Universidade Federal de Minas Gerais
Escola de Engenharia
Mestrado em Engenharia Elétrica

Verificação Automática de Sistemas Descritos Usando a Linguagem Ladder

Alba Francine de Souza Caetano

Trabalho apresentado como requisito parcial para obtenção do título de mestre em Engenharia Elétrica.

Orientador: Prof. Dr. Alair Dias Júnior

Belo Horizonte, Dezembro de 2018

Dedico esta dissertação aos meus preciosos pais, Maria Antéria e José Ademir, por terem tão sabiamente me ensinado a confiar em meus passos e a construir a minha história.

Agradecimentos

Eis que a tí elevo, Senhor, por todas as graças que de vós recebo. Meu agradecimento à Deus, no que tange esta dissertação, é infinito. Agradeço por todos os momentos de superação, por todos os desafios, sabedoria e dissernimento. Agradeço por ter me conduzido e edificado ao longo desta caminhada, minha fé em tí é inabalável.

Sinto-me profundamente agradecida aos meus pais, Maria Antéria e José Ademir, pelos ensinamentos, cumplicidade e por todo amor que de vocês emana, amor este que me fortalece, por vocês sou e serei eternamente grata. Em especial agradeço à minha mãe, que pacientemente me escutou, orou e acompanhou os meus passos, vocês são meu bem mais precioso. Agradeço aos meus queridos irmãos, June e Marcone, por serem meus parceiros para a vida. Agradeço à June por se fazer sempre tão presente, ser tão amorosa e companheira e ao Marcone por ser meu exemplo de perseverança e sabedoria. Vocês quatro são meus maiores incentivadores, são a família que Deus, tão bondosamente, me contemplou.

Agradeço aos meus avós, Nilza, Geraldo e Dede por todo o carinho e amor que tem por mim, por me proporcionarem palavras de conforto e por serem tão especiais. De coração agradeço à toda minha família, especialmente à Margareth, Ricardo e Matheus por serem pessoas tão maravilhosas e estarem sempre presentes em minha vida ao longo do desenvolvimento deste trabalho.

Agradeço ao meu orientador Alair, que mesmo à distância se propôs a me orientar, sempre com muito profissionalismo e grande sabedoria. Sinto-me muito honrada por ter compartilhado de seu conhecimento e por ter me acompanhado e conduzido ao longo do desenvolvimento desta dissertação.

Agradeço à Amanda que foi uma irmã em Belo Horizonte, pela amizade, cumplicidade e carinho, você foi um presente que o mestrado me proporcionou. Aos meus amigos da vida Larissa, Ana, Amanda Machado e Viviane por se fazerem presentes, independente da distância. Aos amigos e colegas da UFMG e do Laboratório de Engenharia

Biomédica, por me acolherem, em especial à Marina e o Manoel.

Muito obrigada à IHM Engenharia, pelo apoio concedido, que foi de fundamental importância para o desenvolvimento deste trabalho.

Agradeço aos funcionários do Colegiado e da Secretaria do Programa de Pós-Graduação em Engenharia Elétrica da UFMG (PPGEE), pela disponibilidade e gentileza. A todos, meus sinceros agradecimentos.

Resumo

Os Controladores Lógicos Programáveis (CLP) foram introduzidos na década de 1960 para realizar a modernização e manutenção do controle, sequenciamento e lógica de intertravamento das plantas industriais e, desde então, tornaram-se indispensáveis na automação e controle industrial. Um CLP é um computador digital construído para operar em ambientes industriais adversos e projetado para ser facilmente operado por equipes de manutenção. Por isso, oferece um conjunto de linguagens de programação direcionadas a esses profissionais. Uma dessas linguagens é a *Ladder Diagram* (LD), que é baseada na lógica de relés e ainda é uma das linguagens de programação de CLP mais usadas. A verificação dos programas de CLP é imprescindível para garantir a operação segura de plantas industriais e vários trabalhos abordaram esse assunto na literatura. No entanto, os métodos disponíveis para verificação de LD usualmente empregam uma abordagem de modelagem ingênua, não levando em conta a ordem de execução das *rungs*, que são linhas de programa em LD. Além disso, a maioria deles se concentra apenas nas operações booleanas, que são um subconjunto dos possíveis comandos disponíveis no LD. Este trabalho aborda esses dois problemas, fornecendo uma metodologia completa para traduzir automaticamente programas em Ladder para o NuSMV, sendo esta uma linguagem formal. A metodologia foi validada com relação à ordem de execução usando exemplos construídos e também construiu-se uma ferramenta para traduzir programas em LD exportado de um CLP amplamente utilizado na indústria para o NuSMV.

Abstract

Programmable Logic Controllers (PLC) were introduced in the 1960's to support modernization and maintenance of control, sequencing and interlocking logic of industrial plants and, since then, they became indispensable in industrial automation and control. A PLC is a digital computer built to operate in harsh industrial environments and designed to be easily operated by maintenance teams. Hence, it offers a set of programming languages directed to these professionals. One of such languages is the Ladder Diagram (LD), which is based on relay logic and still is one of the most used PLC programming languages. Verifying PLC programs is imperative to guaranteeing the safe operation of industrial plants and several works have addressed this issue in the literature. However, available methods for verification of LD usually employs a naive modelling approach, not taking into account the execution order of the rungs, which are program lines in LD. Besides, most of them focus only on Boolean operations, which are a subset of the possible commands available in LD. This work addresses both these issues, providing a complete methodology for translating Ladder Diagrams to NuSMV, which is the formal language. We validated the methodology with respect to execution order using constructed examples and also built a tool to translate LD exported from a PLC suite largely used in industry to NuSMV.

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

ANSI *American National Standards Institute*

BDD *Binary Decision Diagram*

BNF *Backus-Naur Form*

CFC *Continuous Function Chart*

CFG *Context-Free Grammar*

CLP *Controlador Lógico Programável*

CTL *Computation Tree Logic*

DOM *Document Object Model*

FDB *Function Block Diagram*

IEC *International Eletrotechnical Comission*

IEEE *Institute of Electrical and Electronics Engineers*

IL *Instruction List*

LD *Ladder Diagram*

LP *Lógica Proposicional*

LTL *Linear Temporal Logic*

NA *Normalmente Aberto*

NF *Normalmente Fechado*

NuSMV*New Symbolic Model Verifier*

SAT *Boolean Satisfiability Problem*

SMV *Symbolic Model Verifier*

SFC *Sequential Function Chart*

SoC *System-on-Chip*

ST *Structured Text*

VSIDS *Variable State Independent Decaying Sum*

XML *Extensible Markup Language*

Sumário

Capítulo 1

Introdução

Os Controladores Lógicos Programáveis (CLP) estão presentes em uma ampla gama de aplicações na indústria. Esses são, atualmente, indispensáveis no contexto industrial desempenhando inúmeras funções como, a automatização e controle no desenvolvimento e na manutenção de lógicas de sistemas nos quais são implementados. Desde o advento dos CLPs, ao final da década de 60, muitas linguagens têm sido utilizadas para definir programas para máquinas e processos, sendo essas linguagens regulamentadas pela IEC (International Electrotecnical Comission) número 61131-3 (International Electrotechnical Commission, 2013).

A norma IEC 61131 abrange vários aspectos dos controladores lógicos programáveis e é dividida em 9 partes, sendo que uma décima parte ainda está sob elaboração. Cada uma das partes trata de aspectos dos CLPs, desde informações gerais até comunicações e segurança funcional. Em especial, a terceira parte da norma, IEC 61131-3, trata das linguagens de programação do CLP e define seis linguagens: o diagrama ladder (LD – *Ladder Diagram*), o diagrama de blocos de função (FBD - *Function Block Diagram*), texto estruturado (ST - *Structured text*), lista de instruções (IL - *Instruction List*), diagrama de funções sequenciais (SFC - *Sequential Function Chart*) e diagrama de funções contínuas (CFC - *Continuous Function Chart*).

Apesar do ambiente de *design* bem desenvolvido de programas de CLPs, não é incomum que o comportamento do programa implementado na prática, difira do definido pela especificação. Testes podem não alcançar condições de erro durante a execução, e assim, disparidades entre o comportamento previsto e o real podem não ser identificados. Geralmente, a verificação de programas de CLPs é realizada por meio de testes, com vários estímulos de entrada sendo aplicados ao sistema. Porém, verificar por completo

esse tipo de sistema por meio de testes torna-se inviável, levando-se em conta que o número de combinações de entradas para se verificar pode ser elevado. Como muitas vezes esses dispositivos são responsáveis pela automação e controle de processos perigosos e delicados, garantir a corretude do programa deveria ser considerado um ponto chave no fluxo de projeto. No entanto, o que se vê na maior parte dos casos é que a validação do sistema é relegada ao segundo plano e realizada ao fim do projeto como uma atividade adicional, em vez de uma atividade essencial.

Na indústria de microeletrônica, em contrapartida, a verificação e validação do sistema são tratadas como pontos essenciais para o sucesso de um projeto. É fato conhecido que no desenvolvimento de um SoC (*System-on-Chip*) moderno, mais de 70% dos recursos disponíveis são gastos para garantir o funcionamento dentro das especificações (RANJAN; SKALBERG, 2009). Uma consequência natural, portanto, seria adaptar os modelos, técnicas e ferramentas da indústria de microeletrônica para o projeto e verificação formal de sistemas baseados em CLPs. Tal adaptação deve levar em consideração as diferenças entre as duas indústrias, em especial os recursos disponíveis, além da formação e do perfil dos profissionais envolvidos no projeto dos sistemas de automação.

Nesse contexto, verificar programas de CLPs configura-se em uma importante ferramenta que elevará o nível de confiabilidade dos sistemas aos quais será implementado. É interessante ressaltar que a verificação formal automática do sistema, reduzirá consideravelmente o tempo para finalização do projeto e, ao mesmo tempo, reduzirá a probabilidade de falhas no sistema, assegurando que o mesmo atende aos requisitos funcionais e não funcionais especificados (FISHER, 2007).

Desenvolver um método que realize a verificação formal automática de sistemas descritos de CLP é um problema de grande relevância, considerando que, a utilização de testes como meio de verificação de sistemas é, ainda, a abordagem mais adotada.

1.1 Formulação do Problema

Dado o contexto apresentado na seção anterior, pode-se formular uma pergunta norteadora para este trabalho:

Como realizar a verificação formal de programas escritos em Ladder?

A verificação formal de programas de CLPs permite ao usuário do sistema veri-

ficar dado programa quando fornecidas as propriedades de funcionamento deste. Objetiva-se, desta forma, promover uma redução de erros de programação que atingem a etapa de implantação da solução, elevando assim sua confiabilidade. Este é o problema a ser respondido neste trabalho.

1.2 Objetivos

O objetivo principal desta dissertação de mestrado é desenvolver um método e uma ferramenta que permitam a verificação formal de sistemas descritos em linguagem Ladder da norma IEC 61131-3. Dado este objetivo principal, foram elaborados objetivos específicos, de forma a direcionar os esforços do desenvolvimento deste trabalho. Estes objetivos são listados a seguir:

- Pesquisar ferramentas de código aberto que ofereçam suporte à verificação formal;
- Desenvolver um método que sistematiza o processo de tradução de programas em LD, da norma IEC 61131-3, para uma representação formal;
- Aplicar a o método e ferramenta propostos a um estudo de caso de um sistema de automação industrial.

1.3 Justificativa

O trabalho de verificação formal de sistemas descritos, usando a linguagem Ladder da norma IEC 61131-3, contribuirá para a melhoria da qualidade nos serviços prestados, sendo esta na forma de redução dos erros de programação que atingem a etapa de implantação da solução.

1.4 Metodologia

Na Figura 1.1, é exibida a cadeia de desenvolvimento do trabalho. Optou-se por dividir o projeto em duas etapas.

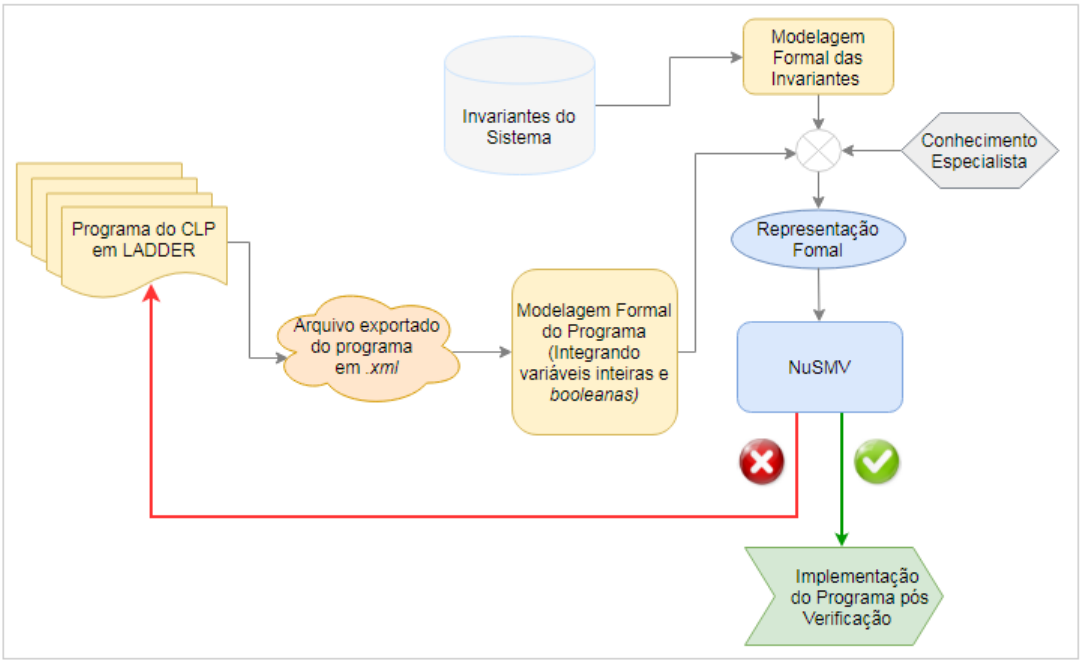

Figura 1.1: Diagrama de Fases do Método Proposto

1.4.1 Etapa 1: Tradução da Linguagem LD para uma Representação Formal

A proposta do projeto é baseada na verificação formal do modelo, que será verificado a partir de propriedades ou invariantes do sistema. Uma vez estabelecidas, as propriedades devem ser atendidas pelo modelo. O fluxo apresentado na Figura 1.1 parte do pressuposto de que as invariantes estejam corretas.

Uma vez que os programas de CLP são utilizados como entrada do fluxo, definidos não só pela norma IEC 61131-3, mas também por padrões internos dos fabricantes dos equipamentos, definiu-se que serão verificados programas do CLP Allen Bradley, modelo RSlogix 5000 -Compact Logix L23E (Allen Bradley, 2013).

1.4.2 Etapa 2: Validação do Modelo de Verificação

O processo de verificação, por meio do uso de ferramenta de verificação formal, tem como resultado um relatório sobre a violação ou não das propriedades descritas. Em caso de violação, analisando-se o resultado, é possível identificar quais foram as combinações das variáveis do sistema que levaram à condição de violação.

Nesta etapa será realizada a descrição de um sistema de automação escrito em linguagem Ladder usando um CLP. Esse sistema será verificado formalmente utilizando

a ferramenta proposta. O método proposto será validado analisando-se os resultados obtidos com o processo de verificação, dado que são exploradas diferentes propriedades de funcionamento do sistema durante sua verificação.

Considerando o resultado da saída do sistema de verificação, será possível validar o modelo desenvolvido, trabalhando com programas que são implementados em ambientes reais.

1.5 Contribuições

A principal contribuição deste trabalho é desenvolver um sistema de verificação formal para programas em lógica Ladder de CLPs, que permita avaliar a corretude dos programas, dadas as invariantes às quais deseja-se verificar. Além desta contribuição principal, também são contribuições a criação de:

1. Uma metodologia que sistematiza a tradução de programas de CLPs escritos em LD;

2. Uma ferramenta que permita a utilização de um verificador de código aberto existente, possibilitando a inserção de propriedades temporais dos sistemas de automação controlados por CLP.

1.6 Estrutura do Texto

O texto deste trabalho está estruturado de forma a facilitar o entendimento da pesquisa realizada. Assim, no capítulo 2, é apresentada a revisão da literatura, que detalha o estado da arte de trabalhos que modelam e verificam, manual e automaticamente, programas de CLP. No capítulo são detalhados os pontos chave de cada um destes trabalhos, e são apresentadas algumas lacunas deixadas por trabalhos discutidos ao longo do capítulo.

No capítulo 3, os conceitos preliminares são apresentados para que o leitor possa se contextualizar e se inteirar a respeito das definições que fundamentam a base teórica desta dissertação. Ao longo do capítulo, são introduzidos alguns tópicos que discutem sobre as áreas do conhecimento que são fundamentais para o efetivo entendimento do projeto desenvolvido.

O capítulo 4 descreve o método proposto para se realizar a verificação formal de programas de CLP. No decorrer do capítulo são apresentadas e analisadas as etapas de desenvolvimento do método. São descritas as estruturas que integram o método desenvolvido, bem como são trabalhados alguns exemplos para melhor entendimento e explanação do sistema de verificação.

É apresentada, no capítulo 5, a verificação formal de um sistema utilizando-se o método proposto. Ao longo do capítulo, detalha-se as propriedades de funcionamento do sistema e discute-se a respeito dos resultados da verificação formal. Ao final do capítulo, evidencia-se a os resultados obtidos por meio da verificação do modelo traduzido para a representação formal.

O trabalho finaliza no capítulo 6, com as considerações finais, onde são apresentadas discussões sobre o método proposto, sobre os resultados obtidos com a verificação do modelo formal gerado a partir do método de tradução desenvolvido. Conclusões e direções para trabalhos futuros também são apresentadas neste capítulo.

Capítulo 2

Revisão da Literatura

O presente capítulo apresenta o estado da arte de trabalhos que modelam e verificam automaticamente programas de CLP. São citados trabalhos que apresentam diferentes abordagem para modelar programas de CLP escritos nas linguagens da norma IEC 61131, também são apresentadas as modelagens que consideram o ciclo de *scan*), a ordem de execução entre *rungs* e interna de uma *rung*, bem como modelos que fazem o uso de variáveis booleanas e blocos de temporização.

Assim sendo, este capítulo foi dividido em duas seções que discutem trabalhos realizados no campo de verificação de programas de CLPs. Na seção 2.1 serão apresentados os trabalhos relacionados que verificam formalmente programas de CLP de forma manual, sendo estes relevantes para o escopo dessa dissertação. Posteriormente, a seção 2.2 relaciona os trabalhos que realizam a tradução e verificação de programas de CLP de forma automática, sendo discutidas algumas lacunas deixadas por trabalhos anteriores, que fazem com que o método de verificação proposto neste trabalho seja relevante em termos aprimoramentos no estado da arte de verificação formal de programas de CLPs.

2.1 Trabalhos Relacionados que Verificam Formalmente Programas de CLPs de forma manual

Muito embora haja trabalhos que abordem métodos de verificação de programas de CLPs, a minoria destes exemplificam abordagens reais de programas de acordo com a Norma IEC 61131-3. Nesse contexto, a aplicabilidade dos modelos desenvolvidos para realizar a tradução automática de programas de CLP, se restringem à operações e uso de blocos lógicos limitados, o que foge do contexto real, em que programas frequentemente

exigem o uso de blocos de temporização para realizar operações de sequenciamento, dentre muitas outras. Outro fator a ser considerado no processo de verificação de programas de CLPs, constitui na sistematização e padronização da tradução do programa em si. Nesse quesito a tradução automática destes programas é indispensável para se evitar erros que podem ocorrer durante modelagem manual de programas.

O primeiro trabalho que realizou a verificação formal de programas de CLPs foi apresentado por (MOON, 1994), que traduziu programas de CLPs escritos em linguagem Ladder para a linguagem de entrada do verificador de modelos SMV.

2.1.1 Principais Métodos utilizados para Modelagem de Programas de CLP

A verificação de modelos constitui em um método formal que, considerando a verificação de programas de CLP, muito trabalho manual e profundo conhecimento sobre a forma de execução de programas ainda são necessários.

Considerando que CLPs são amplamente utilizados para realizar a automação e controle de sistemas, tais componentes são indispensáveis no ambiente industrial. Nesse contexto a verificação dos programas de CLP tem sido o foco principal de vários trabalhos anteriores. Moon (MOON, 1994) apresentou um trabalho pioneiro na verificação formal de CLPs, que introduz um método para traduzir LD para SMV. O método apresentado por ele suporta a tradução de lógicas booleanas e leva em conta o ciclo de varredura do CLP. No entanto, ele não aborda situações rotineiras de programas reais como é o caso de terminações de *rungs* em duas ou mais bobinas, o que facilita a modelagem para a representação formal. Um trabalho semelhante que modela sistemas parecidos é apresentado em (RAUSCH; KROGH, 1998).

Outros pioneiros que verificam programas de CLPs fazendo o uso de outras linguagens da norma IEC 61131-3 como IL (CANET S. COUFFIN; PETIT, 2000; PAVLOVIC; KOLLMANN, 2007) ST (GOURCUFF; FAURE, 2008), e SFC (LAMPÉRIÈRE-COUFFIN; LESAGE, 2000). Estes trabalhos também modelam programas que possuem estruturas, cuja modelagem não considera a ordem de execução interna das *rungs* ou estruturas de programas industriais.

Trabalhos como o de (LAMPéRIèRE-COUFFIN O. ROSSI, 1999), realiza a modelagem manual de programa em linguagem SFC para uma representação formal em

SMV considerando apenas blocos booleanos. Assim também é a abordagem de (ROSSI O. DE SMET; GUENNEC, 2000), que considera apenas variáveis booleanas, mas que exibe um primeiro modelo para a representação de blocos de temporização, apesar de não o incorporar na modelagem dos programas. Rossi, modela programas manualmente e utiliza a representação formal Cadence SMV para representar programas na linguagem Ladder.

O trabalho de (LOBOV et al., 2004) modela o funcionamento do CLP em máquina de estados finitos. Em seu trabalho, Lobov não modela temporizadores, apenas blocos booleanos, sendo a modelagem manual não considerando um modelo real industrial. Basicamente seu trabalho faz uso de um modelo que representa as transições de estado transformando-as para blocos booleanos.

Em (GOURCUFF; FAURE, 2006), é exibido como manualmente se realiza a tradução de programas em ST para a representação formal em NuSMV. Muito embora considere apenas blocos booleanos e exiba apenas um trecho do processo de tradução, constitui em uma primeira abordagem de como sistematicamente traduzir programas de CLP. Posteriormente em um segundo trabalho (GOURCUFF; FAURE, 2008) apresenta uma evolução no processo de modelagem, considerando modelos que demonstrem variáveis booleanas no estado futuro destas, sendo este, um recurso de modelagem da ordem de execução entre *rungs* de programas genéricos.

Thapa (THAPA et al., 2006) apresenta a criação de uma linguagem intermediária para que seja possível traduzir as linguagens na norma IEC 61131-3 para esta representação intermediária, que posteriormente será convertida para SMV. O artigo apresentado por Thapa explica a modelagem em detalhes mostrando a árvore de sintaxe abstrata formada em decorrência da análise gramatical durante o processo de tradução. Já (LJUNGKRANTZ; AKESSON; FABIAN, 2008), modela o comportamento dos blocos booleanos, bem como contatos e bobinas do CLP de acordo com a lógica temporal implícita no comportamento ou combinação destes. Necessariamente, arranjos de contatos e bobinas são traduzidos para uma lógica proposicional equivalente e é associando a esta lógica um comportamento temporal.

No trabalho de (ZHOU et al., 2009), é utilizado o verificador UPPAAL, para realizar a verificação de programas. Trata-se de um trabalho cujo enfoque constitui em definir um *framework* de tradução manual para autômatos temporizados. Dessa forma,

programas em IL são traduzidos para uma linguagem formal modelando-se o funcionamento dos temporizadores.

Blech (BLECH; BIHA, 2011), define uma análise semântica para programas em IL e SFC, onde programas são traduzido para uma representação Coq,sendo esta uma semântica formal. Nos exemplos são considerados apenas blocos booleanos e o trabalho foca em introduzir a sistemática de tradução de programas genéricos para Coq. Já o trabalho apresentado por (FARINES et al., 2011) exibe a modelagem de um programa escrito em Ladder traduzindo-o para uma representação formal denominada Fiacre, usando o *solver* TINA. A implementação não é detalhada, e é realizada de forma manual, em seu trabalho não é exibida a análise temporal dos blocos de temporização.

2.2 Comparação entre as Propostas de Verificação Automática de Programas de CLP

Levando-se em conta que os ambientes de desenvolvimento de programas de CLP são frequentemente específicos do fabricante do dispositivo, é natural que haja certa variabilidade em termos de blocos de funções e recursos fornecidos para o programador.

Considerando os trabalhos que realizam a verificação automática dos programas de controladores lógicos, observa-se que dentre os mais relevantes, (BIALLAS; BRAUER; KOWALEWSKI, 2012), descreve o desenvolvimento de uma plataforma, denominada ARCADE. Esta plataforma constitui em um verificador que suporta as linguagens de programação IL e ST e realiza a verificação dos mesmos. No trabalho apresentado por Biallas, não foi mencionado se a plataforma modela blocos de operações aritméticas ou como considera o ciclo de execução de programas de CLP, muito embora seu trabalho não suporte a linguagem Ladder.

Barbosa (BARBOSA, 2012), discute métodos para se realizar a modelagem automática de programas de CLP exemplificando apenas instruções booleanas em seu sistema de verificação. Ele não verifica automaticamente programas, mas sugere algumas modelagens. No caso desse trabalho não é exemplificado nem mencionado o verificador utilizado. O trabalho tem um enfoque na estrutura e *framework* de tradução em algumas das linguagem de CLPs para o Método B, sendo esta uma representação intermediária. Já o trabalho desenvolvido por (PAKONEN et al., 2013), consiste na criação de uma *toolbox*

para a verificação de blocos lógicos implementados diretamente na ferramenta. Ou seja, o objetivo do programa desenvolvido não consiste em traduzir programas provenientes de um dado software de um CLP, mas sim do *design* da lógica de programa implementada diretamente na ferramenta desenvolvida.

No trabalho apresentado por (THAPA et al., 2006), há restrições para que sejam utilizados apenas blocos lógicos compostos por variáveis booleanas. Da mesma forma (GOURCUFF; FAURE, 2008) considera apenas variáveis booleanas e ainda o uso limitado de *loops*, dentre outros, para sua tradução automática. Em (O. LEANDRO S.; G., 2010) é abordada a metodologia adotada para realizar a tradução, mas não enfatiza-se a sistematização da tradução ou como é realizado o processo de tradução de forma automática. Muito embora estes trabalhos tenham sido pioneiros do ponto de vista de verificação automática de programas de CLP, os mesmo se tornam limitados em termos de aplicações reais.

A verificação automática de sistemas industriais, deveria portanto se aproximar de práticas de programação que sejam regulamentadas pela norma IEC 61131-3, e de fato contemplar variáveis de programas que, em aplicações corriqueiras, fazem uso de blocos aritméticos com variáveis inteiras e booleanas bem como o uso de blocos de temporização.

Capítulo 3

Conceitos Preliminares

No presente capítulo serão apresentados os conceitos e definições necessárias para o entendimento do presente trabalho. Alguns conceitos são introduzidos, pois considera-se que estes mereçam especial atenção para que sejam empregadas de forma consistente, como verificação formal e verificação simbólica de modelos. Além disso, são apresentadas definições básicas, o que garante o entendimento dos tópicos mais complexos que serão discutidos ao longo do trabalho, bem como uma abordagem sobre controladores lógicos programáveis, discutindo suas principais características e aplicações.

3.1 Definições Básicas

3.1.1 Defeito, Erro e Falha

Nesta seção, serão introduzidas definições básicas, sendo estas a de erro, defeito e falha, fundamentadas no padrão IEEE número 610.12-1990 (IEEE, 2010) que serão utilizadas nas seções seguintes posteriores deste trabalho.

Definição 1. ***(Defeito).*** *A incapacidade de um sistema ou componente de executar suas funções necessárias, sendo estas de acordo com os requisitos de desempenho especificados.* □

Definição 2. ***(Erro).*** *A diferença entre um valor ou condição calculada, observada ou medida e o valor verdadeiro, especificado ou condição teoricamente correta.*

Definição 3. ***(Falha).*** *Falha caracteriza-se por ser a manifestação do defeito, considerando-se a especificação dada.* □

3.1.2 Lógica Proposicional e Operadores Lógicos

A sintaxe da Lógica Proposicional (LP) consiste de símbolos e regras que permitem combinar os símbolos para construir sentenças, ou mais especificamente fórmulas. De uma forma geral, lógica proposicional ou cálculo sentencial possui como centro os conectivos lógicos, que permitem combinar afirmações de modo gramaticalmente válido. Os conectivos lógicos caracterizam-se por serem símbolos ou palavras utilizados para modificar uma sentença, ou combinar duas sentenças, de modo gramaticalmente válido. A seguir é apresentada a definição de conectivos ou operadores lógicos.

Definição 4. ***(Operadores Lógicos).*** *São símbolos ou palavras utilizados para modificar uma sentença, ou combinar duas sentenças. De modo gramaticalmente válido, forma-se uma nova sentença cujo sentido depende do significado das sentenças originais.* □

Na sequência é exibida a forma como comumente operadores lógicos são utilizados em operações de lógica matemática.

- Conjunção: $\wedge$;
- Negação: $\neg$;
- Disjunção: $\vee$;
- Implicação: $\Rightarrow$;
- Equivalência: $\Leftrightarrow$;
- Ou Exclusivo: $\bigoplus$.

Estes conectivos serão amplamente utilizados no decorrer deste trabalho, fazendo-se necessário entender a lógica associada à cada operação lógica como mostrado a seguir.

Tabela 3.1: Semântica da Negação

a	$\neg a$
V	F
F	V

Tabela 3.2: Semântica da Conjunção

a	b	$a \wedge b$
F	F	F
F	V	F
V	F	F
V	V	V

Tabela 3.3: Semântica da Disjunção

a	b	$a \vee b$
F	F	F
F	V	V
V	F	V
V	V	V

Tabela 3.4: Semântica da Implicação

a	b	$a \Rightarrow b$
F	F	V
F	V	V
V	F	F
V	V	V

Tabela 3.5: Semântica da Equivalência

a	b	$a \Leftrightarrow b$
F	F	V
F	V	F
V	F	F
V	V	V

As interpretações semânticas dos operadores lógicos de 'negação', 'conjunção', 'disjunção', 'implicação', 'equivalência' e 'ou exclusivo' são vistas respectivamente nas tabelas 3.1, 3.2, 3.3, 3.4, 3.5 e 3.6. Nestas tabelas, são estabelecidas relações entre a variável a ou a e b, onde estas são consideradas verdadeiras (V) ou falsas (F). A coluna da direita de cada uma das tabelas apresenta o resultado para a operação de acordo com o operador lógico utilizado.

3.2 Teste, Verificação e Validação

Dentro do contexto do presente trabalho, considerando que serão verificados programas de CLPs, faz-se necessário diferenciar teste de validação e verificação, conforme define (MYERS, 1979).

Em uma visão prática, quando deseja-se garantir que um dado sistema em desenvolvimento está realmente se comportando conforme proposto, constrói-se para o mesmo um modelo por meio do uso de uma linguagem formal, para que então, com base em um modelo formal possa-se:

1. Descobrir situações em que o sistema em questão se comporta de maneira incorreta, indesejável ou de forma diferente das especificações, por meio de testes;

Tabela 3.6: Semântica da Ou Exclusivo

a	b	$a \bigoplus b$
F	F	F
F	V	V
V	F	V
V	V	F

2. Validar o modelo através de simulações;

3. Realizar provas matemáticas que garantem que este modelo possui as propriedades requisitadas (verificação).

   Na sequencia define-se teste, validação e verificação.

Definição 5. ***(Teste).*** *O teste é destinado a mostrar que o sistema faz o que foi proposto a fazer, assim como destina-se a descobrir se há defeitos no sistema antes de seu uso.* □

Considerando um *software* como exemplificação de sistemas os quais podem ser testados, os resultados são do teste analisados à procura de erros, anomalias ou informações sobre os atributos não funcionais do programa. O processo de teste tem dois objetivos distintos:

1. Demonstrar ao desenvolvedor e ao cliente que o *software* atende a seus requisitos.

2. Descobrir situações em que o *software* se comporta de maneira incorreta, indesejável ou de forma diferente das especificações. O teste de defeitos preocupa-se com a eliminação de comportamentos indesejáveis do sistema, tais como panes, processamentos incorretos e corrupção de dados.

Considerando-se o primeiro objetivo, este conduz à testes de validação, nos quais espera-se que o sistema execute corretamente usando determinado conjunto de casos de teste que refletem o uso esperado do sistema. Já o segundo objetivo leva a testes de defeitos, nos quais os casos de teste são projetados para expor os defeitos. Os casos de teste na busca por defeitos podem ser deliberadamente obscuros e não precisam refletir com precisão a maneira como o sistema costuma ser usado. Claro que não existem limites definidos entre essas duas abordagens de teste. Durante os testes de validação, serão

encontrados defeitos no sistema; durante o teste de defeitos, alguns dos testes mostrarão que o programa corresponde a seus requisitos.

Os testes não podem determinar se o software é livre de defeitos ou de fato se ele se comportará conforme especificado em qualquer situação. É inerentemente possível que em um processo de testes, seja esquecido aquele que poderia descobrir mais problemas no sistema. Como assertivamente dito por (DIJKSTRA, 1972), um dos primeiros colaboradores para o desenvolvimento da engenharia de software.

Barry Boehm (BOEHM, 1979), pioneiro da engenharia de software, expressou sucintamente a indagação:

'Estamos construindo o produto certo?'

A indagação de Boehm, leva à uma reflexão quanto ao processo de validação, cuja definição é data a seguir.

Definição 6. ***(Validação).*** *O processo de validação busca identificar se o sistema em desenvolvimento satisfaz suas especificações e oferece a funcionalidade esperada. A validação, tem por objetivo garantir que o sistema atenda às expectativas do cliente.* □

A validação é essencial porque, nem sempre as especificações de requisitos do sistema refletem os desejos ou necessidades dos clientes e usuários. Portanto, o processo de validação deve ser feito de forma consistente para se garantir que o modelo de requisitos reflita de maneira precisa as necessidades do interessado e forneça uma base sólida para o projeto (PRESSMAN, 2011).

Definição 7. ***(Verificação formal).*** *Por verificação formal entende-se a rigorosa exploração de incoerências no projeto de um sistema. A verificação é realizada por meio da criação de um modelo matemático através da prova formal de teoremas.* □

Dentre as técnicas para verificar formalmente um sistema, a verificação de modelos e a prova por teorema são as duas mais utilizadas (LAMPÉRIÈRE-COUFFIN; LESAGE, 2000). A verificação formal nasceu e se desenvolveu a partir da preocupação da Ciência da Computação em atestar o funcionamento dos seus sistemas - *softwares* ou *hardwares* - pois a simulação e os testes se mostraram insuficientes em inúmeros casos: acidente do foguete Ariane 5 (SPARACO, 1996), aplicação de doses fatais de radiação em pacientes com câncer pelo Therac-25 (LEVESON; TURNER, 1993), um equipamento de radioterapia controlado por computador, etc.

No início dos anos 80,(CLARKE, 1982), e de forma independente, Quielle e Sifakis (QUEILLE; SIFAKIS, 1982), introduziram os algoritmos de verificação de modelos usando lógica temporal, lançando assim, a base do método. De forma geral, o processo de verificação inicia-se assim que os requisitos do sistema estão claros, e prossegue em todas as fases do processo de desenvolvimento.

Nesse contexto pode-se destacar a técnica de Verificação de Modelos (Model Checking) (CLARKE; SISTLA, 1986).

Definição 8. ***(Verificação de Modelo).*** *A verificação de modelos (model checking) é uma técnica para verificar sistemas concorrentes de estados finitos, que tem como principal objetivo verificar se o modelo desenvolvido M satisfaz determinadas propriedades P especificadas pelo usuário. Isto é, se uma fórmula proposicional P é verdadeira em todos ou em alguns estados E da árvore de computação M, ou seja se M satisfaz P.* □

Considerando estado como sendo a atribuição dos valores de todas as variáveis do modelo em um determinado instante. Este método enumera todos os estados alcançáveis, dados os estados iniciais e a regra de transição, e verifica as propriedades de acordo com as especificações fornecidas. Isso pode levar a uma explosão de estados a serem verificados, porém as ferramentas modernas possuem a capacidade de lidar com tal problema. Duas principais lógicas temporais podem ser utilizadas para a descrição das propriedades a serem testadas, conforme descritas na seção 2.2, a lógica temporal LTL (*Linear Temporal Logic*) e CTL (*Computation Tree Logic*).

A especificação, depois de escrita na lógica temporal, pode ser avaliada por uma ferramenta. Caso a especificação seja avaliada como falsa, então a ferramenta constrói um contra-exemplo exibindo as transições dos estados que levaram à especificação a ser infringida, tornando-a falsa. O modelo deve descrever a relação de transição dos estados, por meio das evoluções válidas para a máquina, formando um sistema de transição.

3.3 Lógicas Temporais

Nesta seção são descritas lógicas para especificar propriedades temporais dos sistemas de transição de estados. Na lógica temporal, o tempo não é mencionado explicitamente, pelo contrário, uma fórmula especifica se algum estado é eventualmente alcançado, ou se um estado de erro nunca é atingido.

3.3.1 Lógica Temporal Ramificada

Conceitualmente, as fórmulas da lógica temporal ramificada (*Computation Tree Logic* - CTL) descrevem propriedades de árvores da computação. Forma-se uma árvore de computação ao se designar um estado de uma estrutura de transição de estados como estado inicial e, então, desenrola-se a estrutura numa árvore infinita com o estado inicial na raiz.

Em CTL as fórmulas são compostas de *quantificadores de caminho* e *operadores temporais*. Os quantificadores de caminho são usados para descrever a estrutura de ramificação da árvore de computação. Há dois quantificadores de caminho: **A** (para todos os caminhos de computação - *for all paths*) e **E** (para algum caminho de computação- *for some computation path*). Estes quantificadores em um estado particular são utilizados para especificar que todos ou alguns dos caminhos que começam neste estado possuem alguma propriedade.

Sendo assim os operadores temporais descrevem as propriedades de um dado caminho ao longo da árvore. Neste trabalho são usados, principalmente, os seguintes operadores temporais:

- **F**(*future*- Eventualmente ou no futuro)afirma que uma propriedade é válida em algum estado do caminho.
- **X**(*next*- Próximo estado)requer que uma propriedade seja verdadeira no segundo estado do caminho.
- **G**(*globally*- Sempre)afirma que uma propriedade é sempre válida em todos os estados do caminho.
- **U**(*until*- Até que)Este operador combina duas propriedades. Considerando as propriedades p_1 e p_2, o operador p_1**U** p_1 é válida se houver um caminho onde p_2 torna-se válido, e em cada estado precedente p_1 sempre for válido.

Como exemplo, no estado inicial da Figura 2.2 é válida a fórmula E **G** p. Na CTL, os operadores temporais (F, X, G e U) atuam sobre os possíveis caminhos a partir de um dado estado. Desta forma, cada operador temporal deve ser precedido por um quantificador de caminho (A ou E).

Considerando AP o conjunto de nomes de proposições atômicas e P_i proposições atômicas pertencentes a AP, com $i = 1, 2, ...$ A sintaxe da lógica CTL é dada pela gramática a seguir (CLARKE E.AND PELED, 1999), onde φ e ψ são fórmulas CTL :

$$\varphi, \psi ::= P_1|P_2|...|\neg\varphi|\varphi \wedge \psi|EX\psi|AX\psi|E\psi U\varphi|A\psi U\varphi \tag{3.1}$$

Na notação BNF (*Backus-Naur Form*) (ESSALMI; AYED, 2006) vista em 2.1, nota-se que a partir desse núcleo base pode-se derivar outras fórmulas:

- $EF\psi \rightarrow E(TrueU\psi)$
- $EG\psi \rightarrow \neg AF\neg\psi$
- $AF\psi \rightarrow A(TrueU\psi)$
- $AG\psi \rightarrow \neg EF\neg\psi$

Essas fórmulas derivadas são tidas como as mais comuns. A Figura 3.1 mostra exemplos de árvores de computação que ilustram esse operadores sendo usados relacionados aos estados iniciais.

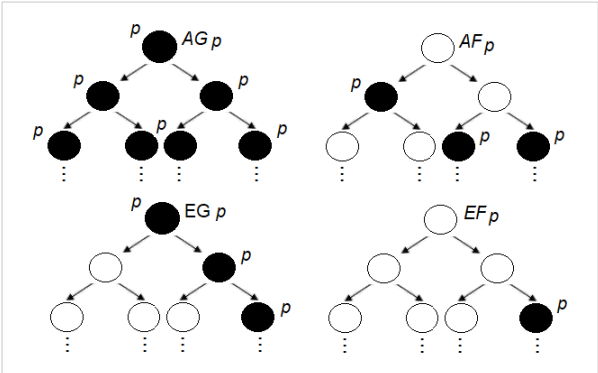

Figura 3.1: Exemplo de Operadores CTL

3.3.2 Lógica Temporal Linear

A lógica temporal linear (*Linear Temporal Logic* - LTL) (PNUELI, 1997) assume o tempo como uma sequência de execução de um sistema onde cada possível caminho de computação é considerado separadamente, raciocinando sobre uma única sequência de

execução. Ao invés de ser interpretado sobre árvores de computação, as fórmulas LTL são interpretadas com respeito a caminhos individuais de computação. Em outras palavras, a lógica temporal linear expressa propriedades sobre uma sequência linear de execução do sistema.

As fórmulas em LTL são compostas de proposições atômicas usando os conectivos booleanos e os operadores temporais. Diferente de CTL, onde cada operador temporal deve ser prefixado com um quantificador de caminho, os conectivos proposicionais e os operadores temporais podem ser aninhados de maneira diferente em LTL. Uma fórmula LTL ξ é definida recursivamente da seguinte forma:

$$\xi ::= ap|False|True|(\neg\xi)|(\xi \vee \xi)|(\xi \wedge \xi)|(\xi \rightarrow \xi)|(X\xi)|(F\xi)|(G\xi)|(\xi_1 U \xi_2) \tag{3.2}$$

onde ap é uma proposição atômica, X, F, G e U são operadores temporais previamente definidos para fórmulas CTL. É importante notar que as fórmulas LTL não possuem quantificadores de caminho explícito. Uma fórmula LTL é considerada verdadeira sobre todo o caminho computacional, isto é, as fórmulas LTL são implicitamente quantificadas universalmente no caminho. Cada fórmula LTL ξ é considerada da forma $\mathrm{A}(\xi)$.

A lógica LTL pode expressar alcançabilidade e segurança, porém LTL não pode expressar a existência de um caminho, ou seja não pode expressar propriedades que dizem respeito a um único caminho. Com isso LTL é implicitamente quantificada universalmente.

3.4 Controlador Lógico Programável

3.4.1 Verificação de Programas de CLPs

No presente trabalho são aplicadas técnicas de verificação de modelos à projetos de automação, sendo esse sistemas baseado em CLPs programados em linguagem LD. Foram desenvolvidos padrões para a verificação, com base em estruturas lógicas de CLPs, visando automatização do processo de verificação dos programas.

A princípio obteve-se o modelo do programa de aplicação que são as diretrizes

de funcionamento do CLP. De forma geral, trata-se dos programas escritos em Ladder, que são modelados em uma linguagem formal. As propriedades de funcionamento do sistema que são então especificadas, e descrevem as sequências lógicas desejadas, como por exemplo, uma lógica de sequenciamento ou as propriedades estruturais do sistema, como condições de segurança. Em função da descrição das especificações ou do conhecimento do sistema, é feita a construção da propriedade formal, escrita na maioria das vezes em CTL.

O resultado da verificação indica se o modelo do sistema está em conformidade com as especificações ou não, sendo uma das fontes de não conformidade o erro de modelagem do sistema, neste caso, deve-se refinar ou corrigir o modelo do sistema para se fazer uma nova verificação. A segunda possibilidade é a existência de um erro do programa de aplicação captado na modelagem. Neste caso, deve-se partir para um reprojeto do programa de CLP antes de uma nova verificação.

3.4.2 Principais Instruções em Linguagem LD

A linguagem Ladder é amplamente utilizada em todos os âmbitos da automatização de sistemas realizada por meio do uso de CLPs, a mesma é constituída por símbolos gráficos, representando contatos e bobinas. Nesse contexto, os contatos e bobinas correspondem a variáveis booleanas armazenadas na memória de dados do CLP. Estes são conectados por ligações em ramos (*rungs* ou *networks*) como em uma lógica de acionamento à relé. As expressões booleanas calculadas a cada ciclo de varredura do CLP correspondem à avaliação lógica sequencial do diagrama de contatos.

A Tabela 2.1, exibe-se o símbolo e a descrição dos contatos e bobinas da lógica LADDER mais comumente utilizados.

A lógica de funcionamento dos comandos em Ladder, baseadas em (ZHENDONG, 1997), assume um diferente nomenclatura para representar um contato NA, NF e uma bobina. Dessa forma tem-se as seguintes associações:

- XIC: Contato normalmente aberto (NA);
- XIO: Contato normalmente fechado (NF);
- OTE: Bobina.

Tabela 3.7: Símbolo e descrição de diagrama de contatos e bobinas

SIMBOLOGIA	DESCRIÇÃO
	Contato Normalmente Aberto (NA). O estado à esquerda do contato é transferido ligação à direita, se o estado de I for verdadeiro.
	Contato Normalmente Fechado (NF). O estado da ligação à esquerda é transferido para a ligação à direita se o estado de I for falso.
	Representa uma conjunção $I_1 \wedge I_2$, onde I_1 e I_2 são variáveis proposicionais.
	Representa uma disjunção $I_1 \vee I_2$, onde I_1 e I_2 são variáveis proposicionais.
	Bobina. O estado da ligação da esquerda é copiado para a variável. Q.

3.5 Ordem de Execução em Programas Escritos em LD

Realizar a verificação formal de programas industriais de CLP não é uma prática comum nos dias de hoje, muito embora seja um técnica comprovadamente eficaz, a verificação formal não é de todo usada durante o desenvolvimento de programas de CLP. Vários motivos podem explicar essa situação: dificuldade para engenheiros de automação escreverem propriedades formais em lógica temporal, a ausência de tradutores automáticos para linguagem formal nos ambientes de desenvolvimento, o que de fato torna a modelagem manual uma árdua tarefa passível de erros durante a tradução, dentre outros fatores.

A base para se realizar a verificação dos programas em Ladder é garantir que o modelo formal gerado a partir da descrição do LD seja equivalente, mesmo que haja algum nível de abstração, à própria descrição do LD. A análise de trabalhos anteriores evidenciou que nenhum destes aborda a semântica da linguagem Ladder por completo. Além disso, até onde concluiu-se a pesquisa, nenhum aspecto da ordem de execução entre e interna de *rungs* da LD foi abordado nos trabalhos anteriores.

Nas seções seguintes, serão discutidos os pontos chaves para se modelar corretamente um programa de CLP em LD. Para que o programa seja modelado em uma

representação formal, devem ser considerados os três níveis de ordem de execução: 1) o ciclo de varredura; 2) a ordem de execução entre as *rungs* de um programa; e finalmente 3) a ordem de execução dentro da própria *rungs*. Estes são pontos críticos para uma modelagem bem-sucedida de um programa em LD.

3.5.1 Ciclo de Varredura em Programas de CLPs

O CLP é um controlador baseado em um microprocessador que testa os sinais de entrada em instâncias de tempo discretos, executa um programa de controle durante um período de tempo e atualiza o valor dos sinais de saída. Os sinais de entrada são geralmente enviados para o controlador por meio de sensores e sinais de comando provenientes da planta ou sistema a ser controlado. Dessa forma as informações de sensores e do controle são aplicados à interface de entrada do dispositivo, no entanto, somente durante a operação de leitura, os sinais são copiados para a memória de entrada. Estas informações, congeladas na memória de entrada, são usadas durante a execução do programa do usuário. Os sinais de saída, calculados na operação anterior, são copiados da memória de saída para a interface de saída do CLP. A interface de saída por sua vez corresponde a um conjunto de portas com valores nominais de operação (tensão, corrente e outros) adequados para o controle do atuador.

As saídas de inicialização, leitura de entradas, execução e atualização das saídas são executadas repetidamente e indefinidamente, de acordo com o ciclo mostrado na Figura 3.2, chamado o ciclo de *scan*. O tempo que cada ciclo leva para ser concluído é o mesmo, caracterizando um ciclo periódico cujo período é chamado de tempo de varredura.

O modo de operação descrito acima, chamado de *single-task*, atende a maioria das aplicações. Para algumas, no entanto, é necessário um recurso adicional chamado *multitask* (MOKADEM B. BERARD; SMET, 2005). Em uma operação normal (execução do programa principal), o tempo mínimo de resposta para um evento é entre 1 ciclo de varredura, se o evento ocorrer no instante exato antes de ser lido pelo CLP e 2 ciclos de varredura, se o evento ocorrer imediatamente após a leitura da entrada do CLP (MOKADEM B. BERARD; SMET, 2005).

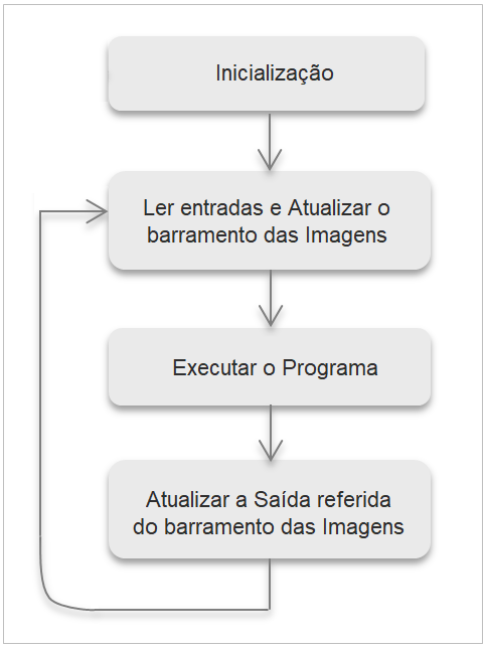

Figura 3.2: Ciclo de Varredura (*Scan*) em CLPs

3.5.2 Ordem de Execução entre *Rungs* de um Programa em LD

A ordem de execução entre as *rungs* de um CLP é um fator muito importante a ser considerado. A Figura 3.3 exibe a forma com a qual o programa de usuário do CLP é lido. As *rungs* em LD são lidas da esquerda para a direita, de cima para baixo conforme indicam as setas em vermelho. Ao final do processo de leitura do programa de usuário, as saídas são atualizadas.

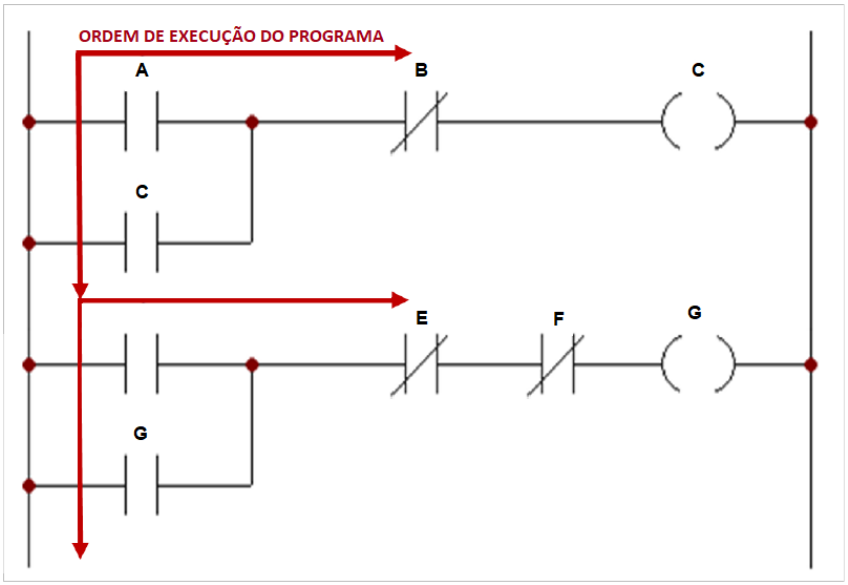

Figura 3.3: Forma de leitura de *rungs* em CLPs

Considerando o trabalho realizado por (MOON, 1994), observa-se que o valor

da memória de referência da bobina será atualizado apenas no próximo ciclo de *scan*, usando a construção *next* do SMV. Isso é suficiente para a maioria dos casos, mas quando mais de uma bobina está referenciando o mesmo endereço de memória de dados presente no programa, essa abordagem não é suficiente. Além disso, quando a memória de dados é manipulada usando instruções especiais que alteram diretamente seus valores, essa modelagem produziria resultados errados.

Rossi (ROSSI O. DE SMET; GUENNEC, 2000) apresenta uma abordagem diferente para modelar programas em LD usando o SMV. O trabalho amplia a gama de operadores suportados para incluir temporizadores, subida e descida de bordas, instruções de salto e qualquer sequenciamento intrincado de *rungs*. Apesar do trabalho tratar a ordem de execução entre *rungs* de uma forma elegante, ele não suporta todos os casos corretamente quando os dados valores correspondentes a bobinas que possui a mesma referência de memórias são modificados entre *rungs*, que é uma prática comum no mundo real da programação em LD.

3.5.3 Ordem de Execução Interna de uma *Rung* em Programas Escritos em LD

Modelar uma *rung* considerando sua ordem interna de execução é um detalhe que passa despercebido quando se trata da modelagem formal de programas em linguagem Ladder. Muito embora tenha-se amplo conhecimento na forma de execução dos programas, a maneira com a qual se modela uma *rung* em representação formal deve observar, por exemplo, a situação ilustrada na Figura 3.4

Considerando a *rung* da Figura 3.4, pode-se observar que, dada a ordem de execução indicadas em (1) e (2), o CLP atualiza primeiramente o estado da bobina *C*. Consequentemente, quando é inicializada a execução da parte da *rung* indicada por (2), deve-se obrigatoriamente considerar o estado do contato da bobina *C*, como sendo correspondente ao da execução (1). Essa análise, conforme mencionado, não foi evidenciada nos trabalhos existentes na literatura, e serão devidamente tratadas no presente trabalho.

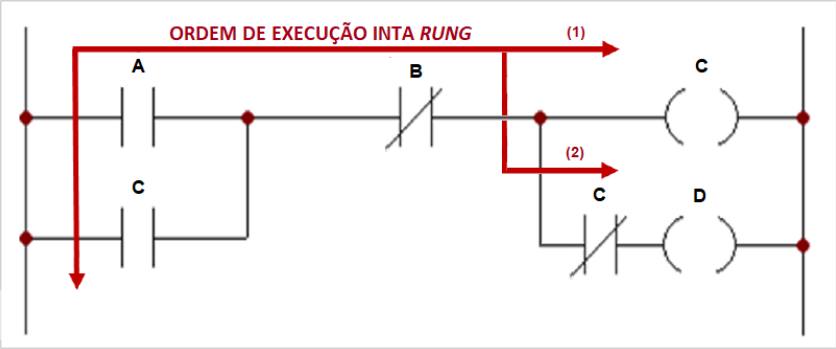

Figura 3.4: Ordem de Execução Intra *Rungs* de um programa de CLP

3.6 NuSMV

O Novo Modelo de Verificação Simbólica (*New Symbolic Model Verifier* - NuSMV) (CIMATTI E. CLARKE, 2000) constitui em uma ferramenta utilizada para verificação de modelos simbólicos cujo código é aberto. A ferramenta foi desenvolvida a partir do Verificador Simbólico de Modelos ()*Symbolic Model Verifier* – SMV), desenvolvido por (MCMILLAN, 1993) durante seu doutorado na Carnegie Mellon University, que é o verificador de modelos baseado em BDDs. O NuSMV é uma linguagem que descreve modelos e os verifica de forma direta, validando as fórmulas em LTL, assim como em CTL. Como entrada a ferramenta recebe um texto que consiste na descrição formal do modelo, bem como especificações de propriedades do modelo descritas em lógica temporal. O resultado da verificação retorna “verdadeiro ”, quando a especificação é satisfazível, e caso não seja, um “contra-exemplo” é retornado, indicando a razão pela qual a especificação representada pelo programa não é satisfeita, sendo a mesma “falsa”.

3.6.1 A Ferramenta de Verificação NuSMV

A ferramenta NuSMV foi escrita em ANSI C, e foi projetada de forma a caracterizar por ser robusta assim como para que os processos industriais mais complexos fossem atendidos com elevado padrão de qualidade em termos de modelagem. Seu código fonte é dividido em módulos, e seus módulos de entrada permitem modelar os sistemas a serem verificados em máquinas de estados finitos, tendo a capacidade de descrever processos que são síncronos ou assíncronos e ainda aqueles que possuem condições de não determinismo. As entradas do NuSMV descrevem as transições dos estados modelados, de forma tal que as relações descrevem as evoluções que são válidas da máquina de estados

finitos, como consequência obtém-se as relações do sistema modelado.

Dessa forma, as possíveis configurações futuras do sistema podem ser identificadas a partir do estado atual. De um modo geral, expressões no cálculo proposicional podem ser usadas para definir as relações de transição, consequentemente há um ganho de flexibilidade, entretanto deve-se atentar para que inconsistências sejam evitadas. Contradições ocorrem, quando por exemplo há a presença de uma contradição lógica, que pode resultar em um *deadlock*. A seguir são descritos os elementos da especificação de um sistema utilizando a linguagem de entrada para a ferramenta NuSMV (CAVADA A. CIMATTI; TCHALTSEV, 2010).

- **MODULE main (módulo principal)** – Em NuSMV toda especificação deve possuir um módulo principal, sem parâmetros, que representa o sistema.
- **VAR -** Declarações das variáveis são realizadas no bloco VAR. As declarações podem ser do tipo enumerado, intervalar, booleano e instâncias de módulos.
- **MODULE** – Módulos são criados para particionar a modelagem do sistema, estes encapsulam as declarações de todas as variáveis, inclusive as de estado e os eventos, que são declarados *boolean*. Cada módulo pode conter a regra de transição dos estados e a especificação das propriedades. A passagem é por referência, sendo possível declarar um módulo com parâmetros para realizar a reutilização de código.
- **ASSIGN -** As regras que determinam a inicialização e transições para o próximo valor de uma variável são declaradas nesta seção. A atribuição direta estabelece um valor inicial para a variável, por meio da comando *init(variável)*. O comando *next(variável)* fornece o valor da variável no próximo estado, a partir do estado atual. A atribuição em *next(variável)* pode ser feita utilizando a regra *case*, onde é possível determinar o próximo valor da variável em função de várias condições.
- **DEFINE -** A seção DEFINE associa uma variável a uma expressão. Uma variável em DEFINE sempre é substituída pela sua definição quando é encontrada na especificação.

- **CTLSPEC -** Propriedades em formato CTL são especificadas precedidas da palavra chave CTLSPEC, sendo estas as propriedades verificadas.

  LTLSPEC - Propriedades em formato LTL são especificadas precedidas da palavra chave LTLSPEC, sendo estas as propriedades verificadas.

```
1   MODULE main
2   VAR
3           request : boolean;
4           state : {ready, busy};
5   ASSIGN
6           init(state) := ready;
7           next(state) :=
8                   case
9                       state = ready & request: busy;
10                      1 : {ready, busy};
11                  esac;
12  LTLSPEC
13          G(request -> F state=busy)
```

Figura 3.5: Modelo de um Sistema Concorrente em NuSMV

A Figura 3.5, exibe o modelo do sistema descrito ou programado na linguagem NuSMV. Um programa em NuSMV pode ter um ou vários módulos e, tal como algumas linguagens de programação, um dos módulos deve ser chamado *"main"*. Nos módulos são declaradas as variáveis e seus respectivos valores. As atribuições normalmente são feitas com um valor inicial para cada variável e em seguida uma especificação do próximo valor por meio de uma expressão formada pelos valores correntes das variáveis.

O programa da Figura 3.5 é um exemplo de um sistema hipotético, que consiste de duas variáveis *request* do tipo booleana e a variável *state* do tipo enumeração. A variável do tipo enumeração depende do valor retornado das operações do programa recebendo como parâmetro *ready, busy*, onde 0 denota *"false"* e 1 representa *"true"*. Considerando que os valores iniciais e subsequentes da variável *request* não são definidos no programa, seus valores são definidos pelo ambiente externo durante a execução do programa.

A variável *state* é inicializada com estado *ready"* e pode ficar *busy* se a variável textitrequest for *true*. Porém se a variável *request* for *"false"* o valor de textitstate fica indeterminado. Observe que na linha 9 da Figura 3.5 caso *ready* e textitrequest sejam verdadeiros, o sinal ":"atribui o estado *busy* para a variável *state*. Já a linha 10 assume que caso a condição da linha 9 não seja satisfeita por *default* a variável *state* permanece com valor inalterado.

A avaliação da expressão formada com o *"case"* é feita de cima para baixo, sendo que a primeira avaliação do lado esquerdo do sinal ":" (dois pontos) que tiver seu valor verdadeiro, atribuirá o valor do lado direito para a variável declarada. Assim sendo o valor "1:" fica como a avaliação padrão, caso nenhuma das avaliações anteriores sejam verdadeiras.

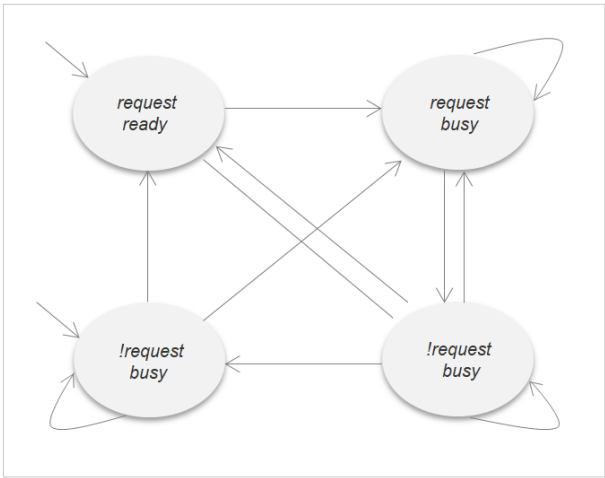

Figura 3.6: Modelagem da Máquina de Estados do Sistema

Este programa denota o sistema de transição da Figura 3.6, onde existem 4 estados, cada um dos estado é definido em função dos valores possíveis para as duas variáveis. No sistema, o nome *"!request"* representa um valor falso para a variável *"request"*. O programa e o sistema de transição são não-determinísticos, isto é, o próximo estado não é unicamente definido. Qualquer transição de estado baseado no comportamento da variável *"state"* vem em pares e o resultado é a transição para um estado sucessor onde *"request"* pode ser falso ou verdadeiro, respectivamente. Observando a Figura 3.6 pode-se notar que a partir do estado *request, busy* o sistema pode caminhar para quatro estados destinos, ele mesmo e mais três outros estado. As especificações em LTL (*Linear Temporal Logic*) são introduzidas pela palavra chave LTLSPEC e são simples fórmulas LTL. Neste exemplo a especificação está declarando que: Para qualquer estado, se o valor de *"request"* é verdadeiro, então eventualmente ocorrerá um estado *"busy"*.

3.7 Etapas do Processo de Compilação de um Programa

Um compilador caracteriza-se por ser um programa de computador que, no escopo deste trabalho constitui em um programa escrito em linguagem Java, que realiza a tradução de um programa fonte em LD para um programa alvo em NuSMV.

Nesse contexto a ferramenta de compilação desenvolvida deve possuir tanto a forma (ou sintaxe) quanto o conteúdo (significado ou semântica) da linguagem de entrada, e ainda, as regras que controlam a sintaxe e o significado na linguagem de saída. Assim sendo, o compilador precisa de um sistema de mapeamento do conteúdo da linguagem-fonte para a linguagem-alvo (COOPER; TORCZON, 2012).

O compilador tem um *front end* para lidar com a linguagem de origem, e um *back end* para lidar com a linguagem alvo. Visto como uma caixa-preta, um compilador deve ser semelhante a Figura 3.7.

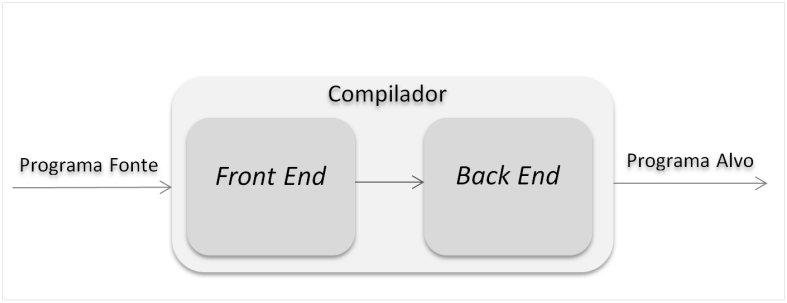

Figura 3.7: Representação em caixa preta de um Compilador

A compilação é composta por duas etapas, a análise e a síntese. Existem três processos de análise, sendo estas a léxica, sintática e a semântica. Já a síntese, constitui na geração do programa alvo.

3.7.1 Análise Léxica

Análise léxica (*scanning*) corresponde ao primeiro estágio de um processo em três partes que o compilador utiliza para traduzir o programa de entrada. O *scanner*, ou analisador léxico, lê um fluxo de caracteres (*tokens*) e produz um fluxo de palavras.

Estes *tokens* são agrupados em palavras e, aplicando-se um conjunto de regras, pode-se determinar as palavras são válida na linguagem-fonte. Se a palavra é válida, o *scanner* atribui-lhe uma categoria sintática, ou classe gramatical.

3.7.2 Análise Sintática

A análise sintática (*parsing*) é o segundo estágio do processo de compilação. O *parser* trabalha com o programa transformado pelo *scanner*; recebendo um conjunto de palavras, cada uma dessas associada a uma categoria sintática (tal como sua classe gramatical).

Se o *parser* determina que o fluxo de entrada é um programa válido, constrói um modelo concreto do programa para uso pelas últimas fases da compilação. Caso contrário, ele informa o problema e a informação de diagnóstico apropriada ao usuário.

O *parser* de um compilador tem como responsabilidade principal reconhecer a sintaxe — ou seja, determinar se o programa sendo compilado é uma sentença válida no modelo sintático da linguagem de programação.

3.7.3 Análise Semântica e Geração do Código Compilado

Corresponde à tarefa de análise final do compilador. Para esta finalidade, se faz necessário ter uma base de conhecimento sobre a computação detalhada que está codificada no programa de entrada, e analisar como o programa interage com arquivos e dispositivos externos.

Dessa forma, deve-se, durante a análise semântica, considerar o contexto no qual uma instrução, ou um conjunto de instruções do programa fonte foram implementadas. Assim, após traduzido, o programa deve se comportar de forma fidedigna ao programa fonte.

Após realizada a análise semântica do programa, deve-se gerar o código compilado, nesta etapa o programa proveniente do analisador semântico é traduzido para um formato de programa que possa ser executado.

Capítulo 4

Método Proposto para Verificação de Programas de CLP

Neste capítulo é abordado o método utilizado para realizar a modelagem e a verificação automática de programas de CLP. O presente capítulo foi subdividido em algumas subseções, para melhor entendimento das etapas desenvolvidas. Iniciando com a apresentação do *framework* do sistema de verificação automática e, na sequência, são detalhadas cada uma das etapas correspondentes ao sistema, para isso serão apresentados alguns exemplos no decorrer do capítulo.

4.1 Relevância do Método proposto

Levando-se em conta os trabalhos discutidos no capítulo 2, observou-se que, muito embora tenham grande contribuição para o a evolução do estado da arte da verificação de programas de CLP, ainda há lacunas a serem preenchidas.

Este trabalho portanto, explora e propõe soluções para estas lacunas. Pode-se destacar entre elas a análise criteriosa de ordem de execução entre e intra *rungs* de programas em Lógica LD. Considerando situações reais de programas implementados em ambiente industrial.

Em última instância será demonstrado o método e as ferramentas utilizadas para realizar a verificação de programas de CLP de forma automática, permitindo avaliar o comportamento de programas quando são verificadas as propriedades de funcionamento deste.

4.2 Método proposto para Verificação Automática de Programas de CLP

O principal resultado deste trabalho é o desenvolvimento da ferramenta de tradução da linguagem Ladder para o NuSMV, utilizando-se o modelo de tradução desenvolvido. Esta seção discutirá o processo de modelagem de programas CLP, bem como das invariantes do sistema para uma representação formal.

Considerando que, o arquivo exportado do CLP constitui na entrada de dados do sistema, e que os blocos da linguagem do programa são definidas não somente pela norma IEC 61131-3, mas também pelos padrões internos dos fabricantes de equipamentos, optou-se por utilizar programas do CLP Allen Bradley, sendo o modelo do CLP o RSlogix 5000 - Compact Logix L23E (Allen Bradley, 2013). A razão pela qual foi escolhido o Compact Logix é devido ao fado de ser um CLP amplamente utilizado em automação de processos e aplicações industriais. Na Figura 4.1, é exibido o *framework* da metodologia de verificação desenvolvida.

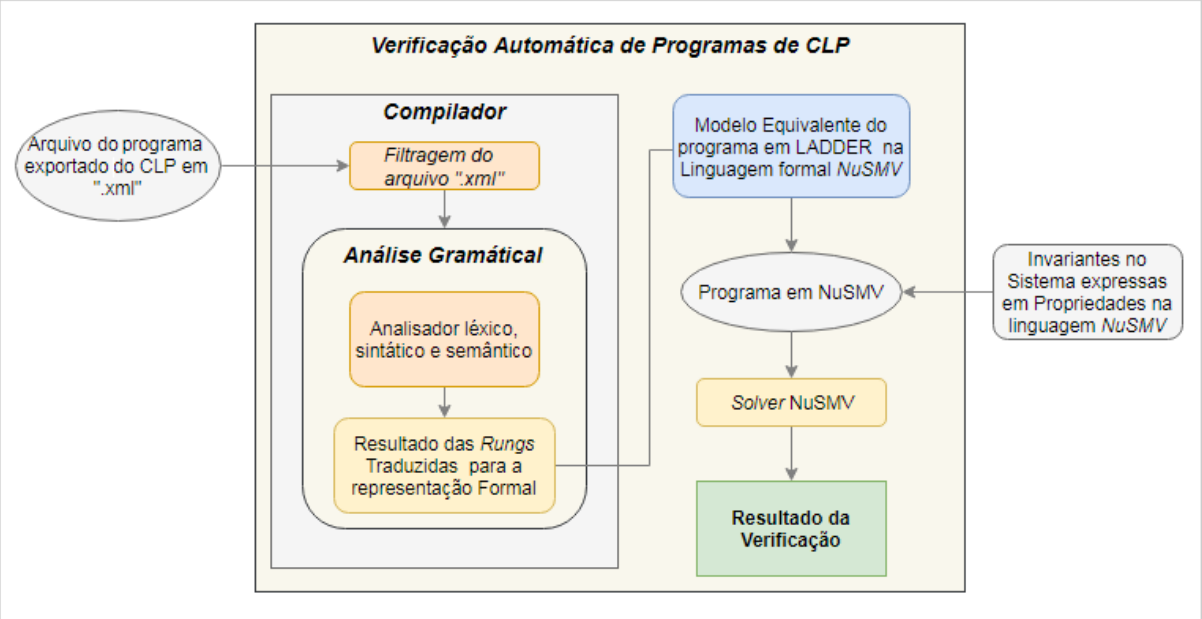

Figura 4.1: Método Proposto de Verificação Automática de CLPs

Para realizar a tradução, identificou-se que era possível exportar o programa em Ladder para um formato de extensão *.xml* (*Extensible Markup Language*). Nesse contexto, decidiu-se por realizar uma análise inicial do arquivo *.xml* e extrair do mesmo apenas as informações necessárias para a tradução e modelagem do programa.

Tratando-se da linguagem formal, estudou-se a possibilidade de se utilizar algumas linguagens que fossem suficientemente representativas, levando-se em conta os programas de CLP a serem verificados. Após análise e levantamento de referenciais bibliográficos, decidiu-se por utilizar a linguagem NuSMV.

Considerando o processo de compilação, a gramática de uma linguagem normalmente refere-se à palavras com base em suas classes gramaticais, muitas vezes chamadas categorias sintáticas. Basear as regras gramaticais em categorias ou classes gramaticais permite que uma única regra descreva muitas sentenças. Muito embora escrever a gramática seja essencialmente teórico, sua implementação requer o uso de um ambiente de desenvolvimento que permita fazer a tradução de forma automática, e este implicou na criação de um programa em linguagem Java, usando recursos como o Javacc (JavaCC, 2018), dentre outros, que resultassem diretamente no programa traduzido para a representação formal sem a necessidade de intermediações.

A Figura 4.1, exibe como foi realizado o processo de sistematização da tradução automática com o método proposto. Considerando-se as mais diversas aplicações nas quais são implementados sistemas automatizados por meio do uso de CLPs, programas desenvolvidos para automatizar sistemas complexos podem resultar em muitas linhas de código. Nas próximas seções, no presente capítulo, serão discutidas distintamente cada um dos passos exibidos na Figura 4.1.

4.3 Analisando-se o Arquivo de Exportação *.xml*

Para se realizar a análise inicial, considera-se o arquivo de exportação do programa como a entrada de dados. Este arquivo exportado possui um formato *.xml*, que tem uma estrutura bem definida, como pode ser visto na Figura 4.2, que exibe um trecho de um programa exportado do CLP.

Na Figura 4.2 é possivel notar que, na sentença destacada em cinza, entre a *tag* nomeada <*text*>, está contida a estrutura de formação das *rungs*. Como, deseja-se acessar as *rungs* do programa, bem como obter por meio destas a lista de variáveis do programa, utilizou-se, em Java uma classe denotada DOM (*Document Object Model*). O DOM, nada mais é que um analisador sintático que realiza o pré processamento do *.xml*, filtrando informações relevantes que serão utilizadas no sistema de verificação.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<RSLogix5000Content SchemaRevision="1.0" SoftwareRevision="19.01" TargetName="MainRoutine" TargetType="Routine"
TargetSubType="RLL" ContainsContext="true" Owner="LVAS, UFMG" ExportDate="Wed Apr 05 17:12:22 2017" ExportOptions="References
DecoratedData Context Dependencies ForceProtectedEncoding AllProjDocTrans">
 <Controller Use="Context" Name="aula_04">
  <DataTypes Use="Context">
  </DataTypes>
  <Tags Use="Context">
   <Tag Name="LED_verde" TagType="Alias" Radix="Decimal" AliasFor="Local:3:O.Data.2" ExternalAccess="Read/Write"/>
   <Tag Name="Sirene" TagType="Alias" Radix="Decimal" AliasFor="Local:3:O.Data.3" ExternalAccess="Read/Write"/>
   <Tag Name="LED_amarelo" TagType="Alias" Radix="Decimal" AliasFor="Local:3:O.Data.1" ExternalAccess="Read/Write"/>
   <Tag Name="Sensor_Fotoeletrico" TagType="Alias" Radix="Decimal" AliasFor="Local:1:I.Data.2" ExternalAccess="Read/Write"/>
   <Tag Name="Sensor_Capacitivo" TagType="Alias" Radix="Decimal" AliasFor="Local:1:I.Data.4" ExternalAccess="Read/Write"/>
   <Tag Name="timer_1" TagType="Base" DataType="TIMER" Constant="false" ExternalAccess="Read/Write">
    <Data>A1 E4 1B 00 B8 0B 00 00 00 00 00 00</Data>
    <Data Format="Decorated">
     <Structure DataType="TIMER">
      <DataValueMember Name="PRE" DataType="DINT" Radix="Decimal" Value="3000"/>
      <DataValueMember Name="ACC" DataType="DINT" Radix="Decimal" Value="0"/>
      <DataValueMember Name="EN" DataType="BOOL" Value="0"/>
      <DataValueMember Name="TT" DataType="BOOL" Value="0"/>
      <DataValueMember Name="DN" DataType="BOOL" Value="0"/>
     </Structure>
    </Data>
   </Tag>
  </Tags>
  <Programs Use="Context">
   <Program Use="Context" Name="MainProgram">
    <Routines Use="Context">
     <Routine Use="Target" Name="MainRoutine" Type="RLL">
      <RLLContent>
       <Rung Number="0" Type="N">
        <Text>
         <![CDATA[XIC(Sensor_Capacitivo)XIO(Sensor_Fotoeletrico)OTE(Sirene);]]>
        </Text>
```

Figura 4.2: Trecho de um programa em LD exportado do CLP RSlogix 5000 - Compact Logix L23E

Desenvolveu-se uma classe em java para que fosse realizada uma filtragem no arquivo *.xml*, tendo como saída, o conteúdo contido entre a *tag* <*text*>, ou seja a estrutura da *rung* em formato de código de exportação do programa em Ladder.

Na Figura 4.3, que exibe um pseudocódigo que descreve a execução da etapa de filtragem do programa exportado em *.xml*.

```
Leia Exported_Program_in_Ladder.xml
para i igual a zero até o comprimento do arquivo .xml acrescente um em i
    capture a expressão entre '<text>' expression '<text>' no arquivo .xml
    adicione a expressão na ListRungs na posição i
```

Figura 4.3: Pseudocódigo que realiza a Filtragem do Arquivo Exportado do CLP

4.4 Compilando a Linguagem Ladder para NuSMV

A análise gramatical constinue em uma etapa crucial que realiza tradução do arquivo filtrado, proveniente do *.xml*, para uma representação formal em NuSMV.

A compilação é constituída fundamentalmente de dois principais processos para que se possa realizar a compilação: análise e síntese (Aho, Ullman, Sethi e Lan, 1986). A parte da análise divide o programa de origem em partes constituintes e a parte de síntese constrói o programa alvo desejado.

Realizar a construção de um compilador corresponde a um exercício de projeto de engenharia. A compilação, no escopo deste trabalho, compreende a um programa de computador escrito em linguagem Java, que traduz programas escritos em LD para NuSMV a fim de prepará-los para execução do processo de verificação. Neste contexto, optou-se por realizar a compilação em duas etapas, sendo estas, 1) Análise Léxica e Sintática e 2) Análise Semântica e processo de Síntese.

Para que seja possível obter arquivo traduzido para a representação formal, de forma fidedigna à ordem de execução, considerando a ordem entre *rungs* e a ordem interna de execução da *rung* devem ser precisamente modelados. Serão abordados alguns exemplos, para melhor assimilação do processo realizado.

4.4.1 Considerando a Ordem de Execução entre *Rungs*

É proposta uma modelagem que considera a ordem na qual o programa do CLP executa as *rungs*. Sabe-se que o elemento terminal de uma *rung* constitui em uma bobina, que por consequência pode ter seu estado alterado ao longo do curso do programa. Considerando os contatos aos quais são atribuídos o mesmo endereço de memória de uma bobina, pode-se ter diferentes *status* para estes contatos considerando a ordem de leitura do programa em Ladder. Desta forma, a análise do ponto no programa onde contatos de bobinas foram posicionados, torna-se mandatório para que estes sejam modelados de acordo com o estado da bobina no ponto analisado. Para isso, a seguir, descreve-se a sequência de análise demonstrando como este processo é realizado.

- Para a análise da ordem de execução foi criada uma estrutura de registros em java para criar um histórico de variáveis. A finalidade é criar um registro em forma de uma tabela de *tags* (nome atribuido à variável) das bobinas ao longo do programa em LD.

- Quando as *rungs* são lidas pelo programa, elas são quebradas em duas *strings*, sendo uma o contexto da *rung*, composta pela estrutura de todos os contatos e

blocos que precedem a bobina. E a outra *string* composta apenas pela bobina, ou bobinas, lembrando que uma *rung* pode ser composta por uma ou mais bobinas.

- Em posse das duas *strings*, analisa-se distintamente a *string* de contexto da *rung*, checando se há alguma *tag* no contexto que esteja registrada na tabela. Em caso negativo, nenhuma das *tags* correspondentes aos contatos presentes no contexto da *rung* necessitam ser modificados. Este é o caso que ocorre quando é lida a primeira *rung* de um programa, considerando que nessa situação não há nenhum registro de bobinas na tabela.

- A partir da segunda *rung* analisada, considerando *string* correspondente ao contexto, caso uma *tag* do contexto seja encontrada na tabela, deve-se alterar o nome da variável presente na *rung* para que a mesma receba o último registro de estado da bobina. A *tag* do contato nesse caso recebe um sufixo, sendo este o de primeira ocorrência da bobina, denotado por *"_OCURRENCE_1"*,

- Em termos da *string* correspondente à bobina, quando analisada, caso não haja uma *tag* correspondente à da bobina presente na tabela, esta deve ser registrada na tabela. Por outro lado, caso a *tag* da bobina seja encontrada na tabela de registros, deve-se alterar o nome da bobina acrescentando o sufixo *"_OCURRENCE_x"*, onde o x do sufixo corresponde ao número de ocorrência de bobinas com o mesmo endereço de *tag*.

Dessa forma, esta análise constitui em um processo que realiza a atualização das *tags* correspondentes aos contatos de bobinas do programa em LD. Dessa forma, cria-se uma nova variável, renomeada da variável original, sempre que uma instrução de saída (bobina) é encontrada na sequência do programa em LD. Cada referência da variável original é substituída pela nova variável durante a análise das seguintes *rungs*. No final do modelo NuSMV, aplica-se a construção *next* para atualizar o valor da variável original, a partir da variável renomeada. Ao fazer isso, é possível modelar qualquer sequência de bobinas e contatos de bobinas que possuem o mesmo endereço de memória.

Para melhor entendimento do método, considere o exemplo do sistema apresentado na Figura. 4.4, que corresponde a um processo de Beneficiamento de Minério de

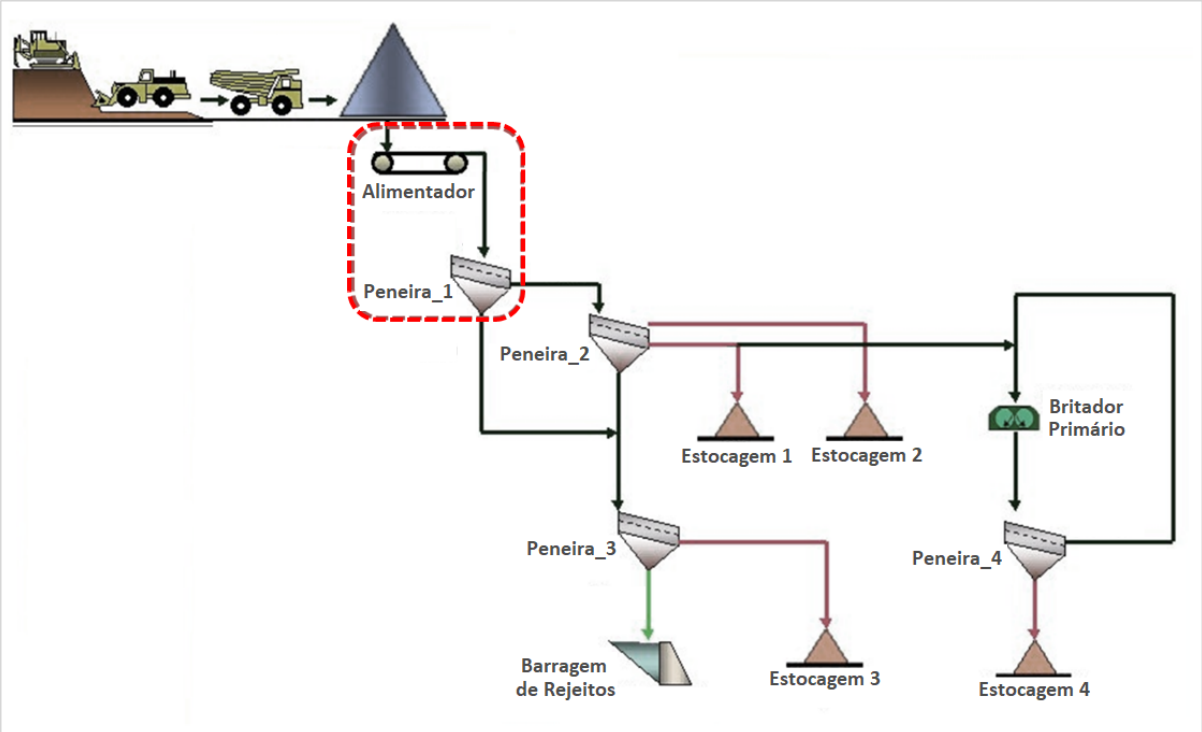

Figura 4.4: Processo de Beneficiamento do Minério de Ferro

Ferro. É analisada uma das etapas do processo (pontilhado vermelho), onde o *Alimentador* transporta o minério de ferro extraído depositando-o na *Peneira_1*. Para esta parte do processo, é imperativo que certas medidas de proteção sejam asseguradas. Entre estas, no exemplo, serão tratadas as seguintes condições:

- O sinal proveniente do *Sensor_Campo* irá garantir que a *Peneira_1* será energizada, um sinal luminoso é acionado (*Ind_Peneira_1*) para indicar o funcionamento da *Peneira_1*;
- A *Peneira_1* é energizada, e transcorrido um período de 5 segundos o *Alimentador* será energizado;
- Após 5 segundos, o *Alimentador* é energizado e um sinal de indicação luminosa é acionado para indicar *Alimentador* ligado (*Ind_Alimentador*).

Analisando a etapa do processo (pontilhado vermelho), obteve-se o correspondente LD do trecho. Na Figura 4.5 observa-se o programa escrito em Ladder, este processo será avaliado para enfatizar a importância de se modelar corretamente a ordem de execução entre *rungs*.

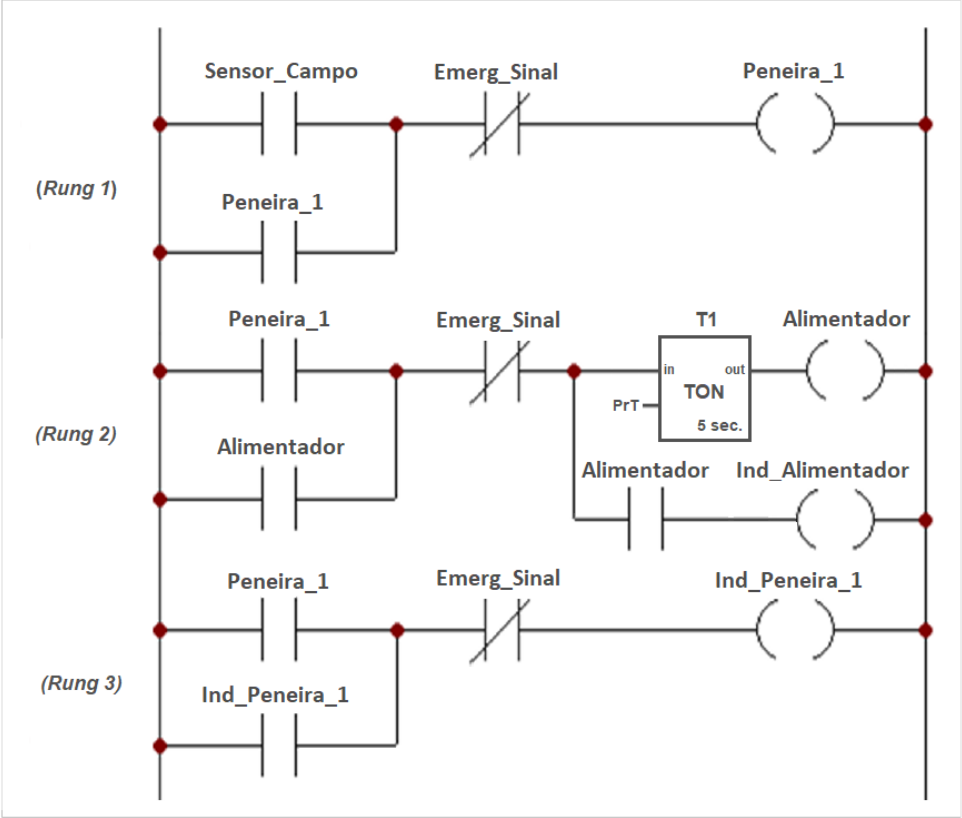

Figura 4.5: Programa LD correspondente para a etapa do processo analisado

Considerando a *Rung 1* da Figura 4.5, observa-se que a bobina de saída é assinalada por *Peneira_1*. Na mesma *rung*, o contato da bobina realimenta a sí propria para, assim que acionada, se mantenha em funcionamento. Analisando a estrutura correspondente desta linha no formato de um arquivo *.xml* exportado do CLP, obtém-se:

[[XIC (Sensor_Campo), XIC (Peneira_1)] XIO(Emerg_Sinal) OTE(Peneira_1);] **(4.1)**

A *Rung 1* acima descreve as estruturas estabelecidas pelas operações lógicas de *conjunção*, caracterizada pelo sequenciamento de contatos e *disjunção*, estabelecida pelos contatos que estão compreendidos entre a estrutura "[,]"presentes na *rung*. Quando essa mesma *rung* é modelada em uma representação formal em linguagem NuSMV, obtém-se a expressão (4.1).

Peneira_1_OCCURRENCE_1 := (Sensor_Campo | Peneira_1) & !Emerg_sinal; **(4.2)**

Note-se que no lado esquerdo do símbolo *":="* o estado futuro da bobina *Peneira_1* é renomeada, dando origem a uma nova variável, com sufixo "_OCCURRENCE_1". O lado direito mantém a variável original.

Desta forma, considerando a ordem de execução entre *rungs*, é mandatório que, todos os contatos de bobinas que estão localizados em *rungs* posteriores à atribuição da bobina, devem ser modelados como uma nova variável, conforme demonstrado em (4.3) pelo modelo NuSMV correspondente da *Rung 3*.

$$\text{Ind\_Peneira\_1\_OCCURRENCE\_1: = (Peneira\_1\_OCCURRENCE\_1 | Ind\_Peneira\_1) \& !Emerg\_Sinal;} \quad \textbf{(4.3)}$$

Ao final da modelagem das *rungs*, aplica-se à última ocorrência da variável o recurso *next* do NuSMV, conforme apresentado em (4.4).

$$\text{next(Peneira\_1) := Peneira\_1\_OCCURRENCE\_1;} \quad \textbf{(4.4)}$$

O recurso *next* atribui o valor da variável à direita do sinal ":="para o estado futuro da variável entre parênteses.

Desta forma, a ordem de execução entre as *rungs* é modelada e, portanto, o modelo correspondente em NuSMV. Um outro exemplo de programa escrito em LD, nas quais a ordem de execução deve ser considerada, pode ser visto no caso hipotético da Figura 4.6. Nas *Rungs 1* e *2*, observa-se que o mesmo endereço de referência é atribuído a duas bobinas distintas. Devido ao fato do estado futuro da bobina na *Rungs 1* não ser equivalente ao estado futuro da bobina na *Rungs 2*, fica evidente que, apenas considerar o estado futuro das bobinas definitivamente não é suficiente.

Assim sendo evidencia-se que modelar bobinas implementando somente o operador *next*, sem necessariamente avaliar a *rung* na qual a bobina esta posicionada, em algumas situações, não irá condizer com a estrutura de execução real de programas em Ladder. Para demonstrar o caso da Figura 4.6, é exibida a modelagem do sistema em NuSMV na Figura 4.7.

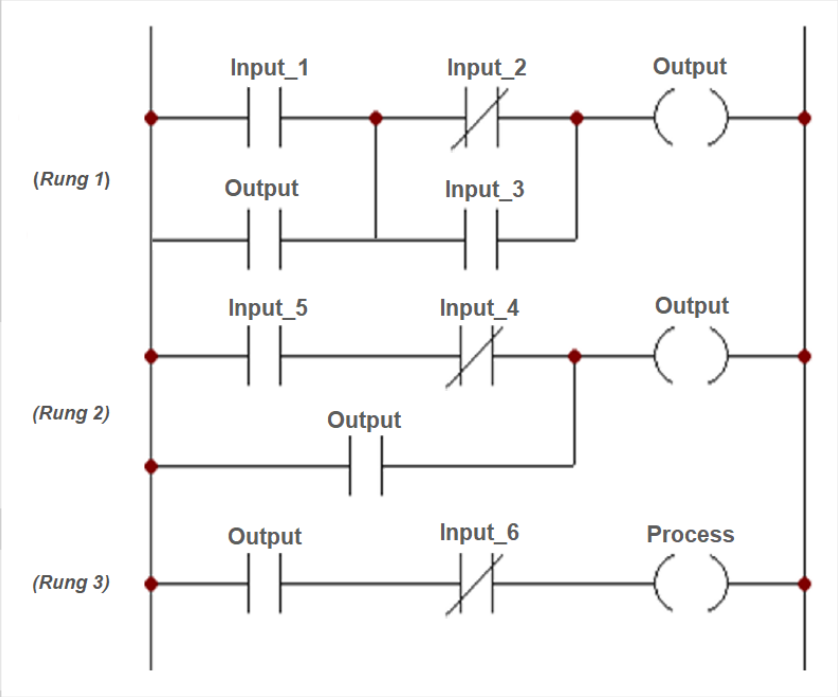

Figura 4.6: Análise da ordem de execução entre *Rungs*

```
Output_OCCURRENCE_1 := (Input_1 | Output) & (!Input_2 | Input_3);
Output_OCCURRENCE_2 := ((Input_5 & !Input_4) | Output_OCCURRENCE_1);
Process_OCCURRENCE_1 := Output_OCCURRENCE_2 & !Input_6;
next(Output) :=  Output_OCCURRENCE_2;
next(Process) := Process_OCCURRENCE_1;
```

Figura 4.7: Modelo de tradução NuSMV resultante para o exemplo da Figura 4.6

Observa-se que, a bobina *Output*, por possuir duas incidências é renomeada *Output_OCCURRENCE_1* e *Output_OCCURRENCE_2*. Sendo assim, os contatos da bobina *Output* da *Rung 2* são renomeados *Output_OCCURRENCE_1* e os contatos da *Rung 3* são renomeados *Output_OCCURRENCE_2*. Ao final do programa em NuSMV aplica-se o recurso *next* para de atualizar o estado futuro das variáveis.

4.4.2 Ordem de Execução Interna de uma *Rung*

Modelar uma *Rung* considerando sua ordem interna de execução é um detalhe que passa despercebido quando se trata da modelagem formal de programas em linguagem Ladder.

Para melhor ilustrar a importância de se considerar a ordem de execução dentro de uma *rung*, considere a Figura 4.8, que descreve o comportamento de um sistema

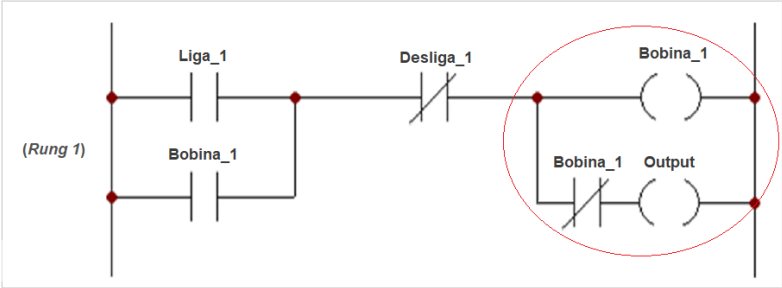

Figura 4.8: Exemplo hipotético de um programa em LD para análise da ordem execução interna de uma *Rung*

```
Bobina_1_OCCURRENCE_1 := (Liga_1 | Bobina_1)& Desliga_1;

Output_OCCURRENCE_1:= Bobina_1_OCCURRENCE_1;

next(Bobina_1):= Bobina_1_OCCURRENCE_1;

next(Output):= Output_OCCURRENCE_1;
```

Figura 4.9: Modelo de Tradução NuSMV equivalente para a *Rung 1* do Programa da Figura 4.8

hipotético, onde a *Rung 1* possui uma bifurcação (circulada em vermelho), sendo o seguimento superior da bifurcação responsável por controlar o *status* da *Bobina_1* e o seguimento inferior constituído por um contato da própria *Bobina_1* que alimenta a bobina *Output*.

Considerando a estrutura apresentada na *Rung 1* da Figura 4.8, observa-se que, através da mesma *rung*, duas bobinas são alimentadas. Portanto, quando a *Bobina_1* é energizada, a bobina *Output* deve ser energizada de acordo com as condições de operação do CLP. Considerando a ordem de execução, na estrutura da *Rung 1* a bobina *Output* será ativada no mesmo ciclo de varredura da *Bobina_1*.

Essa estrutura requer uma análise minuciosa, que exige um conhecimento detalhado do modo como os CLPs executam a estrutura da *rung*. A modelagem correta, portanto, para a *Rung 1* é vista na Figura 4.9.

Observe que a *Rung 1* resultou em duas *rungs* no modelo NuSMV. A modelagem realizada adequadamente, manipula a ordem de execução dentro da *rung*, garantindo que o contato da *Bobina_1*, que precede a bobina *Output*,é denotado no estado correspondente à variável renomeada da *Bobina_1*. Na *Rung 1* também modela-se a estrutura interna da lógica de contatos do programa.

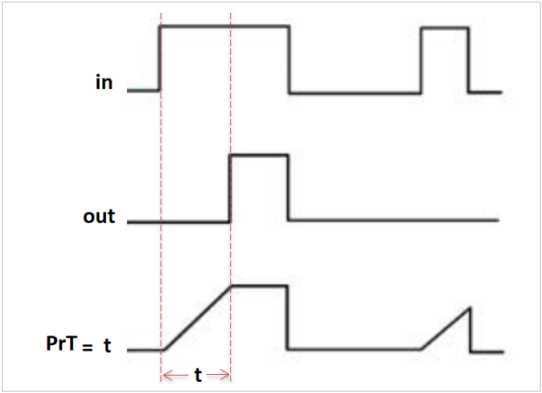

Figura 4.10: Diagrama Operacional de um Temporizador TON

4.4.3 Tradução de Blocos de Temporização

Analisando a *Rung 2* da Figura 4.5, pode-se observar a presença de um bloco de temporização. O bloco de temporização TON (temporizador de atraso na energização - *ON delay*) tem o diagrama de operação como mostrado na Figura 4.10, onde PrT, corresponde ao *preset* (tempo pre-determinado de contagem do temporizador).

Retomando o sistema apresentado na Figura 4.5, será analisada a estrutura apresentada na *Rung 2*. Observa-se que, por meio da mesma *rung*, duas bobinas são energizadas. Portanto, quando a bobina *Alimentador* é energizada, o indicador luminoso *Ind_Alimentador* deve ser energizado de acordo com as condições de operação descritas. Considerando a ordem de execução do CLP, na estrutura da *Rung 2*, o *Ind_Alimentador* será ativado no mesmo ciclo de varredura que a bobina *Alimentador*.

Para se modelar a estrutura da *rung* composta por duas bobinas terminais, considera-se a ordem interna de execução desta. No entanto a tradução de *rungs* com mais de uma bobina terminal, na modelagem proposta, resulta em duas ou mais *rungs* distintas, cujas bobinas recebem o resultado da combinação de contatos e/ou blocos que as precedem. A modelagem resultante para a *Rung 2* é vista na Figura 4.11. Observe que, a *Rung 2* resultou em três *rungs* no modelo NuSMV. A modelagem realizada adequadamente, manipula a ordem de execução dentro da *rung* e também modela como o temporizador TON é inserido na estrutura do programa. Para implementar o modelo da lógica temporal do TON, visto na Figura 4.10, uma abstração do modo de operação da temporização foi realizada discretizando o tempo de contagem. Foi possível transformá-lo em um diagrama de estados representado pela estrutura do *case* do NuSMV, como visto na Figura 4.12.

```
TON_1_OCURRENCE_1 := (Peneira_1_OCCURENCE_1 | Alimentador ) & !Emerg_Sinal;

Alimentador_OCURRENCE_1 := T1.saida;

Ind_Alimentador_OCURRENCE_1 := Alimentador_OCURRENCE_1;

next(TON_1) := TON_1_OCURRENCE_1;

next(Alimentador) := Alimentador_OCURRENCE_1;

next(Ind_Alimentador) := Ind_Alimentador_OCURRENCE_1;
```

Figura 4.11: Modelo de Tradução NuSMV equivalente para a *Rung 2* do Programa em LD do Processo de Beneficiamento de Minério de Ferro

```
MODULE TON (entrada)

VAR
     estado : {pausado, contando,atuado};

 ASSIGN
     init(estado):= pausado;
     next(estado):= case
       estado = pausado & !entrada:{ pausado };
       estado = pausado & entrada: {contando};
       estado = contando & entrada: { atuado, contando};
       estado = contando & !entrada: { pausado };
       estado = atuado & entrada: { atuado };
       estado = atuado & !entrada: { pausado };
  esac;

 DEFINE
     saida:= estado = atuado;
```

Figura 4.12: Módulo para um Temporizador TON modelado em NuSMV

Além de modelar o bloco de contagem de temporizadores, também é necessário declarar o módulo na lista de variáveis no programa em NuSMV, nesse caso o módulo será declarado recebendo como parâmetro o estado futuro da bobina TON_1. Assim, de acordo com a Figura 4.12, o módulo de temporização TON recebe *TON_1_OCCURRENCE_1* como parâmetro, no módulo visto na Figura 4.12, este parâmetro é denominado de *entrada*.

Dentro do módulo de temporização, as transições de estado do temporizador são realizadas e seu *status* é referido ao contato denominado *T1.saida*. Desta forma, tanto a ordem de execução intrínseca da *rung*, bem como a implementação de um bloco de temporização.

4.4.4 Análise Léxica e Sintática

Para verificar a sintaxe do programa de entrada, o compilador precisa comparar a estrutura do programa com uma definição para a linguagem. Isto exige uma definição formal apropriada, um mecanismo eficiente para testar se a entrada atende ou não esta definição e um plano de como proceder caso haja uma entrada ilegal. As ferramentas que automatizam a produção de analisadores léxicos (*scanners*) e analisadores sintáticos (*parsers*) aplicam resultados da teoria de linguagem formal. (COOPER; TORCZON, 2012)

De acordo com um conjunto de regras da gramática é realizada a análise léxica e sintática de forma recursiva, dessa maneira o resultado final da tradução da *rung* é obtido por meio da composição das informações contidas na *rung* em LD original. Exemplificando, considere a *rung* (4.5)

[XIC (Sensor_1) XIO(Alarme_1) OTE(Motor_1);] **(4.5)**

O primeiro passo para entender a sintaxe desta sentença é identificar os *tokens*, sendo estes o conjunto de caracteres de entrada do programa classificados na gramática. Em um compilador, esta tarefa é atribuída ao passo denominado *scanner*. O *scanner* captura um fluxo de *tokens* (ou caracteres) e os converte em um fluxo de palavras classificadas — ou seja, pares na forma (i, g), onde i é a instrução na gramatica e g sua grafia. Um scanner converteria a sentença do exemplo no seguinte fluxo de palavras classificadas:

(*xic instruction*, Sensor_1) (*xio instruction*, Alarme_1) (*ote instruction*, Motor_1) (*final of the rung*, ";")

No próximo passo, o compilador corresponde à construções que especificam a sintaxe da linguagem de entrada. Por exemplo, um conhecimento funcional do programa em Ladder poderia incluir as regras gramaticais da Figura 4.13.

A derivação da lista de *rungs* proveniente do arquivo filtrado, denominada *ListRungs* da Figura 4.13, é lida *rung* por *rung*. A cada passo, é reescrito um termo na *rung*,

```
READ ListRungs
DEFINE TOKEN :
{
 <LPAR:      "(">
| <RPAR:      ")">
| <LBRA:      "[">
| <RBRA:      "]">
| <XIC:       "XIC">
| <XIO:       "XIO">
| <OTE:       "OTE">
| <RUNG:    "Rung">
| <TAG:       <VARIABLE> ( "." <VARIABLE> )* >
| <VARIABLE:    ["a"-"z","A"-"Z","_"] ( ["a"-"z","A"-"Z","0"-"9",",","?","_"] )* ("[" (["0"-"9"])* "]")? >}

BEGIN CASE
            CASE<XIO> <LPAR> TAG <TAG> <RPAR>
                        xio instruction = "( ! " + TAG + " )"
            CASE <XIC> <LPAR> TAG <TAG> <RPAR>
                        xio instruction = TAG;
            CASE <OTE> <LPAR> TAG <TAG> <RPAR>
                        ote instruction = TAG + " := "
;
...
END CASE
```

Figura 4.13: Detalhamento de parte das Regras Gramáticais de Tradução de LD para NuSMV

substituindo-o por um lado direito que pode ser derivado desta regra (observe um trecho da gramática desenvolvida na Figura 4.14). O primeiro passo usa a *xic instruction* para substituir na *rung*. O segundo, a relação entre instruções denominada *input instruction*, estabelecendo a relação entre as instruções de entrada. O terceiro substitui *xio instruction*, enquanto o passo final reescreve a *ote instruction*. Neste ponto, a *rung* gerada pela derivação corresponde ao fluxo de palavras categorizadas produzidas pelo *scanner*.

O processo de encontrar automaticamente as derivações é chamado *parsing* (ou análise sintática), definindo a forma com a qual o compilador desenvolvido usa para analisar sintaticamente o programa de entrada.

Neste caso, executar a análise sintática da *rung* r, dada a gramática, consiste em verificar se r esta contida na gramática G. Sendo assim determinar a estrutura sintática de r em G.

4.4.5 Análise Semântica e processo de Síntese

O compilador desenvolvido precisa realizar uma análise típica de um *scanner* ou um *parser*. Este estudo é, ou realizado ao longo da análise sintática, ou em um passo posterior que é denominado análise semântica, ou de “elaboração semântica”.

A saída da etapa de análise léxica e sintática não implementa em sua essência a consideração da forma de execução de *rungs* em um CLP, basicamente traduz em termos

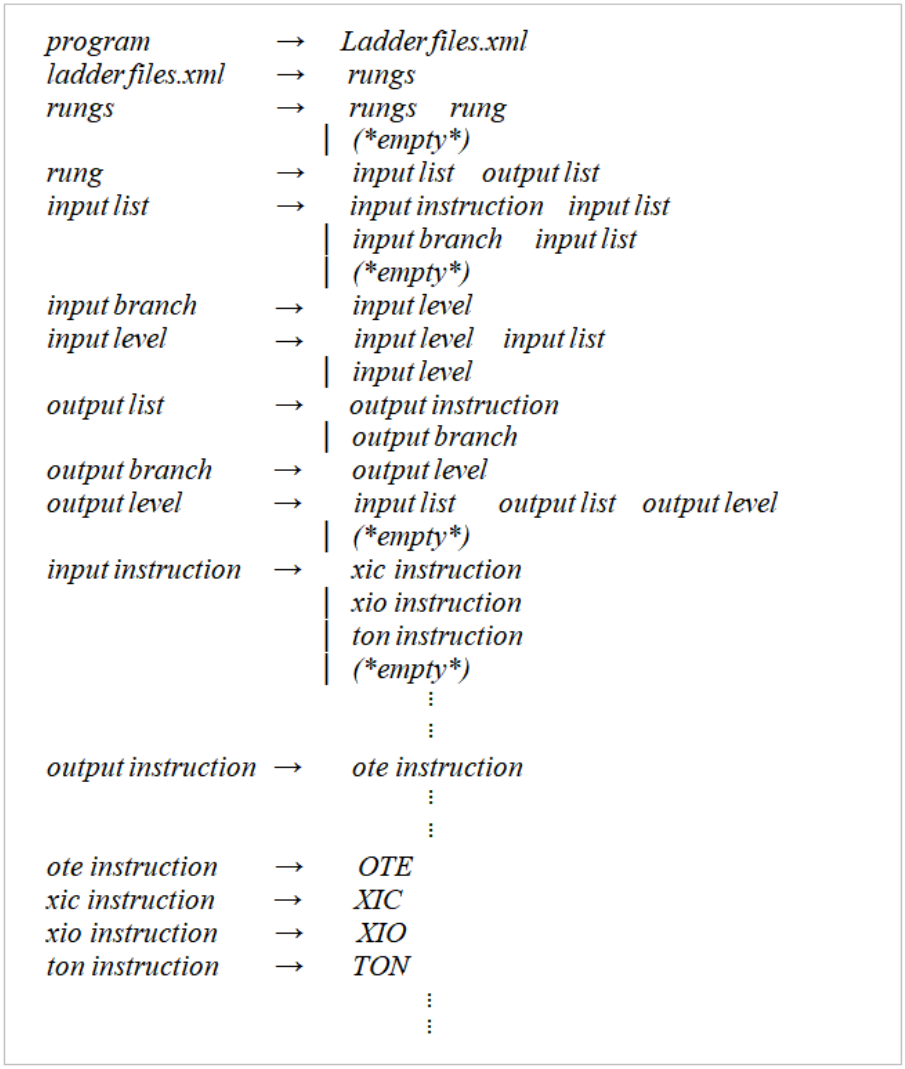

Figura 4.14: Trecho da Gramática LD para NuSMV

literais as *rungs* provenientes do programa filtrado. A etapa de análise semântica leva em conta o contexto da *rung* e implementa a forma de execução de um programa em LD por meio da adequação da ordem de execução interna.

Nessa lógica, a Figura 4.15 exibe o pseudocódigo que descreve como foi feita a implementação da análise considerando o contexto de execução do CLP escrito em LD. A implementação do sistema foi realizada em uma plataforma de desenvolvimento escrita em linguagem Java. Assim como realiza-se a análise semântica, também é obtida a estrutura do programa alvo em NuSMV. Esta etapa correspondente a síntese do programa, que a partir da análise semântica constrói o programa alvo. A seção 4.5 detalha como a estrutura final do programa é salva em um arquivo *.smv*.

Observe que, a Figura 4.15, exibe um trecho da tradução, em formato de pseudocódigo, do sistema implementado. Na linha 4, é criada uma estrutura de tabela em

```
1   List de rungs para NUSMV file
2   faça rungs igual a ListRungs  // ListRungs é proveniente da etapa de análise sintática
3   faça list igual a novo array of strings
4       faça sequence igual a new map of strings
5       faça timers igual a new array of strings
6       inicialize newVirtual igual a zero
7       inicialize temp igual a zero
8       faça rung igual a new string
9       para rung igual a rungs até o comprimento de rungs
10              name é igual ao lado esquerdo da rung splitted no caractere ":="
11              right é igual ao lado direito da rung splitted no caractere ":="
12              se houver o caratere `TON_` em name
13                      some um em newVirtual
14                      string virtu é igual `Virtual` + newVirtual
15                      substitua name por virtu em right
16                      adicione virtu na lista de tag
17                      string timer é igual a `Timer_` + newVirtual + `: TON (` + virtu + `_1);`
18                      adicione timer na lista de tag
19                      name é igual a virtu
20              se sequence não contém name
21                      newName é igual name + `_OCCURRENCE_1`
22                      adicione name em sequence
23              Senão
24                      string suffix é igual ao comprimento de name adicionado de um
25                      string newName é igual name + `_OCCURRENCE` + suffix
26              para string key igual a sequence até o comprimento de sequence
27                      se o valor de key for maior do que zero
28                              string newKey é igual a new Name
29                              substitua key por newKey em right
30              para string tim igual a timers até o comprimento de timers
31                      se houver `TON` em right
32                              temp é igual a temp adicionado de um
33                              string temp_on é igual a `T` + temp + ".saida"
34                              substitua tim por temp_on em right
35              adicione newName na lista de tag
36              adicione newName + `:= ` + right na lista de list
37      para string key igual a sequence até o comprimento de sequence
38              lista de string names é igual a key
39              adicione `next(` + key + `) := ` + names na posição zero em list
40      para integer i igual a um até o comprimento de names adicione um em i
41              Adicione `next(` + names na posição i subtraida de um + `) := ` + names na posição i em list
```

Figura 4.15: Pseudocódigo de Implementação da etapa de Análise Semântica Sensível ao Contexto.

java, para realizar o registro de bobinas do programa e para possibilitar a implementação da ordem de execução do programa em Ladder. Da mesma forma cria-se uma lista de temporizadores, na linha 5, para que estes sejam contabilizados. Cada uma das *rungs*, provenientes da etapa de análise sintática é tratada separadamente analisando-se seu contexto. Para isto, observa-se que nas linhas 10 e 11 da Figura 4.15 a *rung* recebida é separada em duas partes distintas, sendo estas o lado esquerdo do sinal ":="correspondente à bobina e o lado direito que contém a lógica da *rung*.

Esta separação permite que a ordem de execução entre *rungs* e interna de uma *rung* seja realizada. Na linha 12, busca-se pelo caractere reservado "TON_", que indica a existência de um contato de temporização na lógica da *rung*. Entre as linhas 13 e 19 a *rung* é manipulada para se renomear a variável pertencente ao temporizador para que a mesma esteja em um formato compatível com o código NuSMV.

Em seguida, analisa-se os nomes das variáveis (*tag*) presente na lógica da *rung*, buscando-se por variáveis presente na lógica da *rungs* que possam ser contatos de bobinas presentes na tabela criada na linha 4, para que a lógica seja ou não modificada. Na sequência analisa-se o lado da *rung* pertencente à bobina, caso não haja registro da bobina na tabela criada, acrescenta-se o sufixo "_OCCURRENCE_1"na *tag* da bobina. Caso haja registo da bobina na tabela, soma-se uma unidade no último caractere do sufixo, podendo este resultar no sufixo "_OCCURRENCE_2", "_OCCURRENCE_3", ..., "_OCCURRENCE_n".

Assim, na linha 36, as *rungs* modificadas são armazenadas na nova lista que leva em conta a ordem de execução entre e interna das *rungs*. Entre as linha 37 e 41, o recurso *next* do NuSMV é implementado para se modelar o estado futuro das variáveis correspondentes às bobinas.

4.5 Programa em NuSMV e Invariantes do Sistema a ser Verificado

A etapa que precede a verificação do modelo constitui em, adicionar ao programa já traduzido para representação formal as invariantes ou propriedades do sistema. No programa em NuSMV, as invariantes são descridas dentro do campo *SPECS* que, no contexto de verificação podem ser expressas em lógica temporal CTL ou LTL.

Conhecer o sistema ao qual deseja-se verificar torna a descrição das propriedades uma tarefa mais intuitivas. Quando se sabe quais condições devem ser satisfeitas pelo sistema automatizado, para transformar tais propriedades em especificações CTL ou LTL, deve-se conhecer a estrutura básica de operadores de lógica temporal.

4.5.1 Descrição das Invariantes do Sistema a ser Verificado

Considerando algumas descrições de propriedades de segurança do trecho analisado para o processo de Beneficiamento de Minério de Ferro, visto na seção 4.3.1, na sequência descreve-se formalmente suas propriedades. A primeira especificação descreve o seguinte comportamento:

- O sinal proveniente do sensor de campo *Sensor_Campo* irá garantir que a

Peneira_1 será energizada, um sinal luminoso é acionado (*Ind_Peneira_1*) para indicar o funcionamento da *Peneira_1*.

Essa propriedade expressa uma condição de segurança, na qual pode-se inferir que é estabelecida entre a *Peneira_1* e o *Ind_Peneira_1* uma relação de implicação. Em outras palavras, é sempre verdade que, uma vez que a *Peneira_1* foi acionada, o sinal luminoso *Ind_Peneira_1* também será acionado. Para tal, utiliza-se a expressão *AG p*, onde esta apresenta uma situação na qual a propriedade *p* sempre é válida para o sistema.

$$\text{AG } (\textit{Peneira\_1} \ \& \ \textit{Ind\_Peneira\_1})$$

Por outro lado, explorando um pouco mais as propriedades descritas para o sistema, observe a descrição a seguir:

- A *Peneira_1* deve permanecer ligada por um período de 5 segundos antes que o *Alimentador* seja energizado.

Levando-se em conta que, na a descrição acima existe um fator temporal que condiciona a ligação do *Alimentador* apenas após transcorridos 5 segundos da *Peneira_1* ter sido enegizada, pode-se, para esta situação, descrever a seguinte propriedade formal:

$$\text{AG } (\textit{Peneira\_1} \Rightarrow \text{AF } \textit{Alimentador})$$

Esta propriedade diz que, é sempre verdade que estando a *Peneira_1* energizada, em algum momento no futuro (AF) o *Alimentador* será energizado. Nesse caso, não descreve uma lógica temporal em termos de ciclos de varredura, entretanto se, em algum instante de tempo a condição for satisfeita, não será retornado um contra-exemplo indicando que a propriedade foi infringida. Explorando algumas possibilidades de descrição de propriedades, pode-se, também descrever esse comportamento de uma forma alternativa, como:

$$\text{AG } (\ !\ \textit{Alimentador}\ \text{U}\ \textit{T1.atuado})$$

Onde observa-se o operador *'até que'* (U), que nessa situação opera como um fator condicionante. Em outras palavras a propriedade diz que, o *Alimentador* permanece desativado, simbolizado pelo sinal '!' que o precede, até que *TON.atuado*, ou seja, até que o temporizador atinja o status de fim de contagem, acionando assim o *Alimentador*.

4.6 Programa resultante em NuSMV

Como etapa final do sistema de verificação automática, os resultados provenientes da etapa de análise semântica sensível ao contexto são utilizados para compor o programa em NuSMV. A estrutura composicional do programa *.smv* esta condicionada aos blocos utilizados no programa em LD.

```
1   adicionar `MODULE main` ao NuSMV file
2   adicionar `VAR` ao NuSMV file
3   para a string t igual a tag até o comprimento de tag
4       se houver `DI` em t
5               adicionar `  ` + t ao NuSMV file
6       senão
7               adicionar ` ` + t + `: boolean;` ao NuSMV file
8   adicionar `INIT` ao NuSMV file
9   para a string t igual a tag até o comprimento de tag
10      adicionar ` ` + t + `= FALSE &` ao NuSMV file
11  adicionar `ASSIGN` ao NuSMV file
12  para string t1 igual a list até o comprimento de list
13          adicionar ` ` + t1 ao NuSMV file
14  adicionar `SPEC` ao NuSMV file
15  adicionar o conteúdo do SPECS.txt ao NuSMV file
16  se newVirtual for maior do que zero
17          adicione `MODULE TON(input)` ao NuSMV file
18          adicionar o conteúdo do MODULE_TON.txt ao NuSMV file
```

Figura 4.16: Pseudocódigo que salva o programa modelado em um arquivo NuSMV

Assim sendo, a Figura 4.16 exibe, por meio de um pseudocódigo, como a composição do programa em NuSMV é salva em um arquivo *".smv"*. Como pode-se observar na Figura 4.16, os blocos que compõe o programa NuSMV são salvos em um arquivo e a estrutura do programa é formada.

Como é possível identificar na Figura 4.16, são importados dois arquivos (linhas 15 e 18). O arquivo importado da linha 15 corresponde às propriedades descritas em uma representação formal CLT ou LTL, desenvolvida a partir do conhecimento sobre o sistema. Já a linha 18, corresponde ao Módulo de temporização, denominado TON.

Capítulo 5

Verificação de um Sistema utilizando o Método Proposto

Neste capítulo será detalhada a verificação de modelos aplicada à programas do CLP RSlogix 5000 - Compact Logix L23E (Allen Bradley, 2013), utilizando o modelo gerado a partir do sistema de verificação desenvolvido. Para tal, é utilizado um modelo de programa proveniente de uma aplicação real que utiliza o CLP como dispositivo de automatização do sistema.

É abordado um estudo de caso, com o objetivo de explorar a verificação de um sistema em um contexto industrial. O estudo constitui em um sistema de automatização de equipamentos de pátio em uma indústria de mineração. As propriedades do sistema são exploradas, para que se evidencie que o sistema traduzido automaticamente performa tal como deseja-se, dadas as invariantes do sistema.

Nas situações abordadas nesse capítulo as especificações de funcionamento podem descrever as sequências lógicas indesejadas, a lógica sequencial ou as propriedades estruturais do sistema. A partir da descrição das especificações ou do conhecimento do sistema, é feita a construção da propriedade formal, escrita em CTL ou LTL. O resultado da verificação indica se o modelo do sistema está em conformidade com as especificações ou não. Em caso negativo, três são as hipóteses que devem ser levantadas. Primeiramente, uma fonte de não conformidade é o erro de modelagem do sistema. Neste caso, deve-se refinar ou corrigir o modelo do sistema para se fazer uma nova verificação. A segunda hipótese é a existência de um erro do programa de aplicação captado na modelagem. Neste caso, deve-se partir para um reprojeto do programa antes de uma nova verificação. A terceira hipótese corresponde a uma propriedade que foi incorretamente escrita, ou expressa requisitos muito restritos para o sistema. Neste caso, a especificação deve ser

reescrita para uma nova verificação.

5.1 Descrição do Sistema de Automação

Este estudo de caso constitui em um exemplo de aplicação, realizando a representação do contexto do sistema abordado utilizando-se os blocos em linguagem LD explanados e discutidos no presente trabalho.

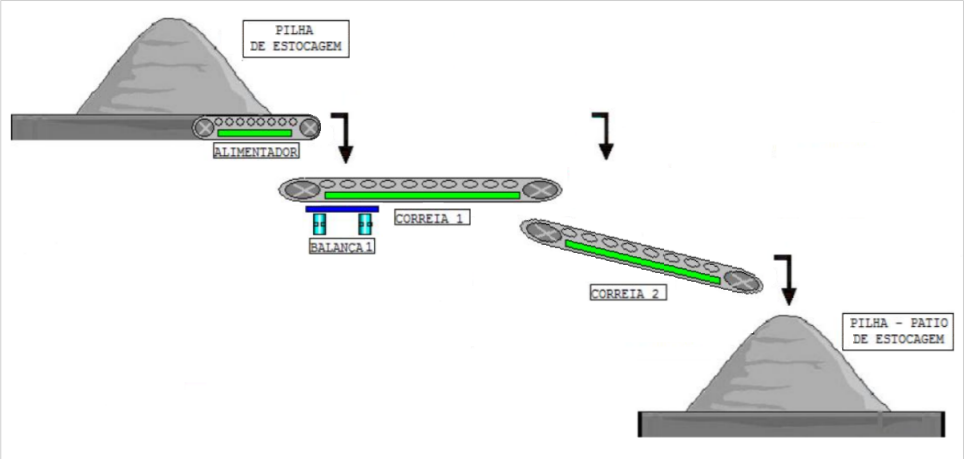

Figura 5.1: Sistema de Automação para Equipamentos de Pátio de uma Mineradora

Considerando o sistema de equipamentos de pátio, foram tratados os acionamentos dos equipamentos ilustrados conforme visto na Figura 5.1.

Na sequência serão detalhadas informações importantes acerca do processo da Figura 5.1. As primeiras propriedades a serem descritas dizem respeito a sequência de acionamento dos equipamentos de pátio. Como o sistema constitui em uma cadeia de correias transportadoras, sendo estas responsáveis por transportar o minério da *Pilha de Estocagem* para o *Pátio de Estocagem*, existe todo um sistema de sequenciamento que assegura o funcionamento do sistema automatizado de transporte tal como requerido. Projetando o cenário do sistema tem-se as seguintes especificações:

1. Inicialmente são energizados os sistemas de *Alarme Sonoro*. O sistema de alarme permanece acionado por 10 segundos, notificando os operadores de campo que o sistema de transporte de correias entrará em funcionamento;

2. Em sequência é acionada a *Correia_2*, o sistema de acionamento executa o processo de *start up* dos motores que movimentam a correia. Uma medida

de segurança para garantir que a correia foi efetivamente acionada, e que não há problemas no sistema elétrico da correia, constitui na realimentação do sistema de controle (CLP) com um sinal proveniente do campo. Este sinal é enviado por um sensor que detecta o movimento da correia. Enquanto a *Correia_2* encontra-se em funcionamento um sistema de indicação de segurança (*Ind_Correia_2*) alerta o estado de funcionamento desta;

3. Dado que a *Correia_2* entrou em estado de funcionamento, esta energiza o temporizador *T1*. O temporizador inicia o processo de contagem para garantir que, apenas após transcorridos 10 segundos a *Correia_1* será energizada. O funcionamento da *Correia_1* também é sinalizada por uma indicação de segurança. Após acionada, a *Correia_1*, energiza o temporizador *T2*, dessa forma a energização do alimentador será postergada em 10 segundos;

4. A última etapa do processo de acionamento sequencial energiza o *Alimentador* por meio do contato de *T2*. O *Alimentador*, por sua vez inicializa o carregamento de minério na *Correia_1*.

Além do acionamento sequencial, como propriedade de funcionamento, o sistema também possui especificações de propriedades de intertravamento e defeitos. A especificações a seguir caracterizam-se como propriedades de intertravamento;

1. Em caso de desligamento por bloqueio de manutenção em uma ou mais correias do sistema, deve-se assegurar que as correias precedentes serão desenergizadas. Por exemplo, caso seja requerida a manutenção da *Correia_2*, a *Correia_1* e o *Alimentador* devem ser desenergizadas.

Já as propriedades de defeitos, são possíveis falhas que podem conduzir o sistema a um funcionamento inapropriado ou incorreto. Sendo estas:

1. Em caso de detecção de rompimento em uma das correias, todo o sistema deve ser desenergizado;

2. Em caso de detecção de sucata ferrosa (fragmento metálico) na correia, as correias devem ser paradas para inspeção. Isso se deve ao fato de que, o material detectado na correia pode levar à quebra de equipamentos que sucedem o processo de beneficiamento do minério;

3. Dentro do projeto do sistema há proteções para o acionamento elétrico dos motores das correias, aqui serão consideradas proteção de sobrecorrente e temperatura nos motores;

4. Caso seja necessário realizar alguma intervenção no processo devido à qualquer eventual falha, o sistema pode ser acionado e desacionado local e remotamente;

5. A não lubrificação nos redutores de velocidade das correias pode causar a quebra do sistema de redução. Caso seja detectada falta de lubrificação o processo deve ser parado para manutenção.

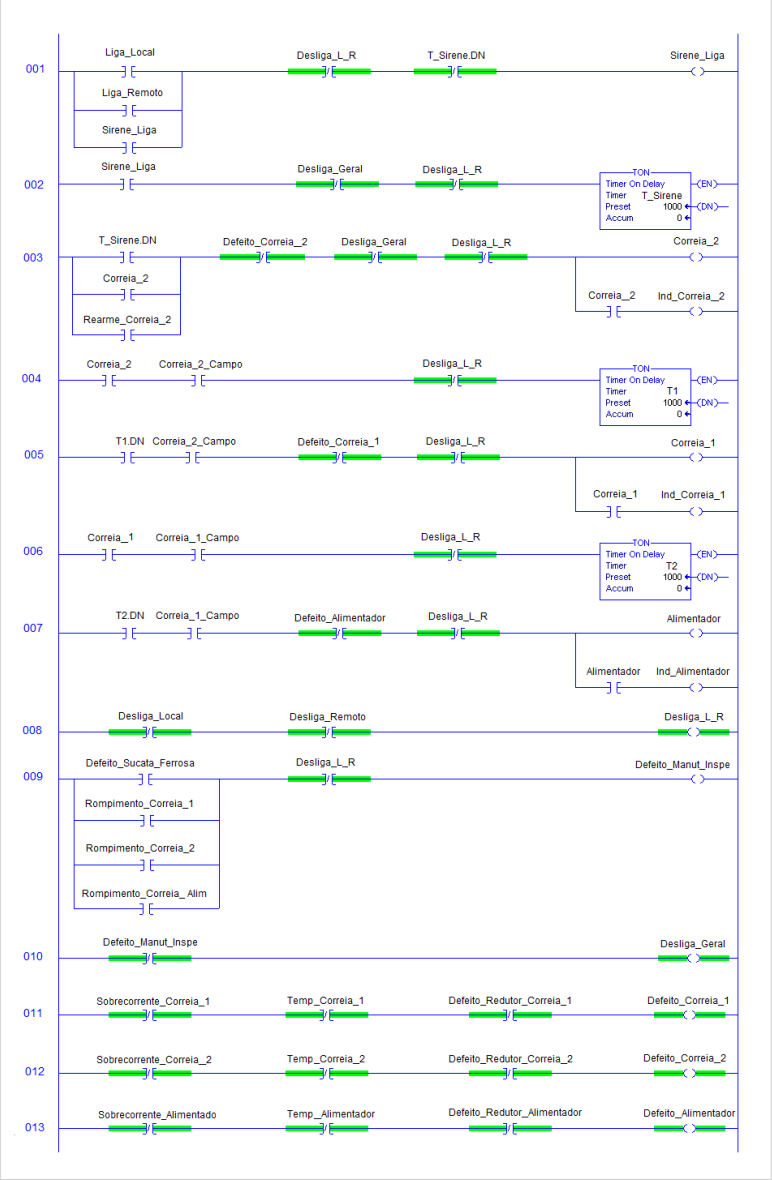

Figura 5.2: Programa em LD do Sistema de Automação para Equipamentos de Pátio de uma Mineradora.

Dadas as propriedades de sequenciamento, intertravamento e defeito do sistema, foi gerado o programa em LD do mesmo. Na Figura 5.2, pode ser visto o resultado final do programa em linguagem Ladder correspondente.

5.2 Verificação do Sistema

5.2.1 Descrição das Invariantes

Considerando as propriedades do sistema, foram elaboradas as descrições das invariantes em uma representação formal em lógica CTL ou LTL. A princípio são descritas as propriedades em função da lógica de sequenciamento de acionamento dos dispositivos, como segue.

- Acionamento do sistema alerta sonoro, *Sirene_liga*.

LTLSPEC ((Liga_Local | Liga_Remoto) ⇒ Sirene_Liga **U** T_Sirene.saida)

No contexto da especificação acima pode-se se concluir que, uma vez que *Liga_Local* ou *Liga_Remoto* sejam acionados, a *Sirene_Liga* será energizada e permanecerá em funcionamento até que *T_Sirene.saida* seja verdadeiro. Essa propriedade, em específico, trata-se de uma expressão em LTL por conter o operador temporal *until* (U).

- Acionamento da *Correia_2*

LTLSPEC ((T_Sirene.saida & !Defeito_Correia_2 & !Desliga_Geral & !Desliga_L_R) ⇒ X Correia_2)

Dado que *T_Sirene.saida* chegou ao final da contagem, a *Correia_2* deve ser energizada. Estando a *Correia_2* energizada, o temporizador *T1* inicia sua contagem.

- Acionamento da *Correia_1*

LTLSPEC ((T1.saida & Correia_2_Campo & !Desliga_L_R & ! Defeito_Correia_1) ⇒ X Correia_1)

A *Correia_2* é energizada pelo contato *T1.saida* do temporizador *T1*, desde que o estado do sensor *Correia_2_Campo*, que detecta a movimentação da *Correia_2* entrou em movimento, esteja ativado. Esta medida de segurança garante que a sequência de ativação dos equipamentos de pátio será executada corretamente.

- Acionamento do *Alimentador*

LTLSPEC ((T2.saida & Correia_1_Campo & !Desliga_L_R & Defeito_Alimentador) ⇒ X Alimentador)

Analogamente, o temporizador *T2.saida* energiza o *Alimentador*, desde que o estado do sensor *Correia_1_Campo* esteja energizado.

Além das propriedades de sequenciamento, as de intertravamento devem garantir que os equipamentos intertravem o funcionamento um do outro, como segue.

CTLSPEC AG (! Correia_2 ⇒ AF !Correia_1 & !Alimentador)

CTLSPEC AG (Correia_1 ⇒ !Alimentador)

Observa-se que caso a *Correia_2* serja desenergizada, isso implica na desenergização da *Correia_1* e do *Alimentador*. Já caso ocorra o desligamento da *Correia_1*, há implicação apenas no desligamento do *Alimentador*.

Tratando-se das propriedades para descrição do comportamento do sistema perante a ocorrência de defeitos, são tratados defeitos que tem implicações gerais e pontuais. No caso dos defeitos locais, são desligados os equipamentos individualmente, em caso de defeitos gerais toda a planta é desenergizada.

- Descrição de propriedades perante a ocorrência de Defeitos Gerais

CTLSPEC AG ((Rompimento_Correia_1 | Rompimento_Correia_2 | Rompimento_Alim | Defeito_Sucata_Ferrosa) ⇒ AF !Alimentador

- Descrição de propriedades perante a ocorrência de Defeitos Locais

LTLSPEC ((Sobrecorrente_Correia_1 & !Temp_Correia_1 & !Defeito_Redutor_Correia_1) ⇒ X !Correia_1)

LTLSPEC ((Sobrecorrente_Correia_2 & !Temp_Correia_2 & !Defeito_Redutor_Correia_2) ⇒ X !Correia_2)

LTLSPEC ((Sobrecorrente_Alimentador & !Temp_Alimentador & !Defeito_Redutor_Alimentador) ⇒ X !Alimentador)

Observe que os defeitos locais caracterizam-se, pela manifestação de defeitos nos dispositivos de acionamento dos equipamentos de pátio. Nessa situação, por exemplo, qualquer eventual defeito na *Correia_2* causado por uma sobrecarga (*Sobrecorrente_Correia_2*) ou temperatura excedendo a admissível (*Temp_Correia_2*) nos motores, isso implica no desligamento da *Correia_2*. Da mesma forma, caso haja defeitos associados aos redutores de velocidade da correia transportadora esta será desenergizada.

5.2.2 Programa Traduzido automaticamente para NuSMV e Verificação do Sistema

Considerando o sistema em linguagem LD visto na Figura 5.2 e as propriedades detalhadas na seção 5.2.1, obteve-se o modelo em NuSMV. A Figura 5.3 exibe um trecho do modelo resultante, sendo exibidos na figura a estrutura ASSIGN e SPEC, o modelo completo encontra-se na Figura A.1 do Apêndice.

```
ASSIGN

 Sirene_Liga_OCURRENCE_1:= ((Liga_Local|Liga_Remoto|Sirene_Liga) & !Desliga_Geral & !Desliga_L_R & !T_Sirene.saida);
 TON_Sirene_OCURRENCE_1:= (Sirene_Liga & !Desliga_Geral & !Desliga_L_R);
 Correia_2_OCURRENCE_1:= ((T_Sirene.saida | Correia_2 | Rearme_Correia_2) & !Defeito_Correia_2 & !Desliga_Geral & !Desliga_L_R);
 Ind_Correia_2_OCURRENCE_1 := Correia_2_OCURRENCE_1;
 TON_1_OCURRENCE_1:= (Correia_2_OCURRENCE_1 & !Desliga_Geral & !Correia_1_Campo & !Desliga_L_R);
 Correia_1_OCURRENCE_1:= (T1.saida & Correia_2_Campo & !Desliga_Geral& !Defeito_Correia_1 & !Desliga_L_R );
  Ind_Correia_1_OCURRENCE_1:= Correia_1_OCURRENCE_1;
 TON_2_OCURRENCE_1:= (Correia_1_OCURRENCE_1 & !Correia_2_Campo & !Desliga_Geral& !Desliga_L_R);
 Alimentador_OCURRENCE_1:= (T2.saida & Correia_1_Campo & !Defeito_Alimentador & !Desliga_Geral& !Desliga_L_R );
 Ind_Alimentador_OCURRENCE_1:= Alimentador_OCURRENCE_1;
 Desliga_L_R_OCURRENCE_1:= (!Desliga_Local & !Desliga_Remoto & !Emergencia);
 Defeito_Manut_Inspe_OCURRENCE_1:= (Defeito_Sucata_Ferrosa | Rompimento_Correia_1 | Rompimento_Correia_2 | Rompimento_Alim);
 Desliga_Geral_OCURRENCE_1:= Defeito_Manut_Inspe_OCURRENCE_1;
 Defeito_Correia_1_OCURRENCE_1:= (!Sobrecorrente_Correia_1 & !Temp_Correia_1 & !Defeito_Redutor_Correia_1);
 Defeito_Correia_2_OCURRENCE_1:= (!Sobrecorrente_Correia_2 & !Temp_Correia_2 & !Defeito_Redutor_Correia_2);
 Defeito_Alimentador_OCURRENCE_1:= (!Sobrecorrente_Alimentador & !Temp_Alimentador & !Defeito_Redutor_Alimentador);
 next(Sirene_Liga) := Sirene_Liga_OCURRENCE_1;
 next(TON_Sirene) := TON_Sirene_OCURRENCE_1;
 next(Correia_2) := Correia_2_OCURRENCE_1;
 next(Ind_Correia_2):= Ind_Correia_2_OCURRENCE_1;
 next(TON_1):= TON_1_OCURRENCE_1;
 next(Correia_1):= Correia_1_OCURRENCE_1;
 next(Ind_Correia_1):= Ind_Correia_1_OCURRENCE_1;
 next(TON_2):= TON_2_OCURRENCE_1;
 next(Alimentador):= Alimentador_OCURRENCE_1;
 next(Ind_Alimentador):= Ind_Alimentador_OCURRENCE_1;
 next(Desliga_L_R):= Desliga_L_R_OCURRENCE_1;
 next(Defeito_Manut_Inspe):= Defeito_Manut_Inspe_OCURRENCE_1;
next(Desliga_Geral):= Desliga_Geral_OCURRENCE_1;
next(Defeito_Correia_1):= Defeito_Correia_1_OCURRENCE_1;
next(Defeito_Correia_2):= Defeito_Correia_2_OCURRENCE_1;
next(Defeito_Alimentador):= Defeito_Alimentador_OCURRENCE_1;

LTLSPEC ((Liga_Local | Liga_Remoto) -> Sirene_Liga U T_Sirene.saida)
CTLSPEC AG( Defeito_Correia_2 -> AF (!Correia_1 & ! Alimentador))
LTLSPEC ((T_Sirene.saida & ! Defeito_Correia_2 & !Desliga_Geral & !Desliga_L_R) -> X Correia_2)
LTLSPEC ((T1.saida & Correia_2_Campo & !Desliga_L_R & !Defeito_Correia_1) -> X Correia_1)
LTLSPEC ((T2.saida & Correia_1_Campo & !Desliga_L_R & !Defeito_Alimentador) -> X Alimentador)
CTLSPEC AG(!Correia_2 -> AF(!Correia_1 & !Alimentador))
CTLSPEC AG(!Correia_1 -> AF !Alimentador)
CTLSPEC AG((Rompimento_Correia_1 | Rompimento_Correia_2 | Rompimento_Alim | Defeito_Sucata_Ferrosa) -> AF !Alimentador )
 LTLSPEC ((!Sobrecorrente_Correia_1 & ! Temp_Correia_1 & !Defeito_Redutor_Correia_1) -> X !Correia_1)
 LTLSPEC ((!Sobrecorrente_Correia_2 & ! Temp_Correia_2 & ! Defeito_Redutor_Correia_2) -> X !Correia_2)
 LTLSPEC ((!Sobrecorrente_Alimentador & ! Temp_Alimentador & ! Defeito_Redutor_Alimentador) -> X !Alimentador)
```

Figura 5.3: Trecho do Modelo resultante em NuSMV do Sistema de Equipamentos de Pátio de uma Mineradora

Para realizar a verificação do modelo em NuSMV utilizou-se, conforme descrito, o *solver* zChaff. O zChaff é um solucionador SAT que tem como alvo a categoria industrial. Ele implementa o algoritmo Chaff (MOSKEWICZ et al., 2001), que inclui a estratégia de decisão VSIDS (*Variable State Independent Decaying Sum*) e um procedimento de propagação de restrição booleana. O zChaff é um *solver* cujo código fonte é

aberto.

```
-- specification AG (Defeito_Correia_2 -> AF (!Correia_1 & !Alimentador)) is true
-- specification AG (!Correia_2 -> AF (!Correia_1 & !Alimentador)) is true
-- specification AG (!Correia_1 -> !Alimentador) is true
-- specification AG ((((Rompimento_Correia_1 | Rompimento_Correia_2) | Rompimento_Alim) | Defeito_Sucata_Ferrosa) -> AF !Alimentador) is true
-- specification ((Liga_Local | Liga_Remoto) -> (Sirene_Liga U T_Sirene.saida)) is true
-- specification ((((T_Sirene.saida & !Defeito_Correia_2) & !Desliga_Geral) & !Desliga_L_R) -> X Correia_2) is true
-- specification ((((T1.saida & Correia_2_Campo) & !Desliga_L_R) & !Defeito_Correia_1) -> X Correia_1) is true
-- specification ((((T2.saida & Correia_1_Campo) & !Desliga_L_R) & !Defeito_Alimentador) -> X Alimentador) is true
-- specification (((!Sobrecorrente_Correia_1 & !Temp_Correia_1) & !Defeito_Redutor_Correia_1) -> X !Correia_1) is true
-- specification (((!Sobrecorrente_Correia_2 & !Temp_Correia_2) & !Defeito_Redutor_Correia_2) -> X !Correia_2) is true
-- specification (((!Sobrecorrente_Alimentador & !Temp_Alimentador) & !Defeito_Redutor_Alimentador) -> X !Alimentador) is true
```

Figura 5.4: Resultado da Verificação do programa em NuSMV do Sistema de Equipamentos de Pátio de uma Mineradora

Considerado as invariantes descritas entre as linhas 38 e 48 da Figura 5.3, e realizando a verificação do sistema utilizando o *solver* zChaff, obteve-se o resultado visto na figura 5.4. Observa-se que todas as expressões retornam *true*, demonstrando que as propriedades verificadas são satisfazíveis pelo modelo gerado em NuSMV.

```
1   -- specification AG (Defeito_Correia_2 -> (!Correia_1 & !Alimentador)) is false
2   -- as demonstrated by the following execution sequence
3   Trace Description: CTL Counterexample
4   Trace Type: Counterexample
5   -> State: 1.1 <-
6     Liga_Local = FALSE
7     Liga_Remoto = FALSE
8     ...
9   -> State: 1.2 <-
10    ...
11  -> State: 1.3 <-
12    ...
13  -> State: 1.4 <-
14    ...
15  -> State: 1.5 <-
16    ...
17  -> State: 1.6 <-
18    Liga_Local = TRUE
19    Desliga_L_R = TRUE
20    Desliga_Local = TRUE
21    Desliga_Geral = TRUE
22    Correia_1 = TRUE
23    Correia_2_Campo = FALSE
24    Ind_Correia_1 = TRUE
25    Defeito_Correia_2 = TRUE
26    Sobrecorrente_Correia_2 = TRUE
27    Defeito_Manut_Inspe = TRUE
28    Sirene_Liga_OCURRENCE_1 = FALSE
29    TON_Sirene_OCURRENCE_1 = FALSE
30    Correia_2_OCURRENCE_1 = FALSE
31    Ind_Correia_2_OCURRENCE_1 = FALSE
31    TON_1_OCURRENCE_1 = FALSE
32    Correia_1_OCURRENCE_1 = FALSE
33    Ind_Correia_1_OCURRENCE_1 = FALSE
34    Desliga_L_R_OCURRENCE_1 = FALSE
35    Defeito_Correia_2_OCURRENCE_1 = FALSE
```

Figura 5.5: Trecho do Contra-exemplo obtido como Resultado da Verificação

Como descrito anteriormente, considerando as propriedades CTL, e observando as linhas 39, 43, 44 e 45 da Figura 5.3, observa-se o uso do operador *estado futuro* (F) da lógica temporal ramificada. Levando-se em conta a forma de execução do CLP sabe-se

que, de acordo com a ordem de execução do entre *rungs* e o ciclo de varredura, quando altera-se o estado das variáveis utilizando-se o recurso *next* em NuSMV, a variável em questão tem seu estado alterado no próximo ciclo de execução do CLP.

Nessas circunstâncias caso não fosse utilizado o operador temporal **F** (*estado futuro*) na propriedade da linha 39, obtém-se o resultado descrito na Figura 5.5 no processo de verificação. A Figura 5.5 exibe apenas o trecho de interesse do contra-exemplo obtido, o resultado completo pode ser visto na Figura A.2.

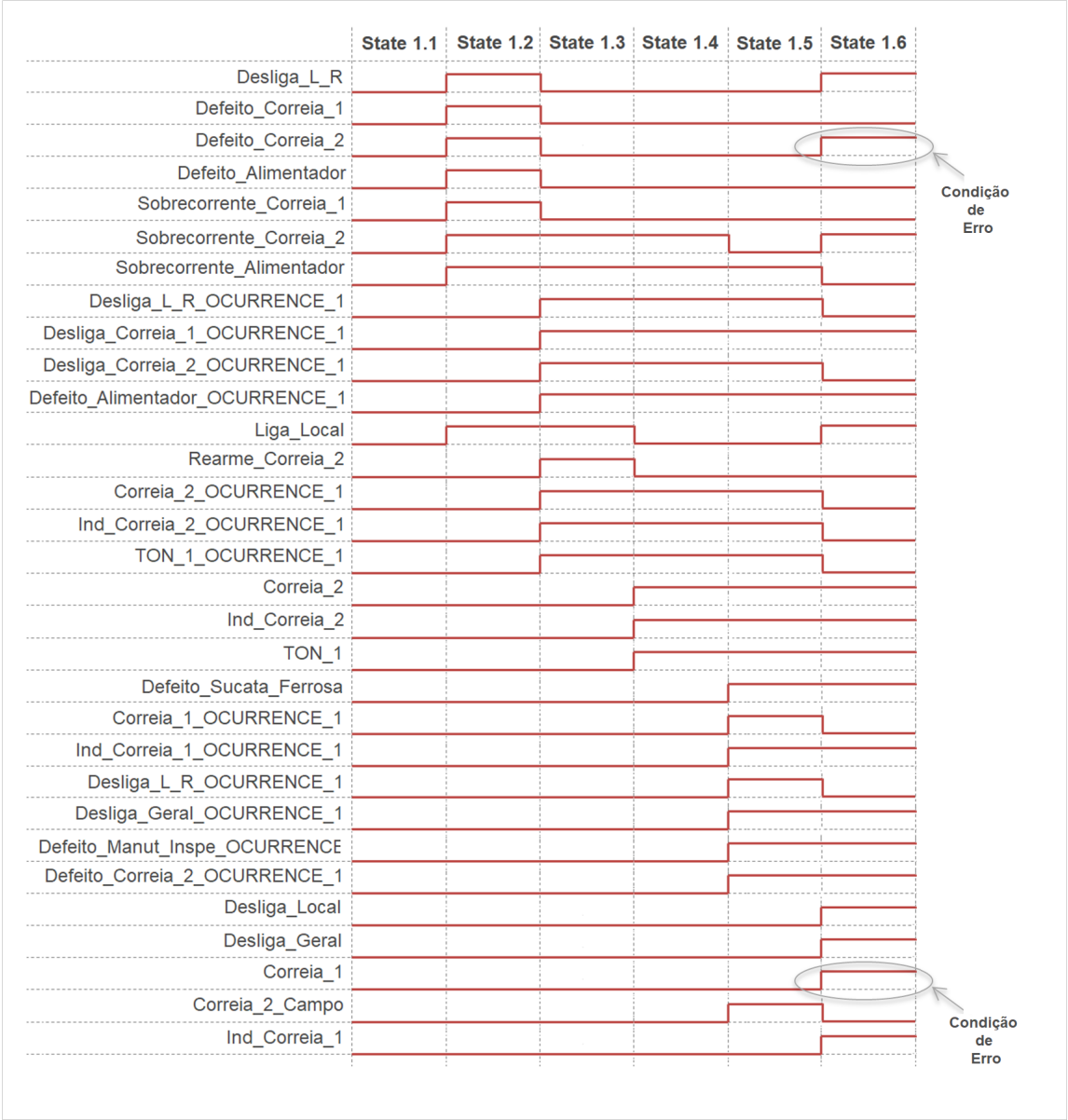

Figura 5.6: Contra-exemplo em diagrama temporal do resultado obtido com a Verificação do Sistema

Observando o contra-exemplo resultante da Figura 5.5, foram suprimidas os estados intermediários para melhor visualização do estado de interesse. Considerando que

a propriedade foi descrita sem que se utilizasse o operador temporal **F**, observa-se que a variável *Defeito_Correia_2* comuta para *TRUE*, no *state 1.6*. Observa-se que, também no *state 1.6* a *Correia_1* encontra-se em *TRUE*. Assim sendo, a propriedade descrita é infringida.

Para melhor visualização, observe a Figura 5.6, que dispõe em um diagrama de tempo os estados das variáveis. Ressalta-se na Figura 5.6 os pontos em que, no *state 1.6*, a condição de erro ocorre.

A descrição errada da propriedade foi propositalmente inserira para que pudesse ser evidenciada a forma com a qual o CLP executa as instruções de programa e realiza a atualização das variáveis. Este evento ocorre devido ao ciclo de execução do CLP que, para que a propriedade seja satisfeita, deve-se considerar que no instante futuro ou no próximo ciclo de execução, a variável Correia_1 será atualizada para FALSE e satisfará a descrição de funcionamento do sistema.

Capítulo 6

Considerações Finais

O presente trabalho tratou do processo de verificação automática de programas de CLP escritos em Diagrama Ladder. Como demonstrou-se ao longo do trabalho, o processo de verificação constitui em uma ferramenta indispensável para aumentar a confiabilidade dos sistemas automatizados por meio do uso de controladores lógicos programáveis. Até o momento, trabalhos desenvolvidos abordando as técnicas de verificação de programas de CLP não exploraram em profundidade a ordem de execução entre *rungs*, bem como a ordem de execução interna da *rung* e suas implicações para o processo de modelagem do programa.

Embora muitos avanços tenham sido feitos em relação ao estado da arte dos sistemas de verificação de programas de CLP, ainda há muito a ser desenvolvido em termos de representatividade de programas reais desenvolvidos para o ambiente industrial. Neste contexto, este trabalho procurou abordar exemplos prático de situações comuns em praticamente todos os programas implementados em sistemas reais. Para tal utilizou-se programas exportados do CLP RSlogix 5000 - Compact Logix L23E (Allen Bradley, 2013).

O objetivo principal desta dissertação de mestrado constituiu no desenvolvimento de um fluxo de trabalho e a criação de uma ferramenta de suporte que permita a verificação de sistemas descritos, automaticamente, usando a linguagem LD da norma IEC 61131-3. Evidenciou-se que, este objetivo foi alcançado com exito, considerando-se os resultados obtidos por meio da verificação do programa em NuSMV.

A sistematização do processo de verificação mostrado neste trabalho, considerando programas em LD como a entrada de fluxo para o trabalho desenvolvido, é uma contribuição para o campo de verificação de programas industriais. Este *fremework* pode

ser facilmente adaptado aos diversos fabricantes de CLP e conduzir o processo de análise e desenvolvimento de modelos que buscam promover resultados corretos em termos de verificação de sistemas.

A estrutura de tradução desenvolvida foi validada através do uso de programas industriais verificado sob diferentes invariantes do sistema, corroborando a eficácia do sistema desenvolvido. O arquivo resultante do processo de tradução em formato *.smv*, evidenciou-se coerente com as aplicações estruturadas na linguagem Ladder, considerados programas cuja composição contém variáveis booleanas, bem como blocos de temporização.

6.1 Trabalhos Futuros

O presente trabalho desenvolvido prevê algumas implementações e aprimoramentos futuros, onde dentre outros podem ser destacados:

1. Além de modelar variáveis booleanas e blocos de temporização, realizar a modelagem eficiente de blocos que manipulam variáveis inteiras está sendo estudada atualmente. Esses blocos são destinados a realizar operações aritméticas, entre outras;

2. Considerando a forma de entrada de invariantes do sistema, sendo estas as propriedades a serem verificadas no programa, encontra-se em fase de desenvolvimento uma interface de escrita de propriedade. Esta interface constitui-se basicamente em um meio de escrita das especificações em uma língua mais próxima da natural. Isto permitiria que a especificação de propriedades fossem descritas por operadores ou desenvolvedores de programas Ladder que não têm conhecimento em profundidade sobre as especificações temporais em linguagem formal.

Referências Bibliográficas

Allen Bradley. *RSlogix 5000: Compact Logix L23E*. [S.l.], 2013. Disponível em: ⟨https://www.rockwellautomation.com/global/literature-library/overview.page?⟩

BARBOSA, D. D. H. Formal verification of plc programs using the b method. *Third International Conference, ABZ 2012*, June 2012.

BIALLAS, S.; BRAUER, J.; KOWALEWSKI, S. Arcade.plc: a verification platform for programmable logic controllers. In: *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. [S.l.: s.n.], 2012. p. 338–341.

BLECH, J.; BIHA, S. O. Verification of plc properties based on formal semantics in coq. *SEFM'11 Proceedings of the 9th international conference on Software Engineering and Formal Methods*, November 2011.

BOEHM, B. W. Software engineering: As it is. In: *Proceedings of the 4th International Conference on Software Engineering, Munich, Germany, September 1979.* [s.n.], 1979. p. 11–21. Disponível em: ⟨http://dl.acm.org/citation.cfm?id=802916⟩.

CANET S. COUFFIN, J. J. L. G.; PETIT, A. Towards the automatic verification of plc programs written in instruction list. *IEEE International Conference on Systems, Man and Cybernetics*, 2000.

CAVADA A. CIMATTI, C. A. J. G. K.-E. O. M. P. M. R. R.; TCHALTSEV, A. *NuSMV 2.6 User Manual*. [S.l.], 2010. Disponível em: ⟨http://nusmv.fbk.eu.⟩

CIMATTI E. CLARKE, F. G. e. a. A. Nusmv: a new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, v. 2, n. 4, p. 410–425, March 2000. ISSN 1433-2779.

CLARKE, E. A. E. E. M. Design and synthesis of synchronization skeletons using branching-time temporal logic. In: *Logic of Programs, Workshop*. Berlin, Heidelberg: Springer-Verlag, 1982. p. 52–71. Disponível em: ⟨http://dl.acm.org/citation.cfm?id=648063.747438⟩.

CLARKE, E. A. E. E. M.; SISTLA, A. P. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, ACM, NY, USA, v. 8, n. 2, p. 244–263, April 1986. ISSN 0164-0925. Disponível em: ⟨http://doi.acm.org/10.1145/5397.5399⟩.

CLARKE E.AND PELED, A. *Model Checking*. [S.l.]: The MIT Press, 1999.

COOPER, K. D.; TORCZON, L. Book. *Engineering a compiler*. 2nd ed. ed. [S.l.]: Amsterdam Elsevier/Morgan Kaufmann, 2012. ISBN 9780120884780.

DIJKSTRA, E. W. The humble programmer. *Commun. ACM*, v. 15, n. 10, p. 859–866, 1972. Disponível em: ⟨https://doi.org/10.1145/355604.361591⟩.

ESSALMI, F.; AYED, L. J. B. Graphical uml view from extended backus-naur form grammars. In: *Sixth IEEE International Conference on Advanced Learning Technologies (ICALT'06)*. [S.l.: s.n.], 2006. p. 544–546. ISSN 2161-3761.

FARINES, J. et al. A model-driven engineering approach to formal verification of plc programs. In: *ETFA2011*. [S.l.: s.n.], 2011. p. 1–8. ISSN 1946-0759.

FISHER, M. S. Software verification and validation. In: _____. [S.l.: s.n.], 2007. cap. 2.

GOURCUFF, O. d. S. V.; FAURE, J.-M. Efficient representation for formal verification of plc programs. *8th International Workshop on Discrete Event Systems*, July 2006.

GOURCUFF, O. D. S. V.; FAURE, J. M. Improving large-sized plc programs verification using abstractions. *Proceedings of the 17th IFAC World Congress*, 2008.

IEEE. *IEEE Std 610.12-1990 - IEEE Standard Glossary of Softtware Engineering*. [S.l.], 2010. Disponível em: ⟨http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=159342⟩.

International Electrotechnical Commission. *Programmable controllers - Part 3: Programming languages*. [S.l.], 2013. Disponível em: ⟨https://webstore.iec.ch/searchform&q=61131⟩.

JavaCC. *JavaCC The Java Parser Generator*. 2018. Https://javacc.org/. Accessed: 2018-10-14.

LAMPÉRIÈRE-COUFFIN, S.; LESAGE, J.-J. Formal verification of the sequential part of plc programs. In: _____. *Discrete Event Systems: Analysis and Control*. Boston, MA: Springer US, 2000. p. 247–254. ISBN 978-1-4615-4493-7. Disponível em: ⟨https://doi.org/10.1007/978-1-4615-4493-7_25⟩.

LAMPéRIèRE-COUFFIN O. ROSSI, J.-M. R. J.-J. L. S. Formal validation of plc programs: A survey. *European Control Conference (ECC)*, September 1999.

LEVESON, N. G.; TURNER, C. S. An investigation of the therac-25 accidents. *Computer*, v. 26, n. 7, p. 18–41, July 1993. ISSN 0018-9162.

LJUNGKRANTZ, O.; AKESSON, K.; FABIAN, M. Formal specification and verification of components for industrial logic control programming. In: *2008 IEEE International Conference on Automation Science and Engineering*. [S.l.: s.n.], 2008. p. 935–940. ISSN 2161-8070.

LOBOV, A. et al. Modelling and verification of plc-based systems programmed with ladder diagrams. *IFAC Proceedings Volumes*, v. 37, n. 4, p. 183 – 188, 2004. ISSN 1474-6670. 11th IFAC Symposium on Information Control Problems in Manufacturing (INCOM 2004), Salvador, Brazil, 5-7 April 2004. Disponível em: ⟨http://www.sciencedirect.com/science/article/pii/S1474667017361165⟩.

MAHAJAN, Y. S.; FU, Z.; MALIK, S. Zchaff2004: An efficient SAT solver. In: *Theory and Applications of Satisfiability Testing, 7th International Conference, SAT 2004, Vancouver, BC, Canada, May 10-13, 2004, Revised Selected Papers*. [s.n.], 2004. p. 360–375. Disponível em: ⟨https://doi.org/10.1007/11527695_27⟩.

MCMILLAN, K. L. *Symbolic model checking*. [S.l.]: Kluwer, 1993. ISBN 978-0-7923-9380-1.

MOKADEM B. BERARD, V. G. J.-M. R. H. B.; SMET, O. de. Verification of a timed multitask system with uppaal. *ETFA'05*, p. 347–354, September 2005.

MOON, I. Modeling programmable logic controllers for logic verification. *IEEE Control Systems Magazine*, v. 14, n. 2, p. 53–59, April 1994. ISSN 1066-033X.

MOSKEWICZ, M. W. et al. Chaff: Engineering an efficient sat solver. In: *DAC*. ACM, 2001. p. 530–535. ISBN 1-58113-297-2. Disponível em: ⟨http://dblp.uni-trier.de/db/conf/dac/dac2001.html\\#MoskewiczMZZM01⟩.

MYERS, G. J. *The Art of Software Testing*. [S.l.]: New York: John Wiley and Sons, 1979. 54-55 p.

NETO, A. Introdução a teste de software. *Engenharia de Software Magazine*, 2015.

O. LEANDRO S., A. P. A. L. K.; G., K. Geração automática de testes de conformidade para programas de controladores lógicos programáveis. *XVIII Congresso Brasileiro de Automática*, 2010.

PAKONEN, A. et al. A toolset for model checking of plc software. In: *2013 IEEE 18th Conference on Emerging Technologies Factory Automation (ETFA)*. [S.l.: s.n.], 2013. p. 1–6. ISSN 1946-0740.

PAVLOVIC, R. P.; KOLLMANN, M. Automated formal verification of plc programms written in il. *VERIFY Workshop Proce*, 2007.

PNUELI, A. Two decades of temporal logic: Achievements and challenges (abstract). p. 78, 1997. Disponível em: ⟨https://doi.org/10.1109/SFCS.1997.646095⟩.

PRESSMAN, R. S. *Software Engineering: A Practitioner's Approach*. [S.l.]: McGraw-hill, 2011.

QUEILLE, J.-. P.; SIFAKIS, J. Specification and verification of concurrent systems in CESAR. In: *International Symposium on Programming, 5th Colloquium, Torino, Italy, April 6-8, 1982, Proceedings*. [s.n.], 1982. p. 337–351. Disponível em: ⟨https://doi.org/10.1007/3-540-11494-7_22⟩.

RANJAN, C. C. R. K.; SKALBERG, S. Beyond verification: leveraging formal for debugging. In: *Proceedings of the 46th Annual Design Automation Conference*. New York, USA: [s.n.], 2009. p. 648–651. ISBN 978-1-60558-497-3. Disponível em: ⟨http://doi.acm.org10.1145/1629911.1630082⟩.

RAUSCH, M.; KROGH, B. H. Formal verification of plc programs. In: *Proceedings of the 1998 American Control Conference. ACC (IEEE Cat. No.98CH36207)*. [S.l.: s.n.], 1998. v. 1, p. 234–238 vol.1. ISSN 0743-1619.

ROSSI O. DE SMET, S. C. J.-J. L. H. P. O.; GUENNEC, H. Formal verification: A tool to improve the safety of control systems. *Fault Detection, Supervision and Safety for Technical Processes (IFAC)*, 2000.

SAT Competition. *SAT Competition 2002*. 2002. Disponível em: ⟨https://www.satcompetition.org/⟩.

SAT Competition. *SAT Competition 2004*. 2004. Disponível em: ⟨https://www.satcompetition.org/⟩.

SPARACO, P. Airbus plans increased production rate. *Aviation Week e Space Technology*, November 1996.

THAPA, D. et al. Verification of plc program using a generic intermediate language (iml). *Proceedings of the ICSCA*, v. 2956, 01 2006.

ZHENDONG, S. Automatic analysis of relay ladder logic program. *Tech. Report CSD-97-969*, 1997.

ZHOU, M. et al. Translation-based model checking for plc programs. In: *2009 33rd Annual IEEE International Computer Software and Applications Conference*. [S.l.: s.n.], 2009. v. 1, p. 553–562. ISSN 0730-3157.

Apêndice A

Apêndice

A Figura A.1 exibe a modelagem completa do programa em NuSMV. Para melhor efeito de visualização foram suprimidas as declarações do estado inicial das variáveis, tendo todas sido inicializadas com estado *FALSE*.

Da mesma forma, com relação à Figura A.2, para efeito de visualização foram suprimidos os estados iniciais das variáveis do sistema.

```
MODULE main

VAR

   Liga_Local: boolean;
   ...
   T1: TON(next(TON_1));
   T2: TON(next(TON_2));
   ...

INIT

   Liga_Local= FALSE &
   ...

ASSIGN

   Sirene_Liga_OCURRENCE_1:= ((Liga_Local|Liga_Remoto|Sirene_Liga) & !Desliga_Geral & !Desliga_L_R & !T_Sirene.saida);
   TON_Sirene_OCURRENCE_1:= (Sirene_Liga & !Desliga_Geral & !Desliga_L_R);
   Correia_2_OCURRENCE_1:= ((T_Sirene.saida | Correia_2 | Rearme_Correia_2) & !Defeito_Correia_2 & !Desliga_Geral & !Desliga_L_R);
   Ind_Correia_2_OCURRENCE_1 := Correia_2_OCURRENCE_1;
   TON_1_OCURRENCE_1:= (Correia_2_OCURRENCE_1 & !Desliga_Geral & !Correia_1_Campo & !Desliga_L_R);
   Correia_1_OCURRENCE_1:= (T1.saida & Correia_2_Campo & !Desliga_Geral& !Defeito_Correia_1 & !Desliga_L_R );
    Ind_Correia_1_OCURRENCE_1:= Correia_1_OCURRENCE_1;
   TON_2_OCURRENCE_1:= (Correia_1_OCURRENCE_1 & !Correia_2_Campo & !Desliga_Geral& !Desliga_L_R);
   Alimentador_OCURRENCE_1:= (T2.saida & Correia_1_Campo & !Defeito_Alimentador & !Desliga_Geral& !Desliga_L_R );
   Ind_Alimentador_OCURRENCE_1:= Alimentador_OCURRENCE_1;
   Desliga_L_R_OCURRENCE_1:= (!Desliga_Local & !Desliga_Remoto & !Emergencia);
   Defeito_Manut_Inspe_OCURRENCE_1:= (Defeito_Sucata_Ferrosa | Rompimento_Correia_1 | Rompimento_Correia_2 | Rompimento_Alim);
   Desliga_Geral_OCURRENCE_1:= Defeito_Manut_Inspe_OCURRENCE_1;
   Defeito_Correia_1_OCURRENCE_1:= (!Sobrecorrente_Correia_1 & !Temp_Correia_1 & !Defeito_Redutor_Correia_1);
   Defeito_Correia_2_OCURRENCE_1:= (!Sobrecorrente_Correia_2 & !Temp_Correia_2 & !Defeito_Redutor_Correia_2);
   Defeito_Alimentador_OCURRENCE_1:= (!Sobrecorrente_Alimentador & !Temp_Alimentador & !Defeito_Redutor_Alimentador);

   next(Sirene_Liga) :=  Sirene_Liga_OCURRENCE_1;
   next(TON_Sirene) := TON_Sirene_OCURRENCE_1;
   next(Correia_2) :=  Correia_2_OCURRENCE_1;
   next(Ind_Correia_2):= Ind_Correia_2_OCURRENCE_1;
   next(TON_1):= TON_1_OCURRENCE_1;
   next(Correia_1):= Correia_1_OCURRENCE_1;
   next(Ind_Correia_1):= Ind_Correia_1_OCURRENCE_1;
   next(TON_2):= TON_2_OCURRENCE_1;
   next(Alimentador):= Alimentador_OCURRENCE_1;
   next(Ind_Alimentador):= Ind_Alimentador_OCURRENCE_1;
   next(Desliga_L_R):= Desliga_L_R_OCURRENCE_1;
   next(Defeito_Manut_Inspe):= Defeito_Manut_Inspe_OCURRENCE_1;
  next(Desliga_Geral):= Desliga_Geral_OCURRENCE_1;
  next(Defeito_Correia_1):= Defeito_Correia_1_OCURRENCE_1;
  next(Defeito_Correia_2):= Defeito_Correia_2_OCURRENCE_1;
  next(Defeito_Alimentador):=Defeito_Alimentador_OCURRENCE_1;

LTLSPEC ((Liga_Local | Liga_Remoto) -> Sirene_Liga U T_Sirene.saida)
CTLSPEC AG( Defeito_Correia_2 -> AF (!Correia_1 & ! Alimentador))
LTLSPEC ((T_Sirene.saida & ! Defeito_Correia_2 & !Desliga_Geral & !Desliga_L_R) -> X Correia_2)
LTLSPEC ((T1.saida & Correia_2_Campo & !Desliga_L_R & !Defeito_Correia_1) -> X Correia_1)
LTLSPEC ((T2.saida & Correia_1_Campo & !Desliga_L_R & !Defeito_Alimentador) -> X Alimentador)
CTLSPEC AG(!Correia_2 -> AF(!Correia_1 & !Alimentador))
CTLSPEC AG(!Correia_1 -> !Alimentador)
CTLSPEC AG((Rompimento_Correia_1 | Rompimento_Correia_2 | Rompimento_Alim | Defeito_Sucata_Ferrosa) -> AF !Alimentador )
 LTLSPEC ((!Sobrecorrente_Correia_1 & ! Temp_Correia_1 & !Defeito_Redutor_Correia_1) -> X !Correia_1)
 LTLSPEC ((!Sobrecorrente_Correia_2 & ! Temp_Correia_2 & ! Defeito_Redutor_Correia_2) -> X !Correia_2)
 LTLSPEC ((!Sobrecorrente_Alimentador & ! Temp_Alimentador & ! Defeito_Redutor_Alimentador) -> X !Alimentador)

MODULE TON (entrada)

VAR
    estado : {pausado, contando,atuado};

ASSIGN
    init(estado):= pausado;
    next(estado):= case
              estado = pausado & !entrada:{ pausado };
              estado = pausado & entrada: {contando};
              estado = contando & entrada: { atuado, contando};
              estado = contando & !entrada: { pausado };
              estado = atuado & entrada: { atuado };
              estado = atuado & !entrada: { pausado };
         esac;

DEFINE
    saida:= estado = atuado;
```

Figura A.1: Modelo equivalente em NuSMV do Sistema de Equipamentos de Pátio de uma Mineradora

```
1   -- specification AG (Defeito_Correia_2 -> (!Correia_1 & !Alimentador)) is false
2   -- as demonstrated by the following execution sequence
3   Trace Description: CTL Counterexample
4   Trace Type: Counterexample
5   -> State: 1.1 <-
6     Liga_Local = FALSE
7     Liga_Remoto = FALSE
8     ...
9   -> State: 1.2 <-
10    Desliga_L_R = TRUE
11    Desliga_Local = TRUE
12    Defeito_Correia_1 = TRUE
13    Defeito_Correia_2 = TRUE
14    Defeito_Alimentador = TRUE
15    Sobrecorrente_Correia_1 = TRUE
16    Sobrecorrente_Correia_2 = TRUE
17    Sobrecorrente_Alimentador = TRUE
18    Desliga_L_R_OCURRENCE_1 = FALSE
19    Defeito_Correia_1_OCURRENCE_1 = FALSE
20    Defeito_Correia_2_OCURRENCE_1 = FALSE
21    Defeito_Alimentador_OCURRENCE_1 = FALSE
22  -> State: 1.3 <-
23    Liga_Local = TRUE
24    Desliga_L_R = FALSE
25    Rearme_Correia_2 = TRUE
26    Defeito_Correia_1 = FALSE
27    Defeito_Correia_2 = FALSE
28    Defeito_Alimentador = FALSE
29    Sirene_Liga_OCURRENCE_1 = TRUE
30    Correia_2_OCURRENCE_1 = TRUE
31    Ind_Correia_2_OCURRENCE_1 = TRUE
31    TON_1_OCURRENCE_1 = TRUE
32  -> State: 1.4 <-
33    Liga_Local = FALSE
34    Sirene_Liga = TRUE
35    Correia_2 = TRUE
36    Rearme_Correia_2 = FALSE
37    Ind_Correia_2 = TRUE
38    TON_1 = TRUE
39    TON_Sirene_OCURRENCE_1 = TRUE
40    T1.estado = contando
41  -> State: 1.5 <-
42    Desliga_Local = FALSE
43    Correia_2_Campo = TRUE
44    Sobrecorrente_Correia_2 = FALSE
45    Defeito_Sucata_Ferrosa = TRUE
46    TON_Sirene = TRUE
47    Correia_1_OCURRENCE_1 = TRUE
48    Ind_Correia_1_OCURRENCE_1 = TRUE
49    Desliga_L_R_OCURRENCE_1 = TRUE
50    Desliga_Geral_OCURRENCE_1 = TRUE
51    Defeito_Manut_Inspe_OCURRENCE_1 = TRUE
52    Defeito_Correia_2_OCURRENCE_1 = TRUE
53    T_Sirene.estado = contando
54    T1.estado = atuado
55    T1.saida = TRUE
56  -> State: 1.6 <-
57    Liga_Local = TRUE
58    Desliga_L_R = TRUE
59    Desliga_Local = TRUE
60    Desliga_Geral = TRUE
61    Correia_1 = TRUE
62    Correia_2_Campo = FALSE
63    Ind_Correia_1 = TRUE
64    Defeito_Correia_2 = TRUE
65    Sobrecorrente_Correia_2 = TRUE
66    Defeito_Manut_Inspe = TRUE
67    Sirene_Liga_OCURRENCE_1 = FALSE
68    TON_Sirene_OCURRENCE_1 = FALSE
69    Correia_2_OCURRENCE_1 = FALSE
70    Ind_Correia_2_OCURRENCE_1 = FALSE
71    TON_1_OCURRENCE_1 = FALSE
72    Correia_1_OCURRENCE_1 = FALSE
73    Ind_Correia_1_OCURRENCE_1 = FALSE
74    Desliga_L_R_OCURRENCE_1 = FALSE
75    Defeito_Correia_2_OCURRENCE_1 = FALSE
```

Figura A.2: Contraexemplo obtido como Resultado da Verificação