Universidade Federal de Campina Grande

Centro de Engenharia Elétrica e Informática

Coordenação de Pós-Graduação em Informática

Dissertação de Mestrado

Geração Automática de Testes de Conformidade para Programas de Controladores Lógicos Programáveis

Kézia de Vasconcelos Oliveira

Campina Grande - PB
Agosto de 2009

Universidade Federal de Campina Grande

Centro de Engenharia Elétrica e Informática

Coordenação de Pós-Graduação em Informática

Geração Automática de Testes de Conformidade para Programas de Controladores Lógicos Programáveis

Kézia de Vasconcelos Oliveira

Dissertação submetida à Coordenação de Pós-Graduação em Informática da Universidade Federal de Campina Grande como parte dos requisitos necessários para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: Ciência da Computação

Linha de Pesquisa: Engenharia de Software

Angelo Perkusich

Leandro Dias da Silva

(Orientadores)

Campina Grande, Paraíba, Brasil

©Kézia de Vasconcelos Oliveira - 2009

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA CENTRAL DA UFCG

O48g

2009 Oliveira, Kézia de Vasconcelos.

Geração Automática de Testes de Conformidade para Programas de Controladores Lógicos Programáveis / Kézia de Vasconcelos Oliveira. - Campina Grande, 2009.

117f. : il.

Dissertação (Mestrado em Ciência da Computação) - Universidade Federal de Campina Grande. Centro de Engenharia Elétrica e Informática.

Orientadores: Prof. Dr. Angelo Perkusich, Prof. Dr. Leandro Dias da Silva.

1. Testes de Conformidade. 2. Controladores Lógicos Programáveis. 3. Autômatos Temporizados. I. Título.

CDU- 004.415.53(043)

"GERAÇÃO AUTOMÁTICA DE TESTES DE CONFORMIDADE PARA PROGRAMAS DE CONTROLADORES LÓGICOS PROGRAMÁVEIS"

KÉZIA DE VASCONCELOS OLIVEIRA

DISSERTAÇÃO APROVADA EM 28.08.2009

LEANDRO DIAS DA SILVA, D.SC
Orientador(a)

ANGELO PERKUSICH, D.SC
Orientador(a)

HYGGO OLIVEIRA DE ALMEIDA, D.SC
Examinador(a)

ANTONIO MARCUS NOGUEIRA LIMA, DR.
Examinador(a)

CAMPINA GRANDE - PB

Resumo

O objetivo deste trabalho é apresentar um método e disponibilizar uma ferramenta que aumente a confiança e a segurança na automação de processos controlados por Controladores Lógicos Programáveis (CLPs). Para tanto, geração e execução automática de casos de teste de conformidade são utilizadas para verificar se o programa do CLP está em conformidade com a especificação. O método consiste em gerar automaticamente modelos de autômatos temporizados, segundo a sintaxe e semântica da ferramenta Uppaal, a partir de Diagramas de Lógica Binária ISA 5.2 (especificação) e programas escritos na linguagem Ladder (implementação). Após a geração destes modelos, testes de conformidade são realizados usando a ferramenta de teste Uppaal-TRON. Estudos de caso são realizados para validar este trabalho.

Abstract

The goal of this work is to present a method and provide a tool to ensure the safety and dependability in the automation of processes controlled by Programmable Logic Controllers (PLCs). Therefore, automatic generation and execution of conformance testing cases are used to verify the conformance between the PLC program and the specification. The method uses the Uppaal tool syntax and semantic to automatically generate timed automata models from ISA 5.2 Binary Logic Diagrams (specification) and Ladder language (implementation). After the models generation, conformance testing is performed using the Uppaal-TRON tool. Case studies are performed to validate this work.

Agradecimentos

Agradeço a Deus, em primeiro lugar, razão de toda a minha história, pelo dom da vida e pela oportunidade de realizar este trabalho, dando-me forças nos momentos difíceis. Obrigada senhor!

Além de dedicar, agradeço a meus pais José Florêncio e Josélia, e as minhas irmãs Kéllen e Kivânia, por todo o incentivo, apoio, confiança e exemplo dado por toda vida. Sendo eles os maiores responsáveis por esta conquista.

A meu filho Arthur Victor, que chegou junto com este projeto e, mesmo pequenino, soube me dar inspiração e apoio nas horas difíceis. E a meu marido Karcius por todo carinho, incentivo, dedicação e paciência.

A minha segunda família Juarez, Socorro e Kelly, por toda confiança e incentivo para a conclusão deste trabalho.

Aos meus parentes que sempre tiveram uma palavra de apoio e ternura.

Aos professores Angelo Perkusich e Leandro Dias da Silva pela orientação necessária ao desenvolvimento desta dissertação e pela contribuição em minha formação acadêmica e profissional.

A Kyller e a Luiz Paulo, pelas contribuições na realização deste trabalho.

Ao pessoal do Laboratório Embedded pela amizade.

A meu avô José e, in memorian, a meus avós Maria, Irene e João pelo exemplo de vida.

A Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES), pelo apoio financeiro.

Aos meus amigos que compreenderam minha ausência.

Em fim, a todos que contribuíram de forma direta ou indireta para a realização deste trabalho.

Conteúdo

1 Introdução **1**

1.1 Descrição do Problema 2

1.2 Objetivos do Trabalho 4

1.3 Resultados e Relevância do Trabalho 5

1.4 Estrutura da Dissertação 5

2 Trabalhos Relacionados **7**

3 Fundamentação Teórica **15**

3.1 Controladores Lógicos Programáveis 15

3.2 Linguagem de Diagramas Ladder 18

3.3 Autômatos Temporizados 22

3.4 Diagramas de Lógica Binária ISA 5.2 27

4 O Método **30**

4.1 Modelagem dos autômatos temporizados 33

4.1.1 Modelagem de variáveis de entrada 34

4.1.2 Modelagem do processo que realiza atualização de variáveis de entrada 36

4.1.3 Modelagem do processamento de sinais de entrada 37

4.1.4 Modelagem de temporizadores 38

4.1.5 Modelagem do ciclo de execução do programa 45

4.1.6 Modelagem do processo que avalia os estados das saídas 55

4.1.7 Modelagem do processamento dos estados das saídas 56

4.1.8 Modelagem do ciclo de varredura de um CLP 57

4.2 Fluxo de execução dos autômatos temporizados 59

4.3 Geração dos modelos de autômatos temporizados 62

5 Estudo de Caso **64**

5.1 Definição do sistema que enche garrafas 64

5.2 Diagrama ISA 5.2 e Programa Ladder para o sistema que enche garrafas . . 65

5.3 Modelagem do sistema que enche garrafas 67

5.4 Testes realizados no sistema que enche garrafas 76

5.4.1 Aplicação do processo de teste para ocorrência do erro 1: O segundo degrau é executado e depois o primeiro degrau é executado 77

5.4.2 Aplicação do processo de teste para ocorrência do erro 2: Retirar o quarto degrau 79

5.4.3 Aplicação do processo de teste para ocorrência do erro 3: Trocar o contato normalmente aberto da variável *LS*, no terceiro degrau, por um contato normalmente fechado 80

5.4.4 Aplicação do processo de teste para ocorrência do erro 4: No quarto degrau a variável *PE* será colocada em paralelo com a variável *TMR1* 81

5.4.5 Aplicação do processo de teste para ocorrência do erro 5 : Alterar o valor de *PT* de *Timer 1* para 0.3 segundo 82

5.5 Definição do Sistema que controla dois semáforos 84

5.6 Diagrama ISA 5.2 e Programa Ladder para o sistema que controla dois semáforos 85

5.7 Modelagem do sistema que controla dois semáforos 89

5.8 Testes realizados no sistema que controla dois semáforos 105

5.8.1 Aplicação do processo de teste para ocorrência do erro 1: O segundo degrau é executado e depois o primeiro degrau é executado 106

5.8.2 Aplicação do processo de teste para ocorrência do erro 2: Retirar o nono degrau 108

5.8.3 Aplicação do processo de teste para ocorrência do erro 3: Alterar o valor de *PT*, de *Timer 3*, para 20 segundos 109

6 Considerações Finais e Trabalhos Futuros **111**

6.1 Trabalhos Futuros 112

References **113**

Lista de Figuras

1.1 Processo de desenvolvimento de um CLP 3

3.1 Estrutura básica de um CLP 16

3.2 Ciclo de Varredura de um CLP 17

3.3 Esquema gráfico da linguagem Ladder 18

3.4 Caminhos de continuidade lógica de um programa Ladder 19

3.5 Exemplo de um degrau de um programa Ladder 21

3.6 Exemplo de um degrau de um programa Ladder com temporizador 21

3.7 Modelagem do funcionamento de uma torneira como uma rede de autômatos temporizados 23

3.8 Diagrama ISA 5.2 para o arranque de um motor 29

4.1 Método utilizado para validar o SIS 31

4.2 Interação entre ferramentas utilizadas no método proposto 33

4.3 Autômato que representa a modelagem de uma variável de entrada 35

4.4 Autômato que representa o processo de atualização de variáveis de entrada . 37

4.5 Autômato que representa a modelagem do processamento dos sinais de entrada 38

4.6 Modelagem de um temporizador TON ou DI como um autômato temporizado 40

4.7 Modelagem de um temporizador TOF ou DT como um autômato temporizado 41

4.8 Modelagem de um temporizador TP ou PO como um autômato temporizado 42

4.9 Esquema utilizado para construir o autômato temporizado que representa a execução da lógica do programa 47

4.10 Modelagem do processo que avalia os estados das saídas 55

4.11 Modelagem do processamento dos estados das saídas 56

4.12 Modelagem do ciclo de varredura de um CLP como um autômato temporizado 58

4.13 Sequência do fluxo de eventos ocorridos na rede de autômatos temporizados 60
4.14 Esquema utilizado para geração dos modelos de autômatos temporizados . 62

5.1 Sistema que enche garrafas - Fonte: (Bryan & Bryan, 1997), página 485 . . 65
5.2 Diagrama ISA 5.2 para o sistema que enche garrafas 66
5.3 Programa Ladder para o sistema que enche garrafas 67
5.4 Autômatos que representam as modelagens das variáveis de entrada do sistema que enche garrafas 68
5.5 Autômato que representa o processo de atualização de variáveis de entrada para o sistema que enche garrafas 69
5.6 Autômato que representa a modelagem do processamento dos sinais de entrada para o sistema que enche garrafas 69
5.7 Autômato que representa o temporizador Timer1 do sistema que enche garrafas 70
5.8 Autômato que representa o temporizador Timer2 do sistema que enche garrafas 71
5.9 Autômato que representam a execução do programa do sistema que enche garrafas 74
5.10 Modelagem do processo que avalia os estados das saídas do sistema que enche garrafas 75
5.11 Modelagem do processamento dos estados das saídas do sistema que enche garrafas 75
5.12 Autômato que representam ciclo de varredura do CLP para o sistema que enche garrafas 76
5.13 Inserção do erro 1 no programa Ladder do sistema que enche garrafas . . . 78
5.14 Inserção do erro 2 no programa Ladder do sistema que enche garrafas . . . 79
5.15 Inserção do erro 3 no programa Ladder do sistema que enche garrafas . . . 80
5.16 Inserção do erro 4 no programa Ladder do sistema que enche garrafas . . . 81
5.17 Inserção do erro 5 no programa Ladder do sistema que enche garrafas . . . 83
5.18 Sistema que controla semáforos 84
5.19 Esquema de funcionamento dos semáforos das ruas 1 e 2 85
5.20 Diagrama ISA 5.2 para o sistema que controla semáforos - Parte 1 86
5.21 Diagrama ISA 5.2 para o sistema que controla semáforos - Parte 2 87

5.22 Programa Ladder para o sistema que controla semáforos - Parte 1 88
5.23 Programa Ladder para o sistema que controla semáforos - Parte 2 89
5.24 Autômato que representa o processo de atualização de variáveis de entrada para o sistema que controla semáforos 90
5.25 Autômato que representa a modelagem do processamento dos sinais de entrada para o sistema que controla semáforos 90
5.26 Autômato que representa o temporizador Timer1 do sistema que controla semáforos 91
5.27 Autômato que representa o temporizador Timer2 do sistema que controla semáforos 92
5.28 Autômato que representa o temporizador Timer3 do sistema que controla semáforos 93
5.29 Autômato que representa o temporizador Timer4 do sistema que controla semáforos 94
5.30 Autômato que representa o temporizador Timer5 do sistema que controla semáforos 95
5.31 Autômato que representa o temporizador Timer6 do sistema que controla semáforos 96
5.32 Autômato que representam a execução do programa do sistema que controla semáforos 102
5.33 Modelagem do processo que avalia os estados das saídas do sistema que controla semáforos 103
5.34 Modelagem do processamento dos estados das saídas do sistema que controla semáforos 103
5.35 Autômato que representam ciclo de varredura do CLP para o sistema que controla semáforos 104
5.36 Inserção do erro 1 no programa Ladder para o sistema que controla dois semáforos 106
5.37 Inserção do erro 2 no programa Ladder para o sistema que controla dois semáforos 108

5.38 Inserção do erro 5 no programa Ladder para o sistema que controla dois semáforos 109

Lista de Tabelas

2.1 Resumo dos trabalhos referentes à modelagem e verificação de programas para CLPs 14

3.1 Elementos da linguagem Ladder 20

3.2 Símbolos ISA 5.2 28

Lista de Algoritmos

1 Geração dos autômatos que modelam variáveis de entrada 35
2 Geração do autômato que modela o processo de atualização de variáveis de entrada 37
3 Geração do autômato que modela o processamento dos sinais de entrada . . 38
4 Geração do autômato que modela temporizadores do tipo TON ou DI . . . 43
5 Geração do autômato que modela temporizadores do tipo TOF ou DT . . . 44
6 Geração do autômato que modela temporizadores do tipo TP ou PO 45
7 Procedimento inicializaAutomato() 50
8 Procedimento criaTransicaoSemTemporizador(Rung ou blocos que determinam saídas r, int indice) 51
9 Procedimento criaFuncaoOutNameTimer(Rung ou blocos que determinam saídas r, Temporizador t, int indice) 51
10 Procedimento criaFuncaoCheckExecutionNameTimer(Rung ou blocos que determinam saídas r, Temporizador t) 52
11 Procedimento criaFuncaoEvaluateOuputNameTimer(Rung ou blocos que determinam saídas r, Temporizador t) 53
12 Geração do autômato que modela o ciclo de execução da lógica do programa 54
13 Geração do autômato que modela o processo de avaliação dos estados das saídas 56
14 Geração do autômato que modelam o processamento dos estados das saídas 57
15 Geração do autômato que modela o ciclo de varredura de um CLP 59

Capítulo 1

Introdução

O desejo humano de controlar processos produtivos é antigo. Sua primeira manifestação se deu com o surgimento da manufatura, onde o homem era responsável pelo controle e pela execução de procedimentos envolvidos no processo manufatureiro. Porém, apesar de inovador para a época, tal método apresentava as seguintes desvantagens: baixa produtividade e qualidade regulada pelo homem. Com o desenvolvimento da máquina a vapor surgiu a idéia de utilizar máquinas para realizar alguns trabalhos manuais. Porém, o objetivo principal ainda não foi alcançado, que era deixar o homem na sua mera condição de ser pensante no processo industrial. Diante desse contexto, no século XX, surgiu o que chamamos de automação industrial, cujo principal objetivo é garantir maior eficiência no controle de processos com redução de custos e o aprimoramento da qualidade. Estes avanços foram possíveis devido ao surgimento da eletricidade e outros controles elétricos e eletrônicos (Capelli, 2006).

Automatizar processos é fundamental para indústrias. Uma das formas de introduzir automação numa indústria é utilizar, nas suas instalações um dispositivo denominado de Controlador Lógico Programável (CLP) (Parr, 2003). Este equipamento eletrônico-digital, compatível com aplicações industriais, é utilizado para controlar processos produtivos e, possui as seguintes vantagens quando comparados a outros dispositivos de controle industrial: baixo custo, fácil programação e manutenção, flexibilidade, maior confiabilidade e dispositivos de entrada e de saída facilmente substituíveis.

Com a inclusão de novas tecnologias nas indústrias tornou-se necessário superar os desafios que a automação industrial moderna trouxe consigo, tais como o de garantir a confi-

ança na operação de processos e a segurança tanto de equipamentos e instalações como de funcionários (Neves *et al.*, 2007), além do meio ambiente.

Com o objetivo de superar os desafios mencionados acima, partes críticas de plantas industriais passaram a ser monitoradas por Sistemas Instrumentados de Segurança (SIS) (Goble & Cheddie, 2005; Gruhn & Cheddie, 2006). Um SIS é responsável pela segurança operacional e tem a finalidade de prevenir a ocorrência de situações indesejadas quando da execução de procedimentos realizados automaticamente ou sob a interferência de operadores humanos. As situações indesejadas, por se tratar de um sistema de tempo real (Cooling, 2003), podem ocorrer devido às falhas em sensores, atuadores, ou executores da lógica de segurança ou pode estar relacionada aos erros na lógica em execução. Então, no caso de falhas, um SIS tem que garantir que a planta, ou parte dela, se manterá em um estado operacional seguro, considerando eventuais danos à instalação, aos seres humanos e ao meio ambiente. As operações de controle de SIS são realizadas por CLPs.

A empresa Petrobras pode ser citada como exemplo de uma indústria que aderiu a utilização de SIS em suas unidades com o intuito de garantir a segurança e a confiança na execução de seus processos.

1.1 Descrição do Problema

Este trabalho de pesquisa está inserido no contexto do projeto SIS em parceria com a empresa Petrobras. O objetivo no contrato deste projeto é desenvolver métodos, técnicas e ferramentas para aumentar a confiança e a segurança nos Sistemas Instrumentados de Segurança.

Este projeto teve início em 2006 e uma ferramenta denominada de SIS foi gerada. Esta ferramenta possui dois módulos:

- Módulo que gera modelos de autômatos temporizados a partir de Diagramas ISA 5.2 (ISA, 1992) e programas escritos na linguagem FBD (de Assis Barbosa *et al.*, 2007; da Silva *et al.*, 2008)
- Módulo que gera e executa casos de teste de conformidade.

Este trabalho de pesquisa é voltado, mais especificamente, para modelagem e verificação de programas para CLPs, parte crítica de um SIS, descritas utilizando Ladder (Bryan & Bryan, 1997).

Na Figura 1.1 é apresentado o processo de desenvolvimento de programas para CLPs adotado pela Petrobras. Primeiramente, a partir de dois arquivos, tabela de causa/efeito e de um texto estruturado que possui informações que não estão presentes na tabela, a companhia gera um conjunto de requisitos que constitui a especificação. Uma vez que a especificação está completa, ou seja, está no formato de Diagramas de Lógica Binária ISA 5.2, o software é desenvolvido e testado por uma empresa terceirizada. Este software é escrito segundo uma das cinco linguagens definidas no padrão IEC 61131-3 (PLCopen, 2004; John & Tiegelkamp, 2001). Finalmente, o programa é compilado e executado em um CLP.

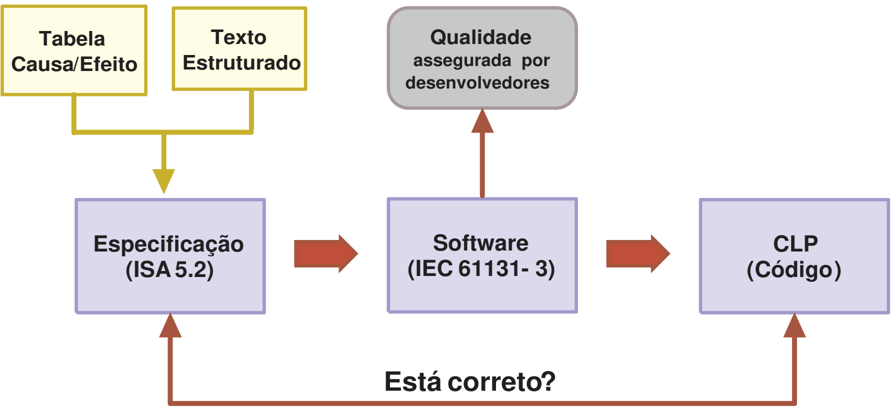


Figura 1.1: Processo de desenvolvimento de um CLP

O processo de teste utilizado pela empresa terceirizada consiste em testar módulos de programas à medida que são desenvolvidos. Desta forma, erros provocados por interações entre as partes previamente testadas podem não ser detectados. Uma medida utilizada para detecção de erros não encontrados durante o processo de teste, consiste na realização de testes de aceitação no próprio SIS a partir de combinações de entrada e análise do comportamento do mesmo. Caso algum erro seja detectado o software é novamente enviado para a equipe de desenvolvimento para que os erros possam ser eliminados. A utilização de testes de aceitação como técnica de validação de programas para CLPs é onerosa, pois, para combinações de grande número de variáveis de entrada tal processo pode levar muito tempo para ser executado.

Diante deste cenário de desenvolvimento, cuja qualidade do software gerado é assegurada apenas pelos testes de aceitação, uma grande questão é levantada: como é possível aumentar a confiança na execução de processos automatizados e a segurança tanto de equipamentos e

instalações como de funcionários numa indústria?. Neste trabalho um método que soluciona tal problema de forma confiável, automática e segura será ilustrado.

Pesquisas têm sido desenvolvidas com o intuito de definir formalismos e técnicas de verificação para avaliarem a confiança e a segurança de programas para CLPs, como pode ser visto em (Frey & Litz, 2000; Moon, 1994; Mader & Wupper, 1999; Zoubek, 2002; Zoubek *et al.*, 2003). Muitas são focadas em verificação automática de modelos (Katoen, 1999), técnica que consiste em verificar automaticamente a validade de propriedades acerca do comportamento de sistemas. Para trabalhos nessa linha de pesquisa, a técnica de verificação automática de modelos utiliza como instrumento de análise autômatos temporizados (Alur & Dill, 1994; Bengtsson & Yi, 2004), como podem ser vistos em (Zoubek *et al.*, 2003; Wang *et al.*, 2007; da Silva *et al.*, 2008), redes de Petri (Murata, 1989), como podem ser evidenciados em (Heiner & Menzel, 1998; Bender *et al.*, 2008) e, sistemas de transição, como podem ser visualizados em (Rausch & Krogh, 1998; Gourcuff *et al.*, 2006).

1.2 Objetivos do Trabalho

O objetivo deste trabalho é desenvolver um método e disponibilizar uma ferramenta que aumente a confiança e a segurança no funcionamento de SIS, mais especificamente nos programas de CLPs. O método proposto tem a finalidade de gerar casos de teste para validar a implementação a partir da especificação.

O método consiste em transformar, de forma automática, Diagramas de Lógica Binária ISA 5.2, que representam a especificação, e programas Ladder, que representam a implementação, em arquivos *eXtensible Markup Language* (XML). Estes arquivos descrevem autômatos temporizados no formato de entrada para a ferramenta Uppaal (Gerd Behrmann & Larsen, 2004). Após a geração destes autômatos e de sua posterior validação na ferramenta Uppaal, testes de conformidade são aplicados, de forma automática, sobre os mesmos com a finalidade de aumentar a confiança e a segurança do sistema. Para tal tarefa utiliza-se a ferramenta de teste Uppaal-TRON (Larsen *et al.*, 2005; Larsen *et al.*, 2007).

No sentido de atingir o objetivo principal, as seguintes etapas foram definidas:

1. Disponibilizar um método e uma ferramenta que gera automaticamente autômatos temporizados a partir da especificação, escrita em Diagramas de Lógica Binária ISA 5.2;

2. Disponibilizar um método e uma ferramenta que gera automaticamente autômatos temporizados a partir da implementação escrita em Ladder;

3. Gerar e executar casos de teste de conformidade a partir da ferramenta Uppaal-TRON. Esta etapa tem a finalidade de verificar se a implementação está em conformidade com a especificação;

4. Integrar as ferramentas acima mencionadas à ferramenta SIS.

1.3 Resultados e Relevância do Trabalho

A principal contribuição no contexto deste trabalho é a disponibilização de um método e de uma ferramenta que aumenta a segurança e a confiança de programas para CLPs. Tal tarefa é realizada através da geração e execução automática de casos de teste para o código do CLP. Portanto, erros existentes na implementação podem ser detectados e corrigidos antes destes terem aceitação de campo. Outro fator importante a ser mencionado é que o método proposto gera automaticamente autômatos temporizados a partir de documentos referentes à especificação, representada por Diagramas de Lógica Binária ISA 5.2, e documentos referentes à implementação, programas Ladder. Além disso, esse método lida com elementos temporais da linguagem Ladder que são explicitamente modelados e analisados em conjunto com a lógica Ladder para um programa completo.

1.4 Estrutura da Dissertação

Este documento está estruturado da seguinte forma: no capítulo 2 são apresentados os principais trabalhos realizados no domínio de modelagem e verificação de programas para Controladores Lógicos Programáveis. No capítulo 3 apresenta-se a fundamentação teórica, onde os principais conceitos relativos a Controladores Lógicos Programáveis (CLPs), Linguagem de Diagramas Ladder (LD), Autômatos Temporizados e Diagramas de Lógica Binária ISA 5.2 são apresentados, a intenção é disponibilizar embasamento teórico para os leitores. No capítulo 4 é apresentado o método introduzido nesta dissertação para aumentar a confiança e a segurança de SIS, mais especificamente de programas para CLPs. Neste capítulo também

são apresentados a modelagem, os algoritmos e a geração de autômatos temporizados a partir de Diagramas de Lógica Binária ISA 5.2 e Diagramas Ladder. No capítulo 5 dois estudos de caso são apresentados. No capítulo 6 apresentam-se as conclusões e os trabalhos futuros.

Capítulo 2

Trabalhos Relacionados

Este capítulo tem a finalidade de relatar os principais trabalhos realizados no domínio de modelagem e verificação de programas, escritos segundo a norma IEC 61131-3, para Controladores Lógicos Programáveis (CLPs).

Os trabalhos de (Frey & Litz, 2000) e (Younis & Frey, 2003) serviram como ponto de partida para o desenvolvimento desta pesquisa, pois, eles retratam os principais tipos de formalismos e métodos de verificação existentes para programas que são executados por CLPs. As metodologias utilizadas por estes dois trabalhos são detalhadas nos dois parágrafos subseqüentes.

Em (Frey & Litz, 2000) é proposto um modelo genérico de projeto de controle de processos para programas de CLPs, com ênfase na técnica de Verificação e Validação (V & V). Uma variedade de trabalhos, que utilizam esta técnica, são categorizados usando os seguintes critérios:

- Metodologia: utilizada para verificar se o processo do controlador é ou não levado em consideração;
- Formalismo: utilizado para descrever formalmente programas para CLPs. Os formalismos usados são: redes de Petri, autômatos, sistemas condição/evento, equações algébricas, linguagens síncronas e lógica de ordem superior;
- Método: utilizado para analisar a validade das propriedades. Simulação, análise de confiabilidade, verificação de modelos e prova de teoremas são exemplos de métodos utilizados para verificação de programas para CLPs.

Em (Younis & Frey, 2003) uma classificação para formalizar programas para CLPs é proposta. Esta classificação utiliza critérios que vão desde a definição da linguagem na qual são escritos os programas para CLPs, até a utilização de um modelo formal que descreve tais programas. Estes critérios também levam em consideração os níveis de formalização que se baseiam na complexidade das estruturas que o processo de formalização pode tratar, bem como no objetivo da formalização que implica quais métodos podem ser aplicados para a geração de modelos formais, tais como engenharia reversa, técnicas de V & V e reengenharia.

Com o intuito de garantir a segurança e a confiança de programas para CLPs torna-se necessário definir uma semântica formal que os descreva (Canet *et al.*, 2000; Heiner & Menzel, 1998; Mader & Wupper, 1999; Rossi & Schnoebelen, 2000). Esta semântica tem o objetivo de gerar um modelo que servirá de entrada para um verificador, onde propriedades poderão ser analisadas. Redes de Petri e autômatos temporizados são exemplos típicos de formalismos utilizados na verificação formal de tais programas (Frey & Litz, 2000; Younis & Frey, 2003).

Um exemplo de modelagem com redes de Petri é apresentado em (Heiner & Menzel, 1998). Neste trabalho é definido um procedimento para transformar fragmentos de um programa escrito na linguagem *Instruction List (IL)* em uma rede de Petri. O modelo do sistema é gerado a partir da composição do modelo de controle e do ambiente, ambas como uma rede de Petri. Após concluída a modelagem, requisitos funcionais e de segurança podem ser verificados por meio da rede gerada utilizando lógica temporal em um verificador de modelos. Em (Bender *et al.*, 2008) uma metodologia dirigida por modelos é apresentada. Nesta metodologia, programas escritos em diagramas Ladder (LD) são modelados e confrontados com metamodelos LD. Estes metamodelos são transformados em redes de Petri que é o formato de entrada da ferramenta Tina, onde esta entrada é simulada e verificada usando propriedades representadas em fórmulas *Linear Temporal Logic (LTL).*

Por outro lado, em diversos trabalhos utiliza-se a metodologia de transformar programas de controle em autômatos temporizados utilizando a técnica de verificação automática de modelos, onde as propriedades a serem verificadas podem ser expressas através da lógica temporal. Nos parágrafos seguintes as idéias centrais definidas nestes trabalhos são apresentadas.

Em (Mader & Wupper, 1999) o método proposto baseia-se na criação de dois modelos de autômatos temporizados para temporizadores. O primeiro modelo utiliza a idéia de como os temporizadores são representados nos CLPs, ou seja, através de blocos de funções. Como tal método utiliza inteiros, os modelos gerados são autômatos infinitos. O segundo modelo utiliza a idéia de como temporizadores funcionam, ou seja, utiliza a idéia intuitiva de um hardware de caixa preta, onde apenas a idéia de contagem de tempo é considerada. Como tal método utiliza relógios, ao invés de inteiros então, os modelos gerados são representados por autômatos finitos temporizados. Os dois modelos são equivalentes e satisfazem a especificação para temporizadores. Porém, o segundo se mostra mais útil quando a questão chave é a verificação de programas para CLPs, pois técnicas de verificação, tais como verificação automática de modelos, podem ser aplicadas. Estes modelos de temporizadores são executados em paralelo com o programa de controle escrito em *IL*. A sincronização é realizada através de operações sobre variáveis temporizadas e chamadas ao temporizador.

Em (Zoubek, 2002), é proposto um método que transforma cada degrau de um programa Ladder em modelos de autômatos temporizados, instrução por instrução. Cada instrução é modelada por um conjunto de transições entre dois ou mais estados. Por fim, estes modelos gerados são conectados entre si, com o auxílio de outros processos, gerando o ciclo de varredura do CLP. Existem particularidades no método proposto, tais como: apenas entradas digitais são consideradas e, para modelá-las é necessário um processo para cada entrada permitindo que todas as possíveis combinações de entradas possam ocorrer. Esta última particularidade causa o problema conhecido como explosão do espaço de estados, pois, para modelar um sistema, diversos autômatos têm que ser gerados. Após a geração destes modelos sua validação pode ser realizada utilizando o verificador de modelos Uppaal. Posteriormente, o resultado desta validação, especificação satisfeita ou não, é retornada aos programadores.

Em (Zoubek *et al.*, 2003), com o intuito de minimizar a explosão de espaço de estados causada pela completa construção do modelo do programa, o autor propõe um novo método que fixa a propriedade e a ser verificada e identifica qual parte do programa é afetada por ela, para que apenas essa parte do programa seja transformada em modelo. Assim, apenas os degraus e as variáveis de entrada que compõem essa parte do programa, junto com o ciclo de varredura, são transformados em uma rede de autômatos temporizados. A comunicação entre estes autômatos é realizada através de canais de sincronização. Após esse passo, ocorre

a simplificação do modelo intermediário gerado até então, através da remoção de transições redundantes, a fim de reduzir o tamanho do modelo. Após a etapa de simplificação, propriedades podem ser validadas no verificador de modelos Uppaal.

Em (Remelhe *et al.*, 2004), programas escritos em *Sequencial Function Chart (SFC)* são transformados em uma rede de autômatos temporizados. Essa transformação, baseada na representação gráfica de programas *SFC*, é realizada pelas seguintes etapas:

1. Verificação de identificadores de *steps* e ações para corretude sintática;

2. Analise sintática de gráficos SFCs quanto às seguintes convenções: um dentre os gráficos que formam o modelo completo do SFC é definido como gráfico principal, os demais são subordinados e executados como ações; cada gráfico pode ativar ou desativar o outro, com exceção do gráfico principal;

3. Conversão de gráficos SFC, através de um analisador gráfico, de acordo com as regras de redução de uma gramática gráfica definida no artigo;

4. Criação de modelos de autômatos temporizados baseados no analisador gráfico. Após a geração destes modelos, propriedades, expressas em lógica temporal *CTL*, são verificadas na ferramenta Uppaal.

Em (da Silva *et al.*, 2008), algoritmos para extração automática de autômatos temporizados a partir de *Function Block Diagrams (FBD)* são ilustrados. O modelo é composto por 4 tipos de autômatos. O primeiro é um modelo para flip-flops e elementos temporizados. O segundo é o modelo do ambiente para entradas físicas. O terceiro modela o ciclo de varredura, e o quarto modela o controle de execução e monitora a convergência de blocos de funções. Em (de Assis Barbosa *et al.*, 2007), Diagramas de Lógica Binária ISA 5.2 são convertidos em autômatos temporizados utilizando uma modelagem similar à metodologia anterior.

Além de redes de Petri e autômatos temporizados, sistemas de transição de estados, específicos para cada tipo de verificador, são utilizados para modelar programas para CLPs. *Symbolic Model Verifier (SMV)*, *Cadence Symbolic Model Verifier (CaSMV)* e *A New Symbolic Model Verifier (NuSMV)* são exemplos de verificadores que utilizam estes tipos de modelos e que realizam verificação de propriedades, expressas em lógica temporal *LTL* e *Computation Tree Logic (CTL)*, no modelo gerado.

Em (Rausch & Krogh, 1998) programas para CLPs são modelados como diagramas de bloco Condição/Evento os quais são transformados em módulos *SMV*, onde propriedades, escritas em lógica temporal *CTL*, podem ser verificadas. O comportamento do programa em cada ciclo de varredura é mantido na modelagem. O objetivo principal deste trabalho é criar uma interface entre a programação e as ferramentas que utilizam *SMV* para que, todas as possíveis condições de funcionamento do programa do controlador sejam validadas antes dele ser utilizado por um sistema real.

Os trabalhos a seguir utilizam a ferramenta *CaSMV* para realizar verificação, através da checagem de propriedades, num modelo formal denominado de estrutura de Kripke. (Rossi & Schnoebelen, 2000) mostra como converter programas Ladder, com presença de temporizadores, em tais modelos. Após a geração destes modelos, a verificação de propriedades representadas por elementos da lógica temporal *LTL*, pode ser realizada. Em (Canet *et al.*, 2000) é apresentado como o comportamento de um programa escrito em *IL* é transformado em uma estrutura de Kripke e, como a verificação de propriedades, segundo a lógica temporal *LTL* pode ser analisada. Em (Bauer & Huuck, 2001) uma metodologia para transformar, automaticamente, programas *SFC* em estruturas de Kripke é ilustrada. A verificação de propriedades escritas em lógica temporal *CTL* e *LTL* pode ser realizada utilizando estas estruturas. Em (Smet *et al.*, 2000) um método para transformar programas para CLPs, escritos em *LD*, *Structured Text (ST)* ou *SFC*, em sistemas de transição é apresentado. Para cada tipo de linguagem existe um procedimento particular que mapeia os programas em sistemas de transição. Assim, para a linguagem Ladder, ocorre mapeamento dos degraus, para a linguagem *ST* cada elemento é descrito como um autômato e, para a linguagem *SFC* ocorre mapeamento da evolução das regras do programa. Após esse passo, propriedades, escritas em lógica temporal *CTL*, podem ser verificadas. É importante lembrar que para cada linguagem o modelo de verificação tem que ser validado independentemente. Em (Smet & Rossi, 2002) o detalhamento da transformação de programas Ladder em um sistema de transição é exibida. Este método se limita a sistemas pequenos e não suporta sistemas complexos.

Em (Gourcuff *et al.*, 2006) programas escritos em *ST*, sem a presença de temporizadores, são convertidos em máquinas de estados finitas. O método proposto é definido em duas etapas: 1) Análise estática: visa derivar, a partir de programas para CLPs, relações de dependência entre variáveis; 2) Transformação de *ST* em módulos *NuSMV*: descreve formal-

mente o comportamento do programa com base em suas entradas e saídas. Desta forma, cada declaração *ST* que dá origem à última relação de dependência é transformada em uma atribuição segundo o modelo *NuSMV*. Após a geração dos modelos, propriedades, escritas em lógica temporal *CTL*, podem ser verificadas na ferramenta *NuSMV*. Apenas propriedades que se referem ao comportamento das saídas são verificadas, pois são elas que geram impacto direto na segurança e na confiança de processos de controle. Este método minimiza o tempo de verificação e não permite a ocorrência do problema conhecido como explosão do espaço de estados.

Outros tipos de formalismo também são utilizados para modelar programas para CLPs. Em (Moon, 1994), segurança e operabilidade de programas escritos em *Relay Ladder Logic (RLL)* são verificadas. O método consiste de modelos de sistemas, asserções e um verificador de modelos que tem a finalidade de verificar se o modelo satisfaz as asserções. Neste trabalho, cada degrau de um programa *RLL* é transformado em uma regra, que por sua vez, representa o modelo do sistema. Em (Younis & Frey, 2004) é sugerida uma metodologia de reengenharia baseada na formalização e visualização de programas para CLPs. Assim, *XML* e outras tecnologias correspondentes são utilizadas para formalização e visualização de programas para CLPs existentes. Para tanto, códigos de programas para CLPs são transformados em arquivos *XML*, validados através do confronto com o *XML schema* e posteriormente identificações de instruções são realizadas. Esta transformação oferece a vantagem de obter uma especificação independente do código do fornecedor. Baseado nesse código, o passo a passo da transformação do modelo formal é planejado. Este modelo pode então ser usado para análise, simulação, validação formal e para reimplementação de um algoritmo otimizado ou outro algoritmo semelhante para outro PLC.

Observando os resultados dos trabalhos analisados podemos verificar que a técnica de verificação automática de modelos é mais utilizada. Porém, esta técnica é inadequada para sistemas complexos pelo fato dela causar o problema conhecido como explosão do espaço de estados. Além disso, esta técnica não é muito utilizada nas indústrias, pois, os contra-exemplos ilustrados para a invalidade de propriedades muitas vezes são de difícil interpretação, fazendo com que tal técnica não seja tão atrativa. Então, diante destes fatos, necessitamos buscar outra técnica de verificação que possa lidar com sistemas complexos. Assim, neste trabalho é proposto a utilização de testes de conformidade como técnica de verificação.

Após definida a técnica de verificação a ser utilizada, necessitamos identificar qual o tipo adequado de formalismo para modelar o comportamento do sistema. Para esta tarefa, algumas premissas foram estabelecidas:

- A representação formal deve possibilitar construção modular dos modelos referentes à especificação (Diagramas ISA 5.2) e a implementação (código Ladder), ou seja, o formalismo deve suportar manipulação de operações lógicas booleanas e lidar com o tempo de forma qualitativa;
- Para tal formalismo, deve haver disponibilidade de ferramentas abertas para: construção, análise, verificação e simulação dos modelos gerados;
- Para tal formalismo deve haver disponibilidade de ferramentas que gerem e executem automaticamente casos de teste de conformidade.

Redes de Petri e autômatos temporizados são tipos de formalismos que satisfazem a primeira premissa. Para a segunda e terceira premissa, apenas o formalismo de autômatos temporizados pode ser aplicado, através do uso das ferramentas Uppaal, para edição e validação dos modelos e a ferramenta Uppaal-TRON para geração e execução de casos de teste de conformidade.

Na tabela 2.1 encontra-se o resumo dos artigos referentes à modelagem e verificação de programas para CLPs. A separação de cada artigo é estabelecida através dos seguintes critérios: tipo de linguagem de programação utilizada, tipo de formalismo utilizado, técnica de verificação utilizada e modelagem de elementos temporizados.

Tabela 2.1: Resumo dos trabalhos referentes à modelagem e verificação de programas para CLPs

Autores	**Linguagem**	**Formalismo**	**Técnica de Verificação**	**Elementos temporizados**
(Moon, 1994)	*RLL*	Regras	Verificação de modelos	Não modela
(Heiner & Menzel, 1998)	*IL*	Redes de Petri	Não utiliza	Não modela
(Rausch & Krogh, 1998)	*SFC*	Diagramas de bloco C/E	Verificação de modelos	Não modela
(Mader & Wupper, 1999)	*IL*	Autômatos temporizados	Não utiliza	Modela
(Canet *et al.*, 2000)	*IL*	Estrutura de Kripke	Verificação de modelos	Não modela
(Rossi & Schnoebelen, 2000)	*LD*	Estrutura de Kripke	Verificação de modelos	Não modela
(Smet *et al.*, 2000)	*LD, ST e SFC*	Sistema de transição	Verificação de modelos	Não modela
(Bauer & Huuck, 2001)	*SFC*	Estrutura de Kripke	Verificação de modelos	Não modela
(Smet & Rossi, 2002)	*LD*	Sistema de transição	Verificação de modelos	Não modela
(Zoubek, 2002)	*LD*	Autômatos temporizados	Verificação de modelos	Não modela
(Zoubek *et al.*, 2003)	*LD*	Autômatos temporizados	Verificação de modelos	Não modela
(Remelhe *et al.*, 2004)	*SFC*	Autômatos temporizados	Verificação de modelos	Não modela
(Younis & Frey, 2004)	Não define	XML	Não utiliza	Não modela
(Gourcuff *et al.*, 2006)	*ST*	Máquina de estados	Verificação de modelos	Não modela
(Bender *et al.*, 2008)	*LD*	Redes de Petri	Verificação de modelos	Não modela
(da Silva *et al.*, 2008)	*FBD*	Autômatos temporizados	Testes de conformidade	Modela

Capítulo 3

Fundamentação Teórica

O objetivo deste capítulo é fornecer um embasamento teórico necessário para o entendimento desse trabalho. São apresentados os principais conceitos relacionados a Controladores Lógicos Programáveis (CLPs), Autômatos Temporizados, Linguagem de Diagramas Ladder (LD), foco deste trabalho, e Diagramas de Lógica Binária ISA 5.2.

3.1 Controladores Lógicos Programáveis

O Controlador Lógico Programável (CLP) surgiu no final da década de 1960, na empresa General Motors, com o intuito de substituir os relés, pois, a cada nova mudança na linha de montagem desta empresa a lógica de controle dos painéis tinha que ser reprogramada, fazendo com que tal tarefa seja onerosa (Parr, 2003). Desde o seu surgimento, até hoje, os CLPs evoluíram passando a ser mais versáteis e de fácil utilização tornando-se assim um dos equipamentos mais atraentes na automação industrial.

Segundo o Nema (Nema - National Electrical Manufacturers Association, 2005), um Controlador Lógico Programável é um aparelho eletrônico digital que utiliza uma memória programável para armazenamento interno de instruções para implementações específicas como lógica, seqüenciamento, temporização, contagem e aritmética; para controlar, através de módulos de entradas e saídas, vários tipos de máquinas ou processos.

As principais vantagens da utilização dos Controladores Lógicos Programáveis, quando comparados a outros dispositivos de controle industrial, são (John & Tiegelkamp, 2001; Parr, 2003):

- Menor espaço ocupado;
- Menor potência elétrica requerida;
- Reutilização;
- Fácil programação;
- Maior confiabilidade;
- Fácil manutenção;
- Maior flexibilidade.

Na Figura 3.1 a estrutura de um CLP é ilustrada. Um CLP é dividido em três partes: entradas e saídas, que podem ser digitais ou analógicas, e CPU, parte responsável por realizar o processamento do programa do usuário e a atualização de dados na memória.

Figura 3.1: Estrutura básica de um CLP

O CLP funciona de forma seqüencial, fazendo um ciclo de varredura (*scan*), em algumas etapas, onde cada etapa é exclusiva, ou seja, quando ela é executada as demais ficam inativas. O tempo total para executar cada ciclo é denominado tempo de varredura (*scan*).

Na Figura 3.2 é ilustrado o ciclo de varredura de um CLP (Bryan & Bryan, 1997; Zoubek *et al.*, 2003). Primeiramente quando o CLP é inicializado ele executa uma série de pré-operações, tais como: desativa todas as saídas e realiza a verificação do funcionamento do CLP, da memória, da configuração interna e da existência de um programa de usuário. Após isso, o ciclo de varredura é inicializado. Seu funcionamento se dá da seguinte forma. Primeiramente, as entradas do sistema são lidas e os seus valores armazenados na memória. Depois disso, o programa é executado e, na última etapa, todas as saídas são ativadas com

base em seus valores armazenados na memória. Um outro ciclo de varredura pode ser novamente iniciado, na mesma seqüência mostrada e sem interrupções.

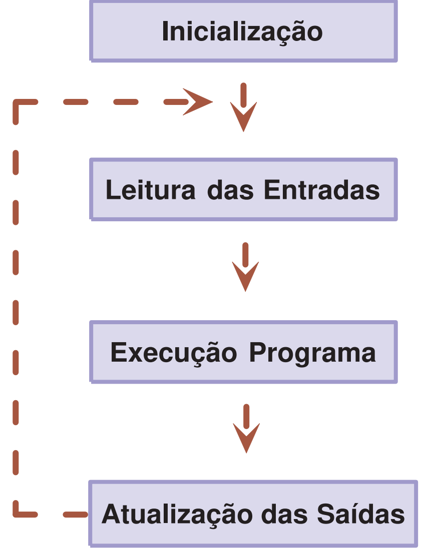


Figura 3.2: Ciclo de Varredura de um CLP

Programas para CLPs podem ser escritos em uma das cinco linguagens definidas pelo padrão IEC 61131-3 (PLCopen, 2004; John & Tiegelkamp, 2001). Veja:

- *Instruction List (IL)* é uma linguagem textual que se assemelha à linguagem *assembler*;
- *Ladder Diagram (Ladder)* é baseada numa representação gráfica de *Relay Ladder Logic (RLL)*;
- *Function Block Diagram (FBD)* expressa o comportamento de um controlador como um conjunto de blocos gráficos interligados;
- *Structured Text (ST)* é uma linguagem de alto nível muito poderosa parecida com a linguagem *Pascal*;
- *Sequential Function Chart (SFC)* modela lógicas de controle baseadas na seqüência temporal de eventos de processo.

3.2 Linguagem de Diagramas Ladder

A linguagem de Diagramas Ladder (LD), ou simplesmente Ladder, é uma das cinco linguagens definida pelo padrão internacional IEC 61131-3 as quais são utilizadas para construir aplicações para CLPs. LD é uma linguagem gráfica, oriunda dos Estados Unidos, que se assemelha muito aos circuitos de relés.

Ladder possui esse nome pelo fato da sua representação se parecer com uma escada, na qual duas barras verticais paralelas, uma esquerda e uma direita, que representam respectivamente o barramento energizado e barramento terra, interligam a lógica de controle que forma os degraus *(rungs)* (Bender *et al.*, 2008). Cada degrau é formado por uma lógica de controle que por sua vez é constituída de linhas e colunas, onde estão localizados os elementos da linguagem, cuja quantidade é definida pelo fabricante do CLP. Estes detalhes podem ser visualizados na Figura 3.3.

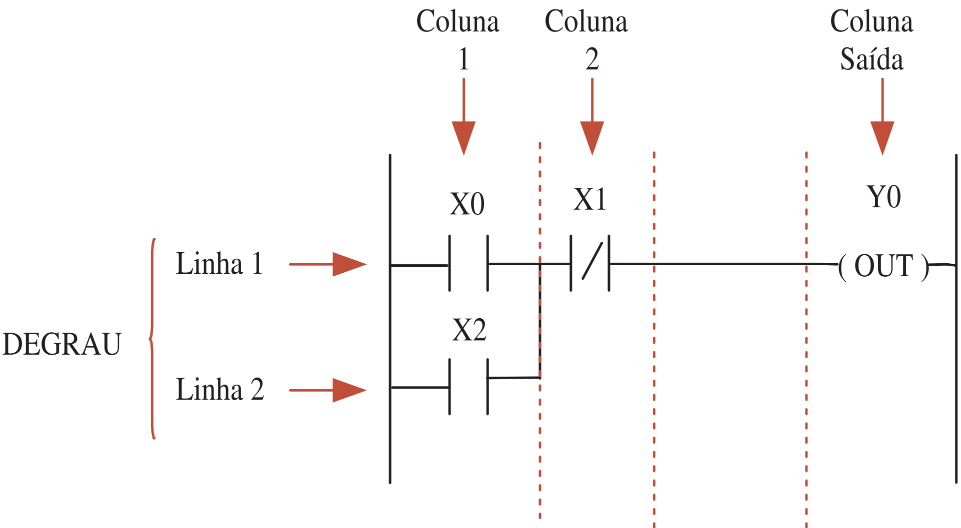


Figura 3.3: Esquema gráfico da linguagem Ladder

Cada lógica de controle deve ser programada de forma que as instruções sejam energizadas no sentido da corrente elétrica entre as duas barras. A corrente flui da esquerda para a direita em cada linha e energiza seqüencialmente cada coluna da linha que está percorrendo. Dizemos que um degrau de um programa Ladder está habilitado (saída energizada), segundo o estado atual de suas variáveis de entrada, quando existe um caminho que gera continuidade lógica entre as duas barras e, que este degrau está desabilitado quando não

há continuidade lógica entre as mesmas (Bryan & Bryan, 1997). A Figura 3.4 ilustra um exemplo de um degrau de um programa Ladder com os possíveis caminhos que provêem continuidade lógica e energizam a saída do degrau.

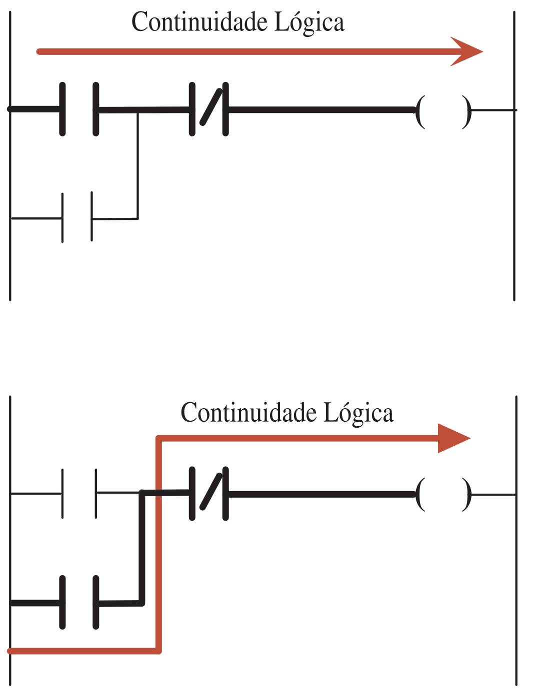


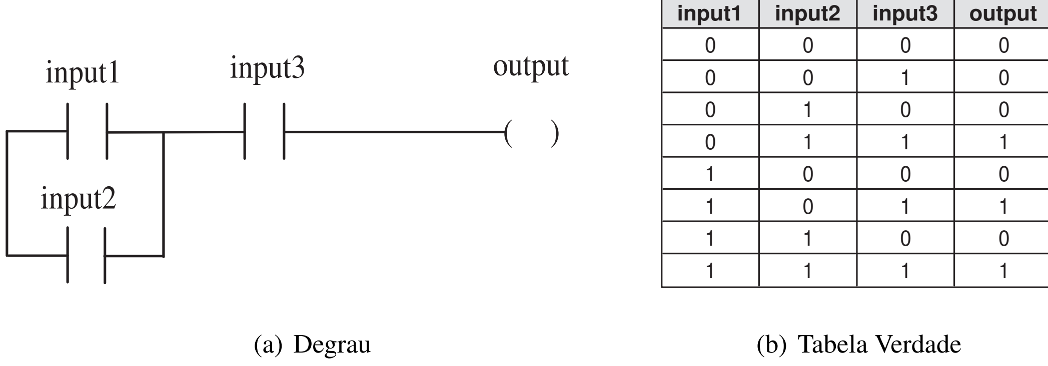
Figura 3.4: Caminhos de continuidade lógica de um programa Ladder

Nos parágrafos acima vimos como ocorre o princípio de funcionamento da linguagem Ladder, agora veremos, na Tabela 3.1, os principais elementos que compõem tal linguagem. Estes elementos podem ser combinados em série, formando uma operação booleana AND, em paralelo, formando uma operação booleana OR, ou uma composição destas combinações.

Tabela 3.1: Elementos da linguagem Ladder

Símbolo	Descrição
InputName —| |—	Contato Normalmente Aberto: representa uma entrada da lógica de controle. O CLP examina o bit especifico correspondente a este símbolo e retorna 0 se a o bit é 0 e retorna 1 se o bit é 1.
InputName —|/|—	Contato Normalmente Fechado: representa uma entrada da lógica de controle. O CLP examina o bit especifico correspondente a este símbolo e retorna 1 se a o bit é 0 e retorna 0 se o bit é 1.
OutputName ( )	Bobina: representa uma saída. É um elemento atuador, o qual é acionado ou desligado pela lógica de controle.
TON IN Q PT	Temporizador TON ou TMR: representa um elemento temporizado. Energiza a saída (Q = 1) depois de um período de tempo (tempo = PT) quando exite lógica 1 na entrada (IN = 1).
TOF IN Q PT	Temporizador TOF: representa um elemento temporizado. Energiza a saída (Q = 1) quando existe lógica 1 na entrada (IN = 1). Quando a lógica na entrada passar a ser 0 (IN = 0) a saída será desenergizada (Q = 0) após um período de tempo (tempo = PT).
TP IN Q PT	Temporizador TP: representa um elemento temporizado. Energiza a saída (Q = 1) durante um período de tempo (tempo = PT) quando existe lógica 1 na entrada (IN = 1). Mesmo que exista lógica 0 na entrada (IN = 0) a saída continua ativada (Q = 1) durante o tempo pré-definido (tempo = PT).

Na Figura 3.5(a), um exemplo de um programa Ladder é ilustrado. As variáveis *input1, input2* e *input3* são chamadas de contatos normalmente abertos e a variável *output* é chamada de bobina. As variáveis *input1* e *input2* são ligadas em paralelo (operação Booleana OR) que por sua vez são ligadas em série com *input3* (operação Booleana AND). Portanto, *output = (input1 or input2) and input3*. Na Figura 3.5(b) a tabela verdade para a saída *output* é ilustrada.

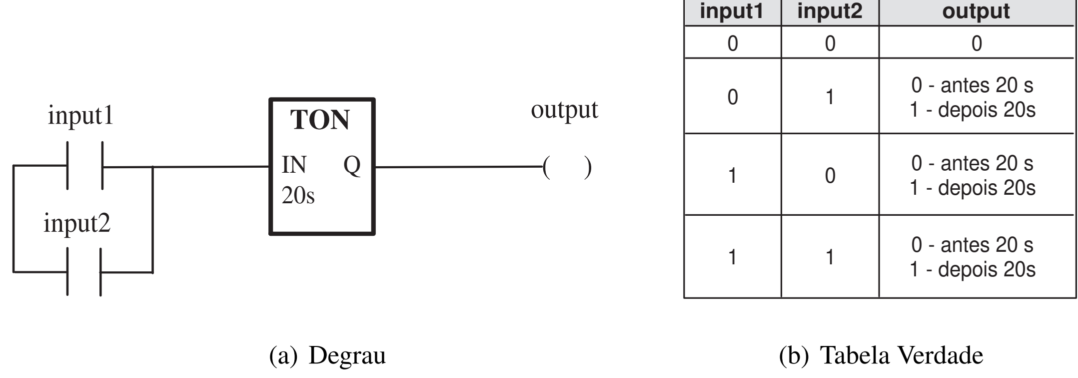


input1	input2	input3	output
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

(a) Degrau (b) Tabela Verdade

Figura 3.5: Exemplo de um degrau de um programa Ladder

Na Figura 3.6(a), outro exemplo de um programa Ladder é ilustrado. As variáveis *input1* e *input2* são chamadas de contatos normalmente abertos e a variável *output* é chamada de bobina. As variáveis *input1* e *input2* são conectadas em paralelo (operação Booleana OR). Como existe um temporizador no degrau então, o valor lógico de *output* depende da saída do temporizador TON. Portanto, se a operação lógica de entrada do temporizador é verdadeira, *input1 or input2 = 1*, então o temporizador inicia sua operação de contagem; quando este tempo chegar em 20s (valor de PT) *output* é ativada (valor lógico 1); caso a operação lógica de entrada seja falsa, *input1 or input2 = 0*, então *output* é desativada (valor lógico 0). Na Figura 3.6(b) a tabela verdade para a saída *output* é ilustrada.



input1	input2	output
0	0	0
0	1	0 - antes 20 s 1 - depois 20s
1	0	0 - antes 20 s 1 - depois 20s
1	1	0 - antes 20 s 1 - depois 20s

(a) Degrau (b) Tabela Verdade

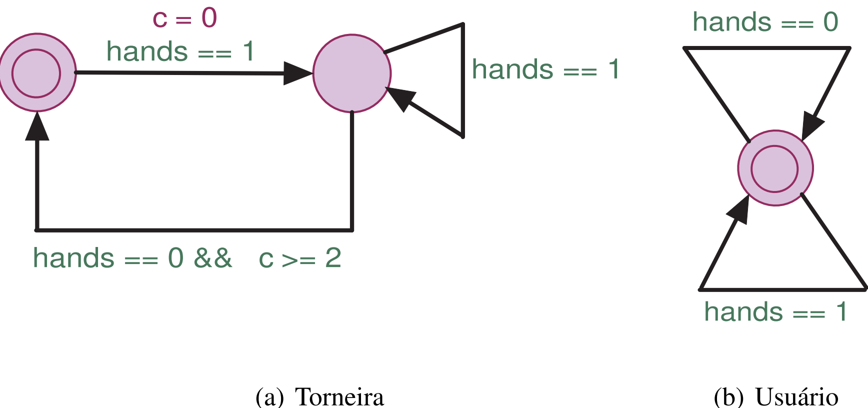
Figura 3.6: Exemplo de um degrau de um programa Ladder com temporizador

3.3 Autômatos Temporizados

Autômatos temporizados são máquinas de estados finitos com restrições de temporização associadas as suas arestas e estados e têm o objetivo de modelar o comportamento de sistemas de tempo real (Alur, 1999; Alur & Dill, 1994; Bengtsson & Yi, 2004). As restrições são construídas a partir de variáveis de controle de tempo, chamadas de relógios. Estes relógios progridem de forma síncrona e são declarados como valores reais, pois o tempo considerado é contínuo.

Neste trabalho focaremos apenas nos modelos de autômatos temporizados utilizados pelas ferramentas Uppaal e Uppaal-TRON, pois estas ferramentas foram utilizadas para modelar e gerar os testes de conformidade nas redes de autômatos geradas.

Na Figura 3.7, uma pequena introdução da sintaxe e semântica dos autômatos temporizados na ferramenta Uppaal é ilustrada. Nessa Figura, a modelagem de uma torneira como um autômato temporizado é apresentada. Esta torneira possui um acionamento automático do fluxo de água quando as mãos a pressionam. O autômato possui duas localidades: *off*, torneira desligada, e *on*, torneira ligada. As expressões próximas aos arcos, ilustradas na cor verde, representam guardas, e expressões do tipo $c = 0$, ilustradas na cor roxa, representam atualização de variáveis. Se o usuário pressionar a torneira ($hands == 1$), então o relógio, representado por *c* é inicializado, *c = 0*, e a torneira é ligada, ou seja, a transição que leva a localidade *off* para a localidade *on* é disparada. Nesta localidade, podem ocorrer duas situações: o usuário pode continuar pressionando a torneira ($hands == 1$), para que ela permaneça ligada; ou o usuário pára de pressionar a torneira e quando dois segundos se passarem ($hands == 0$ && $c >= 2$) a torneira será desligada, ou seja, a transição que leva a localidade *on* para a localidade *off* é disparada.



(a) Torneira (b) Usuário

Figura 3.7: Modelagem do funcionamento de uma torneira como uma rede de autômatos temporizados

A seguir serão mostradas algumas definições formais que facilitarão a compreensão da sintaxe e semântica de autômatos temporizados. As seguintes notações serão utilizadas: C é o conjunto de relógios e B(C) um conjunto de conjunções sobre condições simples da forma x $\bowtie$ c ou x-y $\bowtie$ c, onde x, y $\in$ C, c $\in$ N e $\bowtie \in \{<, \leq, =, \geq, >\}$.

Definição 1: Um autômato temporizado é um sêxtuplo (L, l_0, C, A, E, I), onde:

- L é o conjunto de localidades;
- $l_0 \in$ L é a localidade inicial;
- C é o conjunto de relógios;
- A é o conjunto de ações, co-ações e ações internas;
- E $\subseteq$ L x A x B(C) x 2^c x L é o conjunto de arestas entre localidades com uma ação, uma guarda e um conjunto de relógios que serão restaurados;
- I: L $\rightarrow$ B(C) atribui invariantes às localidades (define o intervalo de tempo que o sistema pode permanecer em um determinado estado).

A valoração do relógio é uma função u: C $\rightarrow R_{\geq 0}$ do conjunto dos relógios para os reais não-negativos. Façamos R^c ser o conjunto de todas as valorações dos relógios. Façamos u_0(x) = 0 para todo x $\in$ C. Notações considerando guardas e invariantes como o conjunto de

valoração do relógio são utilizadas através da utilização u $\in$ I(l) querendo dizer que u satisfaz I(l).

Definição 2: Seja (L, l_0, C, A, E, I) um autômato temporizado. A semântica é definida como um sistema de transição etiquetada $\langle$S, s_0, $\rightarrow\rangle$, onde S $\subseteq$ L x R^c é o conjunto de estados, s_0 = (l_0, u_0) é o estado inicial, e $\rightarrow\subseteq$ S x { $R_{\geq 0}$ $\cup$ A } x S é a relação de transição tal que:

- (l, u) $\rightarrow^d$ (l, u + d) se $\forall$ d´: 0 $\leq$ d´ $\leq$ d $\Rightarrow$ u + d´$\in$ I(l), e

- (l, u) $\rightarrow^a$ (l´, u´) se existe e = (l , a , g, r, l´) $\in$ E tal que u $\in$ g, u´= [r $\mapsto$ 0]u, e u´$\in$ I(l), onde para d $\in$ $R_{\geq 0}$, u + d mapeia cada relógio x em C para o valor u(x) + d, e [r $\mapsto$ 0]u denota a valoração do relógio que leva cada relógio de r até 0, ou seja, todos os relógios são restaurados.

Os autômatos temporizados são compostos freqüentemente por uma rede de autômatos temporizados sobre um conjunto comum de relógios e ações, consistindo em *n* autômatos temporizados, A_i = (L_i, l_i^0, C, A, E_i, I_i), onde 1 $\leq$ i $\leq$ n. Um vetor de localidade é um $\bar{l}$ = (l_1, ..., l_n). As funções invariantes de cada autômato da rede são compostas numa função comum sobre os vetores de posição I($\bar{l}$) = $\bigwedge_i I_i(l_i)$. Denota-se por $\bar{l}[l_i'$ / $l_i]$ o vetor onde o i-ésimo elemento l_i de $\bar{l}$ é substituído por l_i'. A seguir a semântica de uma rede de autômatos temporizados é definida.

Definição 3: Seja A_i = (L_i, l_i^0, C, A, E_i, I_i) uma rede de n autômatos temporizados. E seja $\bar{l_0}$ = (l_1^0, ..., l_n^0) o vetor de localidade inicial. A semântica de uma rede de autômatos temporizados é definida como um sistema de transição $\langle$S, s_0, $\rightarrow\rangle$, onde S = (L_1, ..., L_n) x R^c é o conjunto de estados, s_0 = $(\bar{l_0}, u_0)$ é o estado inicial, e $\rightarrow\subseteq$ S x S é a relação de transição definida por:

- ($\bar{l}$, u) $\rightarrow$ ($\bar{l}$, u + d) se $\forall$ d´: 0 $\leq$ d´ $\leq$ d $\Rightarrow$ u + d´$\in$ I($\bar{l}$), e

- ($\bar{l}$, u) $\rightarrow$ ($\bar{l}[l_i'$ / $l_i]$, u') se existe l_i $\rightarrow^{rgr}$ l_i' tal que u $\in$ g, u' = [r $\mapsto$ 0]u e u'$\in$ I($\bar{l}$)

- ($\bar{l}$, u) $\rightarrow$ ($\bar{l}[l_j'$ / l_j , l_i' / $l_i]$, u') se existe l_i $\rightarrow^{c?g_ir_i}$ l_i' e l_j $\rightarrow^{c!g_jr_j}$ l_j' tal que u $\in$ ($g_i \bigwedge g_j$), u´=´[$r_i \bigcup r_j$ $\mapsto$ 0]u e u'$\in$ I($\bar{l}$)

Os autômatos temporizados em Uppaal são extensões dos autômatos temporizados com as seguintes propriedades adicionais:

- Templates: são definidos com um conjunto de parâmetros que podem ser de diversos tipos (por exemplo, int, chan). Estes parâmetros são substituídos por um dado

argumento na declaração de processos;

- Constantes: são declaradas como *const nome valor*. As constantes por definição não podem ser modificadas e devem ser inteiras;

- Variáveis Inteiras Limitadas: são declaradas como *int[min,max] nome*, onde *min* e *max* são o limite inferior e superior, respectivamente. Os guardas, os invariantes, e as atribuições podem conter expressões que variam sobre variáveis inteiras limitadas. Os limites são avaliados na verificação, e a violação de um limite conduz a um estado inválido que é rejeitado. Se os limites forem omitidos, a escala usada é de *-32768* a *32768*;

- Sincronização Binária: são declarados como *chan c*. Uma aresta etiquetada com *c!* sincroniza-se com outra etiquetada com *c?*. Um par de sincronizações é escolhido de forma não determinística se forem permitidas diversas combinações;

- Canais de Broadcast: são declarados como *broadcast chan c*. Numa sincronização da transmissão um remetente *c!* pode sincronizar-se com um número arbitrário de receptores *c?*. Todo receptor que puder sincronizar no estado atual o deve fazer. Se não houver nenhum receptor, então o remetente pode continuar a executar a ação *c!*, isto é, a emissão da transmissão nunca será bloqueada;

- Canais Urgentes de Sincronização: são declarados quando na declaração do canal a palavra chave *urgent* é utilizada. Os atrasos não devem ocorrer se a sincronização da transmissão num canal urgente é habilitada. As arestas que usam canais urgentes para a sincronização não podem ter restrições de tempo, isto é, não devem possuir relógios nas guardas;

- Localidades Urgentes: são semanticamente equivalentes a adicionar um relógio extra x, que é restaurado em todas as arestas de chegada, e tendo uma invariante na posição. Portanto, o tempo não deve decorrer quando o sistema está numa posição urgente;

- Localidades Committed: são ainda mais restritivas na execução do que as localidades urgentes. Um estado é *comitted* se algumas das localidades no estado forem *comitted*.

Um estado *comitted* não pode atrasar e a próxima transição deve conter uma aresta que parta pelo menos de uma das localidades *comitted*;

- Arrays: são permitidos para os relógios, canais, constantes e variáveis inteiras. São definidos adicionando um tamanho ao nome da variável, por exemplo, chan c[4];
- Inicializadores: são usados para inicializar variáveis inteiras e arrays de variáveis inteiras. Por exemplo, int i: = 2; ou int i[3]: = 1, 2, 3.

As expressões em UPPAAL variam sobre relógios e variáveis inteiras. As expressões são usadas com as seguintes etiquetas:

- Guarda: uma guarda é uma expressão particular que satisfaz as seguintes condições: é livre de efeitos colaterais; é avaliado por uma expressão booleana; somente os relógios, as variáveis inteiras, e as constantes são referenciados (ou arrays destes tipos); os relógios e as diferenças de tempo são comparados somente com expressões inteiras; os guardas sobre os relógios são essencialmente conjunções (as disjunções são permitidas sobre condições inteiras);
- Sincronização: uma etiqueta de sincronização é qualquer expressão da forma *Expression!* ou *Expression?* ou é uma etiqueta vazia. A expressão deve ser livre de efeitos colaterais, avaliada por um canal, e somente referida para inteiros, constantes e canais;
- Assignment: uma etiqueta de atribuição é uma lista de expressões separadas por vírgulas com efeito colateral; as expressões devem somente se referir a relógios, variáveis inteiras, e constantes, e apenas valores inteiros devem ser atribuídos aos relógios;
- Invariante: um invariante é uma expressão que satisfaz as seguintes condições: é livre de efeitos colaterais; apenas relógios, variáveis inteiras e constantes são referenciadas; é uma conjunção de condições da forma $x < e$ ou $x \leq e$, onde x é o relógio referido e e é avaliado como um inteiro.

3.4 Diagramas de Lógica Binária ISA 5.2

O objetivo do padrão ISA 5.2 é prover um método de diagramação lógica de entreligamento binário e seqüenciamento de sistemas. Esta norma destina-se a facilitar a compreensão do funcionamento dos sistemas binários e melhorar a comunicação entre os técnicos, gerentes, projetistas e o pessoal responsável por operar e manter os sistemas (ISA, 1992).

A Norma ISA 5.2 fornece símbolos para representar operações de processos através de funções operacionais binárias. Alguns destes símbolos são ilustrados na Tabela 3.2.

Este padrão simboliza as funções operacionais binárias de um sistema de maneira que pode ser aplicado a qualquer classe de hardware, seja ela eletrônica, elétrica, hidráulica, mecânica, manual, óptica, entre outras.

A leitura de um diagrama ISA 5.2 deve ser feita da esquerda para a direita e de cima para baixo e a alteração no sentido de fluxo convencional deve ser explicitada através de setas.

O diagrama ISA 5.2 de um exemplo introduzido em (Parr, 2003) é ilustrado na Figura 3.8. O exemplo relata o funcionamento do arranque de um motor o qual é acionado e parado com o auxílio dos repectivos botões *StartPB* e *StopPB*. O arranque do motor tem um contato auxiliar, *StarterAux*, que faz com que *Starter* seja energizado, dizendo efetivamente que o motor está funcionando. Se um erro for cometido, devido à ocorrência de uma sobrecarga, ou devido a uma parada de emergência ser pressionada, ou se houver uma falha no abastecimento, o sinal do contato auxiliar será perdido. Este contato não pode ser verificado até 5 segundos após o *Starter* ter sido energizado, para dar tempo para o contato dar partida. *AcceptPB* representa o acionamento manual de um alarme caso algum problema aconteça.

Tabela 3.2: Símbolos ISA 5.2

Símbolo	**Descrição**
inputName HS	Representa uma entrada do Diagrama ISA 5.2.
outputName AC	Representa uma saída do Diagrama ISA 5.2.
AND	Representa uma operação booleana AND em um Diagrama ISA 5.2.
OR	Representa uma operação booleana OR em um Diagrama ISA 5.2.
Di pt	Temporizador com atraso na inicialização da saída (Delay Initiation of output): representa um elemento temporizado. Energiza a saída depois de um período de tempo (tempo = pt) quando existe lógica 1 na entrada.
Dt pt	Temporizador com atraso no desligamento da saída (Delay Termination of output): representa um elemento temporizado. Energiza a saída quando existe lógica 1 na entrada. Quando a lógica na entrada passar a ser 0 a saída será desenergizada após um período de tempo (tempo = pt).
Po pt	Temporizador por pulso (Pulse Output): representa um elemento temporizado. Energiza a saída durante um período de tempo (tempo = pt) quando existe lógica 1 na entrada. Mesmo que exista lógica 0 na entrada a saída continua ativada durante o tempo pré-definido (tempo = pt).

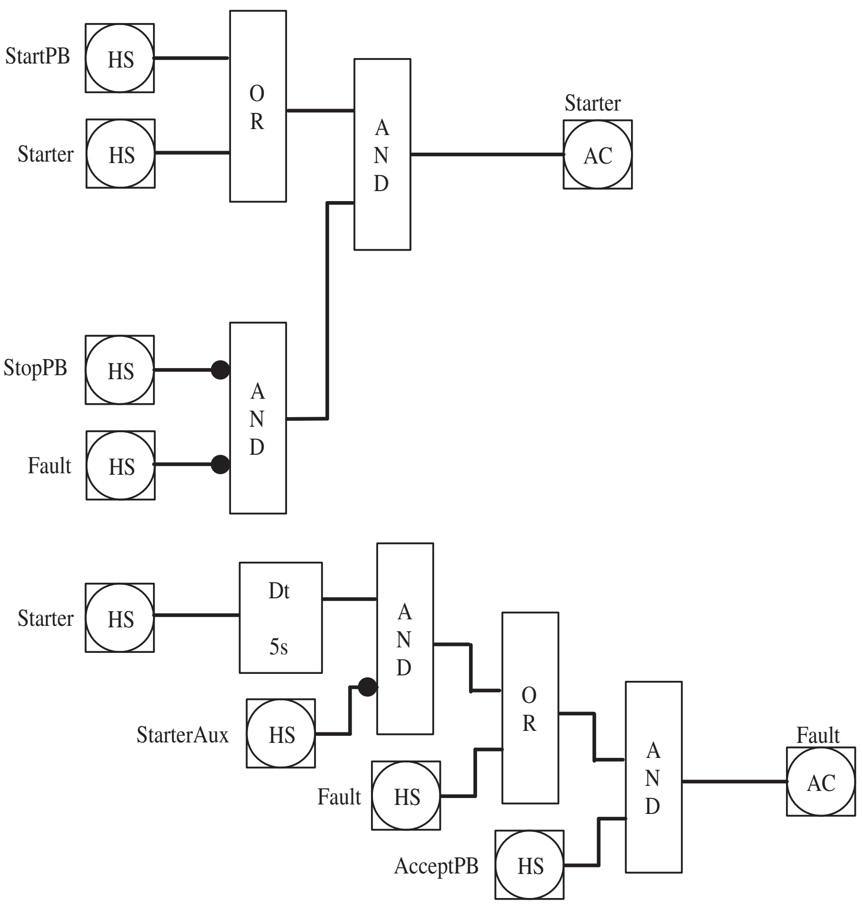


Figura 3.8: Diagrama ISA 5.2 para o arranque de um motor

Capítulo 4

O Método

Neste capítulo é introduzido o método para aumentar a confiança e a segurança de SIS, mais especificamente de programas para Controladores Lógicos Programáveis.

Na Figura 4.1 é ilustrado o método utilizado neste trabalho. Primeiro os arquivos representados pela especificação e implementação do sistema são transformados em arquivos XML. Estes arquivos são compostos por informações tais como: temporizadores, expressões lógicas que determinam saídas e nomes das saídas. A partir das informações contidas nestes arquivos XML são gerados autômatos temporizados referentes à especificação e à implementação, segundo a sintaxe e semântica da ferramenta Uppaal. Após este passo, testes de conformidade são aplicados sobre os modelos de autômatos gerados. Esta tarefa é realizada pela ferramenta de teste Uppaal-TRON, que fornece um veredicto sobre a conformidade entre especificação e implementação do sistema modelado. Por fim, a análise deste veredicto é feita de forma manual e, apenas pessoas que conhecem a ferramenta Uppaal-TRON são capazes de interpretar tais resultados.

Os arquivos que representam a especificação e a implementação do SIS devem ser gerados. O arquivo referente à especificação pode ser gerado a partir da ferramenta *SIS*, onde Diagramas de Lógica Binária ISA 5.2 podem ser manipulados. Já o arquivo referente a implementação tem que ser gerado manualmente e deve conter apenas os elementos da linguagem Ladder descritos na tabela 3.1. Estes arquivos bem como o tempo do ciclo de varredura, dado em microssegundos, representam as entradas para o método.

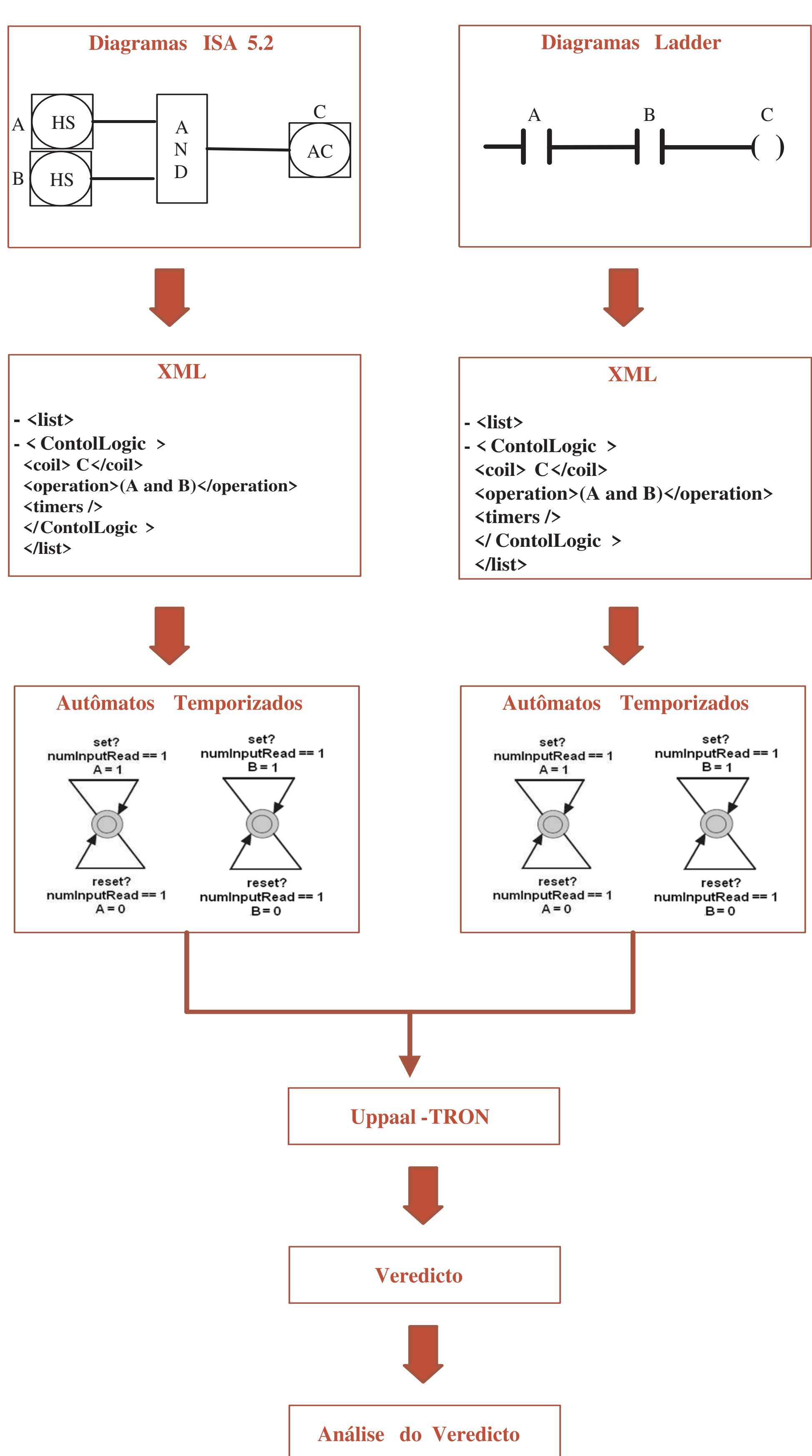


Figura 4.1: Método utilizado para validar o SIS

Para que o processo de teste seja realizado três arquivos são necessário, o modelo que representa a especificação, o modelo que representa a implementação e um arquivo de configuração que contém as seguintes informações: eventos que são caracterizados como entrada, eventos que são categorizados como saída, tempo de precisão necessário para execução de cada evento e total de unidades de tempo para a execução dos eventos.

A ferramenta Uppaal-TRON é uma ferramenta que realiza testes de conformidade de caixa preta em sistemas de tempo real. Estes testes avaliam o comportamento externo do componente de software, sem considerar o comportamento e a estrutura interna do mesmo, ou seja, testes de conformidade de caixa preta avaliam o comportamento das saídas a partir dos valores das entradas sem se importar com o funcionamento interno do sistema.

Na ferramenta TRON os casos de teste são gerados a partir do modelo que representa a especificação (modelo do ambiente) e a execução destes é feita pelo modelo que representa a implementação (*IUT - Implementation Under Test*). Após a execução dos casos de testes um dos três possíveis veredictos é dado:

- *PASSED*: significa que o tempo requerido para a execução dos teste expirou e nenhuma falha no comportamento foi detectada;
- *INCONCLUSIVE*: significa que o estado do ambiente não pode ser modificado com a saída esperada da IUT.
- *FAILED*: significa que a IUT expôs um comportamento errado o qual não pode ser mapeado para o modelo da IUT, isto é, a IUT relatou uma saída errada no tempo errado (cedo ou tarde demais) ou a IUT não respondeu quando foi solicitada para responder com alguma saída.

Na Figura 4.2 a interação entre as ferramentas Uppal-TRON e gerador de autômatos pode ser visualizada. Dois retângulos representam arquivos que devem ser gerados manualmente. Losangos representam as ferramentas. Retângulos simples representam arquivos gerados pelo método e círculos representam saídas produzidas por uma ferramenta externa.

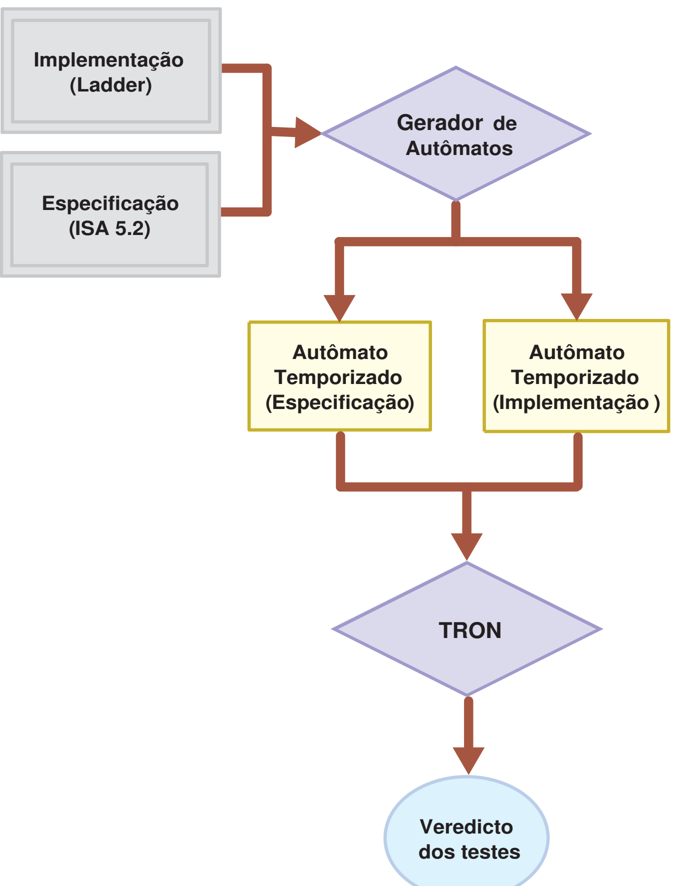


Figura 4.2: Interação entre ferramentas utilizadas no método proposto

4.1 Modelagem dos autômatos temporizados

Nesta seção são ilustrados os modelos de autômatos temporizados gerados bem como a geração automática dos mesmos a partir de diagramas de Lógica Binária ISA 5.2 e Laddder.

A rede de autômatos temporizados gerada para cada arquivo que corresponde a uma entrada para o método, arquivo ISA 5.2 e arquivo Ladder, é composta basicamente de 8 tipos de autômatos temporizados:

- Autômatos que modelam variáveis de entrada booleanas;
- Autômato que modela o processo de atualização das variáveis de entrada;
- Autômato que modela o processamento dos sinais de entrada;
- Autômatos que modelam o comportamento de elementos temporizados;

- Autômato que modela o funcionamento da lógica do programa;
- Autômato que modela o processo de avaliação dos estados das variáveis de saída;
- Autômato que modela o processamento dos sinais dos estados das saídas;
- Autômato que modela o ciclo de varredura do CLP;

As estruturas sintáticas utilizadas na modelagem dos autômatos temporizados gerados são descritas a seguir:

- Localidades iniciais são expressas por dois círculos;
- Localidades do tipo *committed* são expressa por círculos contendo a letra C;
- Expressões ilustradas na cor verde representam guardas;
- Expressões ilustradas na cor roxa representam atualização de variáveis;
- Canais de sincronização são ilustrados na cor azul.

4.1.1 Modelagem de variáveis de entrada

Na Figura 4.3 a modelagem utilizada para variáveis de entrada é ilustrada. Cada variável de entrada é modelada como um autômato de um único estado com duas transições, onde os valores booleanos *0* ou *1* podem ser definidos para as variáveis através da utilização dos canais de sincronização *reset* e *set*, respectivamente. A variável booleana *numInputRead*, utilizada como guarda, determina qual variável de entrada está sendo atualizada. Quando a última variável de entrada é atualizada o valor de *numInputRead* é inicializado para *1*, para que desta forma, no próximo ciclo de varredura, a atualização das variáveis possa ocorrer novamente de forma seqüencial. Um fato importante a ser mencionado é que variáveis de *feedback* e variáveis que representam uma saída de um degrau e em outro degrau representam uma variável de entrada, não são modeladas como autômatos temporizados. As primeiras são utilizadas para armazenar um valor em um ciclo e lida como uma entrada para a mesma rede no próximo ciclo, e as segundas servem para armazenar valores decorrentes de alguma lógica de controle de um programa.

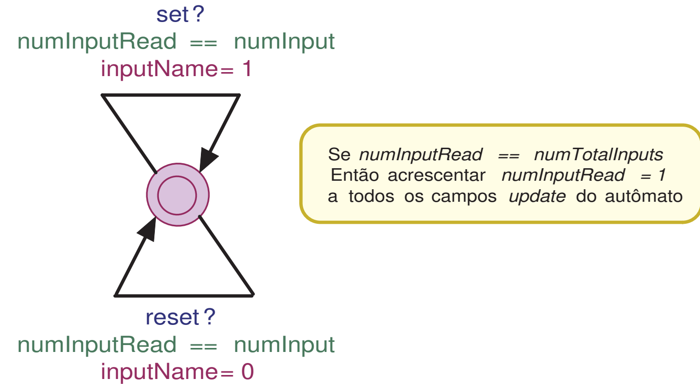


Figura 4.3: Autômato que representa a modelagem de uma variável de entrada

O algoritmo utilizado para modelar variáveis de entrada como autômatos temporizados é ilustrado a seguir:

Algoritmo 1 Geração dos autômatos que modelam variáveis de entrada

Declarar *int numInputRead = 1* em Declarations (do Projeto)

Declarar *broadcast chan set, reset* em Declarations (do Projeto)

for all (Variavel de Entrada v | v não é feedback e v não é entrada em outro degrau) **do**

  Criar um template com o nome *Input_v*

  Declarar *bool v* em Declarations (do Projeto)

  Criar um autômato com apenas um estado $\{s_1\}$, sendo este inicial

  Criar uma transição $s_1 \rightarrow s_1$ atribuindo: *v = 1* ao campo *Update*, *set?* ao campo *Sync* e *numInputRead = (número da entrada a ser lida)* ao campo *Guard*

  Criar uma transição $s_1 \rightarrow s_1$ atribuindo: *v = 0* ao campo *Update*, *reset?* ao campo *Sync* e *numInputRead = (número da entrada a ser lida)* ao campo *Guard*

  if (*numInputRead == número total de entradas*) **then**

    Acrescentar *numInputRead = 1* ao campo *Update* dos arcos das duas transições

  end if

end for

4.1.2 Modelagem do processo que realiza atualização de variáveis de entrada

Na Figura 4.4 a modelagem utilizada para realizar a atualização, de forma seqüencial, das variáveis de entrada pode ser visualizada. O autômato ilustrado nesta Figura envia mensagens *set!* ou *reset!*, para cada autômato que representa uma variável de entrada, informando o valor que deve ser atribuído a esta variável e, para o autômato que processa os sinais de entrada, fazendo com que o valor da variável *numInputRead* seja incrementado. Após todos os valores das variáveis de entrada serem atualizados, *id* == *numInputs*, este autômato envia uma mensagem *start!* ao autômato que representa o ciclo de execução do programa, informando que a execução da lógica do programa pode ser inicializada. O canal *update* é utilizado para sincronizar este autômato com o autômato que representa o ciclo de varredura, indicando que a execução do programa foi realizada e os valores das saídas do sistema foram liberadas. Após a liberação e a avaliação dos valores das saídas do sistema, este autômato envia uma mensagem *start!* ao autômato que representa a avaliação dos estados das saídas, informado que um novo ciclo de varredura foi inicializado.

Na Figura 4.4 o estado inicial do autômato é *committed* pelo fato de que a evolução dos relógios não deve ser considerada durante a atualização de variáveis de entrada e a próxima transição a ser disparada deve partir de uma localidade committed, que neste caso é indicada pela localidade inicial do autômato que representa o ciclo de varredura do CLP, ou pela localidade *commited* do autômato que avalia os estados das saídas.

O algoritmo utilizado para modelar o processo de atualização de variáveis de entrada como um autômato temporizado é ilustrado a seguir:

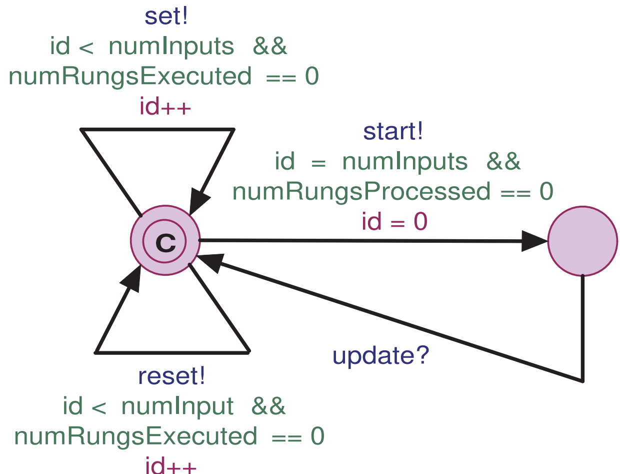


Figura 4.4: Autômato que representa o processo de atualização de variáveis de entrada

---

Algoritmo 2 Geração do autômato que modela o processo de atualização de variáveis de entrada

---

Declarar *int id = 0, numRungsProcessed = 0* e *bool startExecution* em Declarations (do Projeto)

Declarar *const int numInputs = numero de entradas* em Declarations (do Projeto)

Declarar *broadcast chan update, start* em Declarations (do Projeto)

Criar um template com o nome *UpdateInputs*

Criar um autômato com dois estados $\{s_1, s_2\}$, sendo s_1 inicial e *committed*

Criar uma transição $s_1 \rightarrow s_1$ atribuindo: *id++* ao campo *Update*, *set!* ao campo *Sync* e *(id < numInputs && numRungsExecuted == 0)* ao campo *Guard*

Criar uma transição $s_1 \rightarrow s_1$ atribuindo: *id++* ao campo *Update*, *reset!* ao campo *Sync* e *(id < numInputs && numRungsExecuted == 0)* ao campo *Guard*

Criar uma transição $s_1 \rightarrow s_2$ atribuindo: *id = 0* ao campo *Update*, *start!* ao campo *Sync* e *(id == numInputs && numRungsProcessed == 0)* ao campo *Guard*

Criar uma transição $s_2 \rightarrow s_1$ atribuindo *update?* ao campo *Sync*

4.1.3 Modelagem do processamento de sinais de entrada

Na Figura 4.5 o autômato reponsável por incrementar o valor de *numInputRead* toda vez que alguma variável de entrada for atualizada é ilustrado. Este incremento é realizado através do recebimento de mensagens *set!* ou *reset!* do autômato ilustrado na Figura 4.4.

O algoritmo utilizado para modelar o processamento dos sinais de entrada como um

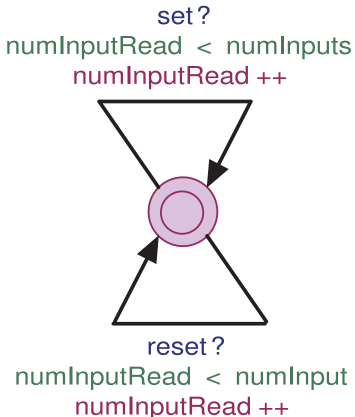


Figura 4.5: Autômato que representa a modelagem do processamento dos sinais de entrada

autômato temporizado é ilustrado a seguir:

Algoritmo 3 Geração do autômato que modela o processamento dos sinais de entrada

Criar um template com o nome *SignalsInputs*

Criar um autômato com apenas um estado $\{s_1\}$, sendo este inicial

Criar uma transição $s_1 \rightarrow s_1$ atribuindo: *numInputRead*++ ao campo *Update*, *set?* ao campo *Sync* e *numInputRead* < *numInputs* ao campo *Guard*

Criar uma transição $s_1 \rightarrow s_1$ atribuindo: *numInputRead*++ ao campo *Update*, *reset?* ao campo *Sync* e *numInputRead* < *numInputs* ao campo *Guard*

4.1.4 Modelagem de temporizadores

Nas Figuras 4.6, 4.7 e 4.8 os autômatos temporizados que representam, respectivamente, o funcionamento dos temporizadores TON ou DI, TOF ou DT e TP ou PO podem ser visualizados. Para que a modelagem dos temporizadores possa ser compreendida os seguintes itens devem ser considerados:

- Os sufixos *NumTimer* e *NameTimer*, utilizados nas variáveis dos modelos dos autômatos, representam, respectivamente, a posição do temporizador no programa (se o temporizador é 1º, é o 2º,...) e o nome do temporizador. Estes sufixos possibilitam a criação de variáveis para cada temporizador.

- A variável *controlNumTimer* e o canal de sincronização *syncNumTimer* realizam a sincronização do autômato que representa o funcionamento do temporizador com o autômato que representa o ciclo de execução do programa;

- A função *inputTimer()* corresponde ao conjunto de operações lógicas que representam a entrada (IN) do temporizador;

- A variável *out_NameTimer* corresponde ao valor lógico da saída (Q) do temporizador;

- A variável *numCyclesNumTimer = PT/(tempo do ciclo de varredura)* determina a quantidade de ciclos de varredura necessária para o temporizador executar;

- O canal *end* determina o término de cada ciclo de execução do temporizador.

Na Figura 4.6 o autômato utilizado para representar o comportamento de temporizadores do tipo TON ou DI é ilustrado. Este autômato possui o seguinte funcionamento: quando o autômato que representa o ciclo de execução do programa enviar uma mensagem de sincronização *syncNumTimer!* e o TON estiver na localidade inicial *L1*, temporizador desligado, e sua entrada estiver energizada, *inputTimer() == 1*, então, o arco que leva a localidade *L1* a localidade *L2* é disparado e as variáveis *numCyclesNumTimer* e *controlNumTimer* são inicializadas. A partir deste ponto seqüências de disparos, tendo como destino as localidades L2, L4 e L5, são executadas até que uma das igualdades *inputTimer() == 0* ou *numCyclesNumTimer == 0* seja estabelecida. Se a igualdade *inputTimer() == 0* for estabelecida, então o temporizador é desligado. Caso a igualdade *numCyclesNumTimer == 0* for obtida e a entrada do temporizador permanecer energizada (*inputTimer() == 1*), então a transição que leva a localidade *L2* a localidade *L3* é disparada e a saída do temporizador com valor lógico *1* é liberada. Caso o temporizador esteja ligado e a entrada do temporizador seja desativada a transição que leva a localidade *L3* a localidade *L1* é disparada e o temporizador é reinicializado.

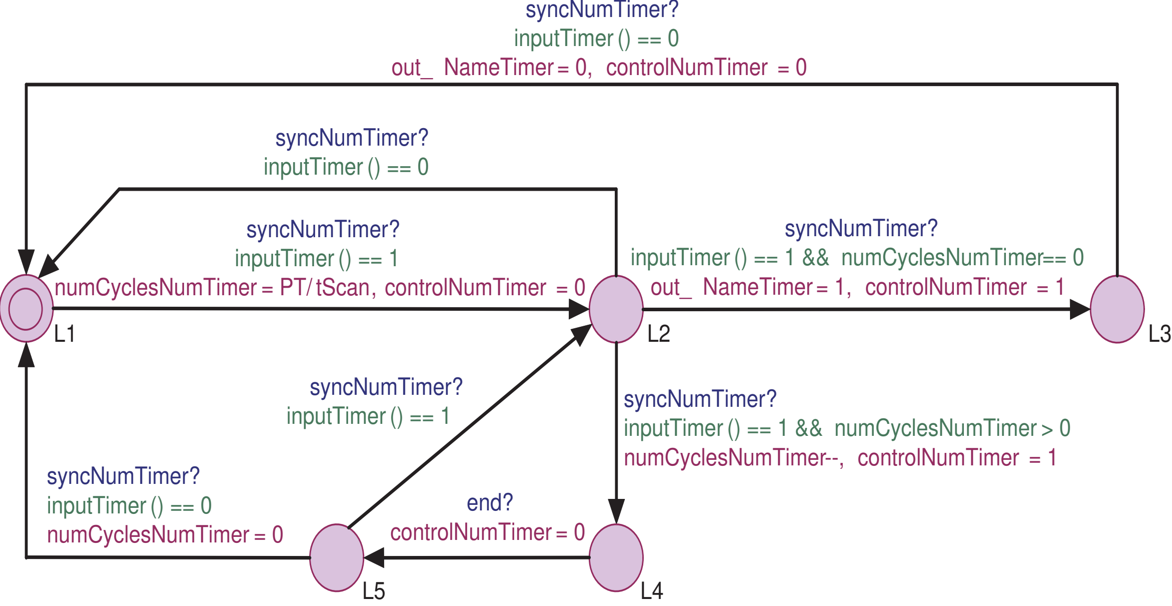


Figura 4.6: Modelagem de um temporizador TON ou DI como um autômato temporizado

Na Figura 4.7 o autômato utilizado para representar o comportamento de temporizadores do tipo TOF ou DT é ilustrado. Este autômato possui o seguinte funcionamento: quando o autômato que representa o ciclo de execução do programa enviar uma mensagem de sincronização *syncNumTimer!* e o TOF estiver na localidade inicial *L1*, temporizador desligado, e sua entrada estiver energizada, *inputTimer() == 1*, então, o arco que leva a localidade *L1* a localidade *L2* é disparado e a saída do temporizador com valor lógico *1* é liberada. Quando a entrada for desenergizada (*inputTimer() == 0*), então, o arco que leva a localidade *L2* a localidade *L3* é disparado e as variáveis *numCyclesNumTimer* e *controlNumTimer* são alocadas, respectivamente, com os valores *PT/tScan* e *0*. A partir deste ponto seqüências de disparos, tendo como destino as localidades L3, L4 e L5, são executadas até que uma das igualdades *inputTimer() == 1* ou *numCyclesNumTimer == 0* seja estabelecida. Se a igualdade *numCyclesNumTimer == 0* for estabelecida e a entrada do temporizador estiver desenergizada, então o arco que leva a localidade *L3* a localidade *L1* é disparado e o temporizador é desligado. Caso a igualdade *inputTimer() == 1* for obtida então o temporizador estará novamente apto a inicializar seu funcionamento.

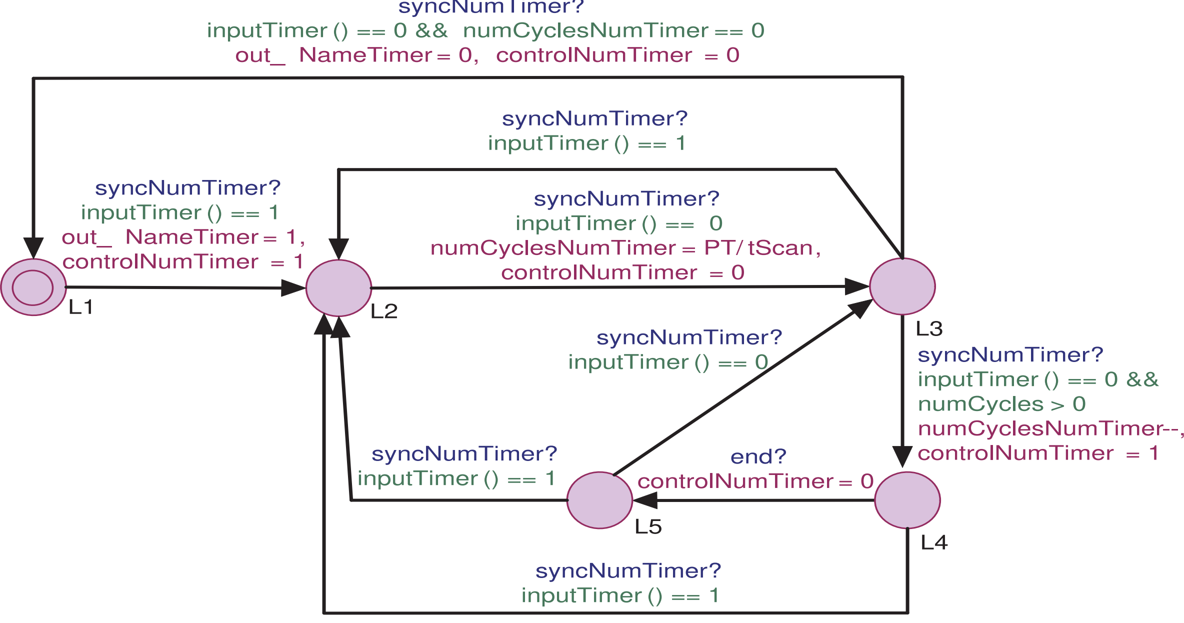


Figura 4.7: Modelagem de um temporizador TOF ou DT como um autômato temporizado

Na Figura 4.8 o autômato utilizado para representar o comportamento de temporizadores do tipo TP ou PO é ilustrado. Este autômato possui o seguinte funcionamento: quando o autômato que representa o ciclo de execução do programa enviar uma mensagem de sincronização *syncNumTimer!* e o TP estiver na localidade inicial *L1*, temporizador desligado, e sua entrada estiver energizada, *inputTimer() == 1* e ele ainda não tiver sido executado *controlNumTimer == 0*, então, o arco que leva a localidade *L1* a localidade *L2* é disparado, as variáveis *numCyclesNumTimer* e *controlNumTimer* são inicializadas e a saída do temporizador é liberada com valor lógico *1*. A partir deste ponto seqüências de disparos, tendo como destino as localidades L2, L3 e L4, são executadas até que a igualdade *numCyclesNumTimer == 0* seja obtida. Quando esta igualdade for obtida então, o temporizador é reinicializado.

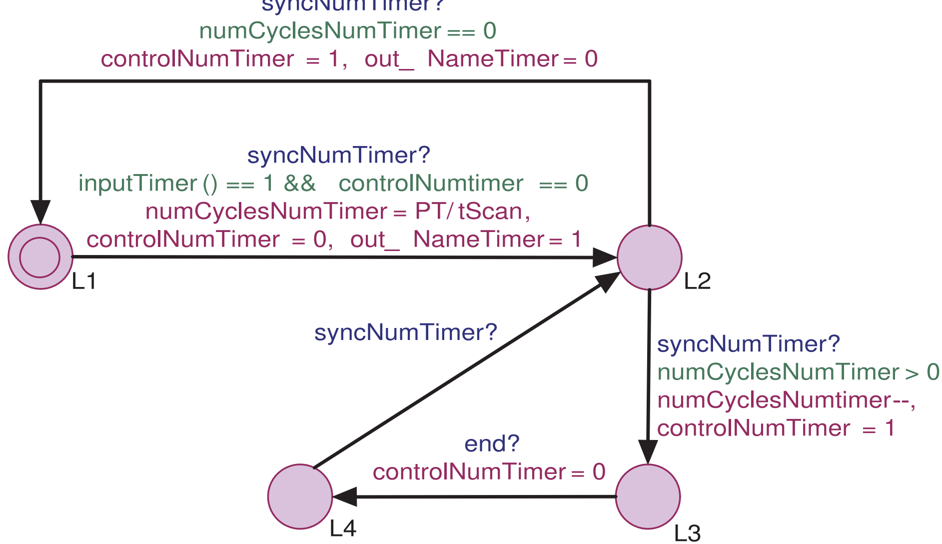


Figura 4.8: Modelagem de um temporizador TP ou PO como um autômato temporizado

Os algoritmos utilizados para modelar o comportamento dos temporizadores TON ou DI, TOF ou DT e TP ou PO, como autômatos temporizados, são ilustrados a seguir:

Algoritmo 4 Geração do autômato que modela temporizadores do tipo TON ou DI

Declarar *bool controlNumTimer, out_NameTimer* e *broadcast chan syncNumTimer*, em Declarations (do Projeto)

Criar um template com o nome *TON_NameTimer*

Declarar *int numCyclesNumTimer* em Declarations (do Projeto)

Declarar *bool inputTimer(){return IN}* em Declarations deste template

Criar um autômato com cinco estados $\{s_1, s_2, s_3, s_4, s_5\}$, sendo s_1 inicial

Criar uma transição $s_1 \rightarrow s_2$ atribuindo: *(numCyclesNumTimer = PT/tScan, controlNumTimer = 0)* ao campo *Update*, *syncNumTimer?* ao campo *Sync* e *inputNomeTimer() == 1* ao campo *Guard*

Criar uma transição $s_2 \rightarrow s_1$ atribuindo: *syncNumTimer?* ao campo *Sync* e *inputNomeTimer() == 0* ao campo *Guard*

Criar uma transição $s_2 \rightarrow s_3$ atribuindo: *(out_NameTimer = 1, controlNumTimer = 1)* ao campo *Update*, *syncNumTimer?* ao campo *Sync* e *(inputTimer() == 1 && numCyclesNumTimer == 0)* ao campo *Guard*

Criar uma transição $s_2 \rightarrow s_4$ atribuindo: *(numCyclesNumTimer−−, controlNumTimer = 1)* ao campo *Update*, *syncNumTimer?* ao campo *Sync* e *(inputTimer() == 1 && numCyclesNumTimer > 0)* ao campo *Guard*

Criar uma transição $s_3 \rightarrow s_1$ atribuindo: *(out_NameTimer = 0, controlNumTimer = 0)* ao campo *Update*, *syncNumTimer?* ao campo *Sync* e *inputTimer() == 0* ao campo *Guard*

Criar uma transição $s_4 \rightarrow s_5$ atribuindo: *controlNumTimer = 0* ao campo *Update*, *end?* ao campo *Sync*

Criar uma transição $s_5 \rightarrow s_1$ atribuindo: *numCyclesNumTimer = 0* ao campo *Update*, *syncNumTimer?* ao campo *Sync* e *inputTimer() == 0* ao campo *Guard*

Criar uma transição $s_5 \rightarrow s_2$ atribuindo: *syncNumTimer?* ao campo *Sync* e *inputTimer() == 1* ao campo *Guard*

Algoritmo 5 Geração do autômato que modela temporizadores do tipo TOF ou DT

Declarar *bool controlNumTimer, out_NameTimer* e *broadcast chan syncNumTimer*, em Declarations (do Projeto)

Criar um template com o nome *TOF_NameTimer*

Declarar *int numCyclesNumTimer* em Declarations (do Projeto)

Declarar *bool inputTimer(){return IN}* em Declarations deste template

Criar um autômato com cinco estados $\{s_1, s_2, s_3, s_4, s_5\}$, sendo s_1 inicial

Criar uma transição $s_1 \rightarrow s_2$ atribuindo: *(out_NameTimer = 1, controlNumTimer = 1)* ao campo *Update*, *syncNumTimer?* ao campo *Sync* e *inputTimer() == 1* ao campo *Guard*

Criar uma transição $s_2 \rightarrow s_3$ atribuindo: *(numCyclesNumTimer = PT/tScan, controlNumTimer = 0)* ao campo *Update*, *syncNumTimer?* ao campo *Sync* e *inputTimer() == 0* ao campo *Guard*

Criar uma transição $s_3 \rightarrow s_1$ atribuindo: *(out_NameTimer = 0, controlNumTimer = 0)* ao campo *Update*, *syncNumTimer?* ao campo *Sync* e *(inputTimer() == 0 && numCyclesNumTimer == 0)* ao campo *Guard*

Criar uma transição $s_3 \rightarrow s_2$ atribuindo: *syncNumTimer?* ao campo *Sync* e *inputTimer() == 1* ao campo *Guard*

Criar uma transição $s_3 \rightarrow s_4$ atribuindo: *(numCyclesNumTimer−−, controlNumTimer = 1)* ao campo *Update*, *syncNumTimer?* ao campo *Sync* e *(inputTimer() == 0 && numCyclesNumTimer > 0)* ao campo *Guard*

Criar uma transição $s_4 \rightarrow s_2$ atribuindo: *syncNumTimer?* ao campo *Sync* e *inputTimer() == 1* ao campo *Guard*

Criar uma transição $s_4 \rightarrow s_5$ atribuindo: *controlNumTimer = 0* ao campo *Update* e *end?* ao campo *Sync*

Criar uma transição $s_5 \rightarrow s_2$ atribuindo: *syncNumTimer?* ao campo *Sync* e *inputTimer() == 1* ao campo *Guard*

Criar uma transição $s_5 \rightarrow s_3$ atribuindo: *syncNumTimer?* ao campo *Sync* e *inputTimer() == 0* ao campo *Guard*

Algoritmo 6 Geração do autômato que modela temporizadores do tipo TP ou PO

Declarar *bool controlNumTimer, out_NameTimer* e *broadcast chan syncNumTimer*, em Declarations (do Projeto)

Criar um template com o nome *TP_NameTimer*

Declarar *int numCyclesNumTimer* em Declarations (do Projeto)

Declarar *bool inputTimer(){return IN}* em Declarations deste template

Criar um autômato com quatro estados $\{s_1, s_2, s_3, s_4\}$, sendo s_1 inicial

Criar uma transição $s_1 \rightarrow s_2$ atribuindo: *(numCyclesNumTimer = PT/tScan, controlNumTimer = 0, out_NameTimer = 1)* ao campo *Update*, *syncNumTimer?* ao campo *Sync* e *inputTimer() == 1 && controlNumTimer == 0* ao campo *Guard*

Criar uma transição $s_2 \rightarrow s_1$ atribuindo: *(controlNumTimer = 1, out_NameTimer = 0)* ao campo *Update*, *syncNumTimer?* ao campo *Sync* e *numCyclesNumTimer == 0* ao campo *Guard*

Criar uma transição $s_2 \rightarrow s_3$ atribuindo: *(numCyclesNumTimer−−, controlNumTimer = 1)* ao campo *Update*, *syncNumTimer?* ao campo *Sync* e *numCyclesNumTimer > 0* ao campo *Guard*

Criar uma transição $s_3 \rightarrow s_4$ atribuindo: *controlNumTimer = 0* ao campo *Update*, *end?* ao campo *Sync*

Criar uma transição $s_4 \rightarrow s_2$ atribuindo: *syncNumTimer?* ao campo *Sync*

4.1.5 Modelagem do ciclo de execução do programa

Na Figura 4.9 o esquema utilizado para a construção do autômato que representa a execução da lógica do programa é ilustrado. Para que a modelagem deste autômato possa ser compreendida os seguintes itens devem ser considerados:

- O sufixo *NameCoil* representa o nome da bobina, elemento de saída, que compõe a lógica de controle de cada degrau. Este sufixo possibilita a criação de funções para cada lógica de controle;

- A função *value_NameCoil* processa o valor da lógica de controle referente ao degrau que *NameCoil* representa. Esta função é utilizada apenas para degraus que não contém elementos temporizados;

- A função *checkExecutionNameTimer()* tem o objetivo de determinar se o elemento temporizado, simbolizado pelo sufixo *NameTimer* da função, está ou não apto a ser executado. Ela é específica para cada tipo de temporizador;

- A função *evaluateOuputNameTimer()* é utilizada para desincronizar o modelo que representa um elemento temporizado, simbolizado pelo sufixo *NameTimer* da função, e o modelo que representa o comportamento do ciclo de execução da lógica do programa, para que o próximo degrau seja executado ou para que o programa termine sua execução. Ela é específica para cada tipo de temporizador;

- A função *outNameTimer()* determina o valor referente a saída do temporizador indicado pelo sufixo *NameTimer* da função;

A idéia utilizada na modelagem consiste em executar seqüencialmente degraus, no caso de diagramas Ladder, e blocos que determinam uma saída, no caso de diagramas ISA 5.2, através do uso de funções cuja execução é realizada a partir do disparo de transições entre localidades distintas que possuam a mensagem *execute?*. Esta mensagem indica que a lógica de controle foi executada durante o ciclo de varredura. A expressão *startExecution == 1* indica que a execução dos degraus ou dos blocos que determinam uma saída pode ser inicializada. A função *setControlVariables()* restaura o valor de todas as variáveis *controlNumTimer* para *0*, caso existam temporizadores na lógica do programa, atualiza o valor de *numRungsProcessed* para *numOutputs*, informando que todos os degraus ou blocos que determinam uma saída foram processados e atualiza o valor de *startExecution* para *0*, informando que a execução do programa foi finalizada. Quando todos os degraus ou blocos que determinam uma saída forem executados uma mensagem *end!* é enviada aos temporizadores do programa, caso existam, informando que suas respectivas execuções não podem mais serem realizadas neste ciclo de varredura. Para o método, três tipos de degraus ou blocos que determinam uma saída são considerados:

- Degraus ou blocos que determinam uma saída cuja lógica de controle não contém elementos temporizados: são modelados por uma transição entre duas localidades consecutivas com rótulos *execute?* e *value_NameCoil()*.

- Degraus ou blocos que determinam uma saída cuja lógica de controle contém apenas um elemento temporizado: são modelados por duas transições e por duas localidades. A primeira transição ocorre entre a mesma localidade e possui rótulos *syncNumtimer!* e checkExecutionNameTimer(). A segunda transição ocorre entre duas localidades consecutivas com rótulos *execute?*, *evaluateOuputNameTimer()* e *outNameTimer()*.

- Degraus ou blocos que determinam uma saída cuja lógica de controle contém mais de um elemento temporizado: sua modelagem é composta por *n + 1* localidades, onde *n* indica o número de temporizadores existentes no degrau ou nos blocos que determinam uma saída. Para este tipo de degrau os n temporizadores são modelados segundo o esquema ilustrado anteriormente e suas execuções são realizadas seqüencialmente. Na última transição, a saída é finalmente computada, através do recebimento da mensagem *execute!*.

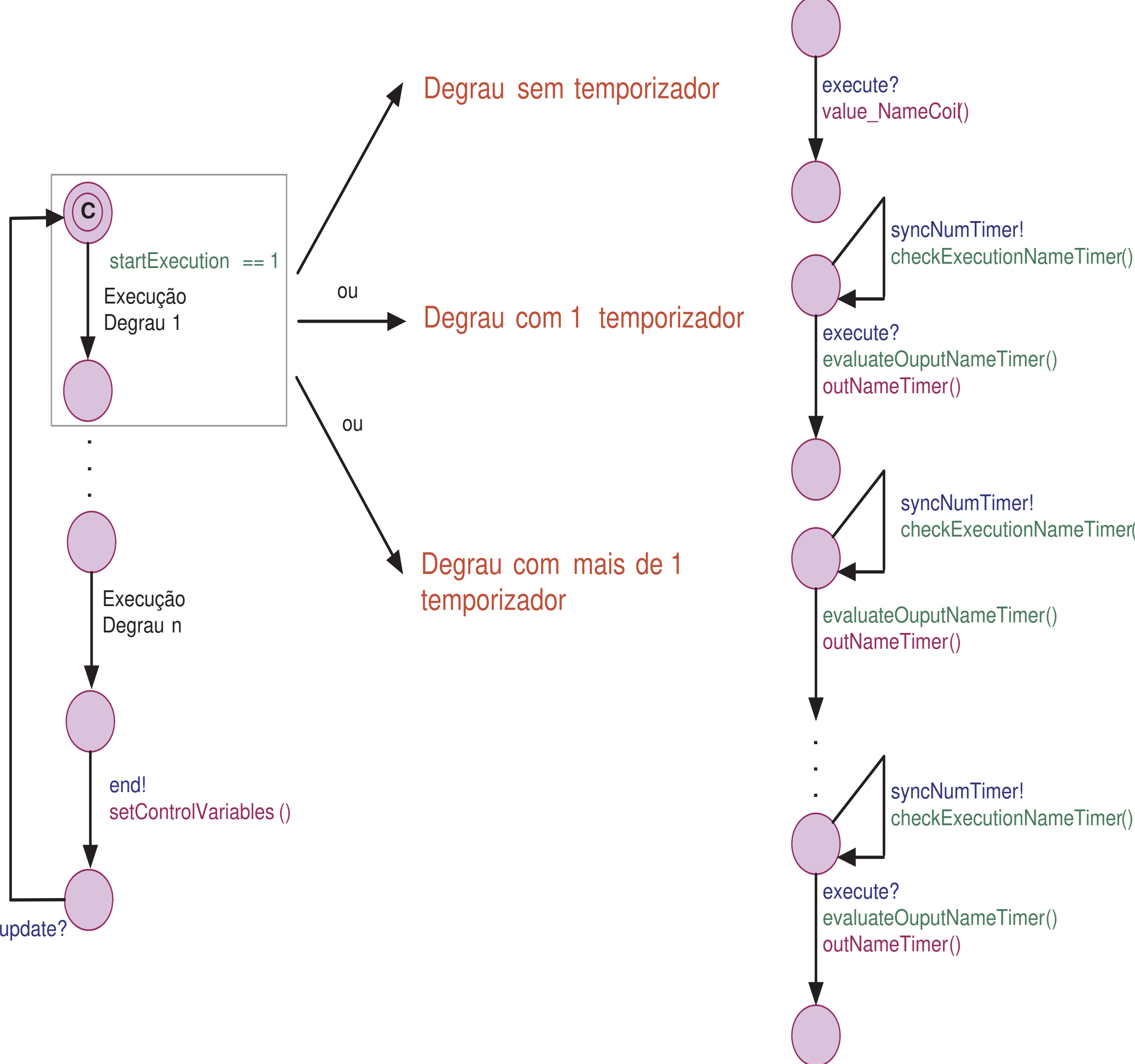


Figura 4.9: Esquema utilizado para construir o autômato temporizado que representa a execução da lógica do programa

Após definido o esquema para a construção do autômato que modela a execução da lógica do sistema, nosso próximo passo é ilustrar como cada função envolvida na representação deste autômato é formada.

A função *checkExecutionNameTimer()* identifica se um determinado temporizador está ou não apto a ser executado. Como ela é específica para cada tipo de temporizador, então sua modelagem dá-se pelo seguinte esquema:

- Função *checkExecutionNameTimer()* para temporizador do tipo TON ou DI:

  bool *checkExecutionNameTimer()*{

  return (IN == 1 && controlNumtimer == 0 && out_NameTimer == 0) ||

  (IN == 0 && controlNumtimer == 0 && out_NameTimer == 1) ||

  (IN == 0 && controlNumtimer == 0 && numCyclesNumTimer > 0) }

  Onde, a primeira expressão lógica indica que a entrada do temporizador está ativada e ele pode ser executado. A segunda expressão lógica indica que o temporizador já executou e sua entrada passou a ser falsa, então ele deve ser reinicializado. A terceira indica que o temporizador ainda necessitava executar mais ciclos para que sua saída fosse liberada, mas sua entrada foi desativada e ele terá que ser reinicializado.

- Função *checkExecutionNameTimer()* para temporizador do tipo TOF ou DT

  bool *checkExecutionNameTimer()*{

  return (IN == 1 && controlNumTimer == 0 && out_ NameTimer == 0) ||

  (IN == 0 && out_NameTimer == 1 && controlNumTimer == 0) }

  Onde, a primeira expressão lógica indica que a entrada do temporizador está ativada então, sua saída com valor lógico *1* pode ser liberada. A segunda indica que a entrada está desativada e que o temporizador pode realizar sua operação de contagem de tempo.

- Função *checkExecutionNameTimer()* para temporizador do tipo TP ou PO

  bool *checkExecutionNameTimer()*{

  return (IN == 1 && controlNumTimer == 0) || (numCyclesNumTimer == 0 &&

out_NameTimer == 1 && controlNumTimer == 0) }

Estas expressões indicam que uma vez que a entrada do temporizador estiver ativada o seu tempo de contagem será executado até mesmo se sua entrada tornar-se falsa.

A função *evaluateOuputNameTimer()* tem o objetivo de desincronizar o modelo que representa um elemento temporizado do modelo que representa a execução da lógica do programa. Como ela é específica para cada tipo de temporizador, então sua modelagem dá-se pelo seguinte esquema:

- Função *evaluateOuputNameTimer()* para temporizador do tipo TON ou DI

  bool *evaluateOuputNameTimer()*{

  return (IN == 1 && controlNumTimer == 1) || (IN == 1 && out_NameTimer == 1) ||

  (IN == 0 && controlNumTimer == 0 && out_NameTimer == 0

  && numCyclesNumTimer == 0)}

  Onde, a primeira e a segunda expressão lógica indicam que o temporizador não pode mais executar apesar de sua entrada estar energizada, pois ele já realizou sua execução durante o tempo pré-estabelecido (*Present Time*). A terceira expressão lógica indica que a entrada do temporizador está desativada, então ele não está apto a executar.

- Função *evaluateOuputNameTimer()* para temporizador do tipo TOF ou DT

  bool *evaluateOuputNameTimer()*{

  return (IN == controlNumTimer == out_NameTimer == 0) || (IN == out_NameTimer == 1) || (IN == out_NameTimer == 0 && controlNumTimer == 1)}

  Onde, a primeira expressão lógica indica que a entrada do temporizador está desativada e ele não liberou uma saída com valor lógico igual a *1* no ciclo anterior, portanto, ele não está apto a ser executado. A segunda expressão lógica indica que a entrada do temporizador está energizada mais sua saída com valor lógico *1* já foi liberada. A terceira indica que a entrada foi desenergizada, mas o temporizador já realizou sua execução durante o tempo pré-estabelecido (*Present Time*).

- Função *evaluateOuputNameTimer()* para temporizador do tipo TP ou PO

bool *evaluateOuputNameTimer()*{

return (IN == 0 && controlNumTimer == 0 && out_NameTimer == 0) ||

(controlNumTimer == 1)}

Estas expressões indicam que a saída do temporizador está desativada ou o temporizador já realizou sua execução durante o tempo pré-estabelecido (*Present Time*).

Os algoritmos utilizados para modelar a execução da lógica do programa como um autômato temporizado são ilustrados a seguir:

Algoritmo 7 Procedimento inicializaAutomato()

Declarar *broadcast chan end*, *int numRungsProcessed = 0, numOutputs = número de varáveis de saída* e *bool outputs[numTotalOutpus]* em Declarations (do Projeto)

for all (Bobina b) **do**

Declarar bool b, output_b em Declarations (do Projeto)

end for

Criar um template com o nome *CycleProgram*

Declarar void setControlVariables() {startExecution = 0; numRungsProcessed = numOutputs; ($\forall$ *controlNumTimer* $\in$ *programa Ladder fazer controlNumTimer = 0*)}

Criar um autômato com *(n + t + 2)* estados, onde *n = número total de degraus ou blocos que determinam saídas do programa sem temporizadores e t indica o número total de temporizadores do programa*, sendo s_1 inicial e committed

int ind = 0 *// variável utilizada para indicar as localidades*

Algoritmo 8 Procedimento criaTransicaoSemTemporizador(Rung ou blocos que determinam saídas r, int indice)

if r.coil() não é variável de entrada **then**

Declarar void value_NameCoil(){ NameCoil = r.controlLogic(); outputs[indice -1] = NameCoil}

else

Declarar void value_NameCoil(){ output_NameCoil = r.controlLogic(); outputs[indice -1] = output_NameCoil}

end if

Criar uma transição $s_{ind} \rightarrow s_{ind+1}$ atribuindo: *value_NameCoil()* ao campo *Update* e *execute?* ao campo *Sync*

if indice == 1 **then**

Acrescentar *startExecution == 1* ao campo *Guard* da transição $s_{ind} \rightarrow s_{ind+1}$

end if

Algoritmo 9 Procedimento criaFuncaoOutNameTimer(Rung ou blocos que determinam saídas r, Temporizador t, int indice)

if t é o último temporizador de r **then**

inTimer = t.IN()

Declarar *void outNameTimer() {NameCoil = out_NameTimer (and / or) (logicaAposTemporizador); outputs[indice -1] = NameCoil}*

else

inTimer = saidaNameTimerAnterior() *// A saída deste temporizador é composta pela lógica que contém o temporizador anterior a ele*

Declarar *void outNameTimer() {saidaNameTimer = out_NameTimer (and / or) (logicaAntesDoProximoTemporizador)}*

end if

Algoritmo 10 Procedimento criaFuncaoCheckExecutionNameTimer(Rung ou blocos que determinam saídas r, Temporizador t)

Considerar NameTimer = t.name() e NumTimer = t.posicao()
if t não é o primeiro temporizador do r **then**
  inTimer = saidaNameTimerAnterior
end if
if t.tipo = 'TON' || t.tipo() == 'DI' **then**
  Declarar, em Declarations deste template, *bool checkExecutionNomeTemporizador(){ (inTimer == 1 && controlNumTimer == 0 && out_NameTimer == 0)* ||
  (inTimer == 0 && controlNumTimer == 0 && out_NameTimer == 1) ||
  (inTimer == 0 && controlNumTimer == 0 && numCyclesNumTimer > 0)}
else
  if t.tipo = 'TOF' || t.tipo() == 'DT' **then**
    Declarar, em Declarations deste template, *bool checkExecutionNomeTemporizador(){ (inTimer == 0 && controlNumTimer == 0 && out_NameTimer == 1)*
    ||
    (inTimer == 1 && controlNumTimer == 0 && out_NameTimer == 0)}
  end if
else
  Declarar, em Declarations deste template, *bool checkExecutionNomeTemporizador(){ (inTimer == 1 && controlNumTimer == 0)* || *(numCyclesNumTimer == 0 && out_NameTimer == 1 && controlNumTimer ==0)}*
end if

Algoritmo 11 Procedimento criaFuncaoEvaluateOuputNameTimer(Rung ou blocos que determinam saídas r, Temporizador t)

Considerar NameTimer = t.name() e NumTimer = t.posicao()
if t não é o primeiro temporizador de r **then**
  inTimer = saidaNameTimerAnterior
end if
if t.tipo = 'TON' || t.tipo() == 'DI' **then**
  Declarar, em Declarations deste template, *bool evaluateOuputNameTimer(){ (inTimer == 0 && controlNumTimer == 0 && out_NameTimer == 0 && numCyclesNumTimer == 0)* || *(inTimer == 1 && controlNumTimer == 1)* || *(inTimer == 1 && out_NameTimer == 1)}*
else
  if t.tipo = 'TOF' || t.tipo() == 'DT' **then**
    Declarar, em Declarations deste template, *bool evaluateOuputNameTimer(){ (inTimer == 0 && controlNumTimer == 0 && out_NameTimer == 0)* || *(inTimer == 1 && out_NameTimer == 1)* || *(inTimer == 0 && controlNumTimer == 1 && out_NameTimer == 0)}*
  end if
else
  Declarar, em Declarations deste template, *bool evaluateOuputNameTimer(){ (inTimer == 0 && controlNumTimer == 0 && out_NameTimer == 0)* || *(controlNumTimer == 1)}*
end if

Algoritmo 12 Geração do autômato que modela o ciclo de execução da lógica do programa

inicializaAutomato() *// cria todas as condições para que este autômato possa ser criado*

for all (rung ou blocos que deteminam saídas r) **do**

ind++

if r não tem temporizador **then**

criaTransicaoSemTemporizador(r,ind)

end if

if r tem temporizador **then**

for all (temporizador t $\in$ r) **do**

bool inTimer, saidaNameTimer, startExecution *// onde NameTimer = t.name()*

criaFuncaoOutNameTimer(r,t,ind)

criaFuncaoCheckExecutionNameTimer(r, t)

criaFuncaoEvaluateOuputNameTimer(r, t)

Criar uma transição $s_{ind} \rightarrow s_{ind}$ atribuindo: *syncNumTimer!* ao campo *Sync* e *checkExecutionNameTimer()* ao campo *Guard*

if ind == 1 **then**

Acrescentar *startExecution == 1* ao campo *Guard* da transição $s_{ind} \rightarrow s_{ind}$

end if

Criar uma transição $s_{ind} \rightarrow s_{ind+1}$ atribuindo: *outNameTimer()* ao campo *Update* e *evaluateOuputNameTimer()* ao campo *Guard*

if t é o último temporizador de r **then**

Acrescentar *execute?* ao campo *Sync* da transição $s_{ind} \rightarrow s_{ind+1}$

end if

end for

end if

end for

Criar uma transição $s_{ind+1} \rightarrow s_{ind+2}$ atribuindo: *setControlVariables()* ao campo *Update* e *end?* ao campo *Sync* e *numOutputRead == ind-1* ao campo *Guard*

Criar uma transição $s_{ind+2} \rightarrow s_1$ atribuindo: *update?* ao campo *Sync*

4.1.6 Modelagem do processo que avalia os estados das saídas

Na Figura 4.10 é apresentado o autômato que avalia os estados das saídas do programa. Este autômato possui o seguinte funcionamento: quando a execução do programa for finalizada e as saídas forem liberadas o autômato que representa o ciclo de varredura envia uma mensagem *startEvaluation!* para este autômato, informado que a avaliação dos estados das saídas pode ser inicializada. Os valores das saídas do programa são avaliadas como energizada, caso a mensagem *high!* seja enviada, e desenergizada, caso a mensagem *low!* seja enviada. Quando os estados de todas as saídas forem avaliados, o autômato que representa o ciclo de varredura envia uma mensagem *endEvaluation!* para este autômato, informado que a avaliação dos estados das saídas deve ser finalizada.

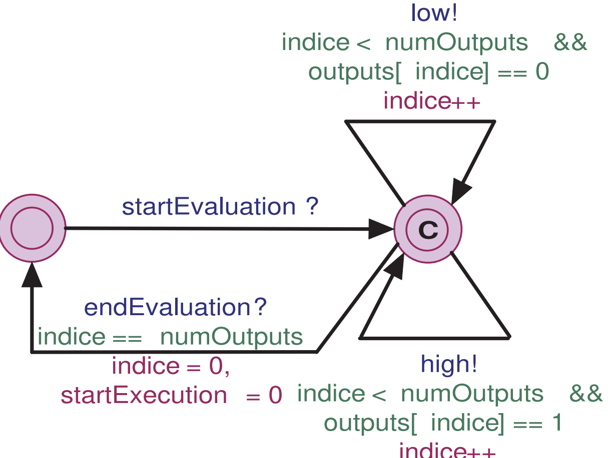


Figura 4.10: Modelagem do processo que avalia os estados das saídas

O algoritmo utilizado para modelar o processo que avalia os estados das saídas como autômato temporizado é ilustrado a seguir:

Algoritmo 13 Geração do autômato que modela o processo de avaliação dos estados das saídas

Declarar *int indice = 0, numOutputs = número total de saídas* em Declarations (do Projeto)

Declarar *chan high, low, startEvaluation, endEvaluation* em Declarations (do Projeto)

Criar um template com o nome *StateOutputs*

Criar um autômato com duas localidades $\{s_1, s_2\}$, sendo $\{s_1\}$ inicial e $\{s_2\}$ *commited*

Criar uma transição $s_1 \rightarrow s_2$ atribuindo: *startEvaluation?* ao campo *Sync*

Criar uma transição $s_2 \rightarrow s_1$ atribuindo: *(indice = 0, startExecution = 0)* ao campo *Update*, *endEvaluation?* ao campo *Sync* e *indice == numOutputs* ao campo *Guard*

Criar uma transição $s_2 \rightarrow s_2$ atribuindo: *indice++* ao campo *Update*, *high!* ao campo *Sync* e *(indice < numOutputs && outputs[indice] == 1)* ao campo *Guard*

Criar uma transição $s_2 \rightarrow s_2$ atribuindo: *indice++* ao campo *Update*, *low!* ao campo *Sync* e *(indice < numOutputs && outputs[indice] == 0)* ao campo *Guard*

4.1.7 Modelagem do processamento dos estados das saídas

Na Figura 4.11 o autômato que processa os estados das saídas é ilustrado. Este processamento é realizado através do recebimento de mensagens *high!* ou *low!* do autômato ilustrado na Figura 4.10.

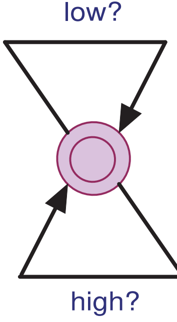


Figura 4.11: Modelagem do processamento dos estados das saídas

O algoritmo utilizado para modelar o processamento dos estados das saídas como autômato temporizado é ilustrado a seguir:

Algoritmo 14 Geração do autômato que modelam o processamento dos estados das saídas

Criar um template com o nome *StateSignalOutputs*

Criar um autômato com apenas uma localidade $\{s_1\}$, sendo esta inicial

Criar uma transição $s_1 \rightarrow s_1$ atribuindo: *high?* ao campo *Sync*

Criar uma transição $s_1 \rightarrow s_1$ atribuindo: *low?* ao campo *Sync*

4.1.8 Modelagem do ciclo de varredura de um CLP

Na Figura 4.12 é ilustrado o ciclo de varredura de um CLP. Sua modelagem obedece a execução das três etapas que compõem o ciclo de varredura: leitura de variáveis, execução do programa e atualização das saídas. O funcionamento deste autômato ocorre da seguinte forma: após as atualizações das variáveis de entrada serem feitas, o autômato que representa o processo de atualização das variáveis envia uma mensagem *start!* ao autômato do ciclo de varredura informando que a execução do programa pode ser inicializada. Então, a transição que leva a localidade *L1* a localidade *L2* é disparada e, a variável *startExecution* é inicializada com valor lógico *1*. Esta variável tem o objetivo de informar que a execução do programa pode ser iniciada. A segunda etapa do ciclo de varredura é executada através do envio da mensagem *execute!* por este autômato ao autômato que representa o ciclo de execução do programa. Ao passo que cada degrau é executado o valor da variável *numRungsExecuted* é incrementado. Esta etapa de processamento não deve ultrapassar o tempo de varredura (time <= tScan). Quando todos os degraus forem executados e o tempo de varredura for atingindo então os valores das saídas são liberadas e o valor de *timer* é restaurado. Após este passo, a avaliação dos estados das saídas é realizada através do envio da mensagem *startEvaluation!* para o autômato que representa a avaliação dos estados das saídas. Quando todos os estados das saídas forem avaliados uma mensagem *endEvaluation!* é enviada ao autômato anterior, informando que a avaliação dos estados das saídas deve ser finalizada. A função *update-Outputs()* libera as saídas do sistema que ainda não foram liberadas durante a execução do programa. A função *checkEndScanCycle()* avalia se todos os temporizadores não estão mais sendo executados ($\forall$ *controlNumTimer* $\in$ *lógica do programa c* $\Rightarrow$ *controlNumTimer* == *0*) e, se a execução do programa foi finalizada (*startExecution* == *0*).

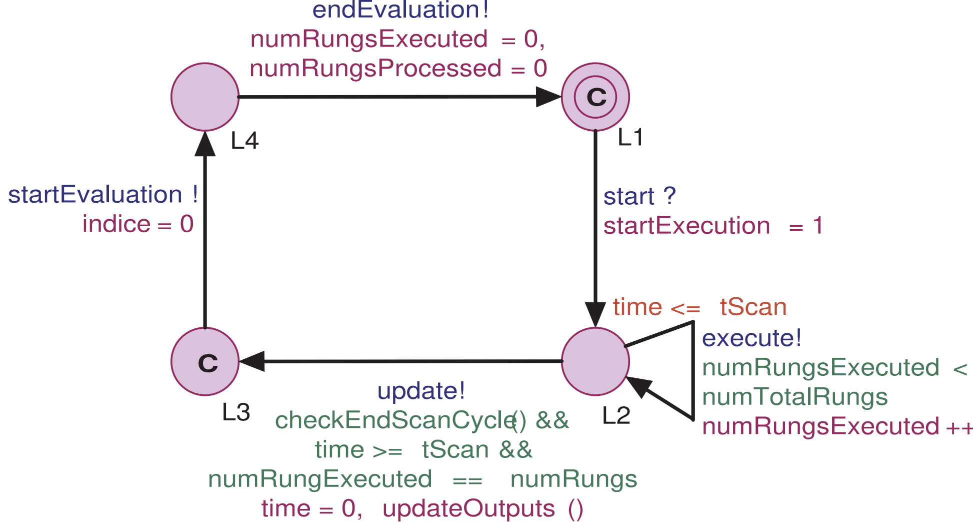


Figura 4.12: Modelagem do ciclo de varredura de um CLP como um autômato temporizado

O algoritmo utilizado para modelar o ciclo de varredura de um CLP como um autômato temporizado é ilustrado a seguir:

Algoritmo 15 Geração do autômato que modela o ciclo de varredura de um CLP

Declarar *int numRungsExecuted = 0, tScan = tempo de scan*, *const int numRungs = número total de degraus* e *clock time*, em Declarations (do Projeto)

Criar um template com o nome *ScanCycle*

Declarar *bool checkEndScanCycle(){return (*$\forall$ *controlNumTimer* $\in$ *programa Ladder* $\mapsto$ *controlNumTimer == 0) && startExecution == 0}* em Declarations deste template

Declarar *void updateOutputs(){* $\forall$ *nameCoil* $\in$ *programa Ladder | nameCoil não é variavel de entrada atribuir nameCoil = output_nameCoil}*

Criar um autômato com quatro localidades $\{s_1$, s_2, s_3 e $s_4\}$, sendo s_1 inicial e committed, s_2 com invariante time <= tScan e s_3 *commited*

Criar uma transição $s_1 \rightarrow s_2$ atribuindo: *(startExecution = 1)* ao campo *Update* e *start?* ao campo *Sync*

Criar uma transição $s_2 \rightarrow s_2$ atribuindo: *numRungsExecuted++* ao campo *Update*, *execute!* ao campo *Sync* e *numRungsExecuted* < *numRungs* ao campo *Guard*

Criar uma transição $s_2 \rightarrow s_3$ atribuindo: *(time = 0, updateOutputs())* ao campo *Update*, *update!* ao campo *Sync* e *(checkEndScanCycle () == 0 && time >= tScan && numRungsExecuted == numRungs)* ao campo *Guard*

Criar uma transição $s_3 \rightarrow s_4$ atribuindo: *indice = 0* ao campo *Update* e *startEvaluation!* ao campo *Sync*

Criar uma transição $s_4 \rightarrow s_1$ atribuindo: *(numRungsExecuted = 0, numRungsProcessed = 0)* ao campo *Update* e *endEvaluation!* ao campo *Sync*

4.2 Fluxo de execução dos autômatos temporizados

Esta seção tem a finalidade de exibir a seqüência do fluxo de eventos ocorridos na rede de autômatos temporizados gerada. Na Figura 4.13 esta seqüência é ilustrada. Retângulos representam os tipos de autômatos e setas representam a emissão de eventos.

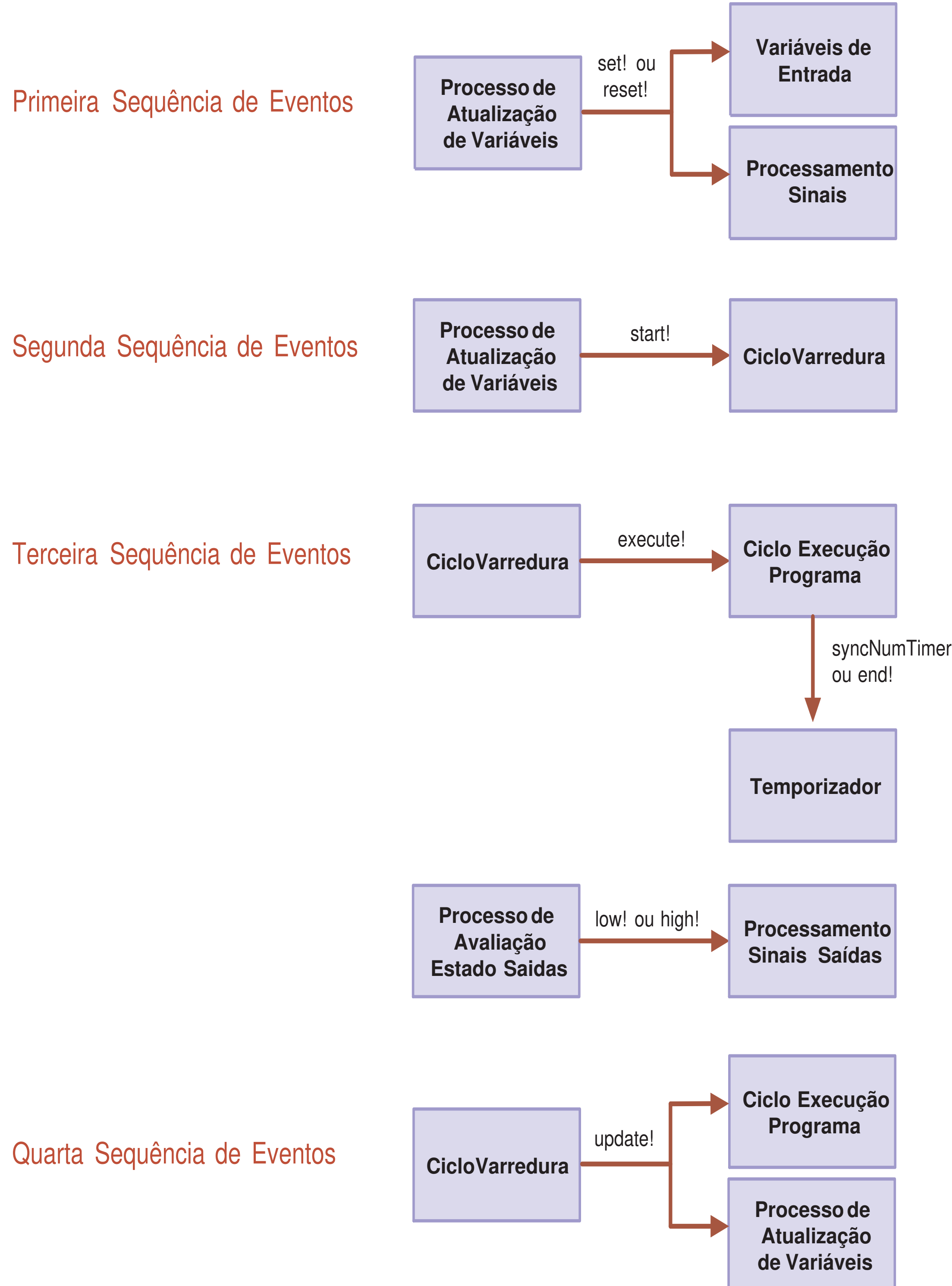


Figura 4.13: Sequência do fluxo de eventos ocorridos na rede de autômatos temporizados

A primeira seqüência de eventos corresponde à atualização dos valores das variáveis de entrada. Este passo é realizado pelo autômato que modela o processo de atualização de variáveis de entrada através do envio de mensagens *set!* ou *reset!* aos autômatos que modelam, respectivamente, variáveis de entrada e o processamento dos sinais de entrada. É importante lembrar que esta seqüência só será realizada se variáveis de entrada forem modeladas como autômatos temporizados, caso isto não ocorra, o fluxo de execução de eventos ocorrerá a partir da segunda sequência de eventos.

Na segunda seqüência de eventos, o autômato que realiza atualização dos valores das variáveis de entrada envia uma mensagem *start!* para o autômato que representa o ciclo de varredura do CLP, informando que a execução da lógica do programa pode ser inicializada.

A terceira seqüência de eventos corresponde à execução seqüencial de todos os degraus ou blocos que determinam saídas. Assim, o autômato que representa o ciclo de varredura do CLP envia mensagens *execute!* para o autômato que representa o ciclo de execução do programa para que a lógica do programa seja executada. Este último autômato envia mensagens para autômatos que representam o comportamento dos temporizadores, caso existam, através das mensagens *syncNumtimer!*, indicando que o temporizador pode executar, e *end!*, indicando que o temporizador deve parar sua execução.

A quarta seqüência de eventos corresponde à finalização da execução do programa e a liberação dos valores das saídas do sistema. Assim, o autômato que representa o ciclo de varredura envia uma mensagem *update!* para os autômatos que representam, respectivamente, o ciclo de execução do programa e o processo de atualização de variáveis, indicando que o processamento da lógica do programa foi completamente executada e as saídas do sistema foram liberadas.

A quinta seqüência de eventos corresponde à avaliação dos estados das saídas do sistema. Esta avaliação tem o objetivo de verificar se cada saída está ou não ativada. Assim, o autômato que representa o ciclo de varredura do CLP envia uma mensagem *startEvaluation!* para o autômato que avalia os estados das saídas. Este autômato envia mensagens *high!* ou *low!* para o autômato que representa o processamento de sinais de saídas.

A sexta seqüência de eventos corresponde ao término da avaliação dos estados das saídas do sistema, desta forma, o autômato que representa o ciclo de varredura do CLP envia uma mensagem *endEvaluation!* para o autômato que avalia os estados das saídas.

4.3 Geração dos modelos de autômatos temporizados

O esquema utilizado para a geração automática dos modelos de autômatos temporizados a partir de diagramas ISA 5.2 e código Ladder é ilustrado na Figura 4.14. As seguintes informações são extraídas destes documentos:

- Variáveis de entrada, representadas por contatos normalmente aberto ou fechado;
- Temporizadores, representados por blocos contendo informações como valor de PT, nome e tipo do temporizador;
- Variáveis de saída, representadas por bobinas;
- Expressões lógicas que determinam cada saída do sistema.

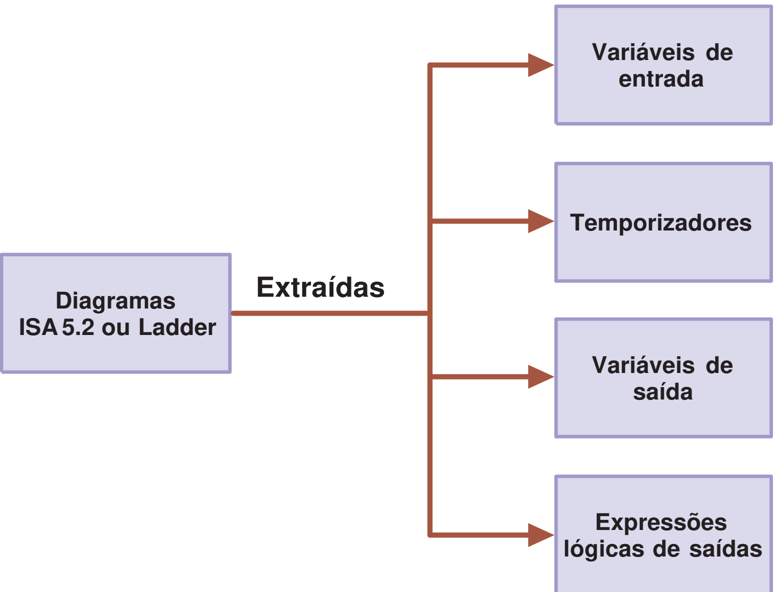


Figura 4.14: Esquema utilizado para geração dos modelos de autômatos temporizados

A partir da extração destas informações são gerados automaticamente autômatos temporizados, referentes a variáveis de entrada, temporizadores, ciclo de execução do programa e ciclo de varredura do CLP. Os autômatos que representam o processo de atualização de variáveis de entrada e o processo de avaliação dos estados das saídas não necessitam de qualquer tipo de informação proveniente dos documentos de especificação e implementação, para serem gerados. Desta forma, independente do sistema a ser modelado, estes autômatos

são gerados automaticamente segundo os modelos ilustrados nas Figuras 4.4, 4.5, 4.10 e 4.11.

O processo de modelagem de variáveis de entrada como modelos de autômatos temporizados é realizado a partir do conjunto formado pelas variáveis de entrada e, do conjunto formado pelas variáveis de saídas do sistema. Como variáveis de *feedback* e variáveis que representam ao mesmo tempo entrada e saída do sistema não devem ser modeladas, então, apenas variáveis de entrada que não pertencem ao conjunto de variáveis de saídas são modeladas.

O processo de modelagem de temporizadores como modelos de autômatos temporizados é realizado a partir do conjunto formado pelos temporizadores do programa. Cada temporizador é modelado segundo seu tipo e sua respectiva posição, detectada no processamento dos documentos de especificação ou implementação.

O processo de modelagem do ciclo de execução do programa como modelo de autômato temporizado é realizado a partir das variáveis de saída, das expressões lógicas que determinam as saídas e dos temporizadores. A modelagem de tal autômato segue um esquema seqüencial de execução de degraus, no caso de diagramas Ladder, e blocos que determinam uma saída, no caso de diagramas ISA 5.2. Desta forma, o valor referente a cada saída do sistema é processado a partir da expressão lógica que a determina junto com a lógica de funcionamento de seus respectivos temporizadores.

O processo de modelagem do ciclo de varredura do CLP como modelo de autômato temporizado é realizado a partir das variáveis de saída, das variáveis de entrada e dos temporizadores. Existe apenas duas partes deste modelo que dependem de informação proveniente dos documentos de especificação ou implementação para serem gerados, a função que libera as saídas dos sistema e, a função que avalia se todos os temporizadores não estão mais sendo executados. Para a primeira função, os valores das saídas que ainda não foram liberados durante a execução do programa devem ser liberados. O conjunto que compõe as saídas que ainda não foram liberadas é formado a partir do conjunto de todas as variáveis de saída que não pertencem ao conjunto de variáveis de entrada, ou seja, as saídas que não são utilizadas como variáveis de entrada no sistema. Para a segunda função, o valor da variável *controlNumTimer* deve ser igual a *0* para todos os temporizadores do programa.

Capítulo 5

Estudo de Caso

Neste capítulo dois estudos de caso são ilustrados para validar o método proposto. O primeiro sistema cuja finalidade é encher garrafas foi retirado do livro (Bryan & Bryan, 1997), página 485. O segundo sistema cujo objetivo é controlar dois semáforos em vias que se cruzam foi adaptado de um exemplo ilustrado em (Olderog & Dierks, 2008), página 192.

5.1 Definição do sistema que enche garrafas

Na Figura 5.1 um sistema que tem a finalidade de encher garrafas é ilustrado. Seu funcionamento ocorre da seguinte forma: uma vez que o botão de inicializar (*PB1*) é pressionado, o motor de auto-realimentação (*M2*) é ligado. Este motor permanecerá ligado até que o botão de parar (*PB2*) seja acionado. O motor *M1* será ativado assim que o sistema for iniciado (*M2 on*); e irá parar quando o sensor (*LS*) detectar uma garrafa na posição correta. Quando a garrafa estiver na posição correta e 0.5 segundos tiverem se passado, o solenóide (*SOL*) irá abrir a válvula para liberar o refrigerante e, o enchimento ocorrerá até que o fotosensor (*PE*) detecte um nível adequado de líquido no interior da garrafa. Após ser enchida, a garrafa permanecerá nesta posição durante 0.7 segundos. Em seguida, o motor *M1* é inicializado. Este irá permanecer ligado até que o sensor detecte outra garrafa.

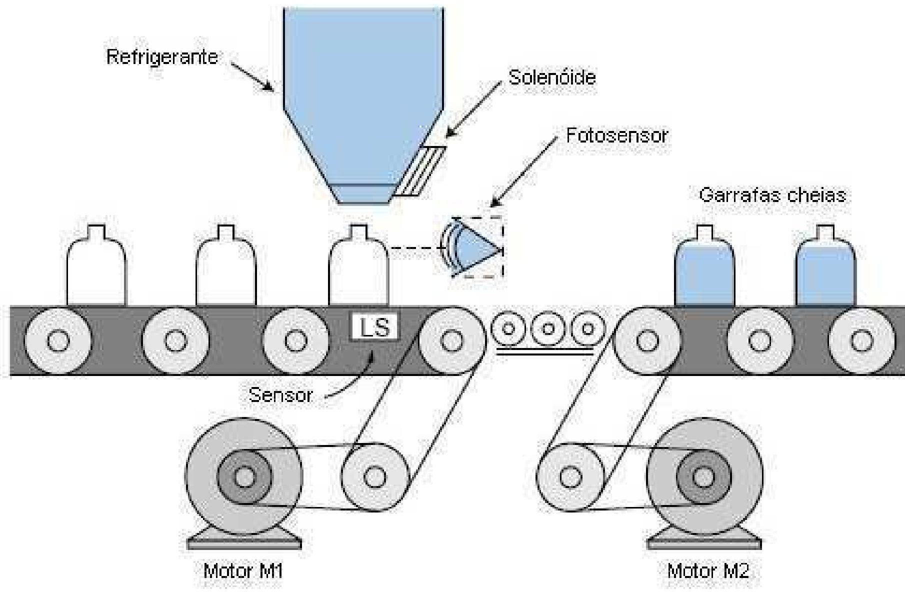


Figura 5.1: Sistema que enche garrafas - Fonte: (Bryan & Bryan, 1997), página 485

5.2 Diagrama ISA 5.2 e Programa Ladder para o sistema que enche garrafas

Nas Figuras 5.2 e 5.3 são ilustrados, respectivamente. a modelagem do sistema que enche garrafas como Diagramas ISA 5.2 e Diagramas Ladder.

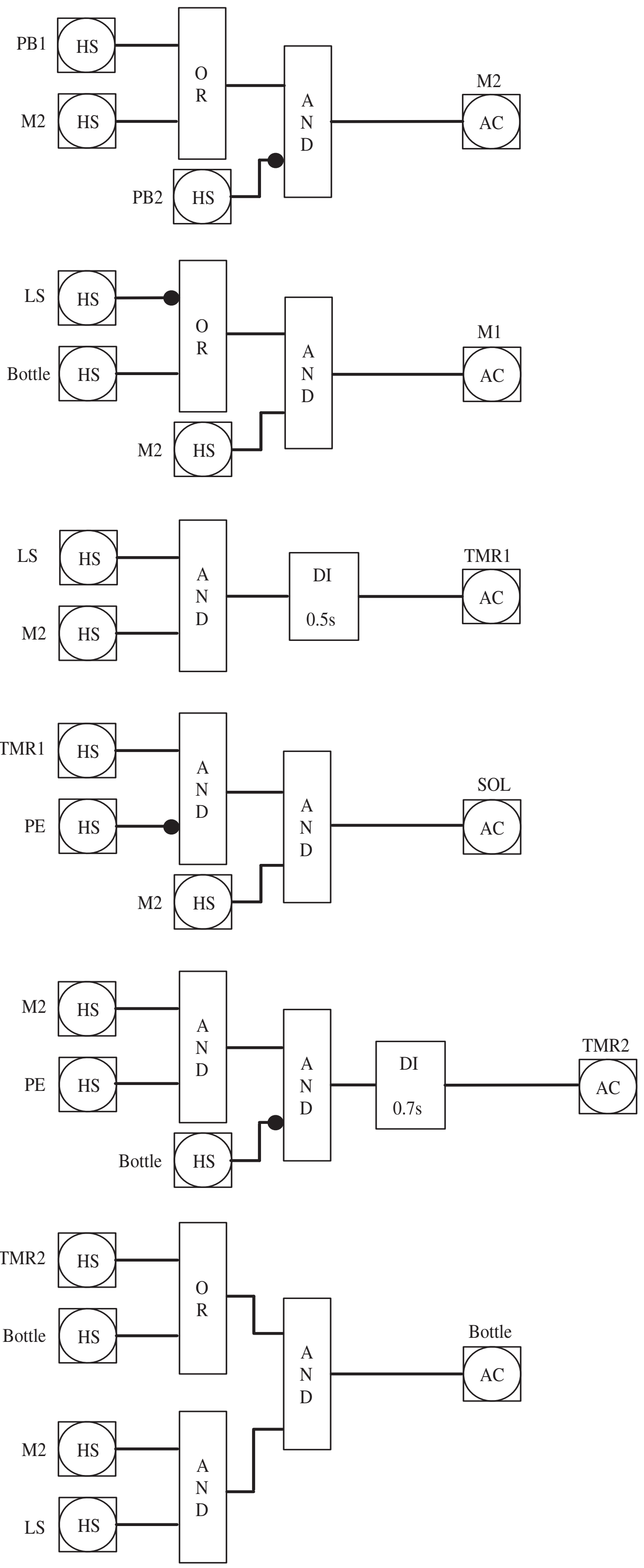


Figura 5.2: Diagrama ISA 5.2 para o sistema que enche garrafas

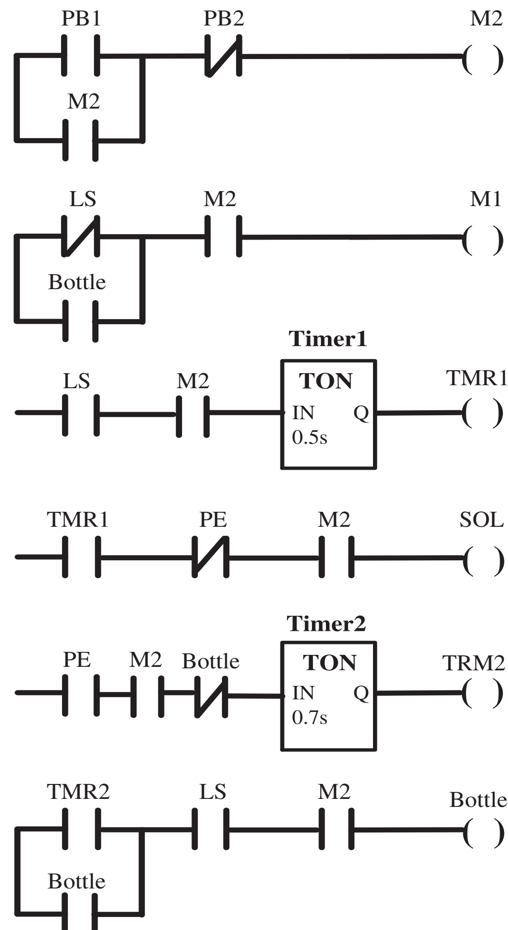
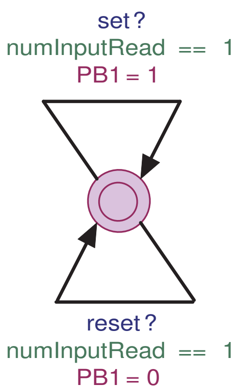


Figura 5.3: Programa Ladder para o sistema que enche garrafas

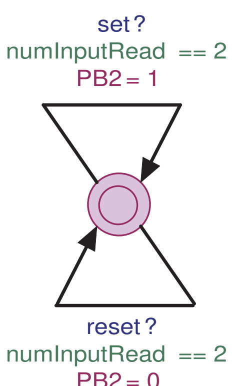
5.3 Modelagem do sistema que enche garrafas

Nas Figuras 5.4 a 5.12 a modelagem do comportamento do sistema que enche garrafas, descrito na seção 5.1, como uma rede de autômatos temporizados é ilustrada.

As Figuras 5.4(a), 5.4(b), 5.4(c) e 5.4(d) apresentam as modelagens das respectivas variáveis entrada, *PB1*, *PB2*, *LS* e *PE* como autômatos temporizados. As variáveis *M2*, *TMR1*, *TMR2* e *Bottle* não são modeladas como autômatos temporizados, pois, as variáveis *M2* e *Bottle* são variáveis de *feedback* e as variáveis *TMR1* e *TMR2* além de serem entradas para o sistema também são saídas do sistema.



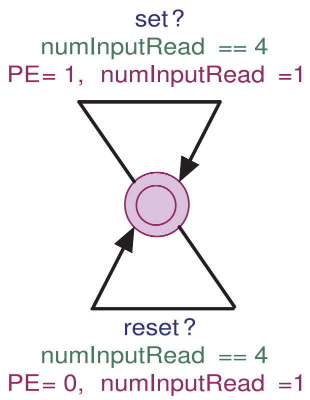
(a) Autômato que representa a molagem da variável de entrada PB1



(b) Autômato que representa a molagem da variável de entrada PB2

set?
numInputRead == 3
LS = 1
reset?
numInputRead == 3
LS = 0

(c) Autômato que representa a molagem da variável de entrada LS



(d) Autômato que representa a molagem da variável de entrada PE

Figura 5.4: Autômatos que representam as modelagens das variáveis de entrada do sistema que enche garrafas

Na Figura 5.5 a modelagem utilizada para realizar a atualização, de forma seqüencial, das variáveis de entrada pode ser visualizada. A variável *numInputs* deve ser inicializada com valor *4*, pois quatro variáveis de entrada foram modeladas como autômatos temporizados. Na Figura 5.6 é apresentado o autômato responsável por incrementar o valor de *numInputRead* toda vez que alguma variável de entrada for atualizada.

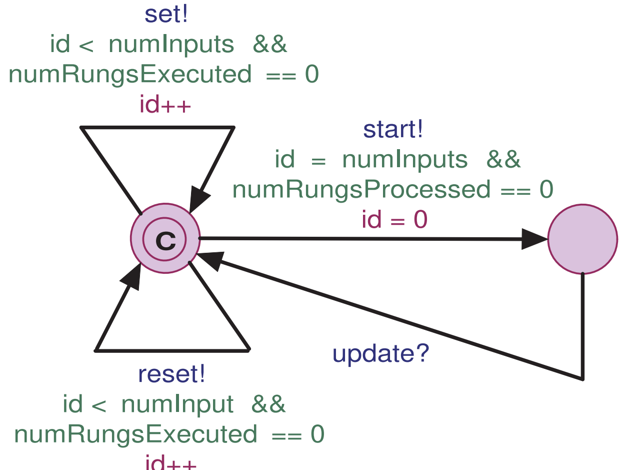


Figura 5.5: Autômato que representa o processo de atualização de variáveis de entrada para o sistema que enche garrafas

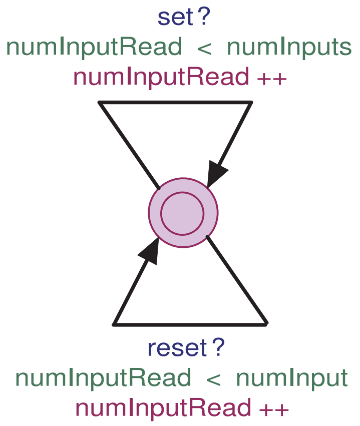


Figura 5.6: Autômato que representa a modelagem do processamento dos sinais de entrada para o sistema que enche garrafas

Na Figura 5.7 o autômato utilizado para representar o comportamento do temporizador TON denominado de *Timer1* é ilustrado. Como ele é o primeiro temporizador encontrado na seqüência da execução do programa, então o sufixo *NumTimer* para este temporizador irá possuir valor *1*. Assim, variáveis como *controlNumTimer* serão transformadas em *control1*. O valor de *PT* (*Present Time*) corresponde a 500 microssegundos logo, a expressão

numCyclesNumTimer = PT/tScan será transformada em *numCycles1 = 500/tScan*. A função *inputTimer()* será declarada da seguinte forma: *bool inputTimer() {return (LS and M2); }*

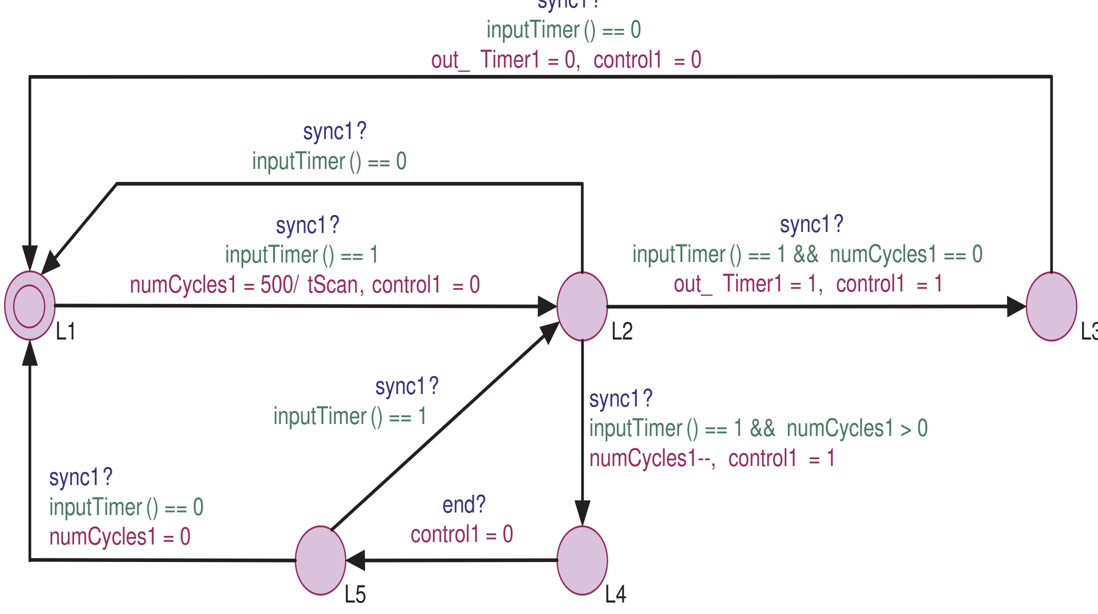


Figura 5.7: Autômato que representa o temporizador Timer1 do sistema que enche garrafas

Na Figura 5.8 o autômato utilizado para representar o comportamento do temporizador TON denominado de *Timer2* é ilustrado. Como ele é o segundo temporizador encontrado na seqüência da execução do programa. O valor de *PT* (*Present Time*) corresponde a 700 microsegundos logo, a expressão *numCyclesNumTimer = PT/tScan* será transformada em *numCycles2 = 700/tScan*. A função *inputTimer()* será declarada da seguinte forma: *bool inputTimer() {return ((PE and M2) and (!Bottle)); }*

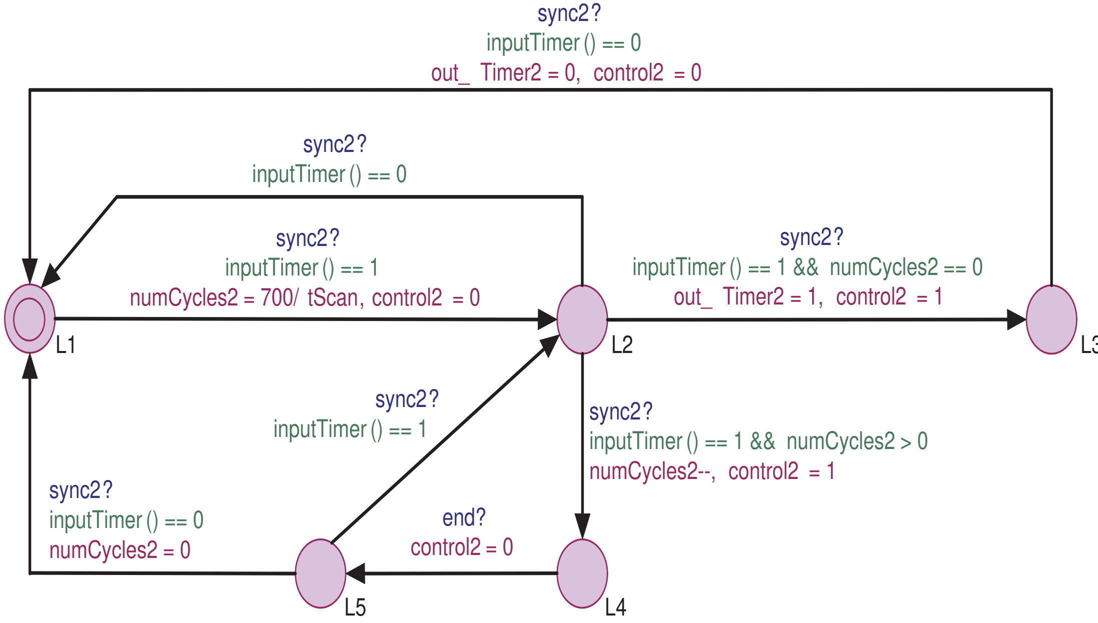


Figura 5.8: Autômato que representa o temporizador Timer2 do sistema que enche garrafas

Na Figura 5.9 é apresentada a modelagem da execução da lógica do programa como um autômato temporizado. A execução dos degraus ocorre seqüencialmente através do disparo de transições entre localidades distintas que possuam a mensagem *execute?*. As funções utilizadas para modelar tal autômato são as seguintes:

- Como os degraus que contém as bobinas *M2*, *M1*, *SOL* e *Bottle* não possuem elementos temporizados, então o valor de cada lógica de controle que compõe tais degraus é determinada através da execução de funções rotuladas por *value_NameCoil()*, onde o sufixo *NameCoil* representa o nome da bobina contida no degrau que está sendo processado. Estas funções são ilustradas a seguir:

  – void value_M2(){

    M2 = ((PB1 or M2) and !PB2);

    outputs[0] = M2;}

  – void value_M1(){

    out_M1 = ((Bottle or !LS) and M2);

    outputs[1] = output_M1;}

- void value_SOL(){

  out_SOL = ((TMR1 and !PE) and M2);

  outputs[3] = output_SOL;}

- void value_Bottle(){

  Bottle = ((TMR2 or Bottle) and (LS and M2));

  outputs[5] = Bottle;}

• Como o degrau que contém a bobina *TMR1* possui 1 elemento temporizado do tipo TON, então a execução de três funções rotuladas por *checkExecutionTimer1()*, *evaluateOuputTimer1()* e *outTimer1()* determinam o valor da lógica de controle do degrau representado por tal bobina. Estas funções são ilustradas a seguir:

  - void outTimer1(){

    TMR1 = (out_Timer1);

    outputs[2] = TMR1;}

  - bool checkExecutionTimer1(){

    return ((LS and M2) == 1 && control1 == 0 && out_Timer1 == 0)

    || ((LS and M2) == 0 && control1 == 0 && out_Timer1 == 1)

    || ((LS and M2) == 0 && control1 == 0 && numCycles1 > 0);}

  - bool evaluateOuputTimer1(){

    return (out_Timer1 == 1 && (LS and M2) == 1) ||

    ((LS and M2) == 0 && control1 == 0 && out_Timer1 == 0

    && numCycles1 == 0) || (control1 == 1 && (LS and M2) == 1);}

• Como o degrau que contém a bobina *TMR2* possui 1 elemento temporizado do tipo TON, então a execução de três funções rotuladas por *checkExecutionTimer2()*, *evaluateOuputTimer2()* e *outTimer2()* determinam o valor da lógica de controle do degrau representado por tal bobina. Estas funções são ilustradas a seguir:

  - void outTimer2(){

    TMR2 = (out_Timer2);

    outputs[4] = TMR2;}

- bool checkExecutionTimer2(){

  return (((PE and M2) and !Bottle) == 1 && control2 == 0 && out_Timer2 == 0) || (((PE and M2) and !Bottle) == 0 && control2 == 0 && out_Timer2 == 1) || (((PE and M2) and !Bottle) == 0 && control2 == 0 && numCycles2 > 0);}

- bool evaluateOuputTimer2(){

  return (out_Timer2 == 1 && ((PE and M2) and !Bottle) == 1) || (control2 == 1 && ((PE and M2) and !Bottle) == 1) || (out_Timer2 == 0 && ((PE and M2) and !Bottle) == 0 && numCycles2 == 0);}

- A função *setControlVariables()* tem a finalidade de restaurar o valor de todas as variáveis *controlNumTimer* para *0*, no caso, control1 e control2, de atualizar o valor de *numRungsProcessed* para *numOutputs*, informando que todos os degraus foram processados e, de atualizar o valor de *startExecution* para *0*. Esta função é ilustrada a seguir:

  void setControlVariables(){

  control1 = control2 = 0;

  numRungsProcessed = numOutputs;

  startExecution = 0;}

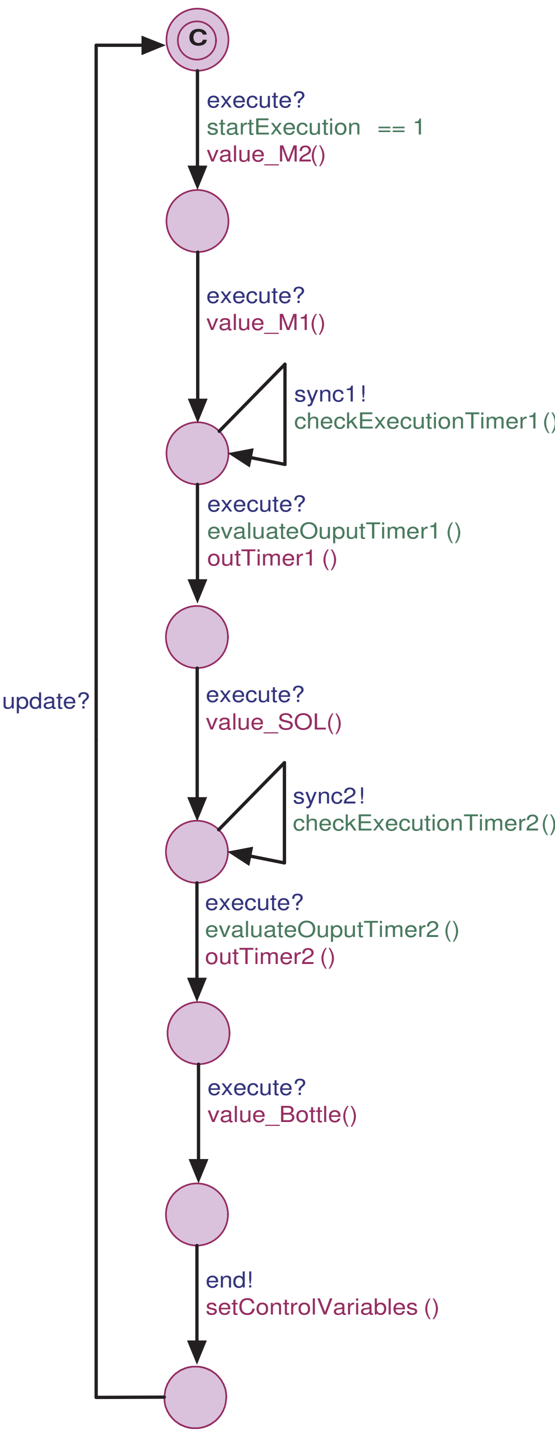


Figura 5.9: Autômato que representam a execução do programa do sistema que enche garrafas

Na Figura 5.10 é apresentado o autômato que avalia os estados das saídas (saída acionada ou não acionada). Na Figura 5.11 é apresentado o autômato que processa os estados das saídas.

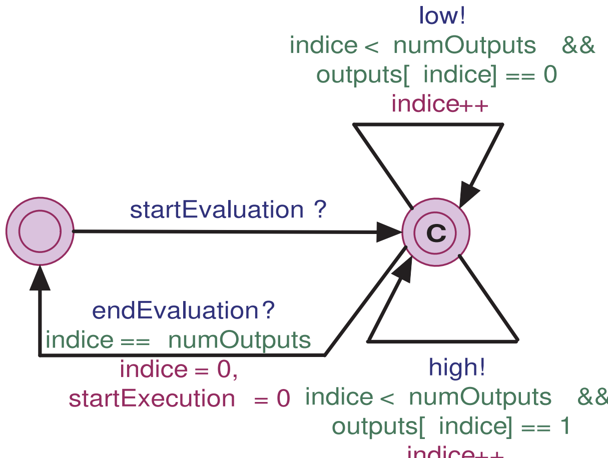


Figura 5.10: Modelagem do processo que avalia os estados das saídas do sistema que enche garrafas

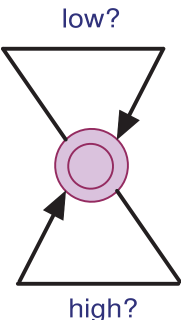


Figura 5.11: Modelagem do processamento dos estados das saídas do sistema que enche garrafas

Na Figura 5.12 é ilustrado o ciclo de varredura de um CLP. A função *checkEndScanCycle()*, utilizada na modelagem de tal autômato, avalia se todos os temporizadores não estão mais sendo executados, no caso, avalia se *control1 == control2 == 0* e, se a execução do programa foi finalizada, *startExecution == 0.* A função *updateOutputs()* libera as saídas do sistema que ainda não foram liberadas durante a execução do programa, no caso, os valores de *M1* e *SOL*. Estas funções são ilustradas a seguir:

```
bool checkEndScanCycle(){
    return (control1 = control2 = 0) && startExecution == 0;}
```

```
void updateOutputs(){
    M1 = out_M1;
    SOL = out_SOL;}
```

endEvaluation!
numRungsExecuted = 0,
numRungsProcessed = 0
L4
L1
startEvaluation !
indice = 0
start ?
startExecution = 1
time <= tScan
execute!
numRungsExecuted <
numTotalRungs
numRungsExecuted ++
L3
update!
checkEndScanCycle() &&
time >= tScan &&
numRungExecuted == numRungs
time = 0, updateOutputs ()
L2

Figura 5.12: Autômato que representam ciclo de varredura do CLP para o sistema que enche garrafas

5.4 Testes realizados no sistema que enche garrafas

No processo de geração e execução dos casos de teste, a partir do uso do mesmo conjunto de valores de entrada, erros existentes na implementação só serão detectados se o conjunto de saídas referentes à especificação for diferente do conjunto de saídas referentes à implementação. É importante lembrar que, dependendo da combinação dos valores dos elementos que compõem o conjunto de entradas, erros podem não ser detectados, sendo necessária a realização de mais de um traço de execução para detecção dos mesmos. Outro fato importante a ser mencionado é que, a análise dos resultados dos testes testes é feita de forma manual e, apenas pessoas que conhecem a ferramenta Uppaal-TRON são capazes de interpretar tais resultados.

Para este estudo de caso, a configuração e o particionamento do modelo foram realizados da seguinte forma:

- Entradas: set(), reset(), start(), end(), sync1(), sync2(), low(), high();

- Saídas: execute(), update(), evaluation(), out_evaluation();
- Precisão: 1 unidade de tempo;
- Tempo de teste: 300 unidades de tempo.

O primeiro modelo de implementação a ser testado foi um modelo cuja representação é fiel à especificação, ou seja, à rede de autômatos que representa o programa Ladder ilustrado na Figura 5.3. O veredicto dado pela ferramenta Uppaal-TRON foi : "*TESTED PASSED: time out for testing*". Assim, durante o tempo programado para a execução dos testes, no caso 300 unidades de tempo, nenhuma anormalidade relacionada ao comportamento do modelo da implementação foi detectada. Portanto, a implementação é fiel à especificação.

Para ilustrar a validade do método, através da detecção de erros existentes na implementação, os seguintes erros foram inseridos no programa Ladder que representa o sistema que enche garrafas:

- Erro 1: Trocar a ordem de execução dos degraus, ou seja, o segundo degrau é executado e depois o primeiro degrau é executado;
- Erro 2: Retirar um degrau, no caso o quarto degrau;
- Erro 3: Trocar um contato normalmente aberto por um contato normalmente fechado, no caso, para o terceiro degrau, a variável *LS* será representada por um contato normalmente fechado;
- Erro 4: Trocar uma operação lógica *and* por uma operação lógica *or*, no caso, para o quarto degrau, a variável *PE* representada por um contato normalmente fechado será colocada em paralelo com a variável representada pelo contato normalmente aberto *TMR1*;
- Erro 5: Alterar o valor de *PT* de *Timer 1* para 0.3 segundos.

5.4.1 Aplicação do processo de teste para ocorrência do erro 1: O segundo degrau é executado e depois o primeiro degrau é executado

Na Figura 5.13 é apresentada a inserção do primeiro tipo de erro no programa Ladder que modela o sistema que enche garrafas.

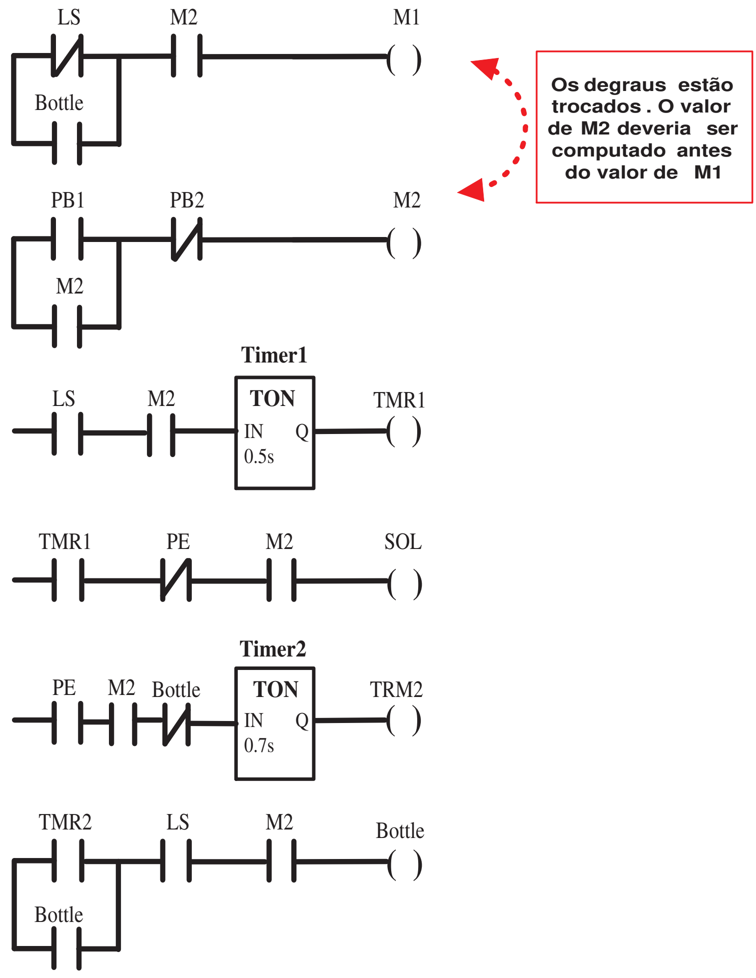


Figura 5.13: Inserção do erro 1 no programa Ladder do sistema que enche garrafas

O veredicto dado pela ferramenta Uppaal-TRON foi : "*TESTED FAILED*". O traço de execução indicou que a saída *high()* era esperada mas a saída *low()* foi encontrada no instante de tempo 8. Em outras palavras, era esperado que o valor da primeira saída do programa Ladder fosse *1* mas seu valor foi *0*. Este valor só pode ser o de *M1* uma vez que o traço de execução gerado para o conjunto de entradas fez com que o motor *M2* fosse acionado, valor lógico *1*. Portanto, a inversão dos degraus causou uma alteração no valor da variável *M1*, ou seja, tal motor deveria está ligado, mas isto não ocorreu.

5.4.2 Aplicação do processo de teste para ocorrência do erro 2: Retirar o quarto degrau

Na Figura 5.14 é apresentada a inserção do segundo tipo de erro no programa Ladder que modela o sistema que enche garrafas.

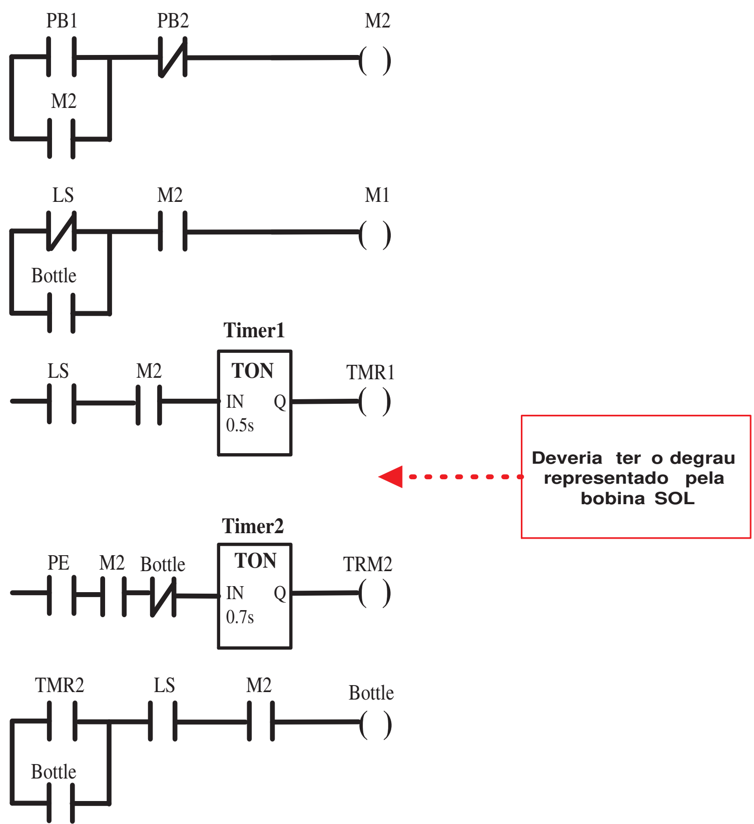


Figura 5.14: Inserção do erro 2 no programa Ladder do sistema que enche garrafas

O veredicto dado pela ferramenta Uppaal-TRON foi : "*TESTED FAILED*". O traço de execução indicou que a saída *execute()* era esperada mas a saída *end()* foi encontrada no instante de tempo 4. Em outras palavras, a execução do último degrau representado pela bobina *Bottle* era esperada, mas o final da execução dos degraus foi executado. Isto aconteceu porque no modelo da implementação existem cinco degraus e no modelo da especificação existem seis blocos que determinam saídas.

5.4.3 Aplicação do processo de teste para ocorrência do erro 3: Trocar o contato normalmente aberto da variável *LS*, no terceiro degrau, por um contato normalmente fechado

Na Figura 5.15 é apresentada a inserção do terceiro tipo de erro no programa Ladder que modela o sistema que enche garrafas.

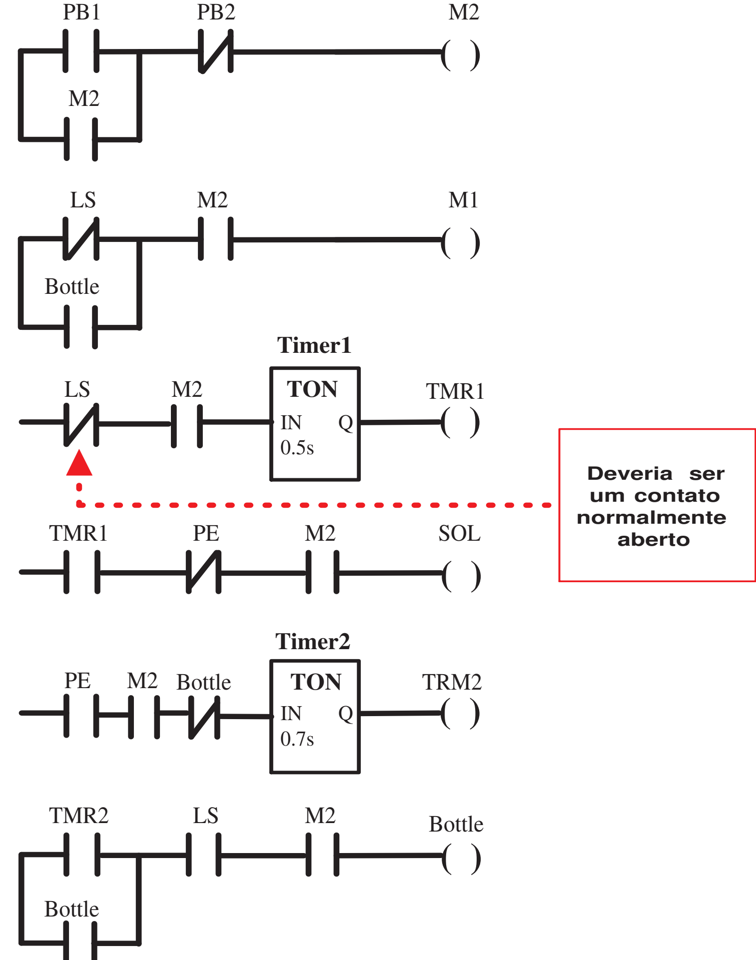


Figura 5.15: Inserção do erro 3 no programa Ladder do sistema que enche garrafas

O veredicto dado pela ferramenta Uppaal-TRON foi : "*TESTED FAILED*". O traço de execução indicou que uma saída *execute()* foi esperada mas uma saída *sync1()* foi encontrada no instante de tempo 27. Em outras palavras era esperado que o temporizador Timer 1 não fosse executado, mas sua entrada *IN = !LS and M2* estava ativada, e ele executou. Através da reconstrução do traço de execução pode-se observar que o motor *M2* estava ligado e o sensor

LS não estava ativado então, a possível explicação para a ocorrência deste fato é a troca do contato normalmente aberto pelo contato normalmente fechado representado pela variável de entrada *LS*.

5.4.4 Aplicação do processo de teste para ocorrência do erro 4: No quarto degrau a variável *PE* será colocada em paralelo com a variável *TMR1*

Na Figura 5.16 é apresentada a inserção do quarto tipo de erro no programa Ladder que modela o sistema que enche garrafas.

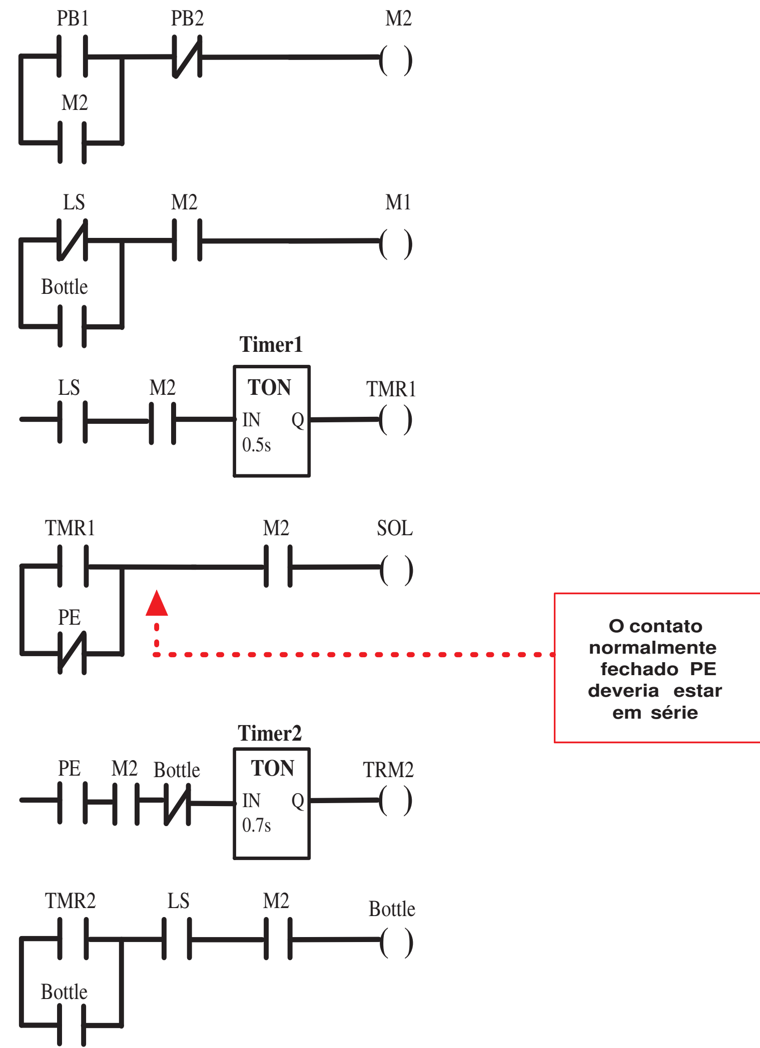


Figura 5.16: Inserção do erro 4 no programa Ladder do sistema que enche garrafas

O veredicto dado pela ferramenta Uppaal-TRON foi : "*TESTED FAILED*". O traço de execução indicou que uma saída *high()* foi esperada, mas uma saída *low()* foi encontrada no instante de tempo 30. Em outras palavras, era esperado que, o valor da bobina representada pela variável *SOL* fosse *1*, mas o valor *0* foi encontrado. Em outras palavras era esperado que, a expressão *(TMR and !PE)* possui-se valor lógico 1, mas seu valor foi *0*. Através da reconstrução do traço de execução pode-se observar que, a bobina representada por *TMR1* estava acionada e o fotosensor *PE* também estava acionado então, a possível explicação para a ocorrência deste fato é a variável *PE* está em paralelo com a variável *TMR1*.

5.4.5 Aplicação do processo de teste para ocorrência do erro 5 : Alterar o valor de *PT* de *Timer 1* para 0.3 segundo

Na Figura 5.17 é apresentada a inserção do quinto tipo de erro inserido no programa Ladder que modela o sistema que enche garrafas.

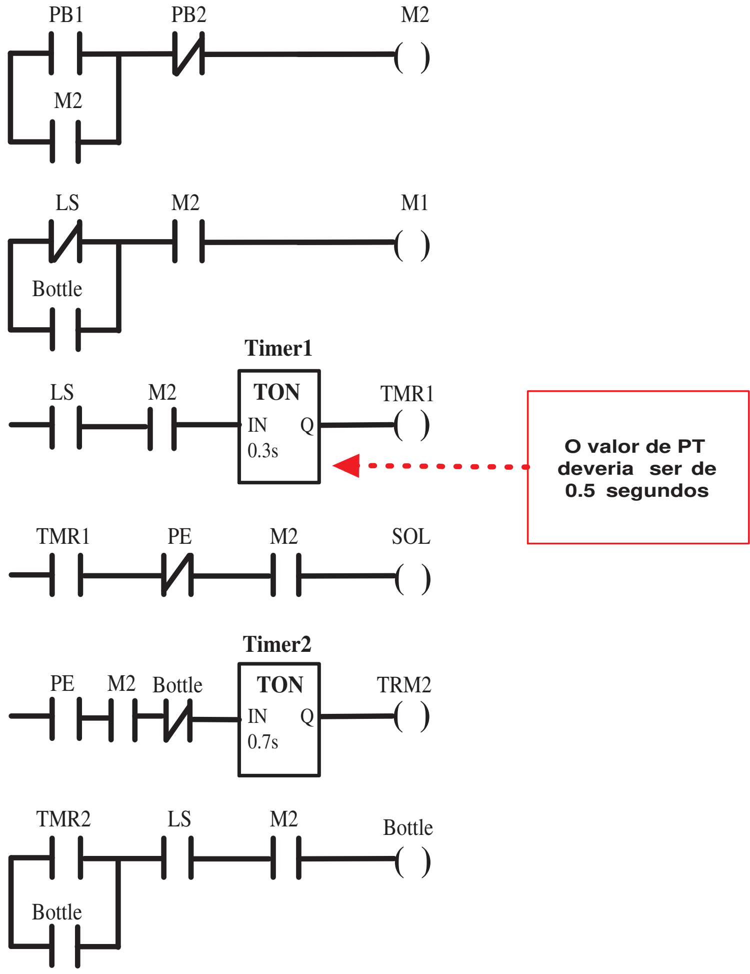


Figura 5.17: Inserção do erro 5 no programa Ladder do sistema que enche garrafas

O veredicto dado pela ferramenta Uppaal-TRON foi : "*TESTED FAILED*". O traço de execução indicou que uma saída *sync1()* foi esperada, mas uma saída *execute()* foi encontrada no instante de tempo 53. Em outras palavras era esperado que o temporizador *Timer 1* continuasse a ser executado, pois a execução de dois ciclo ainda eram necessárias para que sua saída energizada fosse liberada, mas a saída ativada do temporizador *Timer 1* já havia sido liberada.

5.5 Definição do Sistema que controla dois semáforos

Na Figura 5.18 é apresentado um sistema que tem a finalidade de controlar dois semáforos de duas ruas (*1* e *2*) que se cruzam. Este controle é realizado por um CLP que envia sinais para ambos os semáforos indicando qual o estado (verde, amarelo, vermelho) que eles devem permanecer por um determinado período de tempo.

Seu funcionamento ocorre de acordo com o seguinte ciclo de eventos: inicialmente o sinal da rua *1* deve permanecer fechado durante 50 segundos e o sinal da rua *2* deve permanecer aberto durante 40 segundos. Após se passarem 40 segundos o sinal da rua *2* deve permanecer amarelo durante 10 segundos. Quando 10 segundos se passarem o sinal da rua *1* deve permanecer verde durante 20 segundos e o sinal da rua 2 deve permanecer vermelho durante 30 segundos. Após 20 segundos o sinal da rua *1* deve permanecer amarelo durante 10 segundos. Após o termino deste ciclo, outros ciclos, com a mesma configuração, devem ser executados para que o tráfego entre as ruas seja realizado de forma segura. Este esquema é ilustrado na Figura 5.19.

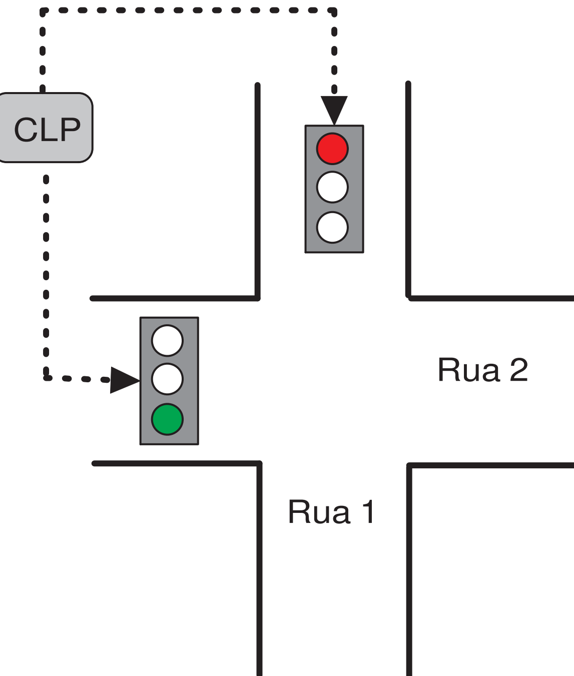


Figura 5.18: Sistema que controla semáforos

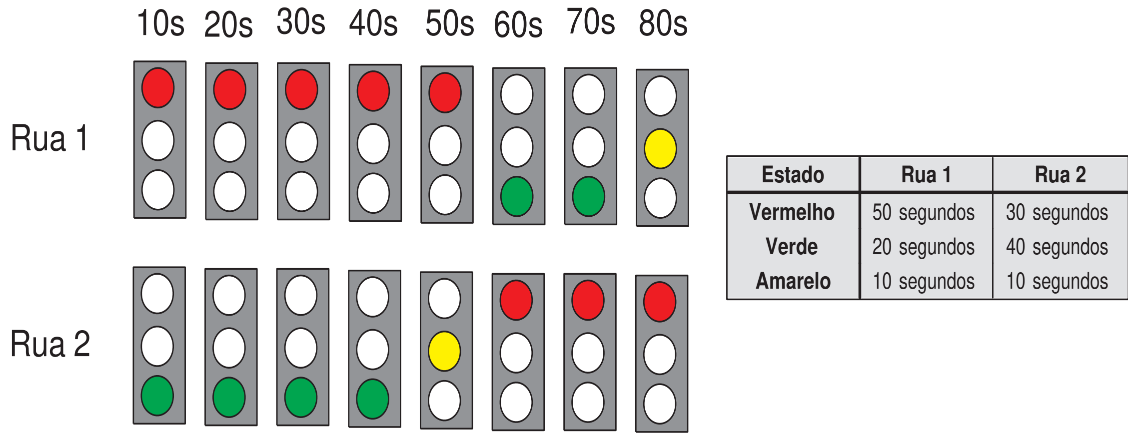


Figura 5.19: Esquema de funcionamento dos semáforos das ruas 1 e 2

5.6 Diagrama ISA 5.2 e Programa Ladder para o sistema que controla dois semáforos

Nas Figuras 5.20 e 5.21 são ilustradas as modelagens do sistema que controla dois semáforos como Diagramas ISA 5.2. Nas Figuras 5.22 e 5.23 são ilustradas as modelagens do sistema que controla dois semáforos como Diagramas Ladder. As variáveis utilizadas nos diagramas possuem a seguinte legenda:

- TMR1 = processa saída sinal vermelho rua 1;
- TMR2 = processa saída sinal verde rua 2;
- TMR3 = processa saída sinal amarelo rua 2;
- TMR4 = processa saída sinal vermelho da rua 2;
- TMR5 = processa saída sinal verde rua 1;
- TMR6 = processa saída sinal amarelo rua 1;
- VmR1 = sinal vermelho da rua 1;
- VdR1 = sinal verde da rua 1;
- AmR1 = sinal amarelo da rua 1;

- VmR2 = sinal vermelho da rua 2;
- VdR2 = sinal verde da rua 2;
- AmR2 = sinal amarelo da rua 2.

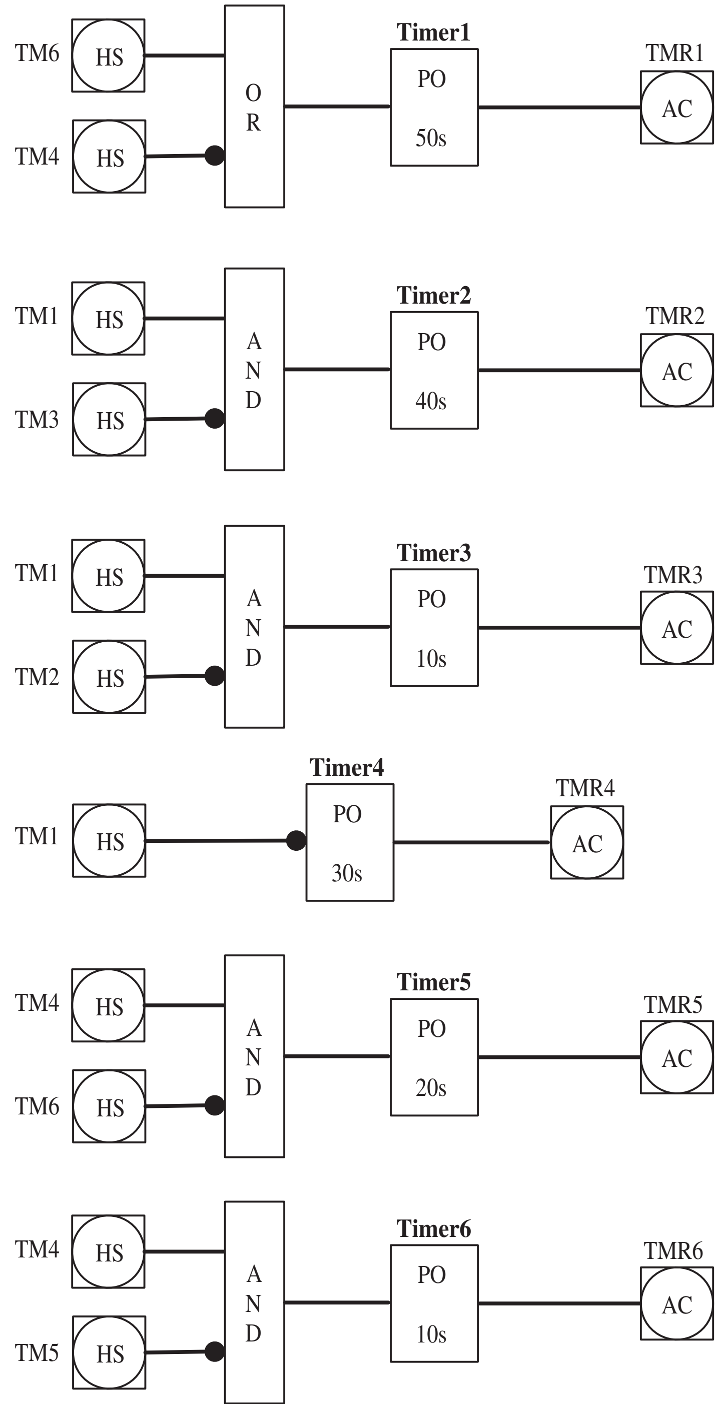


Figura 5.20: Diagrama ISA 5.2 para o sistema que controla semáforos - Parte 1

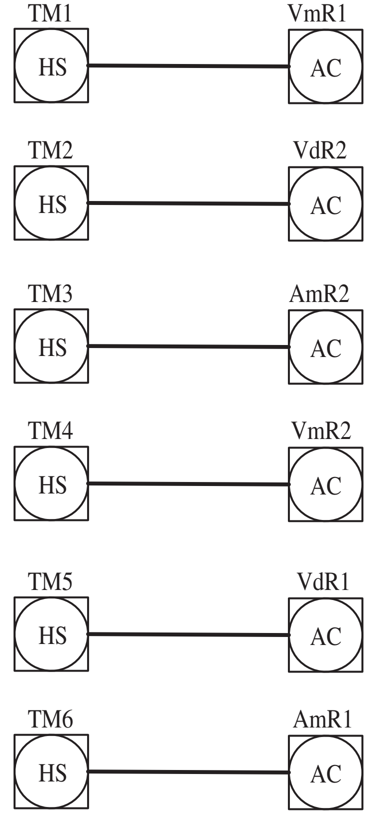


Figura 5.21: Diagrama ISA 5.2 para o sistema que controla semáforos - Parte 2

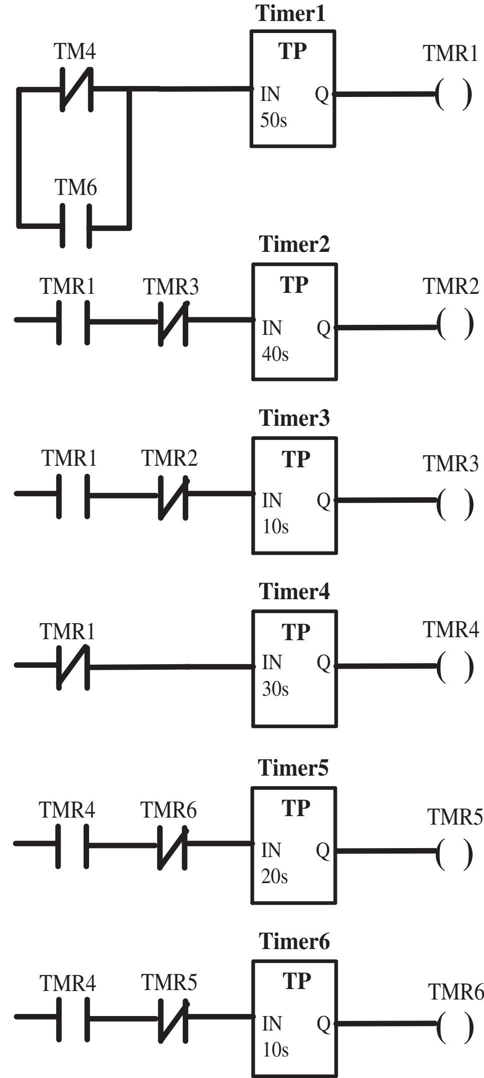


Figura 5.22: Programa Ladder para o sistema que controla semáforos - Parte 1

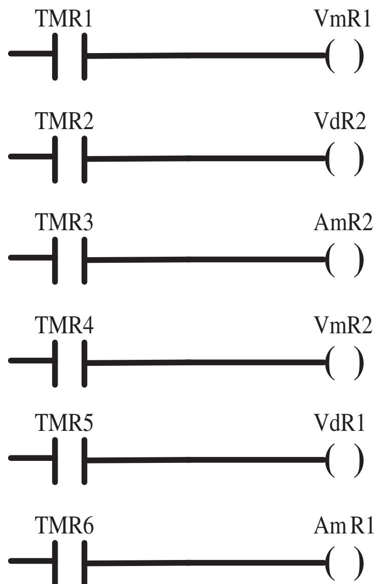


Figura 5.23: Programa Ladder para o sistema que controla semáforos - Parte 2

5.7 Modelagem do sistema que controla dois semáforos

Nas Figuras 5.24 a 5.35 a modelagem do comportamento do sistema que controla dois semáforos, descrito na seção 5.5, como uma rede de autômatos temporizados é ilustrada.

Autômatos que representam variáveis de entrada para o sistema não são modelados para este estudo de caso, pois as variáveis *TMR1*, *TMR2*, *TMR3*, *TMR4*, *TMR5* e *TMR6* além de serem entradas para o sistema também são saídas do sistema.

Na Figura 5.24 a modelagem utilizada para realizar a atualização, de forma sequencial, das variáveis de entrada pode ser visualizada. A variável *numInputs* deve ser inicializa com valor *0*, pois nenhuma variável de entrada foi modelada como autômato temporizado. Na Figura 5.25 o valor da variável *numInputRead* não será incrementado, pois nenhum autômato que modela o comportamento de variáveis de entrada foi gerado.

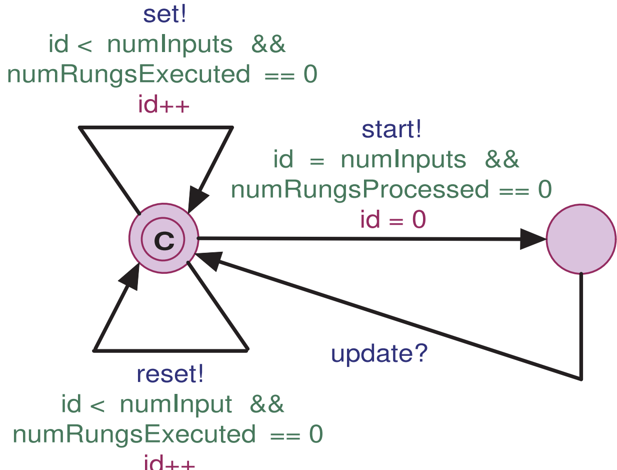


Figura 5.24: Autômato que representa o processo de atualização de variáveis de entrada para o sistema que controla semáforos

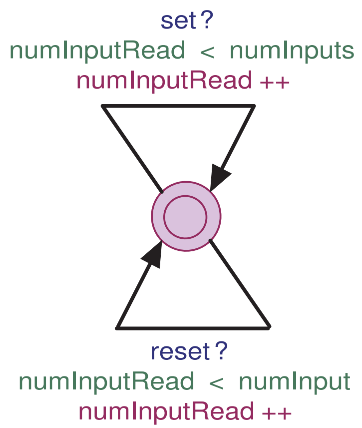


Figura 5.25: Autômato que representa a modelagem do processamento dos sinais de entrada para o sistema que controla semáforos

Na Figura 5.26 o autômato utilizado para representar o comportamento do temporizador TP denominado de *Timer1* é ilustrado. Como ele é o primeiro temporizador encontrado na seqüência da execução do programa, então o sufixo *NumTimer* para este temporizador irá possuir valor *1*. Assim, variáveis como *controlNumTimer* serão transformadas em *control1*. O valor de *PT (Present Time)* corresponde a 50000 microssegundos logo, a expressão *numCyclesNumTimer = PT/tScan* será transformada em *numCycles1 = 50000/tScan*. A função

inputTimer() será declarada da seguinte forma: *bool inputTimer() {return (!TM4 or TM6) }.*

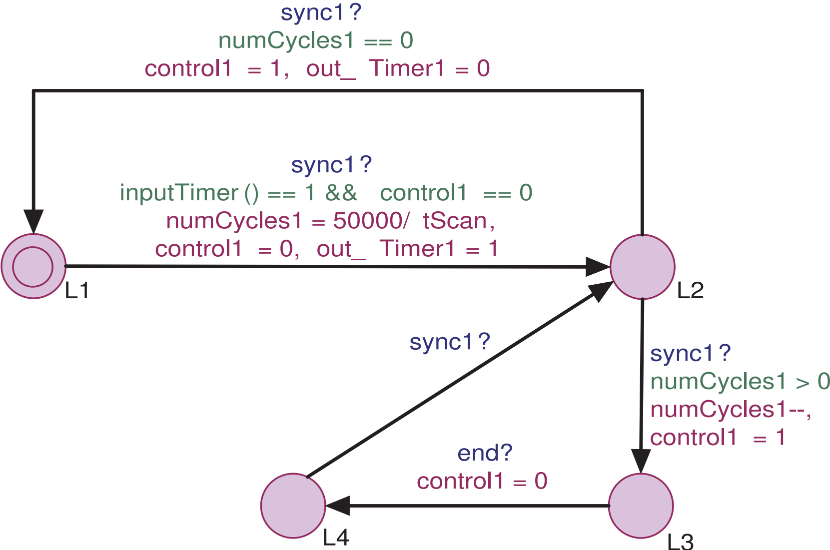


Figura 5.26: Autômato que representa o temporizador Timer1 do sistema que controla semáforos

Na Figura 5.27 o autômato utilizado para representar o comportamento do temporizador TP denominado de *Timer2* é ilustrado. Como ele é o segundo temporizador encontrado na seqüência da execução do programa, então o sufixo *NumTimer* para este temporizador irá possuir valor *2*. O valor de *PT (Present Time)* corresponde a 40000 microssegundos logo, a expressão *numCyclesNumTimer = PT/tScan* será transformada em *numCycles2 = 40000/tScan*. A função *inputTimer()* será declarada da seguinte forma: *bool inputTimer() {return (TMR1 and !TMR3) }.*

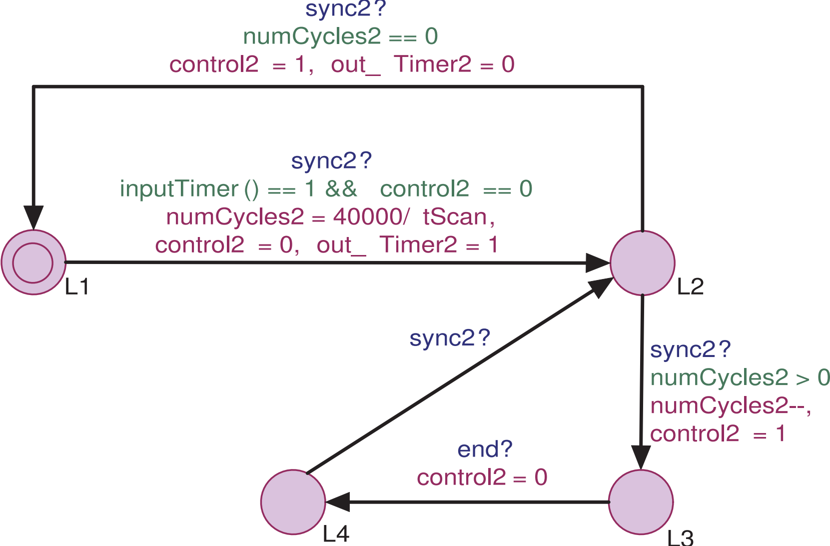


Figura 5.27: Autômato que representa o temporizador Timer2 do sistema que controla semáforos

Na Figura 5.28 o autômato utilizado para representar o comportamento do temporizador TP denominado de *Timer3* é ilustrado. O valor de *PT (Present Time)* corresponde a 10000 microssegundos logo, a expressão *numCyclesNumTimer = PT/tScan* será transformada em *numCycles3 = 10000/tScan*. A função *inputTimer()* será declarada da seguinte forma: *bool inputTimer() {return (TMR1 and !TMR2) }*.

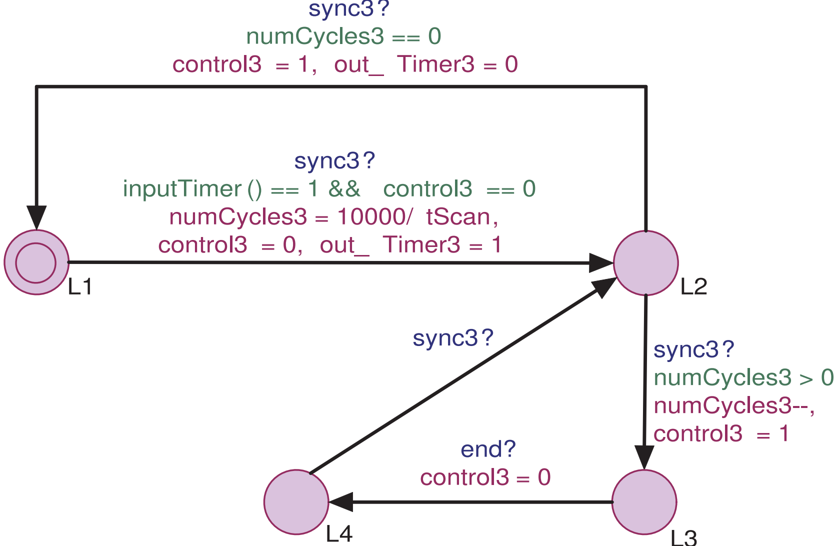


Figura 5.28: Autômato que representa o temporizador Timer3 do sistema que controla semáforos

Na Figura 5.29 o autômato utilizado para representar o comportamento do temporizador TP denominado de *Timer4* é ilustrado. O valor de *PT (Present Time)* corresponde a 30000 microssegundos logo, a expressão *numCyclesNumTimer = PT/tScan* será transformada em *numCycles4 = 30000/tScan*. A função *inputTimer()* será declarada da seguinte forma: *bool inputTimer() {return (!TMR1) }*.

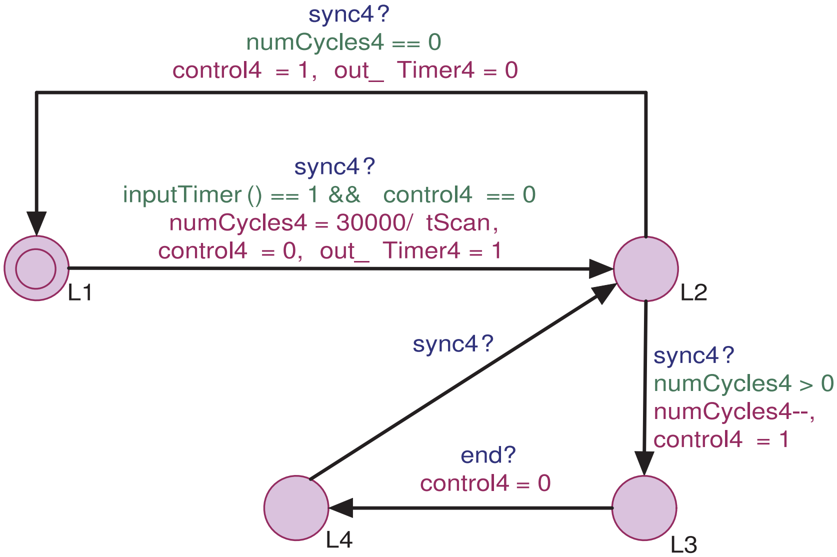


Figura 5.29: Autômato que representa o temporizador Timer4 do sistema que controla semáforos

Na Figura 5.30 o autômato utilizado para representar o comportamento do temporizador TP denominado de *Timer5* é ilustrado. O valor de *PT (Present Time)* corresponde a 20000 microssegundos logo, a expressão *numCyclesNumTimer = PT/tScan* será transformada em *numCycles5 = 20000/tScan*. A função *inputTimer()* será declarada da seguinte forma: *bool inputTimer() {return (TMR4 and !TMR6)}*.

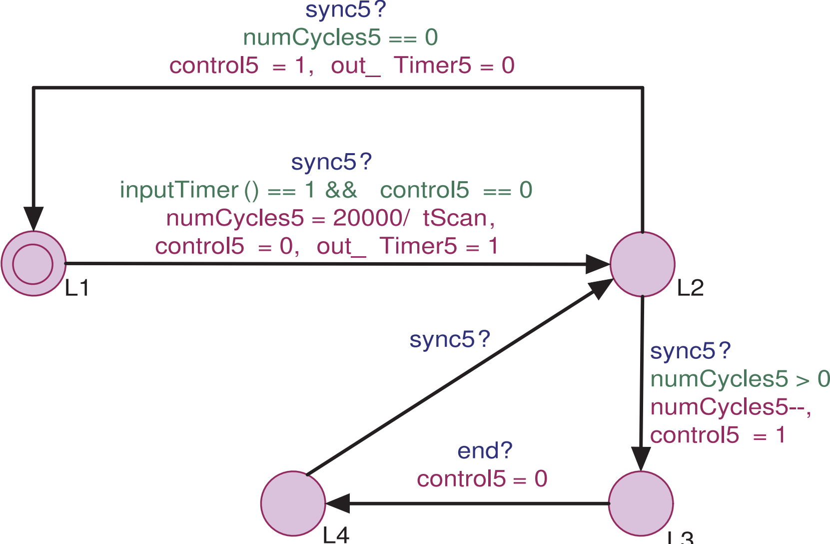


Figura 5.30: Autômato que representa o temporizador Timer5 do sistema que controla semáforos

Na Figura 5.31 o autômato utilizado para representar o comportamento do temporizador TP denominado de *Timer6* é ilustrado. O valor de *PT (Present Time)* corresponde a 10000 microssegundos logo, a expressão *numCyclesNumTimer = PT/tScan* será transformada em *numCycles6 = 10000/tScan*. A função *inputTimer()* será declarada da seguinte forma: *bool inputTimer() {return (TMR4 and !TMR5) }*.

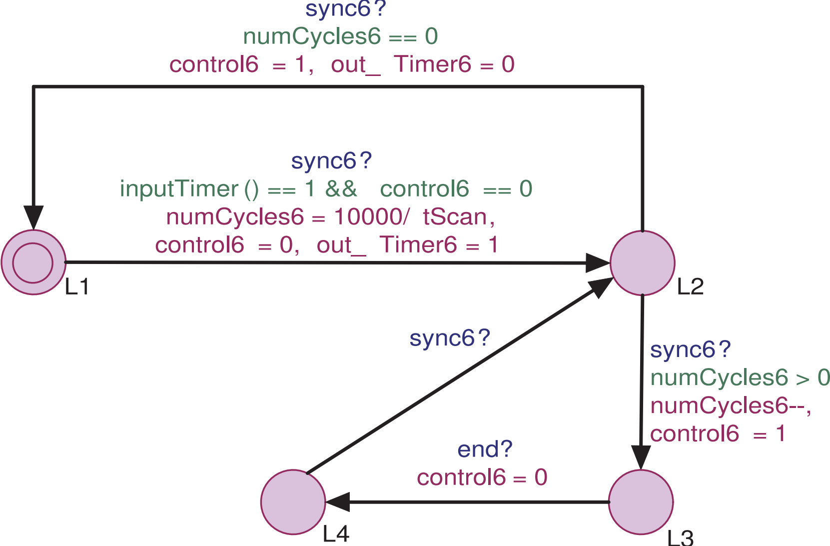


Figura 5.31: Autômato que representa o temporizador Timer6 do sistema que controla semáforos

Na Figura 5.32 é apresentada a modelagem da execução da lógica do programa como um autômato temporizado. A execução dos degraus ocorre seqüencialmente através do disparo de transições entre localidades distintas que possuam a mensagem *execute?*. As funções utilizadas para modelar tal autômato são as seguintes:

- Como o degrau que contém a bobina *TMR1* possui 1 elemento temporizado do tipo TP, então a execução de três funções rotuladas por *checkExecutionTimer1()*, *evaluateOuputTimer1()* e *outTimer1()* determinam o valor da lógica de controle do degrau representado por tal bobina. Estas funções são ilustradas a seguir:
  - void outTimer1(){

    TMR1 = (out_Timer1);

    outputs[0] = TMR1;}
  - bool checkExecutionTimer1(){

    return ((!TMR4 or TMR6) == 1 && control1 == 0) ||

    (numCycles1 == 0 && out_Timer1 == 1 && control1 == 0);}
  - bool evaluateOuputTimer1(){

    return ((control1 == 1) || ((!TMR4 or TMR6) == 0 &&

    control1 == 0 && out_Timer1 == 0);}
- Como o degrau que contém a bobina *TMR2* possui 1 elemento temporizado do tipo TP, então a execução de três funções rotuladas por *checkExecutionTimer2()*, *evaluateOuputTimer2()* e *outTimer2()* determinam o valor da lógica de controle do degrau representado por tal bobina. Estas funções são ilustradas a seguir:
  - void outTimer2(){

    TMR2 = (out_Timer2);

    outputs[1] = TMR2;}
  - bool checkExecutionTimer2(){

    return ((TMR1 and !TMR3) == 1 && control2 == 0) ||

    (numCycles2 == 0 && out_Timer2 == 1 && control2 == 0);}

- bool evaluateOuputTimer2(){

  return ((control2 == 1) || ((TMR1 and !TMR3) == 0 &&

  control2 == 0 && out_Timer2 == 0);}

- Como o degrau que contém a bobina *TMR3* possui 1 elemento temporizado do tipo TP, então a execução de três funções rotuladas por *checkExecutionTimer3()*, *evaluateOuputTimer3()* e *outTimer3()* determinam o valor da lógica de controle do degrau representado por tal bobina. Estas funções são ilustradas a seguir:

  - void outTimer3(){

    TMR3 = (out_Timer3);

    outputs[2] = TMR3;}

  - bool checkExecutionTimer3(){

    return ((TMR1 and !TMR2) == 1 && control3 == 0) ||

    (numCycles3 == 0 && out_Timer3 == 1 && control3 == 0);}

  - bool evaluateOuputTimer3(){

    return ((control3 == 1) || ((TMR1 and !TMR2) == 0 &&

    control3 == 0 && out_Timer3 == 0);}

- Como o degrau que contém a bobina *TRM4* possui 1 elemento temporizado do tipo TP, então a execução de três funções rotuladas por *checkExecutionTimer4()*, *evaluateOuputTimer4()* e *outTimer4()* determinam o valor da lógica de controle do degrau representado por tal bobina. Estas funções são ilustradas a seguir:

  - void outTimer4(){

    TMR4 = (out_Timer4);

    outputs[3] = TMR4;}

    return ((!TMR1) == 1 && control4 == 0) ||

    (numCycles4 == 0 && out_Timer4 == 1 && control4 == 0);}

  - bool evaluateOuputTimer4(){

    return ((control4 == 1) || ((!TMR1) == 0 &&

    control4 == 0 && out_Timer4 == 0);}

- Como o degrau que contém a bobina *TMR5* possui 1 elemento temporizado do tipo TP, então a execução de três funções rotuladas por *checkExecutionTimer5()*, *evaluateOuputTimer5()* e *outTimer5()* determinam o valor da lógica de controle do degrau representado por tal bobina. Estas funções são ilustradas a seguir:

  – void outTimer5(){
      TMR5 = (out_Timer5);
      outputs[4] = TMR5;}

  – bool checkExecutionTimer5(){
      return ((TMR4 and !TMR6) == 1 && control5 == 0) ||
      (numCycles5 == 0 && out_Timer5 == 1 && control5 == 0);}

  – bool evaluateOuputTimer5(){
      return ((control5 == 1) || ((TMR4 and !TMR6) == 0 &&
      control5 == 0 && out_Timer5 == 0);}

- Como o degrau que contém a bobina *TMR6* possui 1 elemento temporizado do tipo TP, então a execução de três funções rotuladas por *checkExecutionTimer6()*, *evaluateOuputTimer6()* e *outTimer6()* determinam o valor da lógica de controle do degrau representado por tal bobina. Estas funções são ilustradas a seguir:

  – void outTimer6(){
      TMR6 = (out_Timer6);
      outputs[5] = TMR6;}

  – bool checkExecutionTimer6(){
      return ((TMR4 and !TMR5) == 1 && control6 == 0) ||
      (numCycles6 == 0 && out_Timer6 == 1 && control6 == 0);}

  – bool evaluateOuputTimer6(){
      return ((control6 == 1) || ((TMR4 and !TMR5) == 0 &&
      control6 == 0 && out_Timer6 == 0);}

- Como os degraus que contém as bobinas *VmR1*, *VdR1*, *AmR1*, *VmR2*, *VdR2* e *AmR2* não possuem elementos temporizados, então os valores das lógicas de controle dos degraus são determinados da seguinte forma:

```
void value_VmR1(){
    out_VmR1 = (TMR1);
     outputs[6] = out_VmR1;}
void value_VdR2(){
    out_VdR2 = (TMR2);
     outputs[7] = out_VdR2;}
void value_AmR2(){
    out_AmR2 = (TMR3);
     outputs[8] = out_AmR2;}
void value_VmR2(){
    out_VmR2 = (TMR4);
     outputs[9] = out_VmR2;}
void value_VdR1(){
    out_VdR1 = (TMR5);
     outputs[10] = out_VdR1;}
void value_AmR1(){
    out_AmR1 = (TMR6);
     outputs[11] = out_AmR1;}
```

- A função *setControlVariables()* tem a finalidade de restaurar o valor de todas as variáveis *controlNumTimer* para *0*, no caso, control1, control2, control3, control4, control5 e control6, de atualizar o valor de *numRungsProcessed* para *numOutputs*, informando que todos os degraus foram processados e, de atualizar o valor de *startExecution* para *0*. Esta função é ilustrada a seguir:

```
void setControlVariables(){
    control1 = control2 = control3 = control4 = control5 = control6 = 0;
    numRungsProcessed = numOutputs;
    startExecution = 0; }
```

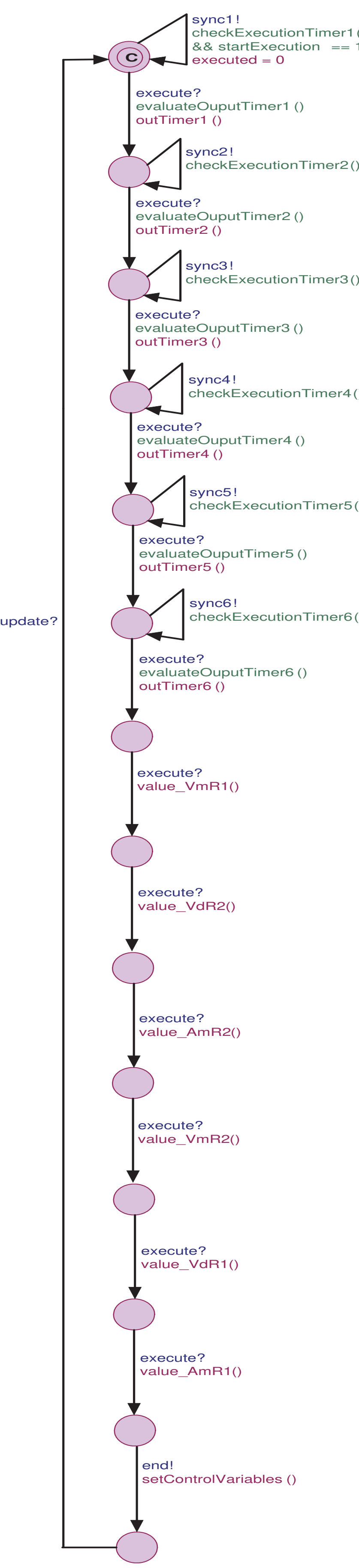


Figura 5.32: Autômato que representam a execução do programa do sistema que controla semáforos

Na Figura 5.33 é apresentado o autômato que avalia os estados das saídas (saída acionada ou não acionada). Na Figura 5.34 é apresentado o autômato que processa os estados das saídas.

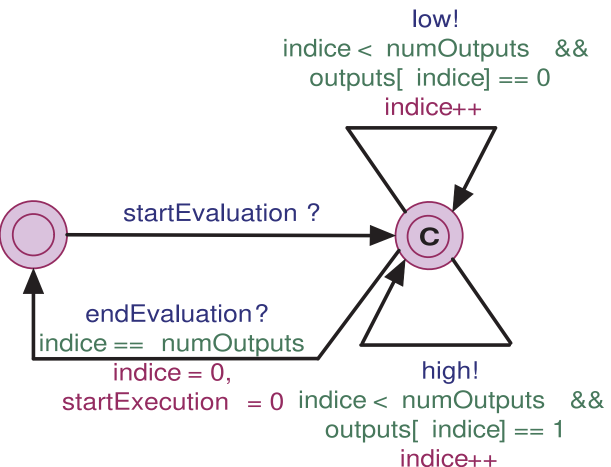


Figura 5.33: Modelagem do processo que avalia os estados das saídas do sistema que controla semáforos

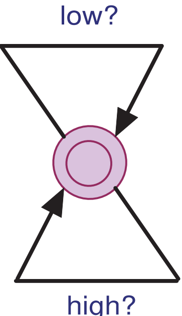


Figura 5.34: Modelagem do processamento dos estados das saídas do sistema que controla semáforos

Na Figura 5.35 é ilustrado o ciclo de varredura de um CLP. A função *checkEndScanCycle()*, utilizada na modelagem de tal autômato, avalia se todos os temporizadores não estão mais sendo executados, no caso, avalia se *control1* == *control2* == *control3* == *control4* == *control5* == *control6* == *0* e, se a execução do programa foi finalizada, *startExecution*

== 0. A função *updateOutputs()* libera as saídas do sistema que ainda não foram liberadas durante a execução do programa, no caso, os valores de *VmR1*, *VdR1*, *AmR1*, *VmR2*, *VdR2* e *AmR2*. Estas funções são ilustradas a seguir:

```
bool checkEndScanCycle(){
    return (control1 == control2 == control3 == control4 == control5 == control6 == 0)
        && startExecution == 0 ;}
    void updateOutputs(){
        VmR1 = out_VmR1;
        VdR1 = out_VdR1;
        AmR1 = out_AmR1;
        VmR2 = out_VmR2;
        VdR2 = out_VdR2;
        AmR2 = out_AmR2;}
```

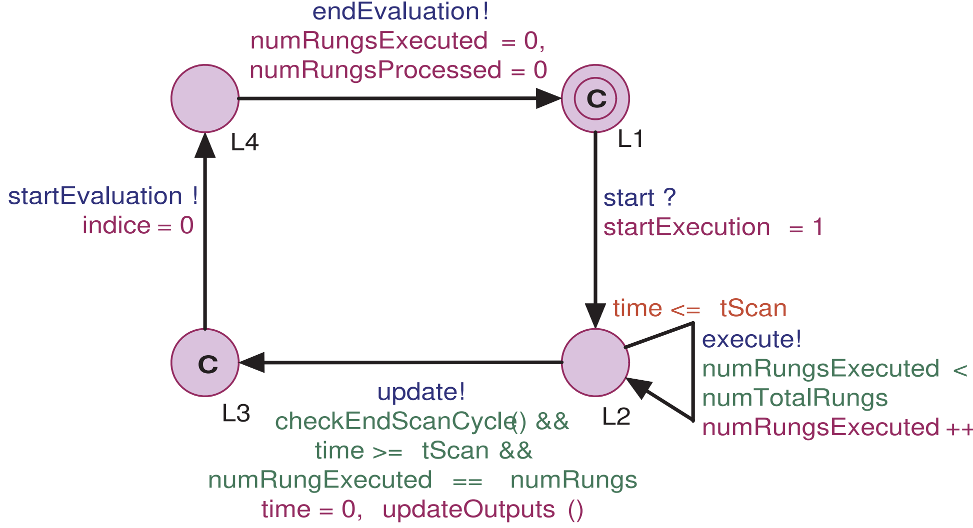


Figura 5.35: Autômato que representam ciclo de varredura do CLP para o sistema que controla semáforos

5.8 Testes realizados no sistema que controla dois semáforos

Para este estudo de caso, a configuração e o particionamento do modelo foram realizados da seguinte forma:

- Entradas: set(), reset(), start(), end(), sync1(), sync2(), sync3(), sync4(), sync5(), sync6(), low(), high().
- Saídas: execute(), update(), evaluation(), out_evaluation();
- Precisão: 1 unidade de tempo;
- Tempo de teste: 300 unidades de tempo.

O primeiro modelo de implementação a ser testado foi um modelo cuja representação é fiel a especificação, ou seja, a rede de autômatos que representa o programa Ladder ilustrado na Figura 5.22. O veredicto dado pela ferramenta Uppaal-TRON foi : "*TESTED PASSED: time out for testing*". Assim, durante o tempo programado para a execução dos testes, no caso 300 unidades de tempo, nenhuma anormalidade relacionada ao comportamento do modelo da implementação foi detectada. Portanto, a implementação é fiel à especificação.

Para ilustrar a validade do método, através da detecção de erros existentes na implementação, os seguintes erros foram inseridos no programa Ladder que representa o sistema que controla dois semáforos:

- Erro 1: Trocar a ordem de execução dos degraus, ou seja, o segundo degrau é executado e depois o primeiro degrau é executado;
- Erro 2: Retirar um degrau, no caso o nono degrau;
- Erro 3: Alterar o valor de *PT* de *Timer 3* para 20 segundos.

5.8.1 Aplicação do processo de teste para ocorrência do erro 1: O segundo degrau é executado e depois o primeiro degrau é executado

Na Figura 5.36 é apresentada a inserção do primeiro tipo de erro no programa Ladder que modela o sistema que controla dois semáforos.

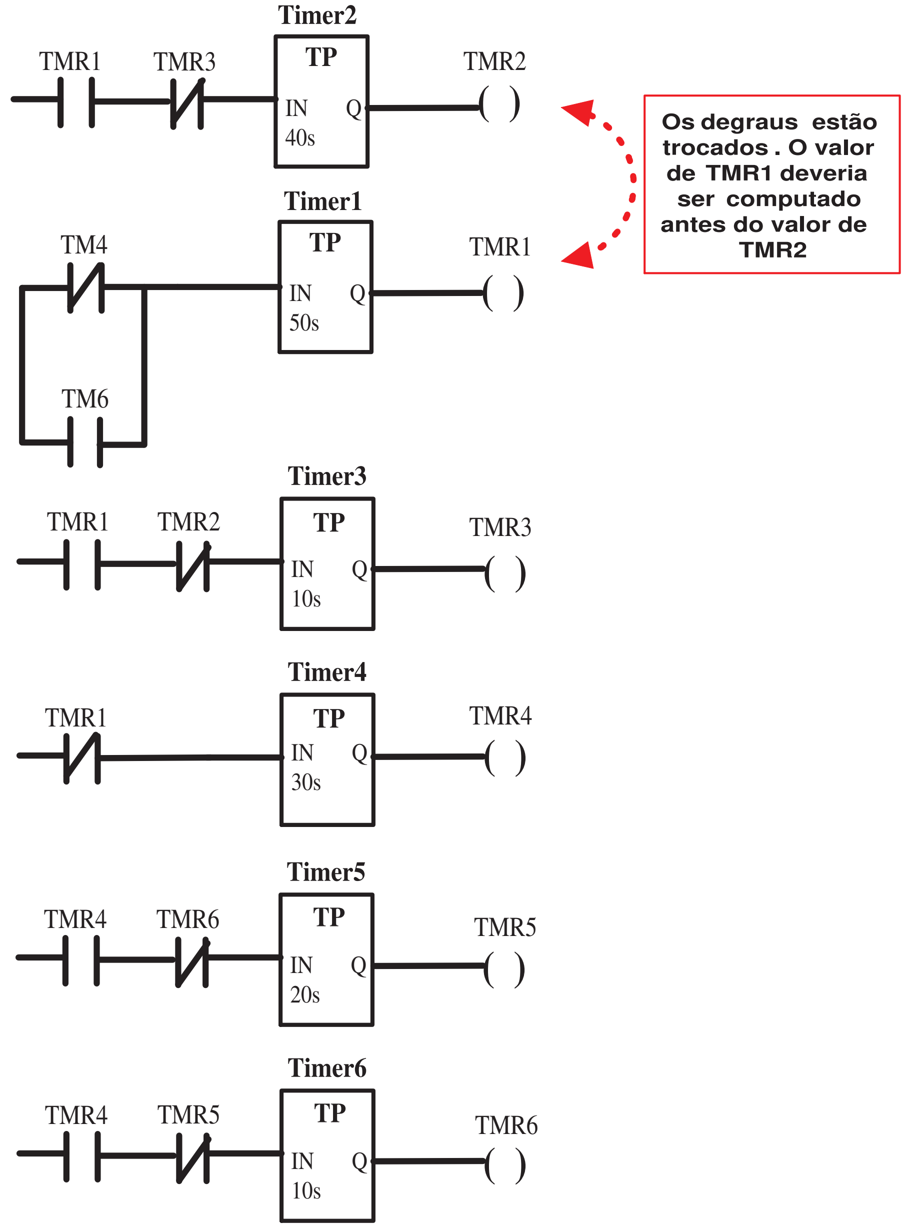


Figura 5.36: Inserção do erro 1 no programa Ladder para o sistema que controla dois semáforos

O veredicto dado pela ferramenta Uppaal-TRON foi : "*TESTED FAILED*". O traço de execução indicou oito erros:

- O primeiro indicou que uma saída *sync1()* foi esperada mas uma saída *execute()* foi encontrada no instante de tempo 2. Em outras palavras, o temporizador *Timer 2* deveria ter executado mas, não executou.
- O segundo indicou que uma saída *execute()* foi esperada mas uma saída *sync2()* foi encontrada no instante de tempo 3. Em outras palavras, o temporizador *Timer 1* já deveria ter começado sua execução, mas só neste instante de tempo é que ele foi executado.
- O terceiro indicou que uma saída *sync2()* foi esperada mas uma saída *execute()* foi encontrada no instante de tempo 4. Em outras palavras, o temporizador *Timer 2* deveria ter executado, mas não executou.
- O quarto indicou que uma saída *execute()* foi esperada mas uma saída *sync3()* foi encontrada no instante de tempo 5. Em outras palavras, o temporizador *Timer 3* não deveria ter executado, mas executou.
- O quinto indicou que uma saída *high()* foi esperada mas uma saída *low()* foi encontrada no instante de tempo 17. O valor da bobina *TMR2* deveria ser *1*, mas uma valor *0* foi encontrado;
- O sexto indicou que uma saída *low()* foi esperada mas uma saída *high()* foi encontrada no instante de tempo 17. O valor da bobina *TMR3* deveria ser *0*, mas uma valor *1* foi encontrado;
- O sétimo indicou que uma saída *high()* foi esperada mas uma saída *low()* foi encontrada no instante de tempo 17. O sinal do semáforo da rua 2 deveria ficar no estado verde, mas não ficou.
- O oitavo indicou que uma saída *low()* foi esperada mas uma saída *high()* foi encontrada no instante de tempo 17. O sinal do semáforo da rua 2 não deveria ter ficado amarelo, mas ficou.

5.8.2 Aplicação do processo de teste para ocorrência do erro 2: Retirar o nono degrau

Na Figura 5.37 é apresentada a inserção do segundo tipo de erro no programa Ladder que modela o sistema que controla dois semáforos.

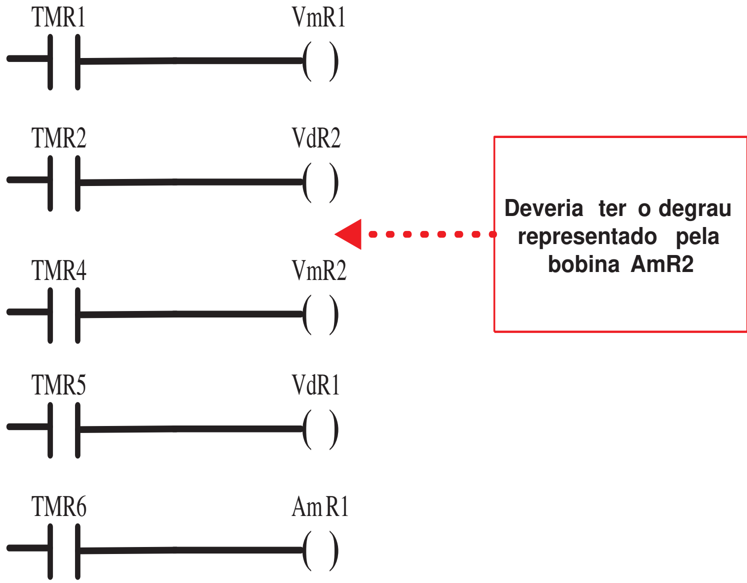


Figura 5.37: Inserção do erro 2 no programa Ladder para o sistema que controla dois semáforos

O veredicto dado pela ferramenta Uppaal-TRON foi : "*TESTED FAILED*". O traço de execução indicou que uma saída *execute()* foi esperada, mas uma saída *end()* foi encontrada no instante de tempo 15. Em outras palavras, a execução do último degrau representado pela bobina *AMR1* era esperada, mas o final da execução dos degraus foi executada. Isto aconteceu porque no modelo da implementação existem onze degraus e no modelo da especificação existem doze blocos que determinam saídas.

5.8.3 Aplicação do processo de teste para ocorrência do erro 3: Alterar o valor de *PT*, de *Timer 3*, para 20 segundos

Na Figura 5.38 é apresentada a inserção do terceiro tipo de erro no programa Ladder que modela o sistema que controla dois semáforos.

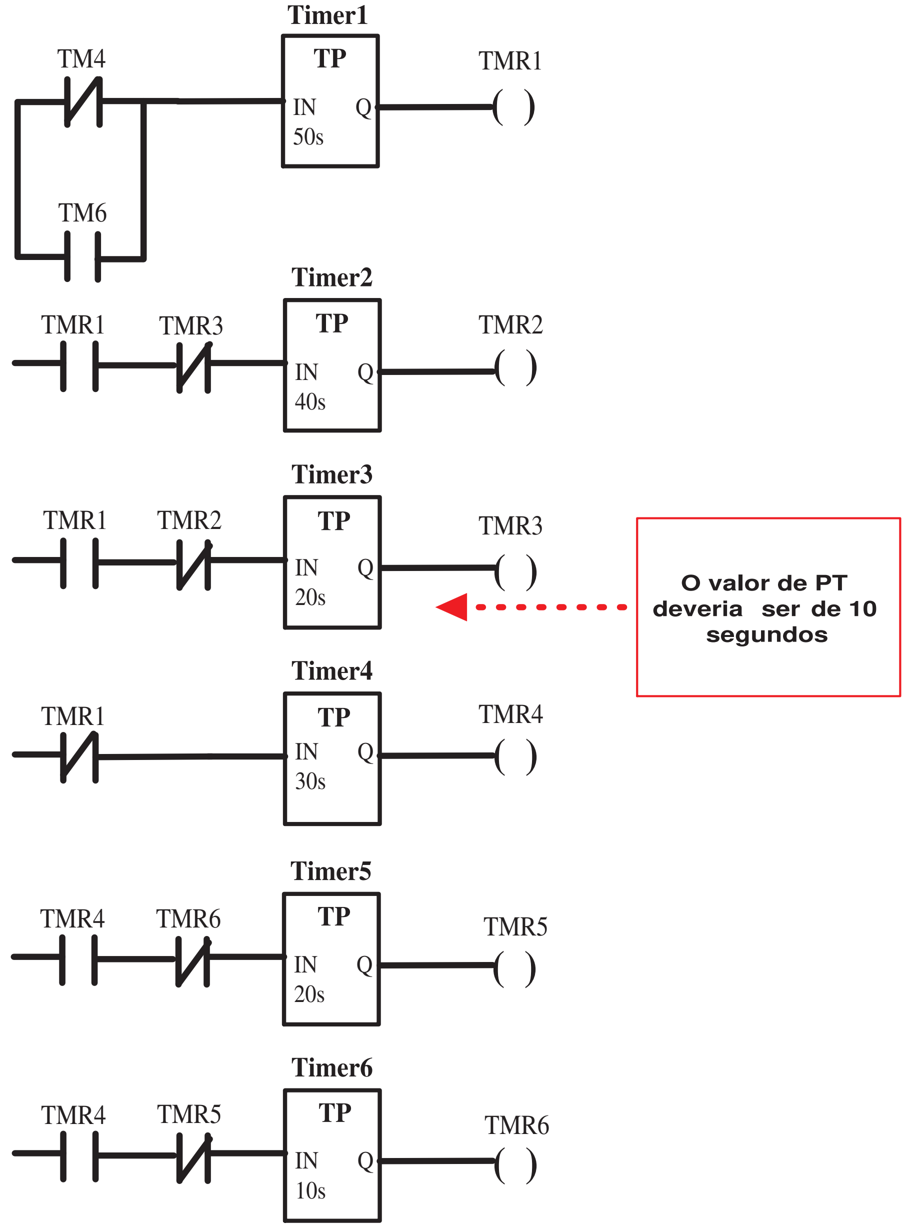


Figura 5.38: Inserção do erro 5 no programa Ladder para o sistema que controla dois semáforos

O veredicto dado pela ferramenta Uppaal-TRON foi : "*TESTED FAILED*". O traço de execução indicou três erros:

- O primeiro indicou que uma saída *sync3()* foi esperada mas uma saída *execute()* foi encontrada no instante de tempo 74. Em outras palavras, o temporizador *Timer 3* deveria ter executado, mas não executou. Este fato ocorreu porque o temporizador *Timer 1* terminou sua execução e liberou uma saída *0*.

- O segundo indicou que uma saída *execute()* foi esperada mas uma saída *sync4()* foi encontrada no instante de tempo 75. Em outras palavras, o temporizador Timer 4 não deveria ter executado, mas executou. Portanto, este fato fez com que o sinal do semáforo da rua 2 ficasse num estado amarelo e vermelho ao mesmo tempo. Este fato não pode ocorrer porque o o sinal do semáforo da rua 1 estava verde.

- O terceiro indicou que uma saída *execute()* foi esperada mas uma saída *sync1()* foi encontrada no instante de tempo 86. Em outras palavras, o temporizador *Timer 1* não deveria ter executado mas, executou. Portanto, este fato fez com que o sinal do semáforo da rua 1 ficasse num estado vermelho e verde ao mesmo tempo. Este fato não pode ocorrer porque o o sinal do semáforo da rua 2 estava verde.

Capítulo 6

Considerações Finais e Trabalhos Futuros

O trabalho aqui apresentado introduziu um método para aumentar a confiança e a segurança no funcionamento de CLPs. Para tanto, autômatos temporizados gerados a partir de Diagramas ISA 5.2 e Diagramas Ladder são confrontados com o objetivo de verificar se a implementação está em conformidade com a especificação. Esta verificação é realizada pela ferramenta de teste Uppaal-TRON para a geração e a execução de casos de teste de conformidade. Portanto, erros na implementação podem ser detectados e corrigidos antes da execução de testes de aceitação em campo.

O principal diferencial deste trabalho para os já conhecidos é que ele se propõe a aumentar a confiança e a segurança no funcionamento de programas para CLPs através da geração automática de casos de teste de conformidade. Além disso, para o método mostra-se como modelar elementos temporais, muito utilizado em sistemas reais, junto com a lógica completa do programa.

O método possibilita a aplicação do processo de teste de forma paralela ao processo de desenvolvimento minimizando o custo adicional para produção de artefatos específicos para teste. Além disso, a complexidade de construção dos modelos de autômatos temporizados gerados é ocultada, uma vez que os desenvolvedores necessitam apenas fornecer os arquivos com a descrição da especificação e da implementação para obter o veredicto dos testes.

As principais dificuldades encontradas foram relacionadas à ferramenta de teste Uppaal-TRON, pois os modelos utilizados no processo de teste devem estar bem particionados. Além disso, discrepâncias entre *timestamps*, mesmo para modelos de implementação fiéis a modelos de especificação, geram um veredicto que indica que os testes falharam. A exibição destes veredictos não é de fácil interpretação para o usuário. Outra dificuldade encontrada está relacionada à geração aleatória de casos de testes. Esta característica faz com que muitas vezes um traço de execução não seja suficiente para encontrar erros existentes na implementação.

6.1 Trabalhos Futuros

Com a finalização do trabalho, tivemos a possibilidade de fazer uma análise do mesmo, a qual resultou no seguinte conjunto de propostas para a sua continuidade:

- Modelar blocos de funções e instruções que lidem com memória, pois estas são muito comum em programas Ladder;
- Aperfeiçoar o método para que ele passe a trabalhar com programação multitarefa. Uma vez que o método ilustrado neste trabalho lida apenas com a execução de um único programa;
- Aperfeiçoar o método para que mais de um CLP possa controlar o mesmo processo. Uma vez que o método ilustrado neste trabalho lida apenas com modelagem de um CLP;
- Submeter o trabalho a outros estudos de caso, tendo como objetivo obter uma maior confiança com relação ao procedimento proposto;
- Gerar um novo verificador que gere e execute casos de teste de conformidade, uma vez que o TRON apresenta limitações.

References

Alur, Rajeev. 1999. Timed Automata. *Theoretical Computer Science*, **126**, 183–235.

Alur, Rajeev, & Dill, David L. 1994. A Theory of Timed Automata. *Theoretical Computer Science*, **126**, 183–235.

Bauer, Nanette, & Huuck, Ralf. 2001. Towards Automatic Verification of Embedded Control Software. *In: Asian Pacific Conference on Quality Software, ser. IEEE.*

Bender, Darlam Fabio, Combemale, Benoît, Crégut, Xavier, Farines, Jean Marie, Berthomieu, Bernard, & Vernadat, François. 2008. Ladder Metamodeling and PLC Program Validation Through Time Petri Nets. *Pages 121–136 of: Ecmda-fa '08: Proceedings of the 4th European Conference on Model Driven Architecture*. Berlin, Heidelberg: Springer-Verlag.

Bengtsson, Johan, & Yi, Wang. 2004. Timed Automata: Semantics, Algorithms and Tools. *Lectures on Concurrency and Petri Nets*, **1633**(April), 8–22.

Bryan, Luis A., & Bryan, Eric A. 1997. *Programmable Controllers: Theory and Implementation*. Industrial Text Company. Illustrator-Kory, Gina.

Canet, Géraud, Couffin, Sandrine, Lesage, Jean-Jacques, Petit, Antoine, & Schnoebelen, Philippe. 2000. Towards the Automatic Verification of PLC Programs Written in Instruction List. *Pages 2449–2454 of: Proceedings of the IEEE International Conference on Systems, Man and Cybernetics (SMC 2000)*. Nashville, Tennessee, USA: Argos Press.

Capelli, Alexandre. 2006. *Automação Industial: Controle do Movimento e Processos Contínuos*. Érica.

Cooling, Jim. 2003. *Software Engineering for Real-Time Systems*. Addison Wesley.

da Silva, Leandro Dias, de Assis Barbosa, Luiz Paulo, Gorgônio, Kyller, Perkusich, Angelo, & Lima, Antônio Marcus Nogueira. 2008. On the Automatic Generation of Timed Automata Models from Function Block Diagrams for Safety Instrumented Systems. *Pages 291–296 of: 34th Annual Conference of the IEEE Industrial Electronics Society (IECON 2008)*. Orlando, USA: IEEE Industrial Electronics Society.

de Assis Barbosa, Luiz Paulo, Gorgonio, Kyller, da Silva, Leandro Dias, Lima, Antonio Marcus Nogueira, & Perkusich, Angelo. 2007. On the Automatic Generation of Timed Automata Models from ISA 5.2 Diagrams. *Emerging Technologies and Factory Automation, 2007. ETFA*, Sept., 406–412.

Frey, George, & Litz, Leothar. 2000. Formal Methods in PLC Programming. *Pages 2431–2436 vol.4 of: IEEE International Conference on Systems, Man, and Cybernetics*, vol. 4.

Gerd Behrmann, Alexandre David, & Larsen, Kim Guldstrand. 2004. A Tutorial on UPPAAL. *Pages 200–237 of: LNCS, Formal Methods for the Design of Real-Time Systems*, vol. 3185. Springer Verlag.

Goble, William M., & Cheddie, Harry. 2005. *Safety Instrumented Systems Verification: Practical Probabilistic Calculations*. ISA.

Gourcuff, Vincent, Smet, Olivier De, & Faure, Jean-Marc. 2006 (July). Efficient Representation for Formal Verification of PLC Programs. *Pages 182–187 of: 8th International Workshop on Discrete Event Systems*.

Gruhn, Paul, & Cheddie, Harry L. 2006. *Safety Instrumented Systems: Design, Analysis, and Justification*. 2nd edn. ISA.

Heiner, Monika, & Menzel, Thomas. 1998 (Aug.). A Petri Net Semantics for the PLC Language Instruction List. *Pages 161–165 of: Procedures of IEEE Workshop on Discrete Event Systems*.

ISA. 1992 (July). *Binary Logic Diagrams for Process Operations*. ISA 5.2-1976 (R1992) edn. ISA - The Instrumentation, Systems, and Automation Society.

John, Karl-Heinz, & Tiegelkamp, Michael. 2001. *IEC 61131-03: Programming Industrial Automation Systems*. Berlin, Germany: Springer-Verlag.

Katoen, Jooster-Pieter. 1999. *Concepts, Algoritms, and Tools for Model Checking*. Lecture Notes of the Course "Mechanised Validation of Parallel Systems".

Larsen, Kim G., Mikucionis, Marius, Nielsen, Brian, & Skou, Arne. 2005. Testing Real-Time Embedded Software Using UPPAAL-TRON: an Industrial Case Study. *Pages 299–306 of: EMSOFT '05: Proceedings of the 5th ACM International Conference on Embedded Software*. New York, NY, USA: ACM.

Larsen, Kim G., Mikucionis, Marius, & Nielsen, Brian. 2007. *Uppaal Tron User Manual. In http://www.cs.aau.dk/ marius/tron/*.

Mader, Angelika, & Wupper, Hanno. 1999 (June). Timed Automaton Models for Simple Programmable Logic Controllers. *In: Proceedings of Euromicro Conference on Real-Time Systems*.

Moon, Il. 1994. Modeling Programmable Logic Controllers for Logic Verification. *IEEE Control Systems Magazine*, **14**(2), 53–59.

Murata, Tadao. 1989. Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, **77**(4), 541–580.

Nema - National Electrical Manufacturers Association. 2005. *Programmable Controllers (PLC), part 1: General Information. In http://www.nema.org*.

Neves, Cleonor, Duarte, Leonardo, Viana, Nairon, & Ferreira, Vicente. 2007. Os Dez Maiores Desafios da Automação Industrial: as Perspectivas para o Futuro. *In: II Congresso de Pesquisa e Inovação da Rede Norte Nordeste de Educação Tecnológica. João Pessoa, Paraíba, Brasil*.

Olderog, Ernst-Rüdiger, & Dierks, Henning. 2008. *Real-Time Systems: Formal Specification and Automatic Verification*. Cambridge.

Parr, Andrew. 2003. *Programmable Controllers an Engineer´s Guide*. 3nd edn. Newnes.

PLCopen. 2004. *IEC 61131-3: a Standard Programming Resource. In http://www.plcopen.org/*. PLCopen For Efficiency in Automation.

Rausch, M., & Krogh, B.H. 1998. Formal Verification of PLC Programs. *Pages 234–238 of: In Procedures of American Control Conference*.

Remelhe, M. P., Lohmann, S., Stursberg, O., Engell, S., & Bauer, N. 2004 (Setembro). Algorithmic Verification of Logic Controllers Given as Sequential Function Charts. *Pages 53–58 of: IEEE International Symposium on Computer Aided Control Systems Design*.

Rossi, Olivier, & Schnoebelen, Philippe. 2000. Formal Modeling of Timed Function Blocks for the Automatic Verification of Ladder Diagram Programs. *Pages 177–182 of:* Engell, Sebastian, Kowalewski, Stefan, & Zaytoon, Janan (eds), *Proceedings of the 4th International Conference on Automation of Mixed Processes: Hybrid Dynamic Systems (ADPM 2000)*. Dortmund, Germany: Shaker Verlag.

Smet, O. De, & Rossi, O. 2002. Verification of a Controller for a Flexible Manufacturing Line Written in Ladder Diagram Via Model-checking. *Pages 4147–4152 vol.5 of: American Control Conference, 2002. proceedings of the 2002*, vol. 5.

Smet, O. De, Couffin, S., Rossi, O., Canet, G., Lesage, J.J., Ph, & Papini, H. 2000. Safe Programming of PLC Using Formal Verification Methods. *Pages 73–78 of: 4th International PLCopen Conference on Industrial Control Programming (ICP'2000)*. PLCOpen, Zaltbommel, The Netherlands.

Wang, R., Song, X., & Gu, M. 2007. Modelling and Verification of Program Logic Controllers Using Timed Automata. *Software, IET*, **1**(4), 127–131.

Younis, M. Bani, & Frey, G. 2003. Formalization of Existing PLC Programs: A Survey. *Proceedings of the IEEE/IMACS Multiconfrence on Computational Engineering in Systems Applications (cesa 2003), Lille, France*, July.

Younis, M.B., & Frey, G. 2004 (June-2 July). Visualization of PLC Programs using XML. *Pages 3082–3087 vol.4 of: Proceedings of the American Control Conference*, vol. 4.

Zoubek, Bohumir. 2002. Automatic Verification of Programs for Control Systems.

Zoubek, Bohumir, Roussel, Jean-Marc, & Kwiatowska, Marta. 2003. Towards Automatic Verification of Ladder Logic Programs. *IMACS Multiconference on Computational Engineering in Systems Applications (CESA)*.