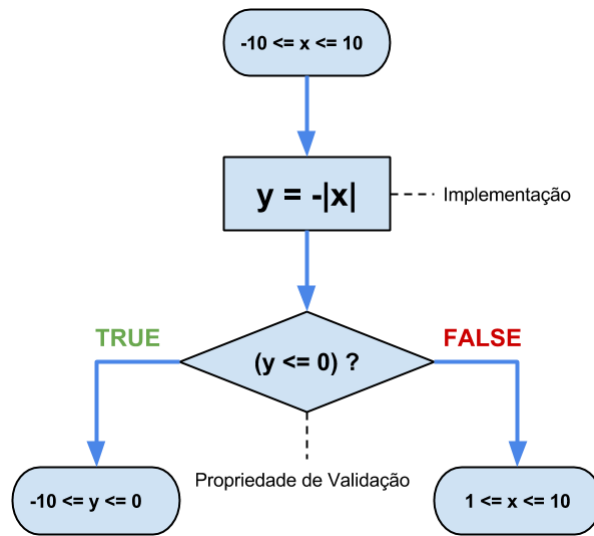


Figura 2.6: Teste formal da funcionalidade módulo - Adaptado de [21]

Neste trabalho, serão considerados 3 principais tipos de diretivas de propriedades:

- Assertion: É uma **afirmação** sobre a validade de uma propriedade. No contexto de hardware, elas garantem que a especificação e a implementação do circuito são consistentes [6].

Na Figura 2.6 se a propriedade de validação ( $y \leq 0$ ) é uma Assertion, ela falharia se a implementação não estivesse dentro das especificações.

- Assumption: É uma **consideração** de uma propriedade. Elas forçam uma propriedade a ser sempre verdadeira. Servem para descrever comportamentos e limitações. No caso da Figura 2.6 assume-se que a entrada está limitada ao intervalo  $[-10, 10]$  [6].
- Cover: É uma **constatação** da possibilidade de ocorrência de uma propriedade. No caso da Figura 2.6 se a Propriedade de Validação fosse um Cover, ela iria mostrar um caso, ou seja, valores de entrada e saída, em que essa propriedade é verdadeira [6].

### System Verilog Assertions

O System Verilog Assertions, ou SVA, é uma extensão da linguagem Verilog. Tem como característica o suporte para a descrição de propriedades, pois possui construções embutidas que facilitam a criação das

mesmas [6]. As principais funções utilizadas neste projeto podem ser observada na Tabela 2.4.

## 2.4.4 Trace

Trace é uma maneira de se representar sequencias de sinais. Pode ser visualizado como uma forma de onda, auxiliando na compreensão das transições dos estados de uma FSM e na relação entre os sinais envolvidos em uma determinada operação. No contexto de verificação de hardware, são importantes pois possibilitam apontar a causa raiz de um erro, mostrando exatamente quais sinais do sistema são defeituosos. A Figura 2.7 possui um exemplo de trace proveniente da ferramenta de verificação de hardware JasperGold®.

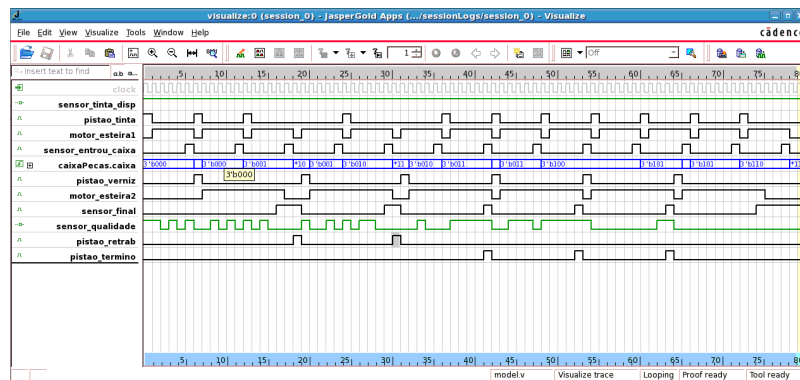


Figura 2.7: Exemplo de trace

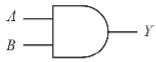

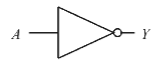
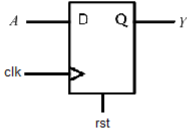
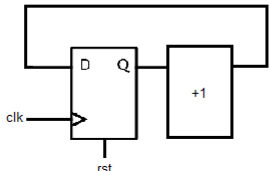
Função Lógica	Circuito	HDL (Verilog)
AND		<pre> module and(input A,             input B,             output Y);     assign Y = A &amp; B; endmodule </pre>
OR		<pre> module or(input A,            input B,            output Y);     assign Y = A   B; endmodule </pre>
NOT		<pre> module not(input A,             output Y);     assign Y = ~A; endmodule </pre>
D-FLIP-FLOP		<pre> module dff(input A,             output Y);     reg Q;     wire D;     assign Y = Q;     assign D = A;     always@(posedge(clk) or posedge(rst)) begin         Q &lt;= (rst) ? 0 : D;     end endmodule </pre>
COUNTER		<pre> module counter(input A,                 output Y);     reg Q;     wire D;     assign Y = Q;     assign D = A;     always@(posedge(clk) or posedge(rst)) begin         Q &lt;= (rst) ? 0 : D + 1;     end endmodule </pre>

Tabela 2.1: Estruturas básicas de circuitos lógicos



Elemento Gráfico	Representação semi gráfica	Representação Gráfica
Contato NA	-----   -----	
Contato NF	-----  /  -----	

Tabela 2.2: Tipos de contatos de um diagrama Ladder - Fonte [17]

Elemento Gráfico	Representação semi gráfica	Representação Gráfica
Bobina	----- ( ) -----	— [ ] —
Bobina Negada É setada para o oposto do estado da linha de potência.	----- ( / ) -----	— [ / ] —

Tabela 2.3: Tipos de bobinas de um diagrama Ladder - Fonte [17]

Função	O que é	Exemplo
\$onehot0(expr)	Retorna TRUE se até no máximo um bit da expressão tem nível lógico TRUE	
\$rose(expr)	Retorna TRUE se o bit menos significativo da expressão mudou de FALSE para TRUE	
\$fell(expr)	Retorna TRUE se o bit menos significativo da expressão mudou de TRUE para FALSE	
\$stable(expr)	Retorna TRUE se o valor da expressão é constante	
expr1 ##N expr2	Representa um atraso temporal de N ciclos (No exemplo: N = 2)	
expr1 ##[m:n] expr2	Representa um atraso temporal que pode ocorrer no intervalo de m a n ciclos (No exemplo: M = 2, N = 4)	
expr1  -> expr2	*([0:\$] significa um intervalo de 0 a infinito)	
expr1  -> expr2	Representa uma implicação (if(expr1) then expr2)	
expr [->N]	Representa N repetições consecutivas da expressão (No exemplo: N = 4, expr = 1)	

Tabela 2.4: Funções úteis para descrição de propriedades em SVA



# Capítulo 3

## Revisão Bibliográfica

Os principais temas deste projeto envolvem Diagramas Ladder e Verificação Formal de Hardware. O primeiro assunto é amplamente abordado em artigos técnico-científicos de Automação Industrial, enquanto o segundo é ainda uma novidade na indústria de Verificação de Hardware. A utilização de Verificação Formal em contexto de diagramas Ladder é um tema pouco estudado, tanto pela falta de comunicação entre as áreas, quanto pelo fato de Verificação Formal ainda ser uma novidade, como dito previamente.

### 3.1 O uso de Verificação Formal

A Verificação por Simulação é o método mais utilizado na Indústria de Verificação de Hardware. Este método, porém, possui um caráter probabilístico [13], pois consiste em gerar vetores aleatórios de valores para alimentar as entradas. A Figura 3.1 representa um gráfico com o número de falhas encontradas pela Simulação no tempo. Por ser um método probabilístico, nem todas as falhas são encontradas. A pergunta é: Quando parar a Simulação?

A Verificação Formal, por sua vez, apresenta uma maneira determinística para detecção de falhas. Para esse determinismo, porém, há um custo: os modelos formais possuem uma complexidade computacional superior aos modelos de simulação. Problemas como explosão de estados (o circuito possui um número tal de estados que torna computacionalmente inviável a modelagem do mesmo) e a dificuldade da descrição de propriedades formais, mantêm este tipo de verificação com uma utilização relativamente baixa no mercado.

Realizando-se uma comparação entre os métodos [13], a verificação Formal possui um melhor ROI (Return of Investment), pois encontra mais falhas em menos tempo. A Figura 3.2 apresenta um gráfico comparativo entre Formal e Simulação.

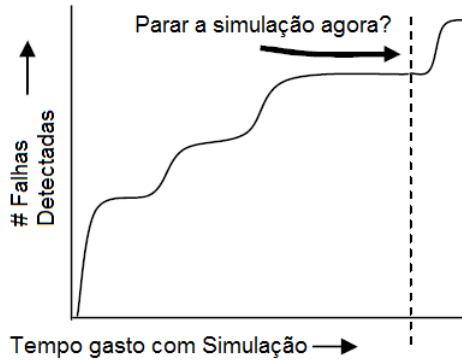


Figura 3.1: Falhas Detectadas em Simulação - Adaptado de [13]

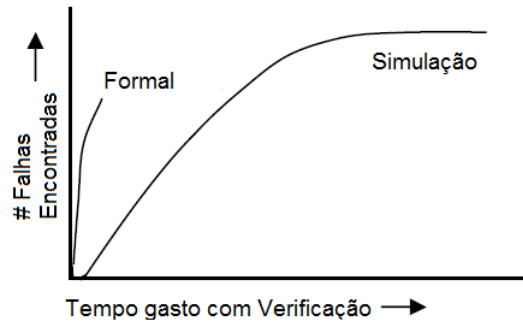


Figura 3.2: Comparação entre Verificação e Simulação - Adaptado de [13]

Para a realização deste projeto, a verificação Formal foi escolhida pela característica de detecção de falhas de forma determinística e exaustiva. Diagramas Ladder possuem tipicamente um número de estados em algumas ordens de grandeza menores que o número de estados dos hardwares atualmente verificáveis, fato que viabiliza a utilização de técnicas formais para verificar este tipo de descrição lógica.

## 3.2 Modelos formais de diagramas Ladder

Uma maneira de se obter modelos que possibilitem a verificação formal de diagramas Ladder, é a utilização da modelagem com abordagem "off-line". Inicialmente, obtém-se o modelo formal da lógica escrita no diagrama Ladder. Esse modelo nada mais é que uma descrição em máquina de estados finitos da lógica Ladder. Devido ao fato de este diagrama não estar conectado a um sistema em funcionamento, é necessária a modelagem do comportamento desse sistema, resultando em dois modelos formais: O primeiro, referente à lógica

do diagrama Ladder e o segundo, referente ao comportamento do sistema [18]. A Figura 3.3 representa as características da modelagem utilizando abordagem off-line. Em 1, ocorre a transformação do código desenvolvido e as especificações do programa, a partir de especificações informais, ou seja, do mundo físico, em um descrição formal. Em 2, as propriedades do sistema ao qual o Ladder interage, também devem ser descritas formalmente. Em 3, a verificação formal pode ser realizada a partir da utilização destes dois modelos.

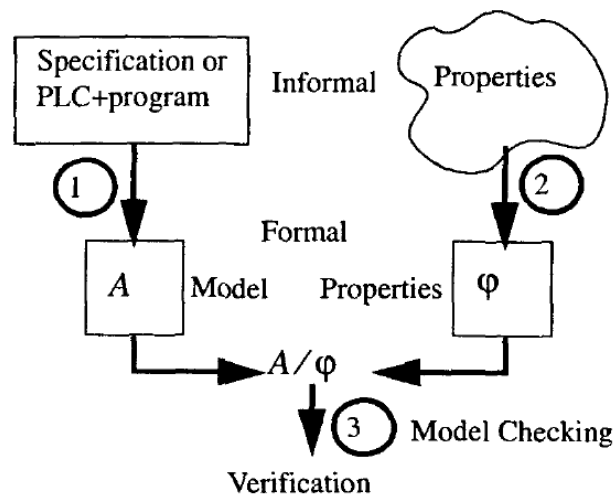


Figura 3.3: Obtendo modelo formal de diagramas Ladder - Fonte [18]

Uma maneira de se obter o Modelo Formal do diagrama Ladder é a conversão de sua lógica para um BDD (Binary Decision Diagram) [18]. O BDD representa um diagrama completo de uma máquina de estados finitos equivalente ao Ladder, possibilitando a utilização de Métodos Formais para a detecção de falhas.

A obtenção de diagramas BDD introduz um nível de complexidade além do desejável para este projeto. Visto que as ferramentas de verificação formal possuem algoritmos para converter descrições em HDL para representações adequadas à técnicas formais, entre elas o BDD, o modelo formal de diagramas Ladder adotado será baseado em descrições HDL da lógica apresentada em Ladder.

# Capítulo 4

## Materiais e Métodos

### 4.1 Materiais

#### 4.1.1 JasperGold®

O JasperGold® é a ferramenta de verificação formal de hardware desenvolvida pela Jasper Design Automation. Uma de suas características mais importantes é a capacidade de verificar de forma exaustiva se um circuito atende as propriedades especificadas pelo usuário. Além disso, a ferramenta possui diversas funcionalidades relacionadas a visualização de formas de onda, facilitando a depuração das falhas detectadas nos circuitos [2].

Devido aos variados tipos de projetos de Hardware, o JasperGold® possui diferentes especializações, denominadas Apps, para melhor atender à demanda da indústria de EDA. Apesar da possibilidade da utilização de cada App separadamente, a aplicação e os resultados deste são, em geral, aditivos entre si, de modo que múltiplos Apps podem ser utilizados em conjunto, criando um ambiente de verificação formal mais abrangente e poderoso [2].

Neste projeto, foi utilizado o App Formal Property Verification (FPV). Trata-se do App mais fundamental do JasperGold®, sendo o que possibilita a verificação e prova de propriedades formais. A Figura 4.1 representa a interface gráfica do JasperGold®, utilizando o App FPV [2]

A Figura 4.2 representa o modelo de uso do FPV. O App é alimentado com o projeto do circuito lógico e as propriedades formais. Tem como saída o resultado da verificação das propriedades aplicadas ao circuito.

A escolha do JasperGold® se deve ao fato de o desenvolvedor deste projeto estar familiarizado com suas funcionalidades e a facilidade de acesso ao mesmo, uma vez que ele possui licenças no laboratório do LECOM do Departamento de Ciência da Computação da UFMG e na própria Jasper Design Automation.

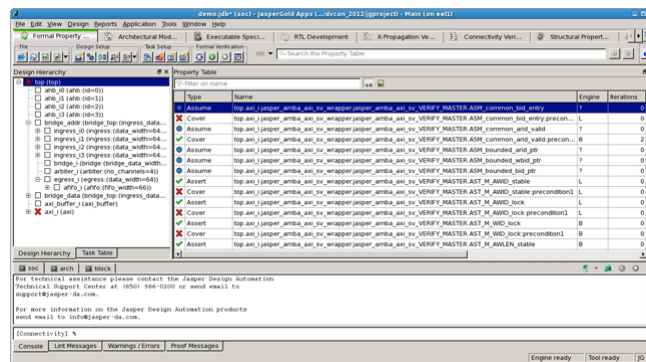


Figura 4.1: Interface gráfica do software JasperGold® - Fonte [2]

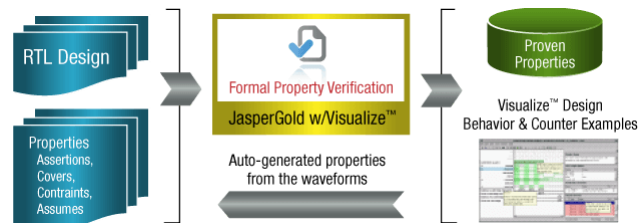


Figura 4.2: Fluxo de verificação do App FPV - Fonte [2]

### 4.1.2 i-TRiLOGI

Para a edição e simulação de diagramas Ladder, foi utilizado o software i-TRiLOGI, desenvolvido pela Triangle Research Internation. A empresa disponibiliza licenças educacionais desta aplicação, que permitem a edição e simulação de diagramas Ladder [9].

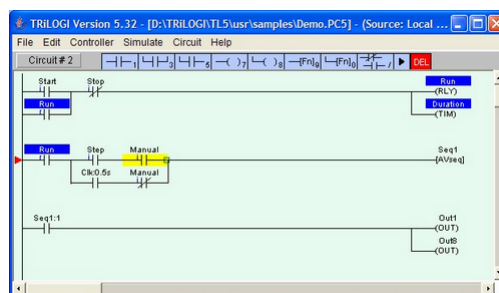


Figura 4.3: Software de edição de diagramas Ladder i-Trilogi - Fonte [9]

Além da disponibilidade de uma licença educacional, a escolha deste software foi motivada por sua capacidade de exportar diagramas Ladder em formato textual, facilitando a conversão automática destes diagramas para

uma representação em Verilog. A conversão será explicada detalhadamente no capítulo de Desenvolvimento.

### 4.1.3 C++

O C++ é uma linguagem de programação orientada a objetos. Foi criada pelo dinamarquês Bjarne Stroustrup e é uma extensão da linguagem C. É considerada como uma linguagem de médio nível, pois encapsula funcionalidades de alto e baixo níveis [10].

Esta linguagem foi utilizada para o desenvolvimento do parser que transforma os diagramas exportados do i-TRiLOGI para Verilog.

Sua utilização é justificada pois o autor deste projeto tem experiência em tal linguagem.

### 4.1.4 TCL

O Tcl, ou Tool Command Language, é uma linguagem de programação de scripts utilizada como padrão na indústria de EDA. É utilizada para automatizar a execução de cadeias de comandos das ferramentas desenvolvidos pelas principais empresas de EDA: Synopsys, Mentor Graphics®, Altera e Cadence [1].

O JasperGold também utiliza esta linguagem para a execução de seus comandos, por esse motivo ela foi utilizada.

## 4.2 Métodos

Para a realização da prova formal de diagramas Ladder, as ferramentas citadas posteriormente devem ser utilizadas em conjunto, como mostrado na Figura 4.4. Um diagrama Ladder, desenvolvido no i-TRiLOGI, deve ser exportado para formato texto. Um parser desenvolvido em C++ cria um arquivo Verilog descrevendo propriedades que capturam as especificações e o comportamento desejados para o sistema a ser verificado. Os arquivos Verilog gerados (descrição do sistema e propriedades) são utilizados como entradas para o JasperGold®, juntamente com um script Tcl para realizar o setup da ferramenta (o fluxo de setup está descrito no capítulo de Desenvolvimento). A partir deste ponto, o FPV verifica se o sistema respeita as propriedades descritas, reportando *Traces* quando apropriados.

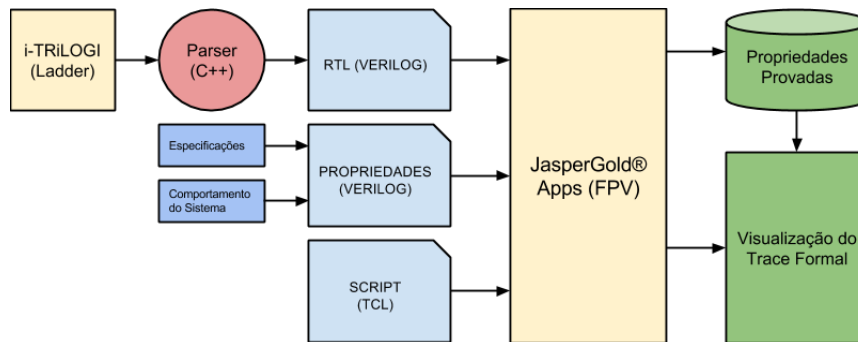


Figura 4.4: Verificação de diagramas Ladder utilizando JasperGold®

#### 4.2.1 Realizando o setup no JasperGold®

No contexto da ferramenta de verificação JasperGold®, o processo de setup consiste em carregar os arquivos HDL que descrevem o sistema, criar propriedades que descrevam as características e comportamentos que se deseja verificar, configurar a ferramenta para interpretar as informações dos arquivos de maneira apropriada e, finalmente, provar as propriedades do sistema. Esse processo pode ser executado manualmente, utilizando a interface gráfica da ferramenta ou executando comandos no terminal. Outra maneira é a utilização de um script contendo os comandos adequados (os mesmos que seriam utilizados para um setup manual). Uma vez que o processo de setup é apenas uma etapa deste trabalho e não constitui seu objeto de estudo, optou-se pelo setup automático. A Figura 4.5 mostra um exemplo do script de setup para JasperGold®. Além de carregar os arquivos HDL e descrever as propriedades, o script deve, no mínimo, definir um sinal para a sincronização dos estados do sistema (clock) e um sinal para indicar que o sistema deve ser colocado em seu estado inicial (reset). A definição dos valores iniciais a serem aplicados é parte da modelagem e, portanto, está incluída na descrição em HDL do sistema.

```
1  # Carrega os arquivos a serem lidos
2  analyze -sv LadderToRtl.v Properties.v
3
4  # Transforma os Verilogs na representação interna da
   Ferramenta
5  elaborate
6
7  # Declaração do sinal de clock
8  clock clk
9
10 # Declaração do sinal de reset
11 reset rst
12
13 # Prova todas as propriedades Formais
14 prove -all
```

Figura 4.5: Exemplo de setup da Ferramenta JasperGold®



# Capítulo 5

## Desenvolvimento

A abordagem "off-line" , apresentada na Seção 3.2, é uma possível maneira de se obter modelos formais de diagramas Ladder. Nessa técnica, são necessários dois modelos para representar corretamente o sistema: um modelo lógico do Diagrama Ladder e um modelo que represente o comportamento e as especificações do sistema [18].

### 5.1 Modelo lógico do diagrama Ladder

O modelo lógico do diagrama Ladder é aquele convertido diretamente da lógica escrita no CLP para Verilog. Como citado na Seção 2.3, as construções dos diagramas Ladder permitem traduções diretas para HDL.

Apesar dessa característica, existe uma construção específica que é problemática para esse tipo de tradução: um intertravamento em um diagrama Ladder, caracterizado por uma bobina cuja lógica de acionamento depende de seu próprio valor, quando estritamente mapeado para uma construção equivalente em Verilog, corresponde a um loop combinacional. A Figura 5.1 mostra uma construção de diagrama Ladder com a característica de intertravamento e a Figura 5.2 a tradução direta para circuito lógico, gerando o loop.



Figura 5.1: Diagrama Ladder com intertravamento

Os loops combinacionais são problemáticos pois sua construção tem característica sequencial, mesmo sendo um elemento combinacional. Analisando

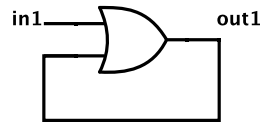


Figura 5.2: Loop combinacional obtido com a transformação direta do diagrama Ladder descrito na Figura 5.1

no contexto de FSMs, as transições não são sincronizadas por um sinal de clock, fato que cria dependências recursivas. As ferramentas de verificação Formal, por sua vez, têm problemas para tratar tais dependências [16]. Existem ainda os casos de instabilidade, onde o loop combinacional não possui um valor constante, como o mostrado na Figura 5.3.

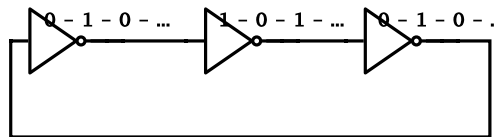


Figura 5.3: Loop combinacional instável - Adaptado de [16]

Com o objetivo de se evitar loops combinacionais, uma técnica que sincroniza os loops foi utilizada [4]. Ela consiste na adição de registradores de estado de modo a quebrar os loops, fato que sincroniza o sistema. A Figura 5.4 representa a quebra do Loop combinacional apresentado pela Figura 5.2. Essa abordagem adiciona novos elementos ao modelo de intertravamento:

- clock: Sinal que sincroniza os estados do loop combinacional. No contexto do diagrama Ladder, cada borda de subida do sinal de clock pode ser considerado como um scan completo do diagrama Ladder. Isso significa que o modelo formal não irá levar em consideração a dependência entre linhas do diagrama Ladder;
- reset: Sinal que é a condição inicial do sistema. No momento que esse sinal é 1, o registrador recebe o valor de `reset_val`;
- reset\_val: Sinal que define o valor inicial do registrador. Tipicamente será 0 quando a bobina for Normalmente Aberta e 1 se a mesma for Normalmente Fechada.
- out1\_in: Sinal interno que representa a quebra do loop combinacional. Realimenta a lógica combinacional;

- out1: Sinal que representa o estado atual da bobina, seu valor é salvo no registrador;
- in1: Sinal que representa uma variável associada a um contato no diagrama Ladder.

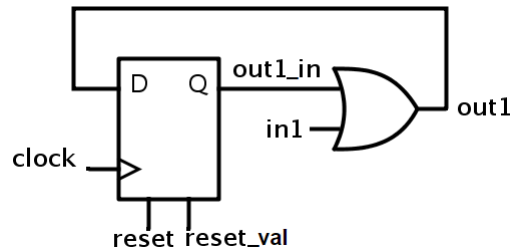


Figura 5.4: Quebra do loop combinacional pela adição de Flop-Flop D

É possível generalizar a quebra de loop para qualquer lógica de intertravamento, como pode ser observado na Figura 5.5. O sinal interno out\_in, irá alimentar a lógica combinacional, onde antes eram realizadas as conexões da bobina out. As entradas in1, in2 e inN são as variáveis associadas aos contatos que fazem parte da lógica que aciona a bobina out. Essas variáveis devem ter seus valores iniciais forçados para 0, no caso em que eles não são associadas a bobinas ou que tenham seu valor forçado por outra regra.

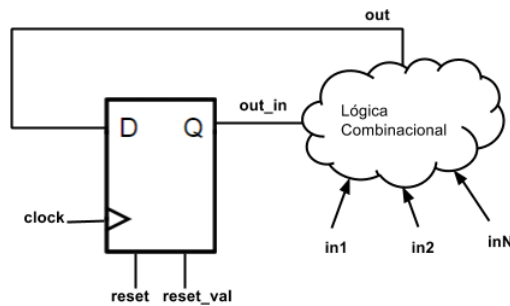


Figura 5.5: Modelagem da quebra de um loop genérico

De uma maneira geral, a conversão entre as representações de Ladder e Verilog ocorre de acordo com a Tabela 5.1.

## 5.2 Conversão Automática para Verilog

Para realizar a conversão automática dos diagramas Ladder para Verilog, foi desenvolvido um programa em C++. Esse programa tem como entrada um

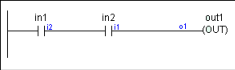
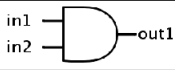
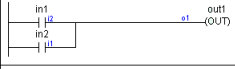
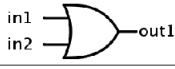
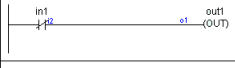
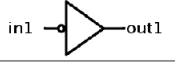
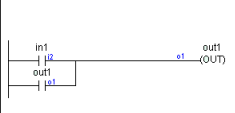
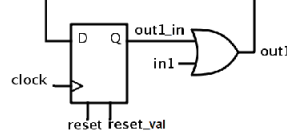
Ladder	Circuito	Verilog
		<pre>assign out1 = in1 &amp;&amp; in2;</pre>
		<pre>assign out1 = in1    in2;</pre>
		<pre>assign out1 = !in1;</pre>
		<pre>assign out1 = out1_in    in1; always @(posedge clock or posedge reset) begin     out1_in &lt;= (reset) ? reset_val : out1; end</pre>

Tabela 5.1: Conversão das funções básicas de diagramas Ladder para Verilog

arquivo texto, exportado do i-TRiLOGI, e como saída um código em Verilog equivalente. Para realizar tal conversão, foi criada uma estrutura de dados que tem o papel de mapear os contatos/bobinas e seus relacionamentos. Cada bobina do diagrama Ladder possui uma expressão associada. Se a expressão de uma bobina possui um contato referente à própria bobina, significa um nó reconvergente. Esse tipo de identificação é necessária para que a quebra de loop seja feita corretamente. A Tabela 5.1 representa a conversão de construções básicas de diagramas Ladder para o código relativo em Verilog.

O arquivo exportado pelo i-TRiLOGI possui as seguintes informações:

- Lista com os nomes dos contatos;
- Lista com os nomes das bobinas;
- Expressões referentes a cada bobina, na ordem em que aparecem no diagrama Ladder;

A sintaxe das expressões no formato i-TRiLOGI é similar à sintaxe das mesmas em Verilog. A diferença está apenas nos caracteres referentes às operações, fato que possibilita a conversão direta, de modo a apenas alterar tais caracteres. A regra seguida é: Os caracteres '\*', '+' e '/' são substituídos por '&&', '||' e '!' respectivamente. Nos casos de loop combinacional, as ocorrências da bobina reconvergente que se encontram à direita da expressão, são substituídas pelo pino interno das mesmas.

O código completo do Parser está disponível no Capítulo 8.

### 5.2.1 Exemplo

Para melhor explicar o funcionamento do algoritmo de conversão automática, será apresentado um exemplo prático da utilização do programa. O diagrama Ladder da Figura 5.6 foi exportado para o formato texto e pode ser observado na Figura 5.7. A Netlist para esse caso pode ser representado pelo diagrama da Figura 5.8. Cada bobina possui uma expressão, no formato exportado do i-TRiLOGI, e a lista das variáveis associadas aos contatos e às bobinas presentes na expressão. Existe intertravamento para as Bobinas1 e Bobina2.



Figura 5.6: Diagrama Ladder de exemplo

```

i-TRiLOGI ver 5.00,Contato1
1,Contato2
2,Contato3
3,Contato4
~
0,Bobina1
1,Bobina2
2,Bobina3
~
~
~
~
Bobina1=(Contato1*/Contato2+Bobina1)*/Bobina2
Bobina2=(Contato1+Bobina2)*/Contato2*(/Bobina3+Contato3)
Bobina3=(Contato1*Contato2+Contato3)*Contato4
$

~END_CIRCUIT~

~END_CUSTFN~
~END_CUSTFNLABEL~
0,0,
1,0,
2,0,
3,0,
4,0,
~END_QUICKTAGS~
~END_DEFINES~
~END_BREAKPOINTS~

```

Figura 5.7: Arquivo referente ao diagrama Ladder da Figura 5.6

Utilizando a regra de conversão direta das expressões entre o formato i-TRiLOGI e Verilog, as expressões podem ser re-escritas como:

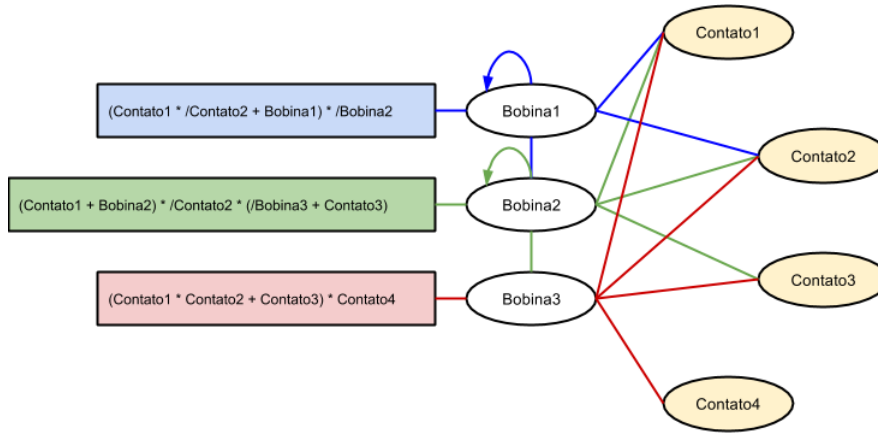


Figura 5.8: Netlist referente ao diagrama Ladder da Figura 5.6

```

assign Bobina1 = (Contato1 && !Contato2 || Bobina1_in) && !Bobina2;
assign Bobina2 = (Contato1 || Bobina2_in) && !Contato2 && (!Bobina3 || Contato3);
assign Bobina3 = (Contato1 && Contato2 || Contato3) && Contato4;

```

Uma vez que a expressão de cada bobina é convertida para a sintaxe do Verilog, basta conectar os sinais da quebra do loop combinacional ao modelo do Flip-Flop D apresentado na Tabela 5.1. O digrama de portas lógicas para o exemplo das Figuras 5.6, 5.7 e 5.8 pode ser observado na Figura 5.9. Nessa Figura, as ligações de clock e reset foram removidas para garantir melhor visualização do circuito.

O próximo passo é criar o modelo do comportamento do Sistema, para que seja possível validar suas propriedades.

### 5.3 Modelo Comportamental

O Modelo Comportamental é uma descrição do comportamento do sistema através da utilização de propriedades. Ele se faz necessário pois na abordagem "off-line" o diagrama Ladder não está conectado a um sistema em funcionamento, o que leva à necessidade de se modelar o ambiente com o qual o diagrama interage [18].

O que é possível inferir do comportamento do diagrama Ladder da Figura 5.10 ?

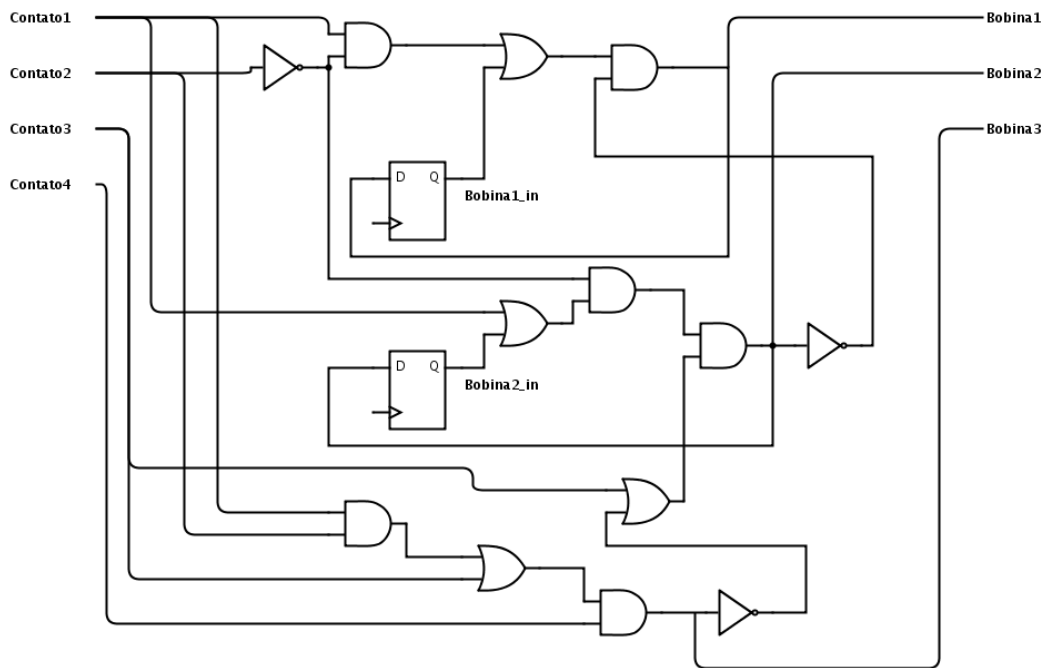


Figura 5.9: Diagrama de portas lógicas referente ao diagrama Ladder da Figura 5.6

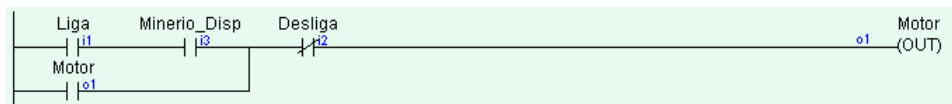


Figura 5.10: Exemplo de diagrama Ladder

- Se as variáveis associadas aos contatos Liga, Minerio\_Dis e Desliga estiverem em 1, 1 e 0 respectivamente, o Motor será ligado;
- O Motor será desligado quando a variável associada ao contato Desliga for 1;

Porém, como se comportam essas variáveis? Quando o motor deve ser ligado? Quando o Desliga é acionado? Para responder a essas perguntas, é necessário o conhecimento de como o sistema controlado funciona.

Supondo que esse sistema é uma esteira que transporta minério entre um silo e um pátio de armazenamento. No momento em que há uma quantidade suficiente de minério no começo da esteira, o operador da mesma pressiona o botão Liga, fazendo a esteira funcionar. Após algum tempo de funcionamento, um sensor, associado à variável Desliga, indica que o minério chegou

ao pátio e desliga o motor. Um segundo operador descarrega a esteira e o processo continua.

É possível abstrair algumas propriedades referentes ao comportamento do modelo:

- Propriedade 1: O operador da esteira deverá pressionar o botão liga apenas se o Motor estiver parado;
- Propriedade 2: O sensor Desliga será acionado algum tempo após o Motor começar a funcionar. Essa é a única condição na qual esse sensor é acionado;

Outras propriedades podem ser necessárias para compreender o comportamento esperado do sistema. Nesse exemplo, além das propriedades 1 e 2, duas outras características são importantes:

- Propriedade 3: Disponibilidade de minério;
- Propriedade 4: Tempo gasto pelo segundo operador para descarregar a esteira;

No entanto, as informações e suposições feitas até agora não são suficientes para determinar as duas últimas propriedades. Em uma aplicação real, esses comportamentos são mensuráveis e poderiam ser estimados. Nesse exemplo, valores concretos para estas foram escolhidos arbitrariamente. Para realizar a verificação da validade das mesmas na lógica do sistema, é necessário expressá-las em um formato reconhecido pela ferramenta JasperGold®. Por esse motivo, as mesmas foram descritas utilizando propriedades SVA, apresentadas na Seção 2.4.3.

### 5.3.1 Formalizando as propriedades

Para descrever em SVA as propriedades relacionadas ao comportamento do sistema, utiliza-se a diretiva Assumption. Esse tipo de propriedade descreve uma restrição no modelo, de forma a reduzir o espaço de estados da análise e concentrar os esforços apenas sobre os comportamentos permitidos ao sistema. No contexto do exemplo da Figura 5.10, a propriedade será analisada para explicar o porquê de sua descrição formal em SVA tem o formato apresentado.

- Propriedade 1: Como o operador só pode mandar mais minério para o silo se a esteira estiver parada e se houver minério disponível, essa propriedade pode ser formalizada como:



*assume property (Liga == !Motor && Minerio\_Disp);*

- Propriedade 3: Não há informação suficiente para determinar essa propriedade. Para o caso, pode se assumir que há sempre minério disponível:

*assume property (Minerio\_Disp == 1);*

Outra opção seria permitir que a ferramenta explore todas as possibilidades de valores para o sinal `Minerio_Disp` durante a verificação. Para realizar a descrição de tal comportamento, basta não escrever propriedades que restrinjam o valor desse sinal. Para a implementação deste exemplo, foi escolhido que sempre haverá minério disponível.

- Propriedade 4: Tal qual a propriedade 3, não há informação suficiente para que essa propriedade possa ser determinada. Será assumido que o segundo operador sempre limpa a esteira imediatamente, e o tempo que ele gasta realizando a tarefa é desprezível. Ele não possui uma propriedade descrita, pois seu comportamento será descrito quando o comportamento dos equipamentos for criado.

A propriedade 2 possui uma peculiaridade que a diferencia das demais: Ela descreve um comportamento que modela um outro sistema completo, correspondente ao motor e à esteira. Esse subsistema possui seu próprio conjunto de funcionalidades e características operacionais, portanto, para descrevê-los em SVA, seria necessária uma construção excessivamente aparelhada e complexa, ou haveria a necessidade de se realizar considerações muito fortes. Nesses casos, uma solução alternativa é a criação de um circuito que apresente um comportamento semelhante ao subsistema em questão. Os sinais desse novo circuito são utilizados para alimentar o sistema e as propriedades que se deseja verificar. Apesar de exigir um volume maior de trabalho na criação de descrições, essa solução facilita o entendimento do sistema e das relações entre os componentes.

### 5.3.2 Comportamento de Equipamentos

Há uma infinidade de equipamentos tipicamente utilizados em processos que utilizam diagramas Ladder. Como a modelagem de um conjunto amplo desses equipamentos foge ao escopo deste projeto, apenas dois sistemas dessa classe serão considerados: uma esteira de transporte e um container de armazenamento.

## Esteira

A Figura 5.11 representa um esquemático da implementação da esteira. Ela funciona da seguinte maneira:

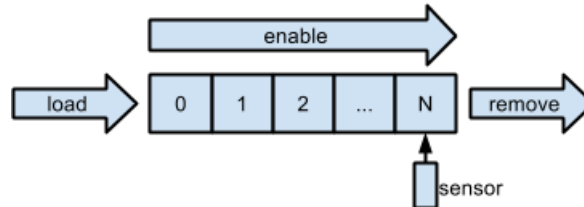


Figura 5.11: Esquemático do funcionamento da esteira

- A esteira é representada por um array de bits;
- O sinal 'load' é o responsável por carregar um item na primeira posição do array;
- A esteira comporta apenas 1 item por vez;
- Enquanto o sinal 'enable' for 1, a esteira leva o item da sua posição inicial (0) para a posição final (N), passando pelas posições intermediárias. Esse comportamento é implementado fazendo-se uma operação de deslocamento (shift) para a direita a cada ciclo de clock. A conversão entre ciclos de clock e tempo não será abordada profundamente neste trabalho, porém uma consideração simples foi feita: é possível separar as transições lógicas em duas fases, a primeira corresponde às transições de estado e a segunda a uma propagação de tempo causada pelas transições [7]. Isso faz com que análise de tempo para cada clock seja diferente, mesmo que eles possuam tamanhos iguais no Trace. A Figura 5.12 demonstra tal comportamento.
- Se o sinal 'enable' for 0 e existir uma peça na esteira, a peça ficará naquela posição até que 'enable' seja 1 novamente;
- Quando um item chega ao final da esteira, o sinal 'sensor' é ativado;
- Quando o sinal 'remove' é acionado, o item é retirado da última posição da esteira. Se não houver itens na esteira e o 'remove' for acionado, a mesma continua vazia;
- O tamanho da esteira pode ser parametrizado na instanciação da mesma;

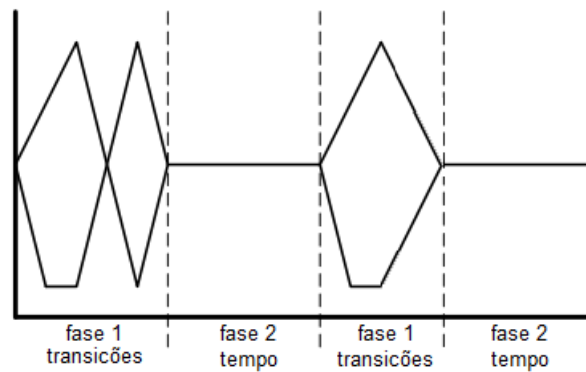


Figura 5.12: Análise de tempo em duas fases - Adaptado de [7]

No exemplo da Figura 5.10, o Motor seria ligado ao sinal 'enable', o botão Liga seria ligado ao sinal 'load' e o sinal Desliga seria ligado ao 'sensor'. Para esse exemplo especificamente, remove seria 1, por conta da Propriedade 4.

### Container

A Figura 5.13 representa um esquemático da implementação do container. Ele funciona da seguinte maneira:

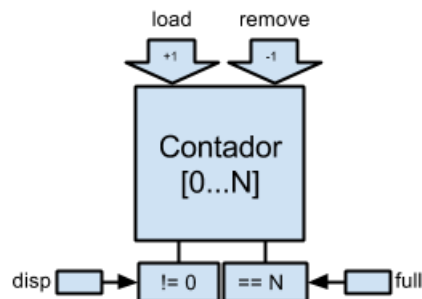


Figura 5.13: Esquemático do funcionamento do container

- O container é representado por um contador;
- O sinal 'load' é o responsável por adicionar um item (+1 ao contador);
- O sinal 'remove' é o responsável por remover um item (-1 ao contador);
- O container pode conter vários itens;
- Sempre que há um item (contador  $\neq 0$ ), o sinal 'disp' é ativado;

- Quando a caixa está cheia (contador == N), o sinal 'full' é ativado;
- O tamanho do container pode ser parametrizado na instanciação do mesmo;

Em termos práticos: eventos que iriam inserir itens no container devem ser conectados ao pino 'load'. Eventos que irão retirar itens da caixa, devem ser conectados ao pino 'remove'. Sensores que verificam se há itens disponíveis e se a caixa está cheia, serão conectados aos pinos 'disp' e 'full' respectivamente.

## 5.4 Modelo de Especificações

O modelo do diagrama Ladder representa o sistema controlado por essa implementação. O próximo passo é verificar se a mesma atende às especificações do projeto. Para isso, serão utilizadas propriedades do tipo Assertion e Cover, tais como descritas na Seção 2.4.3.

A Figura 5.14 representa o diagrama Ladder do sistema de controle de um motor que pode ser acionado em dois sentidos, porém, para ilustrar o tipo de falhas lógicas que serão depuradas, há um erro no diagrama que seria inadmissível em uma aplicação real.



Figura 5.14: Ligação errada de motor de duas rotações

A diretiva Cover pode ser utilizada para provar que um determinado comportamento é possível para o sistema sob verificação. No exemplo da Figura 5.14 poderia ser desejável verificar que o acionamento do botão Liga\_Horario ativa a bobina Rot\_Horario. A estrutura de um cover para verificar esse comportamento seria:

```
cover property(Liga_Horario |-> Rot_Horario);
```

Analogamente, o Cover para verificar a relação entre Liga\_AntiHorario e Rot\_AntiHorario seria:

*cover property(Liga\_AntiHorario  $\mid$ - > Rot\_AntiHorario);*

Essas propriedades mostrarão que a sequência de acionamento de cada motor é permitida pela lógica do sistema, porém não garantem que ambas as sequências não possam ser executadas simultaneamente. Pelas regras de segurança, a configuração de um sentido de rotação deve excluir o outro. Essa regra pode ser expressa na forma de uma Assertion:

*assert property(!(Rot\_Horario && Rot\_AntiHorario));*

Apenas inspecionando o diagrama Ladder que controla o motor, sabemos que a Assertion não é respeitada, pois nada impede que ambos sentidos de rotação sejam ligados simultaneamente. Assim, o sistema viola uma de suas especificações. Em uma aplicação real, esse erro poderia ter consequências desastrosas se não fosse detectado e corrigido a tempo.

# Capítulo 6

## Resultados

Os modelos desenvolvidos foram submetidos a três testes distintos. O primeiro, um teste simples, com o objetivo de validar a transformação do diagrama Ladder para Verilog. O segundo, mais complexo que o primeiro, tem o objetivo de validar o modelo comportamental, utilizando os equipamentos esteira e container. O terceiro e último teste, tem o intuito de mostrar que a união entre ferramentas de verificação formal e a automação industrial pode formar uma poderosa ferramenta de validação de sistemas.

### 6.1 Motor de Duas Rotações

O primeiro problema abordado é o acionamento de um Motor de Duas Rotações. Dois push buttons, Liga\_Horario e Liga\_AntiHorario são acionados de modo a escolher o sentido de rotação do motor. Esses botões acionam duas bobinas: Rot\_Horario e Rot\_AntiHorario. Deve se evitar que as bobinas estejam acionadas simultaneamente. Esse problema foi escolhido pois é simples e possibilita demonstrar como utilizar as propriedades de verificação. Essas propriedades podem guiar o desenvolvedor do código a melhorar sua implementação, de acordo com os traces das propriedades. Não é necessário criar um modelo de comportamentos para esse caso, pois o acionamento do motor já descreve todo seu comportamento.

As seguintes propriedades de especificação foram criadas para o problema:

- Propriedade 1: Para garantir que as bobinas de rotação não sejam acionadas simultaneamente:

*assert property(!(Rot\_AntiHorario && Rot\_Horario));*

- Propriedade 2: Para observar que o botão Liga\_Horario aciona a bobina Rot\_Horario:

*cover property(Liga\_Horario |-> Rot\_Horario);*

- Propriedade 3: Para observar que o botão Liga\_AntiHorario aciona a bobina Rot\_AntiHorario:

*cover property(Liga\_AntiHorario  $\mid\rightarrow$  Rot\_AntiHorario);*

- Propriedade 4: Para observar que se a bobina Rot\_Horario for ligada, a bobina Rot\_AntiHorario pode ser ligada posteriormente:

*cover property(Rot\_Horario  $\mid\rightarrow$  ##[1 : \$] (Rot\_AntiHorario && !Rot\_Horario);*

- Propriedade 5: Para observar que se a bobina Rot\_AntiHorario for ligada, a bobina Rot\_Horario pode ser ligada posteriormente:

*cover property(Rot\_AntiHorario  $\mid\rightarrow$  ##[1 : \$] (Rot\_Horario && !Rot\_AntiHorario);*

### 6.1.1 Teste 1

O primeiro diagrama Ladder desenvolvido pode ser observado na Figura 6.1. Ele consiste em utilizar as bobinas do motor de modo a intertravar e bloquear a rotação para sentidos opostos simultaneamente. A Figura 6.2 mostra as Propriedades provadas pela Ferramenta JasperGold®. A primeira propriedade de cima para baixo, representa a Propriedade 1 de especificação e assim sucessivamente. É possível observar que a Propriedade 1 foi violada, ou seja, existe um caso que faz com que as bobinas possam ser acionadas simultaneamente. A Ferramenta mostra o trace da Figura 6.3. Ele mostra que, se as variáveis associadas aos contatos Liga\_Horario e Liga\_AntiHorario forem 1 simultaneamente, as bobinas de ambas rotações serão ligadas simultaneamente.

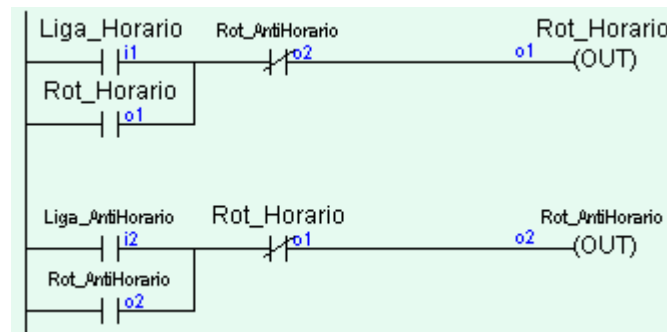


Figura 6.1: Primeira implementação do acionamento do motor de duas rotações

As outras propriedades foram verificadas e estão de acordo com as especificações. Porém é necessário que todas as propriedades sejam aprovadas para que todas as especificações sejam corretamente contempladas.

▼	Type	▼	Name	▼	Engine	Bound
✗	Assert		top._assert_1		Ht	2
✓	Cover		top._cover_2		Ht	2
✓	Cover		top._cover_3		Ht	2
✓	Cover		top._cover_4		Ht	4
✓	Cover		top._cover_5		Ht	4

Figura 6.2: Tabela com o resultado da prova das propriedades de especificação, de acordo com o diagrama Ladder da Figura 6.1

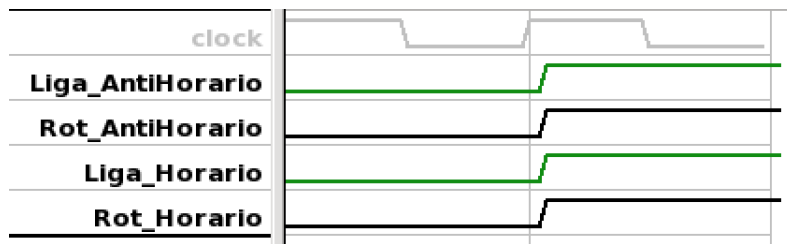


Figura 6.3: Trace da condição que viola a Propriedade 1

### 6.1.2 Teste 2

A partir do trace do exposto pela Propriedade 1, é necessário modificar o diagrama Ladder, de modo que o acionamento simultâneo dos push buttons não permita o também acionamento simultâneo dos sentidos de rotação. A Figura 6.4 representa a mudança utilizada. Ela fez com que os contatos dos push buttons sejam mutuamente exclusivos, fazendo com que o acionamento de ambos simultaneamente não deixe energia fluir para as bobinas. A Figura 6.5 mostra que nem todas as propriedades foram satisfeitas para esse caso, apesar de as Propriedades 1, 2 e 3 terem sido corretamente provadas.

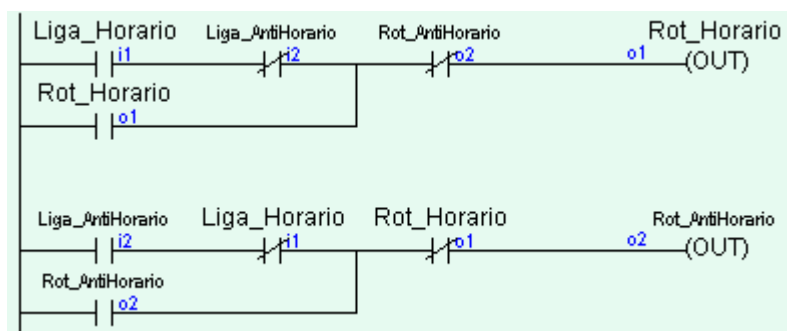


Figura 6.4: Implementação com Trava nos acionamentos de entrada



▼	Type	▼	Name	▼	Engine	Bound
✓	Assert		top._assert_1		Hp (1)	Infinite
✓	Cover		top._cover_2		Ht	2
✓	Cover		top._cover_3		Ht	2
✗	Cover		top._cover_4		Hp (4)	Infinite
✗	Cover		top._cover_5		Hp (4)	Infinite

Figura 6.5: Tabela com o resultado da prova das propriedades de especificação, de acordo com o diagrama Ladder da Figura 6.4

As propriedades 4 e 5 são referentes à capacidade do sistema de, após acionar um sentido de rotação, ser capaz de desligar o atual sentido e acionar o outro. Portanto, o diagrama proposto, apesar de bloquear o acionamento simultâneo das rotações, não funciona como o especificado.

### 6.1.3 Teste 3

Para resolver o problema das Propriedades 4 e 5, foi realizada uma implementação como a da Figura 6.6. O contato normalmente fechado das bobinas foi retirado, pois eram eles que não permitiam a uma segunda bobina ligar, se uma primeira já tivesse sido acionada. No lugar destes contatos, foram inseridos os contatos normalmente fechados dos push buttons, de modo que ligar um sentido, abre o contato do outro. A Tabela de propriedades provadas no JasperGold® ficou como representado na Figura 6.7. Todas as propriedades de especificação foram corretamente comprovadas, mostrando que a implementação funciona de acordo com a especificação.

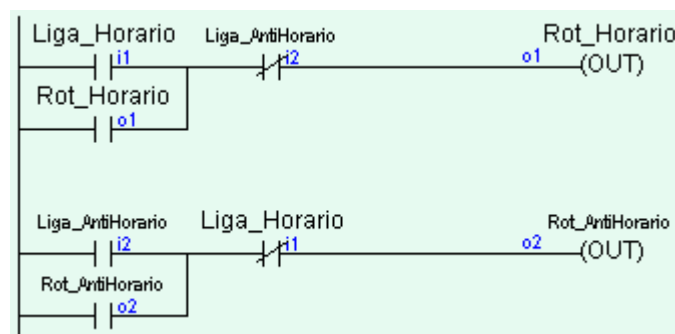


Figura 6.6: Implementação com linhas mutualmente exclusivas

▼	Type	▼	Name	▼	Engine	Bound
✓	Assert		top._assert_1		Hp (1)	Infinite
✓	Cover		top._cover_2		Ht	2
✓	Cover		top._cover_3		Ht	2
✓	Cover		top._cover_4		Ht	4
✓	Cover		top._cover_5		Ht	4

Figura 6.7: Tabela com o resultado da prova das Propriedades de Especificação, de acordo com o diagrama Ladder da Figura 6.6

## 6.2 Esteira Transportadora

A Esteira Transportadora é um exemplo adaptado de uma aula prática da disciplina Informática Industrial, fornecida pelo curso de Controle e Automação da Universidade Federal de Minas Gerais [12]. Ela foi utilizada para validar a descrição de modelos comportamentais, bem como a implementação dos equipamentos esteira e container. Para esse caso, o container se comporta como uma caixa.

A Figura 6.8 representa a planta da esteira transportadora. Ela se comporta da seguinte maneira:

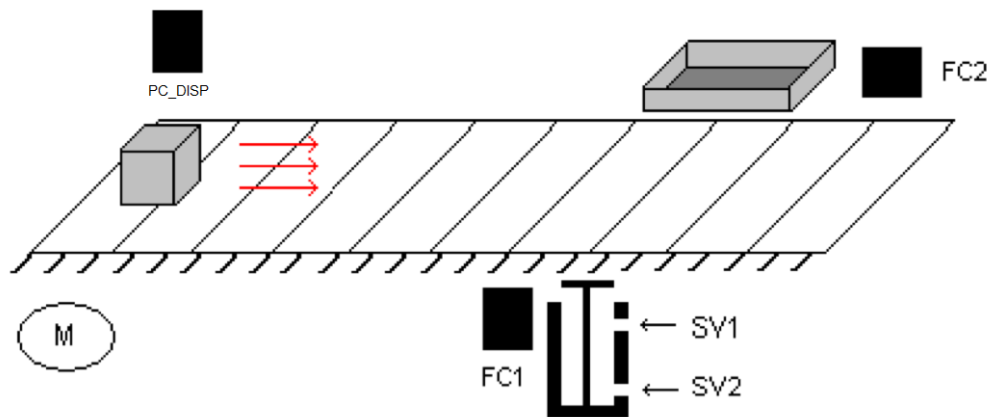


Figura 6.8: Planta referente à esteira transportadora

- Quando há uma peça disponível na esteira, um sensor 'PC\_DISP' é acionado;
- Ao ser acionado o comando 'START' pelo operador da esteira, o motor da mesma é energizado e a válvula solenóide SV1 é energizada, provocando o recuo do pistão de expulsão;

- A peça é transportada pela esteira, até que chegue no sensor de proximidade FC1, provocando o desligamento do motor da esteira e o acionamento da válvula solenóide SV2, ação que expulsa a peça da esteira.
- Uma vez expulsa, a peça ativa o sensor de proximidade FC2, causando desenergização da válvula solenóide SV2 e acionando a válvula SV1 (provocando o recuo do pistão);
- A peça é armazenada em uma caixa.

As seguintes propriedades de especificação foram criadas para o problema:

- Propriedade 1: Para garantir que as válvulas solenóides não sejam acionadas simultaneamente:

*assert property(!(SV1 && SV2));*

- Propriedade 2: Para garantir que o motor não estará funcionando quando a peça estiver sendo expulsa:

*assert property(!(Motor && SV2));*

- Propriedade 3: Para garantir que sempre que o botão START foi pressionado, uma peça irá parar no FC1:

*assert property(@(posedge clock) START | - > ##[0 : \$] FC1);*

- Propriedade 4: Para garantir que sempre que o botão START foi pressionado, uma peça irá parar no FC2:

*assert property(@(posedge clock) START | - > ##[0 : \$] FC2);*

- Propriedade 5: Para observar uma peça chegando a FC2:

*cover property(FC2);*

- Propriedade 6: Para observar a caixa cheia:

*cover property(Caixa.full);*

Para a obtenção do modelo formal completo do sistema, é necessário criar o seu modelo de comportamento. Para modelar o comportamento do sensor FC2, basta criar uma propriedade que afirma que toda vez que SV2 for acionado, após um ciclo haverá uma peça em FC2:

*assume property(@(posedge clock) \$rose(SV2) - > ##1 FC2);*

Para modelar o comportamento do operador que pressiona o botão START, foi utilizada a propriedade:

$$\text{assume property}(@(\text{posedge clock}) \text{START} == \$\text{rose}(\text{PC\_DISP}));$$

Basicamente, toda vez que há uma peça disponível, o operador vai estar pronto para apertar START.

Para modelar a esteira, foi utilizado o modelo de comportamento da esteira, e para modelar a caixa, foi utilizado o modelo de comportamento de container, abordados nas Seções 5.3.2 e 5.3.2 respectivamente.

### 6.2.1 Comportamento da Esteira

Como o comprimento da esteira não foi especificado, e não é um fator crítico para a validação do sistema, foi utilizada uma esteira de 4 ciclos. O Motor (M) é o responsável por fazer a esteira movimentar, então seu sinal foi conectado ao 'enable' da esteira. O sensor PC\_DISP informa que há peças no início da esteira, por isso ele foi conectado ao 'load'. Finalmente, a válvula solenóide SV2 é responsável por remover itens da esteira, por tal motivo, ele foi conectado ao 'remove'.

### 6.2.2 Comportamento da Caixa

A capacidade da caixa também não foi especificada. Visto que não há um valor preferencial para sua grandeza, assume-se arbitrariamente que a caixa comporta sete peças. O sensor FC2 foi conectado ao 'load', já que é ele que indica que uma peça entrou. Não existe nenhum elemento que remove peças da mesma, então o 'remove' é conectado a 0, ou seja, nada pode tirar itens da caixa.

### 6.2.3 Validando o modelo de Comportamento

A Figura 6.9 contém o diagrama de estados do sistema. Ela ilustra a sequência de funcionamento, desde o acionamento do botão Liga (no caso, START) até a chegada da peça em FC2. Esse ciclo se repete até que o sistema esteja desligado. A Figura 6.10 representa o trace resultante da prova da propriedade 6. Como essa é uma propriedade do tipo Cover, seu trace prova que o comportamento desejado é permitido pela implementação, comprovando que o modelo comportamental e a descrição da esteira estão de acordo com o sistema real. Note que o sinal Esteira se refere ao sinal Motor.

Há uma diferença no comportamento do sinal Liga em relação ao START. Como o problema foi adaptado, a adição do sensor PC\_DISP fez com que a forma de onda desse sinal, em relação ao do problema original, fosse diferente. Portanto essa diferença não é caracterizada como um erro, pois nesse ponto específico, os sistemas são diferentes.

Há outras diferenças: O trace formal possui as transições sincronizadas por um sinal de clock, enquanto o diagrama de estados não possui nenhum tipo de sincronização; o tempo que SV2 gasta para empurrar a peça da esteira para a caixa foi considerado desprezível no modelo formal, sendo que SV2 gasta um tempo não desprezível no diagrama de estados (essa especificação não foi explicitada na descrição do problema).

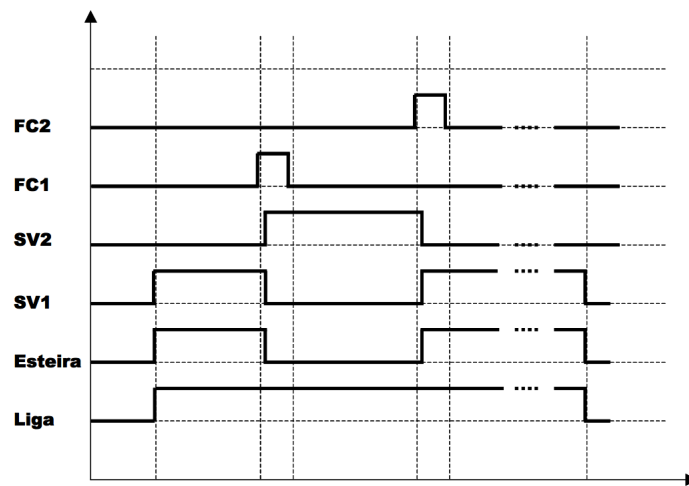


Figura 6.9: Diagrama de estados da esteira transportadora

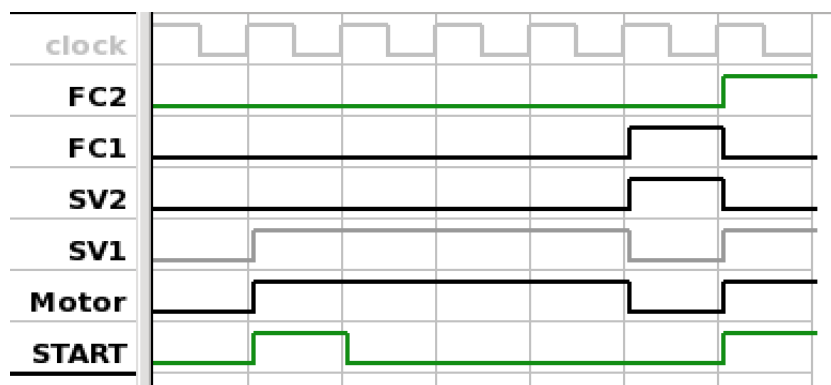


Figura 6.10: Trace obtido a partir da prova formal da Propriedade 5

Como o sistema joga peças numa caixa, o modelo do comportamento da mesma foi utilizado para observar seu enchimento, conforme pode ser observado no trace representado pela Figura 6.11. O sistema apresenta um comportamento cíclico até que a caixa contenha sete elementos. Nesse estado, a caixa se encontra cheia, bloqueando o sistema de depositar mais peças.

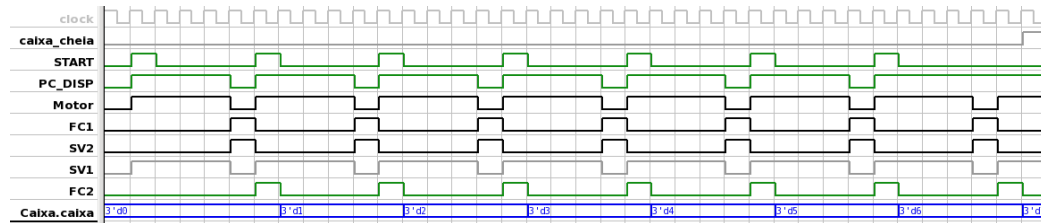


Figura 6.11: Trace obtido a partir da prova formal da Propriedade 6

### 6.3 Detecção de Deadlock

O terceiro caso de teste foi criado pelo supervisor deste projeto, com o intuito de mostrar um caso em que a ferramenta detecta, a partir da correta especificações de propriedades, estados que podem levar um sistema ao deadlock. A Figura 6.12 representa um processo no qual uma peça é pintada e envernizada. O processo funciona da seguinte maneira:

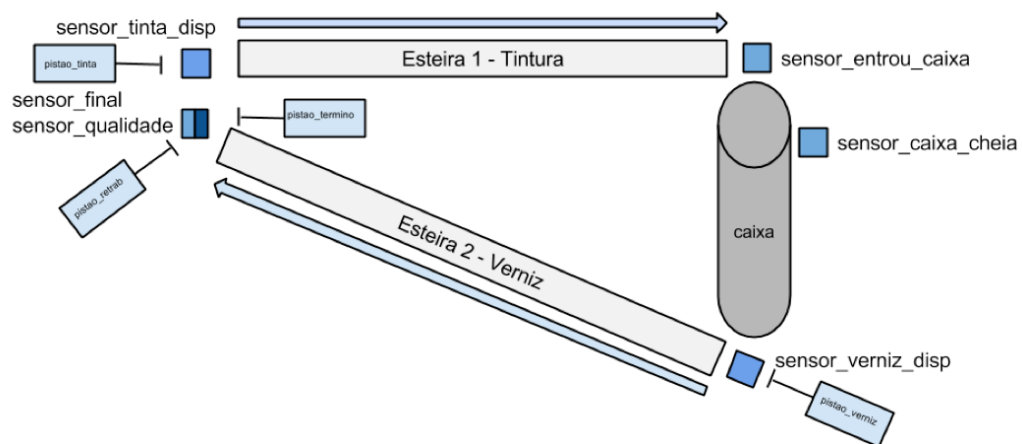


Figura 6.12: Processo de pintura e verniz

- O sensor\_tinta\_disp é um sensor na entrada do sistema que detecta se há uma peça disponível para ser pintada. Caso há peças nesse sensor, o pistao\_tinta empurra a mesma para a esteira1;
- A esteira1 possui borrifadores de tinta. A peça é pintada à medida que é levada para uma caixa de armazenamento de peças. Ao chegar no final da esteira, a peça entra na caixa, ativando o sensor\_entrou\_caixa;
- Quando há uma ou mais peças na caixa, o sensor\_verniz\_disp é ativado, fazendo com que o pistao\_verniz empurre a peça pintada para a esteira2;
- A esteira2 realiza a operação de envernizar as peças. É uma operação mais lenta em relação à esteira1;
- Quando a peça chega ao sensor\_final, um teste de qualidade é realizado. Se o teste for positivo, o pistao\_termino é acionado, expulsando a peça para um pátio de estocagem. Para o caso em que a qualidade da peça não é aprovada, ela deve ser pintada e envernizada novamente. O pistao\_retrab retorna a peça para a esteira1;

As seguintes especificações devem ser respeitadas:

- Propriedade 1: A esteira1 não pode funcionar se a caixa estiver cheia;

*assert property(!(motor\_esteira1 && caixa\_cheia));*

- Propriedade 2: O pistao\_tinta e o pistao\_retrab não podem empurrar peças para a esteira1 se a mesma estiver em movimento ou se a caixa estiver cheia;

*assert property(!(pistao\_tinta&&(motor\_esteira1 || caixa.full)));*

*assert property(!(pistao\_retrab&&(motor\_esteira1 || caixa.full)));*

- Propriedade 3: O pistao\_verniz não pode empurrar peças para a esteira2 se a mesma estiver em movimento ou se houver peças no sensor\_final;

*assert property(!(pistao\_verniz&&(motor\_esteira2 || sensor\_final)));*

- Propriedade 4: A prioridade é da peça que está no retrabalho;

*assert property(!(retrab &&pistao\_tinta));*

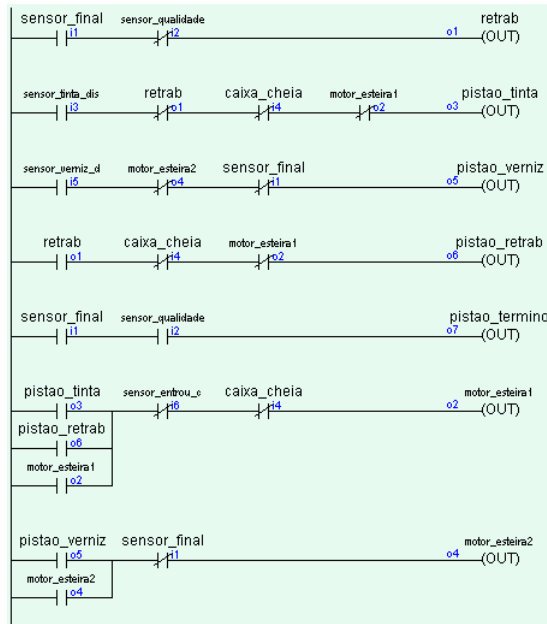


Figura 6.13: Diagrama Ladder para o controle do sistema

O diagrama Ladder da Figura 6.13 foi desenvolvido para automatizar o sistema.

O modelo comportamental utilizou duas instâncias de comportamento de esteira: uma para esteira de tinta e outra para esteira de verniz. Uma instância de comportamento de container também foi utilizada: para representar a caixa de peças. Para esse caso específico, o comprimento das esteiras é importante para a análise do sistema. Propositamente, foi feito com que o comprimento da esteira1 fosse 2 vezes menor do que o da esteira2. Essa decisão se deve ao fato de reproduzir um modelo de produtor/consumidor, onde o produtor é mais rápido que o consumidor. Como há possibilidade de retrabalho, há também a possibilidade de deadlock.

### 6.3.1 Comportamento das Esteiras

Na esteira1, representando a esteira de tinta, o sinal motor\_esteira1 foi conectado ao 'enable', pois ele é o responsável por fazer a esteira movimentar. A expressão (pistao\_tinta || pistao\_re trab) foi conectado ao 'load' da esteira, pois esses pistões são os responsáveis por inserir peças nessa esteira. Como a esteira joga peças numa caixa, sem a necessidade de um elemento que empurre as mesmas, o evento de retirar peças da esteira, conectado ao 'remove', é igual a '1'. Quando a peça chega ao final da esteira, o sensor\_entrou\_



caixa identifica que existe uma peça saindo da esteira. Por esse motivo, ele foi conectado ao 'sensor' da esteira. O comprimento da esteira foi arbitrado em três unidades, sendo que cada peça demora um ciclo para percorrer uma unidade.

Para a esteira2, representando a esteira de verniz, o sinal motor\_esteira2 foi conectado ao 'enable'. O pistao\_verniz foi conectado ao 'load'. A expressão (pistao\_retrab || pistao\_termino) foi conectado ao 'remove', pois esses são os pistões responsáveis por tirar peças da esteira. O sensor que indica que há peças no final da esteira é o sensor\_final, elemento que foi conectado ao 'sensor'. O comprimento utilizado foi de seis unidades, duas vezes maior do que o comprimento da esteira de pintura.

### 6.3.2 Comportamento da Caixa

A caixa foi conectada da seguinte maneira: O sensor\_entrou\_caixa foi conectado ao 'load', pois ele indica que entrou uma peça na caixa. O pistao\_verniz remove peças da caixa, por esse motivo ele foi conectado ao 'remove'. Quando há uma peça na caixa, o sensor\_verniz\_disp deve ser acionado, portanto ele foi ligado ao 'disp'. O sensor de caixa\_cheia foi conectado ao sinal 'full' da caixa, pois ele indica que a caixa está com capacidade máxima. A capacidade da caixa foi arbitrada para 7 elementos. Esse número foi escolhido para que o trace da prova formal não seja muito extenso, impossibilitando a publicação do mesmo em uma página.

### 6.3.3 Verificando a ocorrência de Deadlock

As Propriedades 1 a 4 foram provadas corretamente pelo JasperGold®, mostrando que o sistema funciona como o especificado. Apesar de estar dentro das especificações, nenhuma das propriedades verificadas impede que o sistema entre em deadlock. Analisando a estrutura da montagem, percebe-se que a planta atinge um estado em que o trabalho não está finalizado, porém não há qualquer possibilidade da continuação do seu funcionamento se houver uma ausência prolongado de acionamento dos sinais pistao\_termino ou pistao\_retrab. Foi considerado que o sistema estará em deadlock se ambos pistões permanecerem inativos por uma quantidades de ciclos superior ao necessário para percorrer todas as etapas do processo. Para o exemplo, essa quantidade corresponde a nove ciclos (três da esteira de pintura e seis da esteira de verniz). A esse intervalo característico é dado o nome de latência do sistema. A seguir, uma propriedade para verificar se o sistema pode atingir as condições de deadlock especificadas:

```
assert property((pistao_retrab || pistao_termino)[- > latencia]);
```

Em uma análise de alto nível, essa propriedade corresponde à pergunta: o pistao\_retrab ou o pistao\_termino serão acionados dentro de um intervalo igual a um período de latência do sistema? Caso o resultado seja negativo, há algum conjunto de estímulos válidos que podem levar ao deadlock. Quando a propriedade foi validada usando o JasperGold<sup>®</sup>, o resultado foi uma violação da mesma. O trace de um caso que leva o sistema a deadlock pode ser visto na Figura 6.14.

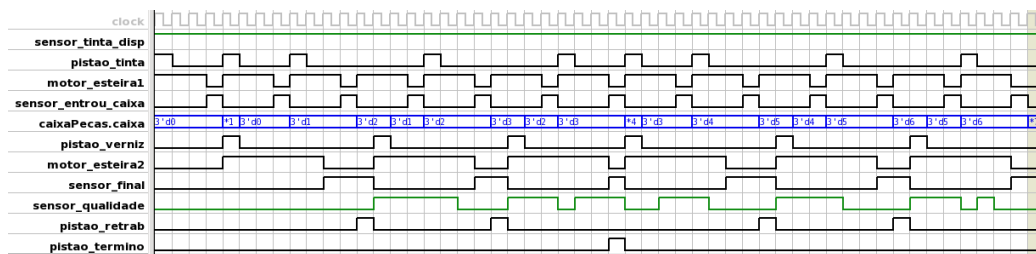


Figura 6.14: Trace de deadlock no sistema representado pela Figura 6.12

O Trace mostra que a caixa está se enchendo, enquanto ocorrem retrabalhos nas peças com qualidade reprovada. A Figura 6.15 representa um retrabalho ocorrendo. A peça em questão será re-enviada para a esteira de pintura. A caixa se enche, à medida que novas peças são pintadas e outras são retrabalhadas. Há um caso em que a qualidade da peça está boa, então a mesma é expulsa do sistema, fato demonstrado na Figura 6.16. O deadlock vai ocorrer no momento em que a caixa está cheia, porém há uma peça defeituosa que necessita ser retrabalhada. Nesse momento, o pistao\_retrab não pode empurrar a peça para a esteira1, pois a caixa está cheia e o pistao\_termino não pode expulsar a peça do sistema pois a mesma é defeituosa. A Figura 6.17 representa o sistema em deadlock.

O fluxo completo que possibilitou a obtenção dos modelos desse exemplo, pode ser observado no Capítulo 8.

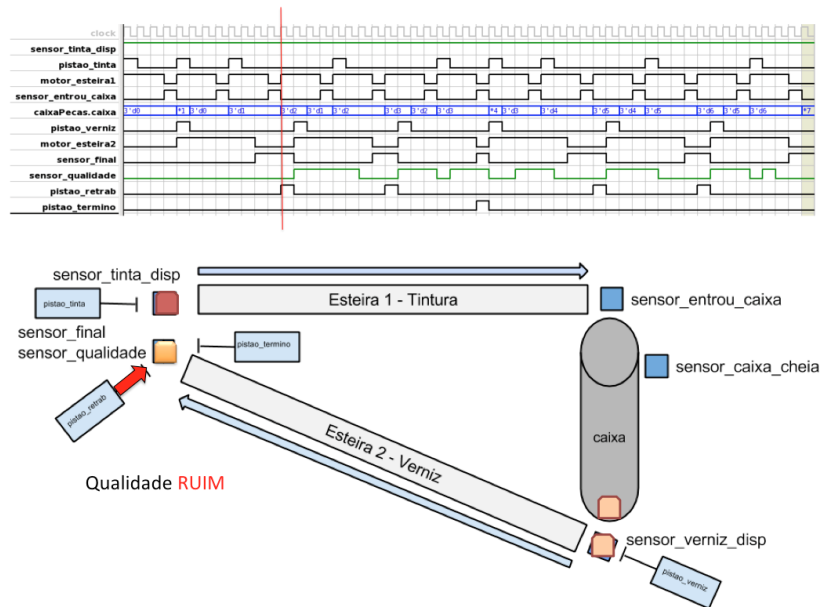


Figura 6.15: Um evento de retrabalho ocorrendo de acordo com o Trace

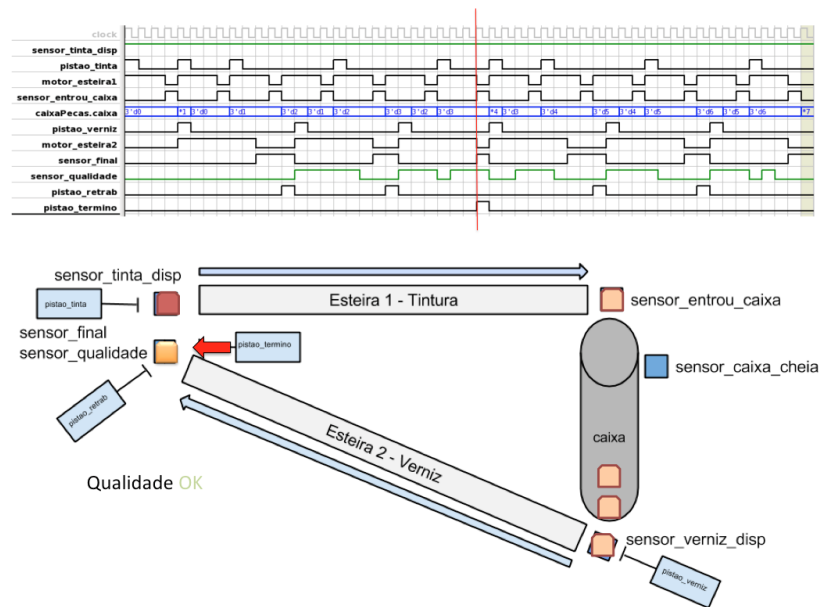


Figura 6.16: Um evento de expulsão de peça ocorrendo de acordo com o Trace

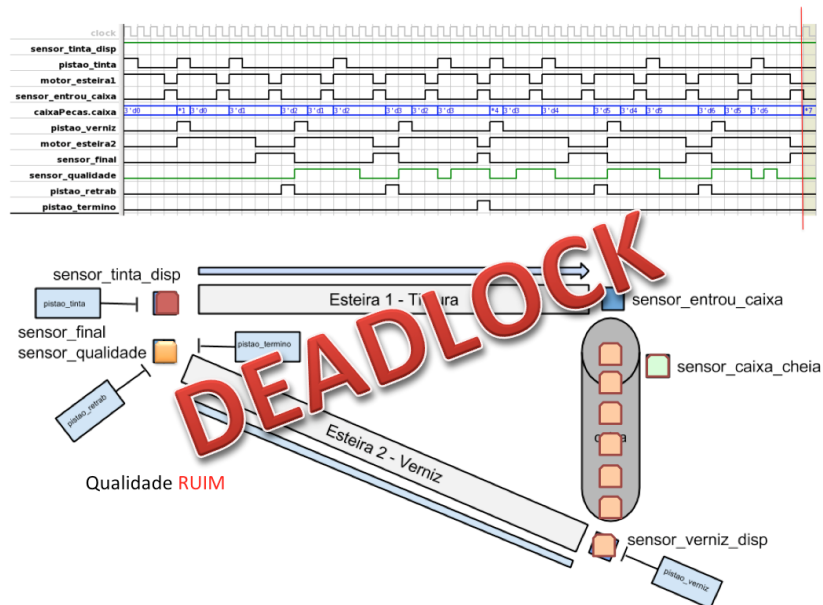


Figura 6.17: O deadlock ocorrendo de acordo com o Trace

# Capítulo 7

## Conclusões

### 7.1 Considerações Finais

O trabalho desenvolvido propôs uma nova maneira de se testar o funcionamento de sistemas controlados por diagramas Ladder. A verificação formal foi utilizada, possibilitando a detecção de falhas de maneira determinística e removendo a necessidade do teste aleatório nas entradas do sistema. Para que a verificação formal fosse aplicada, dois tipos de modelo foram criados: O modelo da lógica do diagrama Ladder e o modelo de comportamentos do sistema. Enquanto o primeiro é uma conversão direta da lógica programada no CLP, o segundo se mostrou um importante meio de se obter modelos comportamentais de qualquer tipo de sistema, seja ele controlado por um diagrama Ladder, um código em outras linguagens de programação ou até mesmo sistemas biológicos.

Os softwares de verificação formal são caros, no caso do JasperGold® uma licença custa em média US\$ 25.000 por ano [2]. O retorno de investimento, por sua vez, é alto, pois falhas críticas podem ser encontradas durante a etapa de verificação. Segurança e a garantia de que o sistema não irá parar (gerando prejuízos), são exemplos dos ganhos mostrados neste projeto com a utilização desse tipo de ferramenta.

### 7.2 Propostas de Continuidade

#### 7.2.1 Biblioteca de Comportamento de Equipamentos

O projeto se limitou a propor a implementação do comportamento de dois tipos básicos de equipamentos: esteira e container. Um possível trabalho futuro seria a implementação de uma biblioteca completa de equipamentos,

começando por equipamentos mais simples, encontrados em laboratórios de faculdades, até uma possível modelagem de equipamentos de ponta, utilizados na automação de processos reais.

### **7.2.2 Estudo sobre Especificações de Tempo**

Os modelos formais desenvolvidos no projeto são capazes de especificar apenas número de ciclos. Um estudo mais rebuscado pode ser realizado com o intuito de possibilitar uma conversão entre número de ciclos e o tempo real gasto pelas operações, de modo a possibilitar verificações baseadas em limitações de tempo.

### **7.2.3 Modelagem das Linguagens da IEC-61131-3**

O diagrama Ladder é apenas uma linguagem especificada pela IEC-61131-3, existem ainda: Lista de Instruções (IL), Texto Estruturado (ST), Diagrama de Fluxo de Blocos (FBD) e Grafos de Sequência (SFC). Um possível estudo posterior seria a modelagem de cada uma dessas linguagens para HDL.

### **7.2.4 Desenvolvimento de Interface Gráfica**

Criar uma interface gráfica que permita ao usuário a criação de propriedades formais, sem a necessidade do conhecimento da sintaxe das mesmas.

# Capítulo 8

## Anexos

### 8.1 Código do Parser

O Parser que transforma o arquivo exportado pelo i-TRiLOGI em código Verilog está documentado nesta seção. O código foi desenvolvido em C++, por tal motivo existem dois arquivos: o cabeçalho (.h), com as definições do Parser e a implementação (.cpp), com o código que efetua a conversão propriamente dita.

#### 8.1.1 Cabeçalho

O código representado na Figura 8.1 possui a definição das classes utilizadas no Parser. A classe Net é a representação de um contato e/ou bobina no diagrama Ladder. A classe Netlist possui os relacionamentos entre as Nets.

#### 8.1.2 Implementação

A implementação do Parser é descrito nas Figuras 8.2, 8.3, 8.4 e 8.5.

### 8.2 Códigos dos Modelos de Comportamento

#### 8.2.1 Container

A implementação em Verilog do modelo de comportamento do container pode ser observa na Figura 8.6.

### **8.2.2 Esteira**

A implementação em Verilog do modelo de comportamento da esteira pode ser observada na Figura 8.7.

## **8.3 Exemplo completo de transformação**

O Exemplo apresentado na Seção 6.3 será utilizado para demonstrar um exemplo completo da transformação. Ele foi escolhido por se tratar do exemplo mais completo.

### **8.3.1 Diagrama Ladder**

O diagrama Ladder desenvolvido para o controle do processo descrito na Seção 6.3 pode ser observado na Figura 8.8.

### **8.3.2 Formato i-TRiLOGI**

O diagrama Ladder da Figura 8.8 foi exportado para o formato texto. O resultado pode ser observado na Figura 8.9.

### **8.3.3 Verilog - Modelo do Diagrama Ladder**

O Parser foi utilizado para transformar o arquivo texto exportado pelo i-TRiLOGI em uma representação equivalente em Verilog. O resultado da transformação é mostrado na Figura 8.10. O modelo do Flip-Flop D utilizado na quebra de loop combinacional é descrito pela Figura 8.11.

### **8.3.4 Verilog - Modelo de Comportamentos**

A operação de ligar os pinos no modelo de comportamentos é realizada de maneira manual. O Verilog utilizado do modelo de comportamentos pode ser observado na Figura 8.12.

### **8.3.5 Verilog - Especificações**

A propriedade de verificação da ocorrência de deadlock, descrita em Verilog é observada na Figura 8.13.



```

#ifndef NETLIST_H
#define NETLIST_H

#include <iostream> // std::string
#include <vector>    // std::vector
#include <set>       // std::set
#include <map>       // std::map
#include <list>      // std::list

namespace Expression {

    class Net;

    typedef std::map<const std::string, const Net*> NameToVarMap;
    typedef std::set<const Net*> NetSet;
    typedef std::multimap<const Net*, const std::string> NetToExpressionMap;
    typedef std::list<std::string> ExpressionList;

    class Net {
    public:
        Net(const std::string& name);
        const std::string& getName() const { return m_name; }
        const std::string getReconvName() const { return m_name + "_in"; }

    private:
        const std::string m_name;
    };

    class NetList {
    public:
        // Cria a NetList a partir do arquivo passado como parâmetro.
        NetList(const char* fileName);
        ~NetList();

        const NetToExpressionMap& getNetToExprMap() const { return m_netToExpr; }
        // Todas as Nets
        const NetSet& getAllNets() const { return m_allNets; }
        // Sao as nets que levam uma atribuicao (bobinas)
        const NetSet& getAttNets() const { return m_attNets; }
        // Sao as nets que possuem reconvergencia
        const NetSet& getReconvNets() const { return m_reconvNets; }
        // Sao as nets de entrada (switches)
        const NetSet& getInputNets() const { return m_inputNets; }
        // Sao as linhas de expressao
        const ExpressionList& getExprLines() const { return m_exprLines; }

    private:
        // Registra a Net e retorna o ponteiro para a net registrada, de nome netName
        const Net* registerNet(const std::string& netName,
                               bool& newNet);

        // Metodo para fazer o parse do arquivo simples criado manualmente
        // para debug
        void parseDummyFile(const char* fileName);

        // Metodo para fazer o parse do arquivo exportado do Trilogi Ladder
        void parseTrilogiFile(const char* fileName);

        // Metodo que faz o parse de uma linha (atribuicao + expressao)
        void parseLine(const std::string& line);

        // Metodo que faz o parse de uma expressao
        void parseExpression(const std::string& expression);

        // Metodo que converte a expressao do Trilogi para Verilog
        const std::string convertToVerilogExpression(const std::string& ladExpression) const;

        // Mapa para acessar a Net a partir de seu nome. Toda net
        // visivel criada deve ser registrada neste mapa.
        NameToVarMap m_netToVar;

        // Estrutura com as Nets de atribuicao. Utilizada para calcular as reconvergencias
        NetSet m_attNets;

        // Estrutura com as Nets de entrada apenas
        NetSet m_inputNets;

        // Estrutura com todas as Nets
        NetSet m_allNets;

        // Estrutura com as Nets reconvergentes
        NetSet m_reconvNets;

        // Estrutura com as Expressoes de cada atribuicao
        NetToExpressionMap m_netToExpr;

        // Estrutura que salva a linha de cada expressão
        ExpressionList m_exprLines;
    };
};
#endif // NETLIST_H

```

Figura 8.1: Cabeçalho do Parser em C++

```

#include "netlist.h"
#include <fstream>
#include <assert.h>

#define DEBUG_PARSER(code) if (0) {std::cerr << code << std::endl;}

using namespace Expression;

Net::Net(const std::string& name) : m_name(name)
{}

NetList::NetList(const char* fileName) {
    // Faz o parse do trilogi file
    parseTrilogiFile(fileName);
}

NetList::~NetList() {
    // Libera memoria alocada pelas nets
    for (NameToVarMap::iterator it = m_netToVar.begin();
         it != m_netToVar.end(); ++it) {
        delete it->second;
    }
}

const Net*
NetList::registerNet(const std::string& netName, bool& newNet) {
    const Net* currNet = NULL;
    NameToVarMap::const_iterator find = m_netToVar.find(netName);
    if (find != m_netToVar.end()) {
        currNet = find->second;
        newNet = false;
    } else {
        currNet = new Net(netName);
        m_netToVar.insert(std::make_pair(netName, currNet));
        m_allNets.insert(currNet);
        newNet = true;
    }
    return currNet;
}

```

Figura 8.2: Implementação Parser - parte 1

```

namespace {
    const char negOp()    { return '!'; }
    const char andOp()    { return '&'; }
    const char orOp()     { return '|'; }
    const char* attOp()   { return "<="; }
    size_t     attOpSz()  { return std::string(attOp()).size(); }

    bool isOr(const std::string& str) {
        assert (str.size() == 1);
        return (str[0] == orOp());
    }
    bool isAnd(const std::string& str) {
        assert (str.size() == 1);
        return (str[0] == andOp());
    }
    bool isNeg(const std::string& str) {
        // Neg operator comes attached to a signal name:
        // !SigName. Do NOT assert the size
        // assert (str.size() == 1);
        return str[0] == negOp();
    }
    bool isOperator(const char op) {
        return (op == negOp() || op == andOp() || op == orOp());
    }
    bool isSyntax(const char op) {
        return (op == '(' || op == ')' || op == ' ');
    }
}

void
NetList::parseExpression(const std::string& expression) {
    std::string currVar;
    for (std::string::const_iterator it = expression.begin();
         it != expression.end(); ++it) {

        if (!isOperator(*it) && !isSyntax(*it)) {
            currVar += *it;
        } else if (!currVar.empty()) {
            DEBUG_PARSER("currVar = /" << currVar << "/");
            bool isNew = true;
            const Net* pNet = registerNet(currVar, isNew);
            if (!isNew) {
                if (m_attNets.find(pNet) != m_attNets.end()) {
                    m_reconvNets.insert(pNet);
                }
            } else {
                if (m_attNets.find(pNet) == m_attNets.end()) {
                    m_inputNets.insert(pNet);
                }
            }
            currVar.clear();
        }
    }
}

```

Figura 8.3: Implementação Parser - parte 2

```

void
NetList::parseLine(const std::string& line) {

    // Primeiramente salva a variavel de atribuicao
    size_t att = line.find(attOp());
    if (att != std::string::npos) {
        const std::string varAtt = line.substr(0, att-1);
        DEBUG_PARSER("varAtt = /" << varAtt << "/");

        bool isNew = true;
        const Net* attNet = registerNet(varAtt, isNew);

        // Registra a net como sendo de atribuicao
        if (isNew) {
            m_attNets.insert(attNet);
        }

        // Agora faz o parse da expressao a qual a variavel foi atribuida
        const std::string expression = line.substr(att + attOpSz() + 1, std::string::npos);

        DEBUG_PARSER("expression = /" << expression << "/");

        // Pega as Nets da expressao e registra as mesmas
        parseExpression(expression);
        m_netToExpr.insert(std::make_pair(attNet, expression));
        const std::string fullExpr = attNet->getName() + " = " + expression;
        m_exprLines.push_back(fullExpr);
    }
}

void
NetList::parseDummyFile(const char* fileName) {
    std::ifstream inputFile;
    inputFile.open(fileName, std::ifstream::in);

    // Start reading the file
    while (inputFile.good()) {
        std::string line;
        std::getline(inputFile, line);
        parseLine(line);
    }

    inputFile.close();
}

void
NetList::parseTrilogiFile(const char* fileName) {
    std::ifstream inputFile;
    inputFile.open(fileName, std::ifstream::in);

    while (inputFile.good()) {
        std::string line;
        std::getline(inputFile, line);
        if (line.find("=") != std::string::npos) {
            const std::string verilogExpr = convertToVerilogExpression(line);
            parseLine(verilogExpr);
        }
    }
}

```

Figura 8.4: Implementação Parser - parte 3

```

const std::string
NetList::convertToVerilogExpression(const std::string& ladExpr) const {
    std::string converted;
    for (std::string::const_iterator it = ladExpr.begin();
         it != ladExpr.end(); ++it) {

        if (*it == '*') {
            converted.append(" & ");
        } else if (*it == '+') {
            converted.append(" | ");
        } else if (*it == '/') {
            converted.append("/");
        } else if (*it == '=') {
            converted.append(" <= ");
        } else if ((int)*it != 0) {
            converted.append(1, *it);
        }
    }

    DEBUG_PARSER("converted expr = /" << converted << "/");
    return converted;
}
}

```

Figura 8.5: Implementação Parser - parte 4

```

// Descreve o comportamento de um container que se comporta como uma caixa
// A caixa possui um sensor que indica que existe elementos dentro da mesma
// e outro sensor que indica se ela está completamente cheia
module caixa(input clock, // Sincroniza o sistema
             input reset, // Define o estado inicial da caixa (Vazia)
             input insert, // Evento de inserção de itens na caixa
             input remove, // Evento de remoção de itens na caixa
             output disp, // Sensor indicando que ha itens na caixa
             output full, // Sensor indicando que a caixa esta cheia
             output estado_caixa); // Interface para acessar o estado da caixa

// A quantidade de itens que podem ser inseridos na caixa
parameter size = 3;
// Array indicando o numero de itens da caixa;
reg [size:0] caixa;
always@(posedge(clock) or posedge(reset)) begin
    // Caixa inicialmente vazia
    if (reset) begin
        caixa <= 0;
    // Evento de inserção: incrementa um item da caixa
    end else if ($rose(insert) & !$rose(remove)) begin
        caixa <= caixa + 1;
    // Evento de remoção: remove um item da caixa
    end else if ($rose(remove) & !$rose(insert)) begin
        caixa <= caixa - 1;
    end
end
// Reduce & para verificar que a caixa está cheia (todos os bits do array
// são iguais a 1)
assign full = (& caixa);
// Liga a interface
assign estado_caixa = caixa;
// Item disponivel na caixa
assign disp = (caixa != 0);

endmodule

```

Figura 8.6: Implementação da Caixa

```

// Descreve o comportamento de um container que se comporta como uma esteira
// A esteira é representada por um array de bits. Quando o bit = 1, significa
// que existe uma peça naquela posicao da esteira. Este array tera apenas um
// bit por posicao, ou sera vazio (onehot0):
// [000]->(load)->[100]->[010]->[001]->(remove)->[000]
module esteira(input clock, // Sincroniza o sistema
               input reset, // Define estado inicial (vazio)
               input enable, // Mantém a esteira funcionando (faz com que a peça se movimente)
               input load, // Insere uma peça na esteira [100]
               input remove, // Remove a peça quando ela estiver na ultima posicao da esteira
               output sensor); // Sensor que indica que o item chegou ao final da esteira
//output estado_esteira; // Interface para acessar o estado atual da esteira

// A extensão da esteira (numero de ciclos que leva para o item entrar e sair)
parameter tamanho_esteira = 4;
// Array indicando o estado atual da esteira
reg [tamanho_esteira-1:0] esteira;
always@(posedge(clock) or posedge(reset)) begin
    // Define estado inicial da esteira. Ela sera resetada no primeiro ciclo de funcionamento
    // do sistema ou se existe uma peça na ultima posicao e um evento de remocao é chamado
    if (reset || remove & sensor) begin
        esteira <= 0;
    // Enquanto o motor funciona, faz o shift do bit do array para simular a movimentacao
    // da esteira
    end else if (enable & !sensor) begin
        esteira <= (esteira >> 1);

        // Insere o item na primeira posicao da esteira
    end else if (load && !enable) begin
        esteira[tamanho_esteira-1] <= 1;
    end
end
// Peça chegou ao final da esteira
assign sensor = esteira[0] == 1;
// Liga a interface
//assign estado_esteira = esteira;
endmodule

```

Figura 8.7: Implementação da Esteira

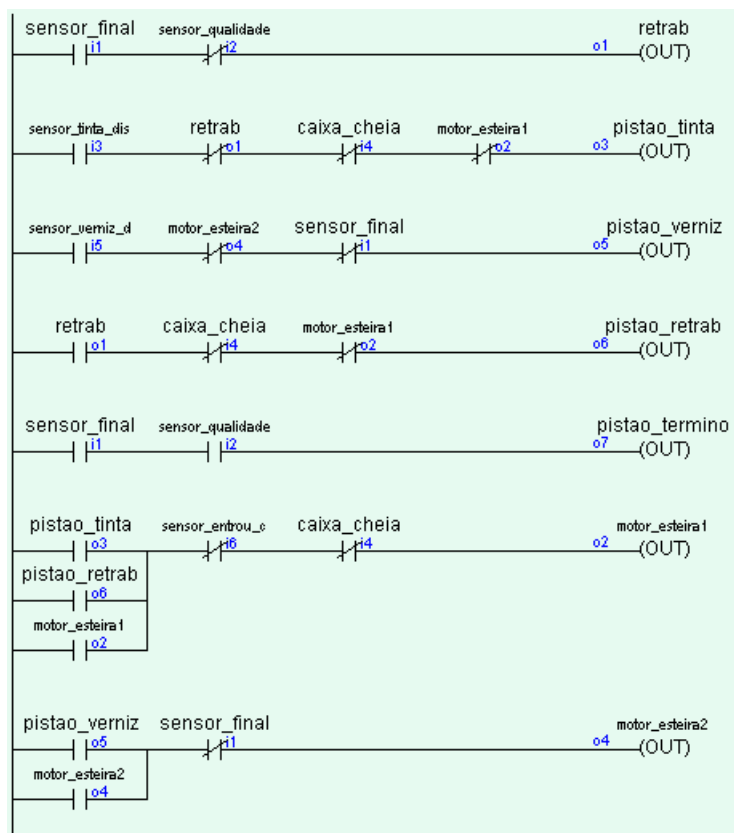


Figura 8.8: Diagrama Ladder para controle do pátio de pintura

```

00TRILOGI Ver 5.00,sensor_final
1,sensor_qualidade
2,sensor_tinta_dis
3,caixa_cheia
4,sensor_verniz_d
5,sensor_entrou_c
~
0,retrab
1,motor_esteira1
2,pistao_tinta
3,motor_esteira2
4,pistao_verniz
5,pistao_retrab
6,pistao_termino
~
~
~
retrab=sensor_final*/sensor_qualidade
pistao_tinta=sensor_tinta_dis*/retrab*/caixa_cheia*/motor_esteira1
pistao_verniz=sensor_verniz_d*/motor_esteira2*/sensor_final
pistao_retrab=retrab*/caixa_cheia*/motor_esteira1
pistao_termino=sensor_final*sensor_qualidade
motor_esteira1=(pistao_tinta+(pistao_retrab+motor_esteira1))*/sensor_entrou_c*/caixa_cheia
motor_esteira2=(pistao_verniz+motor_esteira2)*/sensor_final
~END_CIRCUIT~
~END_CUSTFN~
~END_CUSTFNLABEL~
0,0,
1,0,
2,0,
3,0,
4,0,
~END_QUICKTAGS~
~END_DEFINES~
~END_BREAKPOINTS~

```

Figura 8.9: Diagrama Ladder da Figura 8.8 em formato texto



```

module top(input clock, reset);

    // INICIO MODELO DO DIAGRAMA LADDER //

    // Sensores
    logic sensor_tinta_disp,
        sensor_entrou_caixa,
        sensor_verniz_disp,
        sensor_qualidade,
        sensor_final,
        caixa_cheia;

    // Bobinas sem realimentacao
    logic pistao_tinta,
        pistao_verniz,
        pistao_termino,
        pistao_retrab ;

    // Variaveis internas
    logic retrab;

    // Bobinas com realimentacao
    logic motor_esteiral, motor_esteiral_in,
        motor_esteira2, motor_esteira2_in;

    assign retrab      = sensor_final      & ~sensor_qualidade;
    assign pistao_tinta = sensor_tinta_disp & ~retrab      & ~caixa_cheia & ~motor_esteiral_in;
    assign pistao_verniz = sensor_verniz_disp & ~motor_esteira2_in & ~sensor_final;
    assign pistao_retrab = retrab          & ~caixa_cheia & ~motor_esteiral_in;
    assign pistao_termino = sensor_final    & sensor_qualidade;

    assign motor_esteiral = (pistao_tinta || pistao_retrab || motor_esteiral_in) & ~sensor_entrou_caixa & ~caixa_cheia;
    assign motor_esteira2 = (pistao_verniz || motor_esteira2_in) & ~sensor_final;

    linha l1(motor_esteiral, clock, reset, 0, motor_esteiral_in);
    linha l2(motor_esteira2, clock, reset, 0, motor_esteira2_in);

    // FIM MODELO DO DIAGRAMA LADDER //

```

Figura 8.10: Modelo do Diagrama Ladder em Verilog

```

// Modelo D Latch
module linha(data_in, // Dados
    clock, // Clock
    reset, // Reset
    reset_value, // Valor de Reset
    data_out); // Saida do Latch

    input data_in;
    input clock;
    input reset;
    input reset_value;

    output data_out;
    reg data_out;

    // Flop Assincrono
    always@(posedge(clock) or posedge(reset)) begin
        data_out <= (reset) ? reset_value : data_in;
    end

endmodule

```

Figura 8.11: Flip-Flop D utilizado na quebra de loop combinacional

```

//          ...          //
// INICIO PROPRIEDADES DO SISTEMA //

// Variaveis referentes ao tempo gasto pela esteira
parameter tempo_esteiral = 3;
parameter tempo_esteira2 = 6;
parameter tamanho_caixa1 = 2;

// Variaveis internas para auxiliar na visualizacao das propriedades
logic [tempo_esteiral:0] esteiral;
logic [tempo_esteira2:0] esteira2;
logic [tamanho_caixa1:0] caixa1;

// Conecta a instrumentação na caixa
caixa #(tamanho_caixa1) caixaPecas(clock,
                                   reset,
                                   sensor_entrou_caixa,
                                   pistao_verniz,
                                   sensor_verniz_disp,
                                   caixa_cheia);

// Conecta a instrumentação na esteira de tinta
esteira #(tempo_esteiral) esteiraTinta(clock,
                                       reset,
                                       motor_esteiral,
                                       (pistao_tinta | pistao_retrab),
                                       1,
                                       sensor_entrou_caixa,
                                       esteiral);

// Conecta a instrumentacao na esteira de verniz
esteira #(tempo_esteira2) esteiraVerniz(clock,
                                       reset,
                                       motor_esteira2,
                                       pistao_verniz,
                                       (pistao_retrab | pistao_termino),
                                       sensor_final,
                                       esteira2);

```

Figura 8.12: Ligação do Modelo de Comportamentos

```

// 0 sensor de qualidade nao pode mudar de valor para uma mesma peca
assume property (@(posedge clock) sensor_final |-> $stable(sensor_qualidade));

// Sempre haverá tinta disponível
assume property (sensor_tinta_disp);

// Verifica a propriedade de deadlock - se ambos os pistoes nao ativarem depois de uma
// latencia do sistema, entao o sistema esta em deadlock
assert property (caixa_cheia |-> (pistao_retrab | pistao_termino)[->tempo_esteiral + tempo_esteiral];

// FIM PROPRIEDADES DO SISTEMA    //
endmodule

```

Figura 8.13: Propriedade de Deadlock

# Referências Bibliográficas

- [1] Altera. Tcl scripting. Disponível em [http://www.altera.com/literature/hb/qts/qts\\_qii52003.pdf](http://www.altera.com/literature/hb/qts/qts_qii52003.pdf), August 2014.
- [2] Jasper Design Automation. Jaspergold. Disponível em <http://www.jasperda.com/>, July 2014.
- [3] Programmable Logic Controller. The New Interconversion Algorithm between Ladder Diagram and Instruction List LIDandan Qixian Tree , forest The convert ideas o ./ ladder diagram to instmction list. pages 820–823.
- [4] Prof. Sujit Dey. Vlsi advanced topics - combinational loop example. Disponível em [http://mesdat.ucsd.edu/wshaoxuan/ece260S10/combinational\\_loop.html](http://mesdat.ucsd.edu/wshaoxuan/ece260S10/combinational_loop.html), October 2014.
- [5] Philip R. Moorby Donald E. Thomas. *The Verilog Hardware Description Language - Fifth Edition*. Kluwer Academic Publishers, 101 Philip Drive, Assinippi Park, Norwell, Massachusetts 02061 USA, 2002.
- [6] Khizar A. Khan Faisal I. Haque, Jonathan Michelson. *The Art of Verification with System Verilog Assertions*. Verification Central, 5178 Mowry Ave, 2137, Fremont, CA 94538, 2006.
- [7] Marc Constantijn Willem Geilen. Formal techniques for verification of complex real-time systems. Disponível em <http://www.es.ele.tue.nl/mgeilen/publications/thesis.pdf>, December 2014.
- [8] David J. Lacey Harry D. Foster, Adam C. Krolnik. *Assertion-Based Design 2nd Edition*. Kluwer Academic Publishers, 101 Philip Drive, Assinippi Park, Norwell, Massachusetts 02061 USA, 2004.
- [9] iTRiLOGI. [www.triplc.com/trilogi.htm](http://www.triplc.com/trilogi.htm). Disponível em [www.triplc.com/trilogi.htm](http://www.triplc.com/trilogi.htm), May 2014.
- [10] Cory Janssen. C++ programming language. Disponível em <http://www.techopedia.com/definition/26184/c-programming-language>, August 2014.

- [11] Karl-Heinz John and Michael Tiegelkamp. *IEC 61131-3 : programming industrial automation systems : concepts and programming languages, requirements for programming systems, aids to decision-making tools*. Springer-Verlag, Berlin, Heidelberg, 2001.
- [12] Prof. Luiz Themystoklitz Santos Mendes. 6a aula - programação convencional de clp. Disponível em [http://www.cpdee.ufmg.br/~luizt/lab\\_inf\\_ind/aula06.pdf](http://www.cpdee.ufmg.br/~luizt/lab_inf_ind/aula06.pdf), October 2014.
- [13] R.S. Mitra. Strategies for mainstream usage of formal verification. *2008 45th ACM/IEEE Design Automation Conference*, pages 800–805, 2008.
- [14] Elthon Alex Da Silva Oliveira, Leandro Dias Da Silva, Kyller Gorgonio, Angelo Perkusich, and Aldenor Falcao Martins. Obtaining formal models from Ladder diagrams. *2011 9th IEEE International Conference on Industrial Informatics*, pages 796–801, 2011.
- [15] Eng. Alexandre Bernadino Prof. Paulo Jorge Oliveira. Linguagens de programação de plc's: Ladder diagrams. Disponível em [http://users.isr.ist.utl.pt/~pjcro/cadeiras/api0203/pdf\\_files/api\\_c3\\_ld\\_1.pdf](http://users.isr.ist.utl.pt/~pjcro/cadeiras/api0203/pdf_files/api_c3_ld_1.pdf), July 2014.
- [16] Saurabh Verma Sanjay Churiwala, Chandan Kumar. Exploring the types of combinational loops. Disponível em [http://www.eetindia.co.in/STATIC/PDF/201003/EEIOL\\_2010MAR18\\_-SIG\\_TA\\_01.pdf?SOURCES=DOWNLOAD](http://www.eetindia.co.in/STATIC/PDF/201003/EEIOL_2010MAR18_-SIG_TA_01.pdf?SOURCES=DOWNLOAD), October 2014.
- [17] Constantino Seixas. Iec 61131-3 ladder: Suporte de curso. Disponível em <http://www.cpdee.ufmg.br/~seixas/PaginaII/Download/DownloadFiles/IEC1131Ladder.PDF>, July 2014.
- [18] O. De Smet and O. Rossi. Verification of a controller for a flexible manufacturing line written in Ladder Diagram via model-checking. *Proceedings of the 2002 American Control Conference (IEEE Cat. No.CH37301)*, 5:47–52, 2002.
- [19] Yunfeng Tao Yunfeng Tao. An introduction to assertion-based verification. *2009 IEEE 8th International Conference on ASIC*, pages 1–6, 2009.
- [20] Frank Vahid. *Sistemas Digitais: Projeto, Otimização e HDLs*. Artmed, Av. Jerônimo de Ornelas, 670 - Santana, 2008.
- [21] Fernando Vanini. Engenharia de software: Conceitos básicos de verificação formal. Disponível em <http://www.ic.unicamp.br/~vanini/MC747/Verifica2014>.