Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Coordenação de Pós-Graduação em Ciência da Computação

Validação Visual de Programas Ladder Baseada em Modelos

Leonardo Rodrigues Sampaio

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande - Campus I 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
(Orientadores)

Campina Grande, Paraíba, Brasil

Resumo

Sistemas de controle são freqüentemente utilizados na indústria na realização de tarefas críticas. Falhas nestas operações muitas vezes acarretam em perdas e podem colocar a segurança da planta em risco. Portanto, garantir a corretude de programas implementados em sistemas de controle é necessário.

Técnicas de verificação e validação desenvolvidas pela academia muitas vezes primam por aspectos técnicos mas possuem pouca aceitação na indústria devido ao necessário domínio de conceitos avançados. A academia procura aplicar métodos formais na criação de ferramentas neste contexto, permitindo a análise e adaptação genérica. No entanto, o nível de conhecimento sobre conceitos como modelagem e abstração impedem uma vasta adoção das mesmas.

O objetivo deste trabalho é a introdução de um método amigável de verificação visual de conformidade, ocultando detalhes de implementação dos usuários durante a fase de teste de programas Ladder implantados em sistemas de controle e permitindo sua operação sem a necessidade de treinamento específico em métodos formais.

Abstract

Control systems are frequently used in industry in the execution of critical tasks. Flaws in this operations may cause expensive losses and can put the plant safety at risk. Therefore, it is necessary to define techniques, methods and tools to increase the dependability of control.

Verification and validation techniques developed by academy often excel in technical aspects but have low acceptance in industry due to the necessary knowledge of advanced concepts. Academy applies formal methods in tools in this context, allowing analysis and generic adaptation. However, the knowledge level about concepts such as modeling and abstraction prevents a wide adoption of such tools.

The goal of this work is to introduce a user-friendly visual conformance verification method, to hide implementation details from users during the test phase of Ladder programs without the need of specific training in formal methods.

Agradecimentos

Primeiramente a Deus, sem ele, nada seria possível. A minha família pelo amor e apoio incondicionais. A minha namorada, Francisca Carvalho, que mesmo quando longe, sempre esteve me apoiando e incentivando.

Aos antigos companheiros de Sucesso e atuais de Remanso, pelo suporte oferecido tanto na partida quando na chegada.

As amizades que fiz em Maceió, João Pessoa, Natal, Recife, Patos e em especial Campina Grande durante esses dois anos, muito obrigado, anfitriões melhores não poderia ter conhecido.

Aos amigos do Embedded dentre muitos outros que ficam aqui supracitados, maravilhosas pessoas com as quais tive o prazer de conviver, meu muito obrigado pela força, amizade e companheirismo.

Aos professores do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, que proporcionaram discussões de alto nível que em muito contribuíram ao meu senso crítico científico. Em particular aos meus orientadores, Angelo Perkusich e Leandro Dias, pelo norte dado à pesquisa e a este trabalho.

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

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Computadores têm se tornado presentes em diversas áreas da produção do conhecimento humano e são utilizados cada vez mais para prover produtos e serviços de alta qualidade e confiabilidade, controlando e otimizando processos produtivos. O avanço tecnológico constante na indústria deu origem à chamada automação industrial, que teve início ainda na década de 20, com a criação das linhas de montagem automobilísticas, e se desenvolveu a passos largos a partir da década de 60 com o avanço da microeletrônica.

A implantação do processo de automação geralmente ocorre com a utilização dos chamados sistemas de controle, que são grupos de dispositivos responsáveis pela regulagem, manutenção e comando de outros dispositivos ou sistemas, estando estes em constante interação com o ambiente físico ou planta através de sensores e atuadores.

Controladores Lógico Programáveis (CLP) são sistemas de controle industriais baseados em sensores e atuadores e controlados por um programa do usuário, geralmente escrito em uma das cinco linguagens definidas no padrão IEC 61131-3 [22]. Estes sistemas foram projetados inicialmente com o objetivo de substituir os sistemas de controle baseados em relés, e tiveram como principais requisitos [5]:

- Serem competitivos em termos de custo - só haveria inicialmente uma vantagem no uso de tais sistemas se o custo/benefício de implantação (em relação aos sistemas até então utilizados) o justificasse;

- Robustos o suficiente para operarem em ambientes hostis - sistemas computacionais como PCs ou semelhantes não atenderiam facilmente a tal requisito, por isso a neces-

sidade de uma arquitetura própria;

- Facilmente substituíveis - os sistemas deveriam ser desenvolvidos de uma forma modular, a fim de facilitar procedimentos em caso de falha ou manutenção;

- Reusáveis - mudanças de requisitos e/ou aplicações deveriam ser refletidas facilmente nos sistemas;

- O método de programação a ser utilizado deveria ser simples o suficiente para ser entendido pelo pessoal da planta de fábrica.

Estes dispositivos são utilizados em diversos setores na indústria química, de manufatura, metalúrgica, de beneficiamento de alimentos, dentre outras. Muitas vezes também atuam no monitoramento de outros dispositivos de forma a assegurar sua correta operação, como é o caso dos chamados Sistemas Instrumentados de Segurança (SIS), que são responsáveis por manter a planta em um estado operacional seguro, atuando como dispositivos de contenção. A Petrobras é um exemplo de empresa que utiliza os SIS em suas refinarias, implantando os mesmos através de CLPs.

CLPs são largamente utilizados na indústria devido a sua flexibilidade, baixo custo, fácil programação e manutenção [23]. Muitas vezes, tarefas críticas, cujas perdas são dispendiosas, são controladas por esses dispositivos e necessitam de um alto nível de confiança no funcionamento, tolerância a falhas e operação contínua.

Para alcançar estes objetivos é necessário levar em conta tanto requisitos funcionais durante a fase de análise quanto questões inerentes ao processo de desenvolvimento de software, como a verificação de conformidade entre especificação e implementação e a etapa de teste. O processo de desenvolvimento típico de um sistema de controle implementado em um CLP é composto por uma etapa de especificação de requisitos, seguida pela implementação do programa em uma das linguagens de programação para CLPs do padrão IEC 61131-3, e posterior compilação e implantação no dispositivo.

Na Figura 1.1, ilustra-se o processo adotado na empresa Petrobrás, onde a etapa de especificação de requisitos gera como documentação uma tabela de causa/efeito e um diagrama ISA 5.2. A implementação é realizada na linguagem Ladder ou FBD, e por fim é realizada a implantação e teste da aplicação no CLP a ser utilizado em campo.

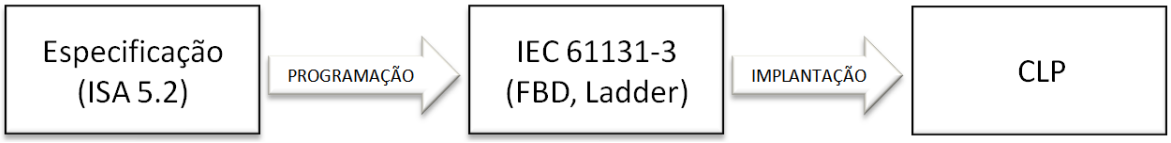


Figura 1.1: Processo de desenvolvimento.

Para realizar a verificação de conformidade de tais sistemas finais em relação a sua especificação são aplicadas técnicas conhecidas pela indústria, como por exemplo os chamados Testes de Aceitação de Fábrica (TAF). Um TAF consiste na reprodução no chão de fábrica das entradas esperadas durante a operação em um ambiente de produção do sistema, verificando-se as saídas obtidas e comparando-as ao comportamento esperado presente na especificação [8].

1.1 Descrição do problema

Métodos de teste como o TAF são ineficientes devido ao tempo necessário para a reprodução de requisitos como temporizações, por exemplo, e por operarem sobre o produto final. Muitas vezes correções *in loco* do programa, já executando na configuração final do sistema, são realizadas no chão de fábrica, e a verificação de conformidade do software modificado com os requisitos, quando realizada, fica a cargo apenas dos programadores e engenheiros envolvidos nesta etapa de implantação.

Sabe-se que quanto mais cedo for detectada uma falha no ciclo de desenvolvimento de software menor o custo de correção [21]. Portanto, retardar a etapa de teste para o final do processo irá incorrer em um custo adicional que pode ser evitado.

Soluções propostas na literatura relativas a métodos de validação e verificação de sistemas de controle geralmente empregam técnicas de testes, checagem de modelos, análise estática e *slicing* dinâmico de programas, dentre outras. Neste contexto, dois ramos de pesquisa se destacam: a definição de um modelo formal do programa e posterior geração automática de código e a verificação *a posteriori* do código, sem a alteração do fluxo de desenvolvimento clássico. Detalhes sobre os trabalhos analisados durante a pesquisa são discutidos no Capítulo 2.

Existem diversos trabalhos que abordam a problemática da verificação de conformidade

entre especificação e implementação, no entanto as técnicas propostas pela academia [21; 24; 18; 9; 25] muitas vezes exigem um conhecimento aprofundado do operador do sistema sobre técnicas de verificação e validação. Pessoal especializado deve operar as ferramentas de validação de forma que seja possível executar-se o método corretamente e se possa interpretar as saídas obtidas com sucesso. Para a correta utilização desses sistemas de validação se faz necessário o conhecimento sobre todo o processo: as linguagens e formalismos intermediários utilizados, como são especificadas entradas e saídas, que limitações são inerentes e que tipo de adaptações e abstrações devem ser feitas para que o mesmo seja efetivo.

Devido a esse impedimento em termos de treinamento de pessoal ainda existe uma resistência na adoção de tais técnicas. O que se espera na indústria é que programadores e engenheiros, profundos conhecedores dos tipos de diagramas com que lidam diariamente, não precisem aprender outros tipos de especificações e formalismos para resolver problemas que são originalmente inerentes apenas aos seus instrumentos de trabalho rotineiros.

1.2 Objetivos do Trabalho

O foco do trabalho consiste na coleta de informações de rastreabilidade durante a etapa de geração de uma rede de autômatos temporizados, que representa os diagramas de especificação e implementação para uma posterior interpretação a partir de um traço de execução sobre essa rede que descreve a execução de um caso de teste. Este processo permite a identificação de partes do modelo de especificação excitadas e por conseqüência que partes do modelo original Ladder são problemáticas, de forma visual e automática.

Neste trabalho propõe-se a adaptação de um método de geração e execução automatizada de testes de conformidade entre modelos de especificação (ISA 5.2) e implementação (Ladder) de sistemas de controle (implementados em CLPs) [19] de forma que uma eventual falha na implementação possa ser facilmente identificada a partir dos traços de execução gerados na etapa de teste. Pretende-se excluir do processo a necessidade de um especialista na interpretação das saídas geradas pelo método de teste.

Além dos modelos de implementação e especificação intermediários inicialmente gerados [19], será agora criado um terceiro modelo XML que faz a associação entre os elementos dos dois primeiros e os componentes textuais do diagrama Ladder. Este modelo adicional

permite a associação, por exemplo, entre uma determinada bobina (representada em uma rede de autômatos temporizados) e seu símbolo, nome e linha no arquivo Ladder. Caso ocorra um erro durante a execução dos testes automatizados é utilizado este arquivo para realizar a associação entre os eventos descritos no traço de execução do caso de teste aos elementos do programa Ladder, e posteriormente destacar visualmente tais elementos.

1.3 Relevância do trabalho

Neste trabalho é apresentado um método de validação automatizado e visual de programas Ladder, permitindo-se a geração e execução de casos de teste de conformidade e posterior avaliação dos resultados obtidos através dos traços gerados pelos casos de teste que indiquem erro, fornecendo-se como entrada ao método apenas os diagramas Ladder e ISA 5.2.

A principal contribuição do trabalho se dá pela criação de um método de teste automatizado e visual que provê uma ferramenta de baixa complexidade de operação. Um artigo que apresenta o método proposto neste trabalho foi aceito para apresentação na *IEEE International Conference on Industrial Technology* 2011 (ICIT 2011[1]) a se realizar em março do corrente ano. O trabalho também contribuiu para o avanço na pesquisa em verificação e validação de software para sistemas de controle realizada no Laboratório de Sistemas Embarcados e Computação Pervasiva (Embedded), pertencente ao Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande.

1.4 Estrutura da dissertação

A estrutura restante do trabalho está organizada da seguinte forma: no Capítulo 2 os trabalhos relacionados são discutidos; no Capítulo 3 é feita uma introdução sobre o funcionamento de um CLP, ao padrão IEC 61131-3, ao formalismo de autômatos temporizados, ao conceito de rastreabilidade entre modelo e código e as linguagens Ladder e ISA 5.2. No Capítulo 4 é detalhado o método proposto; no Capítulo 5 é feita uma introdução à modelagem sobre a qual se baseia o trabalho; no Capítulo 6 são demonstrados dois estudos de caso aplicando o método desenvolvido e por fim no Capítulo 7 são feitas as conclusões.

[1] http://icit2011.auburn.edu/

Capítulo 2

Trabalhos relacionados

Neste capítulo são discutidas soluções propostas na literatura relativas a métodos de validação e verificação de sistemas de controle escritos utilizando linguagens do padrão IEC 61131-3, além de outras contribuições relativas a abordagens de rastreabilidade entre modelo, código e testes de conformidade. Neste contexto, dois ramos de pesquisa se destacam: a definição de um modelo formal do programa e posterior geração automática de código [24; 18; 9; 25] e a verificação *a posteori* do código, sem a alteração do fluxo de desenvolvimento clássico, que é a abordagem aqui adotada. A seguir são apresentados alguns trabalhos que seguem esta última linha.

Em [6] são apresentados diversos aspectos sobre os quais a pesquisa acadêmica e na indústria diferem. Dentre estes aspectos destaca-se o grande interesse na indústria pelo desenvolvimento de aplicações para verificação e validação de programas para CLPs utilizando simulação e emulação, que não são o principal foco da academia devido a não utilização de métodos formais neste contexto.

A criação de ferramentas que não tratem as linguagens utilizadas com algum nível de formalismo tendem a ser específicas ao problema e não suportarem uma análise genérica. No entanto, trabalhos que unam os dois mundos podem trazer uma contribuição a ambas as vertentes de pesquisa.

Em [4] é proposto um método baseado em modelos para verificação de propriedades temporais extraídas automaticamente de diagramas Ladder. Os autores propõem-se a verificar propriedades genéricas sobre os diagramas (como ausência de concorrência) e apenas elementos simples da linguagem Ladder são levados em consideração no subconjunto da lin-

guagem definido no metamodelo (não são considerados elementos temporizados, por exemplo). A conversão entre sintaxe textual e metamodelo também não foi implementada, o que torna o método dependente em vários aspectos da intervenção manual do usuário.

Em [16] é proposto um método de programação para CLPs que incorpora simulação, permitindo a validação visual do comportamento dos programas a partir da sincronização com um modelo virtual 3D da planta de fábrica. O funcionamento do método depende da especificação de um modelo de planta baseado no formalismo *DEVS* e um mapeamento entre este modelo de simulação e o programa de controle em si. A proposta dos autores cria uma série de novas etapas ainda durante a fase de desenvolvimento, o que pode demandar uma extensa fase de adaptação de processos, adequando-se mais a um ambiente de prototipagem do que ao de verificação de software propriamente.

Em [17] é apresentado um estudo de caso da aplicação de um gerador de casos de teste automatizado (UPPAAL-Tron[1]) *online*, ou seja, capaz de gerar e executar testes evento por evento em tempo real. A execução da ferramenta após a modelagem do comportamento esperado do sistema e do ambiente e posterior criação de uma interface de comunicação com o sistema de controle mostrou diversas divergências entre especificação e implementação mesmo com um reduzido tempo de execução. A aplicação do método depende da modelagem do sistema na ferramenta UPPAAL[2], utilizando autômatos temporizados, e um conhecimento sobre a operação do testador, o que a desqualifica como amigável a programadores de sistemas para CLPs e demais agentes de negócio da indústria.

Em [7] é proposto um método que utiliza abordagem semelhante a [17], no entanto a etapa de geração dos modelos UPPAAL é feita automaticamente, e ao invés de se verificar propriedades sobre a rede de autômatos são realizados testes de conformidade, utilizando-se como gabarito um modelo gerado a partir de uma especificação do programa em ISA 5.2 e como modelo de implementação uma rede de autômatos no formato UPPAAL gerada automaticamente a partir de um programa FBD. Em [19], método semelhante é aplicado a programas descritos na linguagem Ladder.

Os processos de teste propostos em [17; 7; 19] necessitam de uma etapa final de interpretação da saída da ferramenta de teste utilizada caso seja expedido um veredicto de falha,

[1] http://www.cs.aau.dk/~marius/tron/

[2] http://www.uppaal.com/

etapa esta não realizada automaticamente e de responsabilidade de um especialista.

Em [11] é proposto um método de validação e depuração de programas descritos através de *statecharts* [12], utilizando-se a técnica de *slicing* dinâmico [1] do código gerado automaticamente a partir dos modelos. A rastreabilidade entre modelo e código é realizada a partir da introdução de *tags* no código Java gerado a fim de relacionar os trechos de código problemáticos identificados pela técnica de depuração aos modelos originais. O estudo de caso proposto demonstrou que nem todos os erros puderam ser identificados (15% deles) devido a limitações da técnica de depuração, no entanto a implementação de rastreabilidade entre código Java e modelos *statechart* mostrou-se capaz de relacionar corretamente os trechos de programa selecionados às seções do modelo.

Em [10] é proposto um método que proporciona eficiência na aplicação de checagem de modelos para verificação de programas para CLPs escritos em linguagem ST. Os modelos gerados para o *model checker* NuSMV[3] a partir do programa ST possuem apenas os estados significativos para a checagem armazenados em memória, fazendo com que o espaço de estados a ser explorado seja reduzido drasticamente. A técnica só leva em consideração programas sem restrições de tempo e formados por variáveis booleanas. Os autores levantam como um dos problemas na adoção de checagem de modelos na indústria a interpretação dos resultados e contra-exemplos obtidos a partir das ferramentas, no entanto nada foi proposto neste sentido.

Existe uma demanda da indústria ainda não atendida em relação à criação de métodos e ferramentas de validação eficientes e de fácil utilização. Este último requisito, no entanto, geralmente é ignorado em diversas abordagens como foi possível observar neste levantamento bibliográfico. O presente trabalho propõe a criação de um método e teste de conformidade que automatiza tarefas realizadas na indústria ao tempo em que busca aproximar sua utilização da realidade do chão de fábrica.

[3] http://nusmv.fbk.eu/

Capítulo 3

Fundamentação Teórica

Neste capítulo é exposta a base teórica do presente trabalho. São apresentados os conceitos referentes a Controladores Lógico Programáveis, o padrão IEC 61131, as linguagens de especificação ISA 5.2 e de programação Ladder e rastreabilidade modelo-código.

3.1 Controladores Lógico Programáveis

Controladores lógico programáveis (CLPs) são computadores industriais que utilizam circuitos integrados ao invés de dispositivos eletromecânicos para implementar funções de controle. Estes dispositivos foram projetados para atuarem sobre condições adversas existentes no ambiente industrial (altas temperaturas, umidade, ruídos eletromagnéticos, dentre outros) substituindo os circuitos de relés por uma interface mais facilmente programável e extensível [5]. As primeiras aplicações datam do final da década de 1960, nas linhas de montagem da General Motors, cujo interesse na utilização destes dispositivos devia-se à constante necessidade de atualização das lógicas de controle implementadas através de circuitos de relés em suas linhas de montagem, onde até então era necessário alterar fisicamente as instalações elétricas a medida que novos produtos iam sendo lançados.

Um CLP é composto de uma unidade central de processamento, áreas de memória onde fica armazenado o programa do usuário após a inicialização do sistema, e cartões de entrada e saída, muitas vezes montados em *racks*, ou seja, extensões físicas do CLP que permitem a expansão da quantidade de portas de I/O.

As conexões de entrada e saída do CLP provêem isolamento de corrente entre a unidade

central de processamento e as portas de I/O, evitando interferências e problemas na execução da lógica de controle. Uma possível falha em um dispositivo ligado ao CLP ou de um cartão de I/O não irá interromper o funcionamento de toda a planta, por exemplo.

O funcionamento básico de um CLP se dá a partir da implantação de um programa do usuário, geralmente descrito em uma das cinco linguagens do padrão IEC 61131-3 (descrito em detalhes mais adiante), e a ligação aos sensores e atuadores da planta às respectivas entradas e saídas, analógicas ou digitais. Um CLP, após inicializado, fica em um laço constante executando o programa do usuário, como é ilustrado na Figura 3.1.

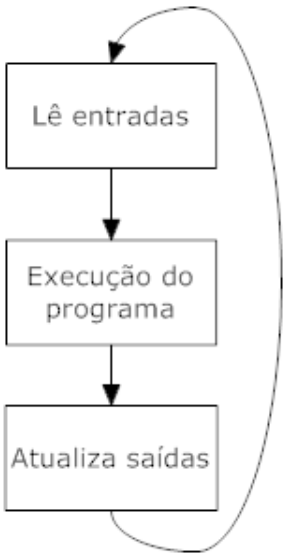


Figura 3.1: Sequência de execução do programa

Cada execução ocorre dentro do chamado tempo de varredura do programa, que é o tempo necessário para que todas as instruções do programa do usuário sejam executadas até o fim. Este valor é determinado pela velocidade de processamento do CLP e pelo tamanho do programa. O tempo de varredura total (tempo de *scan*) inclui tanto o tempo de varredura do programa como o tempo necessário para ler o estado das entradas e atualizar o valor das saídas [20].

Assim que a última instrução é executada o processo recomeça com a primeira, e ao invés de estar em constante interação com o ambiente, o CLP só considera o estado das entradas antes de iniciar a execução do programa e atualiza o valor das saídas apenas após o fim. Se durante a execução alguma entrada tiver seu valor alterado isto não será refletido no valor em memória da variável que representa a entrada.

O tempo de *scan*, portanto, limita a velocidade dos sinais com os quais um CLP pode trabalhar com segurança. Se uma entrada mudar seu valor de 1 para 0 e retornar a 1 em um intervalo de tempo menor que o tempo de *scan*, por exemplo, o CLP só levará em consideração o último estado ao iniciar um novo ciclo, ou seja, 1, ignorando a mudança de estado que ocorreu durante a execução anterior do programa.

3.2 Padrão IEC 61131-3

O padrão IEC 61131 define requisitos para programação e implantação de CLPs, abrangendo tanto *hardware* quanto *software*, e foi introduzido com o objetivo de reduzir custos devido a complexidade crescente dos sistemas, que exigiam treinamento de pessoal e programas cada vez maiores. Padronizando processos e reutilizando *software* seria possível reduzir custos e ainda assim manter vantagem competitiva entre fabricantes levando em conta outros aspectos fora do padrão, como requisitos adicionais necessários apenas em determinados segmentos de mercado ou implementações específicas de *hardware* no CLP [20].

O padrão é dividido em cinco seções, sendo a seção 61131-3 definida como um guia de melhores práticas de programação para CLPs, ao invés de um conjunto de regras rígidas a serem seguidas por fabricantes e usuários. Fica a cargo de quem implementa decidir que nível de observância ao padrão deve ser levado em conta [15].

Dentro da seção 61131-3 existem duas subdivisões, elementos comuns e linguagens de programação [22]. Os elementos comuns dividem-se em tipos de dados (inteiros, booleanos, palavras, datas, etc.), variáveis, configurações, recursos, tarefas, funções e blocos de funções. A segunda subseção define as cinco linguagens de programação do padrão, duas delas gráficas e duas textuais, que são:

- *Instruction List*, ou IL, uma linguagem textual cuja sintaxe assemelha-se à linguagem de programação Assembly;
- *Function Block Diagram*, ou FBD, linguagem gráfica cuja lógica é expressa por blocos gráficos interligados, que descrevem o comportamento de funções, blocos de funções e programas;

- *Ladder Diagram*, ou simplesmente Ladder, uma linguagem gráfica com elementos semelhantes aos circuitos de relés (mais detalhes na Seção 3.4);

- *Structured Text*, ou ST, linguagem cuja sintaxe textual assemelha-se as linguagens de programação estruturadas Ada, Pascal e C;

- *Sequential Function Charts*, ou SFCs, representam graficamente o comportamento seqüencial de programas, permitindo descrever em diferentes níveis de detalhamento sua estrutura interna. Os programas consistem em uma rede de funções ou blocos de funções responsáveis por trocar dados.

O método proposto neste trabalho leva em consideração a linguagem de programação Ladder, tanto na implementação da ferramenta desenvolvida quanto nos estudos de caso utilizados para fins de avaliação.

3.3 Diagramas de Lógica Binária ISA 5.2

O padrão ISA 5.2 define um método de diagramação lógica de sistemas binários de intertravamento e sequenciamento para o início, operação, alarme e interrupção de processos ou equipamentos diversos na indústria [14]. Os diagramas têm como objetivo facilitar a comunicação e entendimento do funcionamento de sistemas binários entre os diversos agentes da indústria (técnicos, gerentes, desenhistas, etc.).

O padrão simboliza operações binárias de maneira genérica, sendo aplicável a várias classes de dispositivos, tanto eletrônicos quanto elétricos, de fluidos, pneumáticos, hidráulicos, dentre outros. Os diagramas podem conter apenas os chamados elementos básicos, ou podem se utilizar de elementos não básicos que tornem as representações de sistemas mais concisas. O nível de detalhamento desejado deve ser definido de acordo com o uso do diagrama. Anotações e demais informações não lógicas podem ser utilizadas para facilitar o entendimento. Na Tabela 3.1 são detalhados os elementos lógicos ISA 5.2 utilizados neste trabalho.

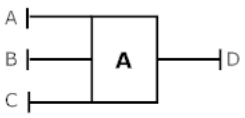
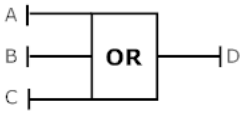
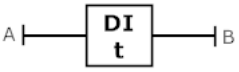
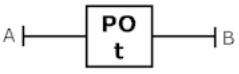
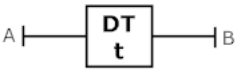
Função	**Símbolo**	**Definição**
Entrada		Representa uma entrada. Opcionalmente é possível definir o nome do instrumento ou dispositivo de entrada em "D N".
Saída		Representa uma saída. Opcionalmente é possível definir o nome do instrumento ou dispositivo de saída em "D N".
Negação		Representa a negação da lógica existente em A, ou seja, (B) = **NÃO** (A).
Conjunção		Uma conjunção ou **E** lógico. O valor de D é igual a (A) **E** (B) **E** (C).
Disjunção		Uma disjunção ou **OU** lógico. O valor de D é igual a (A) **OU** (B) **OU** (C).
Temporizadores		Temporizador de atraso de iniciação. O valor de B será igual a 1 quando o valor de A se tornar 1 e houver decorrido um tempo *t*. Assim que o valor de A se tornar 0 o valor de B também será 0.
		Temporizador de pulso. Existindo lógica 1 em A, independente de seu valor futuro, fará com que B seja 1 durante o intervalo de tempo *t*.
		Temporizador de atraso de desligamento. O valor de B será 1 assim que o valor de A for 1, quando o valor de A se tornar 0 o valor de B também será 0 após decorrido o tempo *t*.

Tabela 3.1: Elementos ISA 5.2

Na Figura 3.2 é ilustrado um exemplo de diagrama ISA 5.2 extraido de [7]. O sistema descrito faz parte de uma unidade de geração de hidrogênio, descrevendo parte do controle de nível de um tanque, mais especificamente o controle de uma bomba.

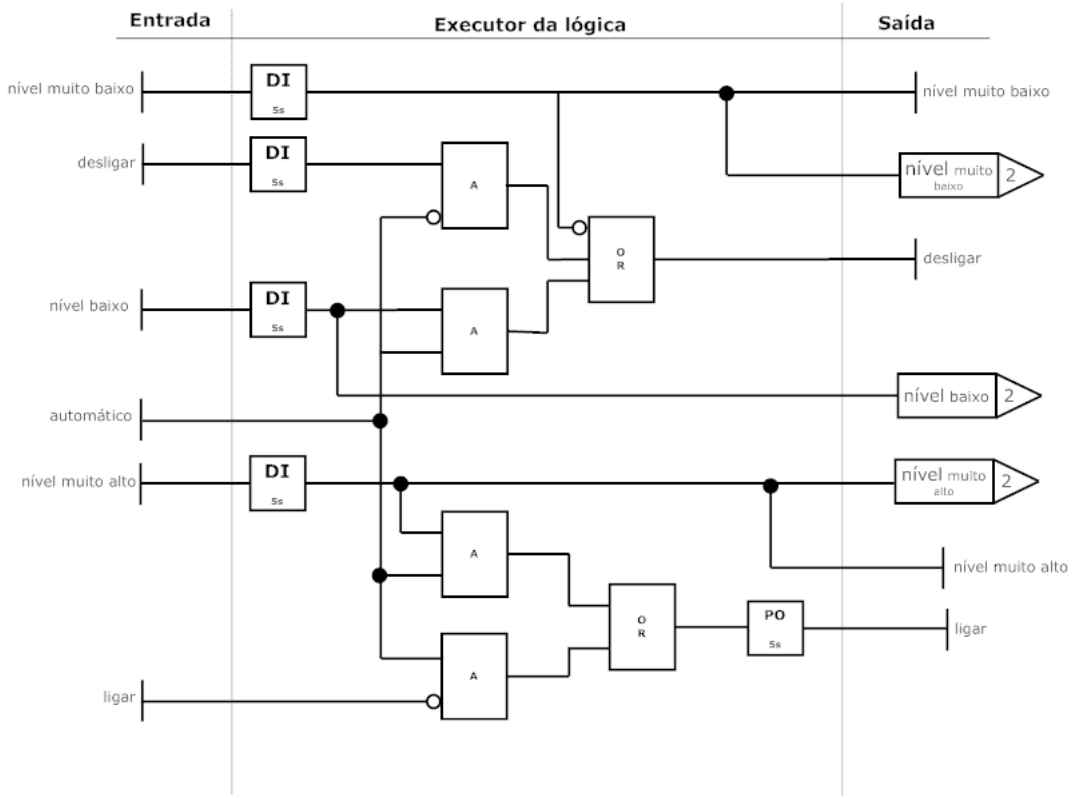


Figura 3.2: Exemplo de um diagrama ISA 5.2

A bomba pode ser operada tanto manualmente quanto automaticamente, de acordo com os níveis detectados pelos sensores instalados no tanque. Independente do modo de operação, caso o nível do tanque seja baixo a bomba é desligada por segurança.

No modo automático, a bomba é ativada caso os sensores indiquem nível alto, e desativada caso o nível seja baixo. Indicadores luminosos informam ao operador do sistema o estado da bomba e alarmes são ativados de acordo com o nível do tanque (saídas 1 e 3). Por segurança, os sinais dos sensores devem ter duração mínima de 5 segundos (indicado pelos temporizadores DI) para produzirem os comandos de ligar ou desligar a bomba e ativar os alarmes.

As setas próximas às saídas (nível baixo, muito baixo e muito alto) indicam a continuação da lógica em outro diagrama, no caso, o segundo. Através destas indicações é possível subdividir a lógica a ser implementada em diversos diagramas.

3.4 Linguagem Ladder

Ladder é uma das cinco linguagens definidas no padrão IEC 61131, que sumariza requisitos a serem implementados e seguidos por sistemas utilizando CLPs modernos, tanto em *hardware* quanto em *software*. A linguagem foi criada a partir da transcrição dos símbolos e expressões existentes nos antigos circuitos lógicos de relés, com o objetivo de manter a programação de CLPs simples e próxima do método utilizado até então [5]. Houve uma evolução da linguagem ao longo dos anos, permitindo o suporte de funções e blocos, além das instruções básicas existentes nos circuitos de relés.

As duas barras existentes na esquerda (L) e direita (N) em cada programa Ladder representam, respectivamente, os barramentos energizado e terra, havendo passagem de corrente elétrica no sentido da esquerda para a direita. A continuidade lógica de um programa Ladder se dá da esquerda para a direita, no sentido da corrente, caso a lógica resultante do programa seja satisfeita.

A linguagem recebeu este nome devido ao formato dos programas se assemelhar a uma escada (*ladder*, em inglês), sendo as lógicas de controle entre os barramentos energizado e terra os degraus. O programa é executado em seqüencia e caso haja continuidade lógica em um degrau, sua saída é energizada.

Na Tabela 3.2 são detalhados os elementos Ladder abordados neste trabalho. Cada entrada ou saída lógica existente no programa pode ser mapeada para um endereço físico de saída no CLP, de modo que a alteração dos valores das variáveis existentes no programa (e que representam tais portas de I/O) sejam refletidas externamente.

A colocação de elementos em série em um degrau define uma conjunção (operação E, Figura 3.3a) e em paralelo a disjunção (operação OU, Figura 3.3b). É possível também associar ambas, como é ilustrado na Figura 3.3c

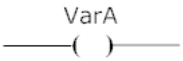
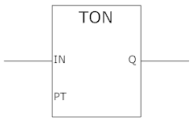
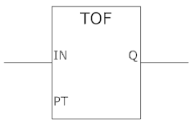
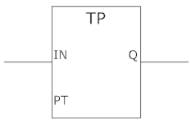
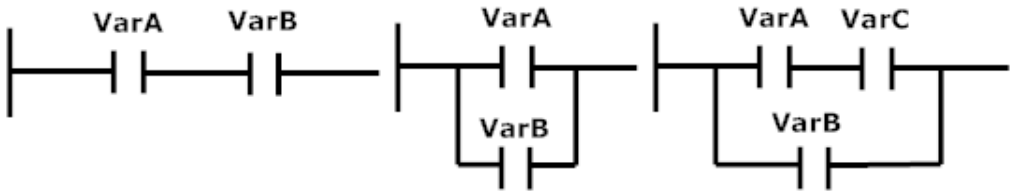
Função	**Símbolo**	**Definição**
Entrada		Contato normalmente aberto. Retorna o valor booleano existente em uma referência (1 caso a referência seja 1 e vice-versa).
		Contato normalmente fechado. Retorna o inverso do valor booleano existente em uma referência (1 caso a referência seja 0 e vice-versa).
Saída		Bobina de saída. Controla uma saída real ou interna. Caso a lógica resultante no degrau seja 1 o valor da bobina será 1, e vice-versa.
Temporizadores		Temporizador TON. Após um intervalo de tempo determinado (valor de PT) a saída (Q) é ativada, caso exista lógica 1 na entrada (IN).
		Temporizador TOF. Energiza a saída (Q) caso exista lógica 1 na entrada (IN). Caso o valor da entrada se torne 0, após um intervalo de tempo determinado (valor de PT) a saída é desativada.
		Temporizador TP. Energiza a saída (Q) caso exista lógica 1 na entrada (IN) durante o tempo pré-determinado (PT). Mesmo que a lógica da entrada se torne 0 a saída continuará a ser energizada durante o intervalo de tempo PT.

Tabela 3.2: Elementos Ladder



(a) E (b) OU (c) Operações encadeadas

Figura 3.3: Operações lógicas E e OU em um diagrama Ladder

Na Figura 3.4 é ilustrado um exemplo de programa Ladder extraído de [20].

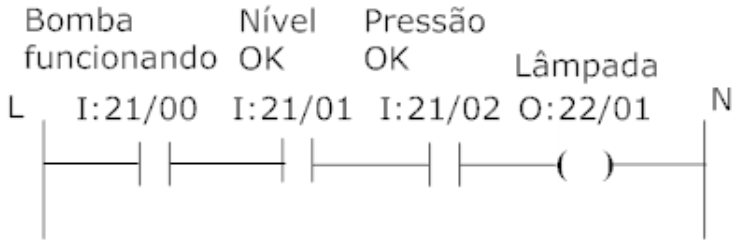


Figura 3.4: Exemplo de diagrama Ladder

O funcionamento de uma unidade hidráulica é controlado por um programa Ladder simples, responsável por indicar através de uma lâmpada sua correta operação. Três sensores são conectados ao CLP, e responsáveis por indicar se:

- A bomba está funcionando (a bomba foi energizada)
- Existe óleo no tanque (indicado por um sensor de nível)
- A pressão do óleo é adequada (indicado por um sensor de pressão)

Caso as três situações sejam verdadeiras ao mesmo tempo, a saída será liberada (ou seja, a lâmpada deverá ser acesa).

No programa Ladder mostrado na Figura 3.4 temos a entrada I:21/00 representando a primeira condição (bomba funcionando), a segunda entrada I:21/01 representando a segunda condição (nível satisfatório) e a terceira entrada I:21/02 representando a terceira condição (pressão satisfatória). Caso a lógica resultante do degrau seja satisfeita, ou seja, (I:21/00 AND I:21/01 AND I:21/02 == 1), a saída (O:21/02) deverá ser energizada assim que o degrau for avaliado, fazendo com que a lâmpada seja acesa.

Na Figura 3.5 é representada uma instalação típica de um CLP ligado aos sensores e atuadores do programa descrito.

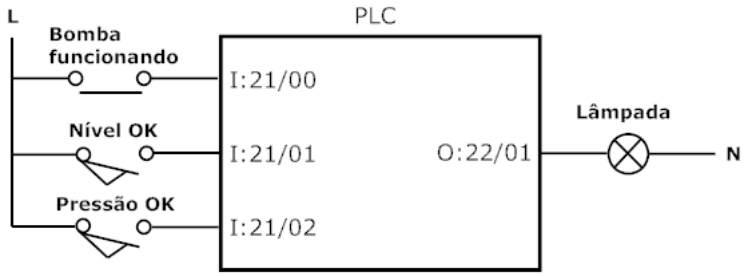


Figura 3.5: Instalação do CLP implementando o controle da bomba hidráulica

3.5 Autômatos temporizados

Programas para CLPs se encaixam na categoria de sistemas de tempo real, pois suas interações com o ambiente de produção podem obedecer restrições de tempo. Tais sistemas podem ser modelados como TIOTS (*Timed Input/Output Transition Systems*), que são sistemas de transição em que cada ação é do tipo entrada ou saída e existem rótulos indicando um determinado atraso de tempo existente entre cada uma delas.

Muitas pesquisas têm sido desenvolvidas aplicando autômatos temporizados na modelagem de sistemas de tempo real [3; 2; 27]. Este formalismo pode ser definido como uma máquina de estados com extensões temporais, que permitem seguir a progressão do tempo através de variáveis auxiliares discretas de relógio e adicionar guardas entre transições que levem o tempo em consideração [13]. A seguir é feita a definição formal de autômatos temporizados.

Sendo A um conjunto de ações, assumindo que há uma ação distinguível e inobservável $t \notin A$, denota-se A_t o conjunto $A \cup \{t\}$. Sendo X um conjunto de variáveis com $\mathbb{R}_{\geq 0}$-valores chamadas relógios. Sendo *G(X)* um conjunto de guardas sobre os relógios formadas por conjunções de restrições do tipo x $\bowtie$ c, e sendo *U(X)* um conjunto de atribuições que sinalizam atualizações de relógios do tipo x := c, onde x $\in X$, c $\in \mathbb{N}$ e $\bowtie \in \{\leq, <, =, >, \geq\}$. Um autômato temporizado sobre (A,X) é uma tupla (L,l_0,I,E) onde:

- L é um conjunto de localidades e $l_0 \in L$ é uma localidade inicial;

- *I*:*L* -> *G(X)* atribui invariantes as localidades;
- *E* é um conjunto de arestas de forma que $E \subseteq L \times G(X) \times A_t \times U(X) \times L$.

O modelo de autômatos temporizados do UPPAAL, utilizado neste trabalho como formalismo intermediário de entrada para a ferramenta de geração e execução de casos de teste, é definido com algumas extensões sobre o formalismo original, que são:

- *Templates* (ou modelos), permitindo que autômatos sejam especificados utilizando-se um conjunto de parâmetros de tipos diversos (inteiros, canais de sincronização, booleanos, dentre outros). Utilizando-se *templates* é possível utilizar instâncias diferentes de um processo, cujo valor de suas variáveis locais será igual aos valores passados como argumento na declaração;
- Constantes - declarações de variáveis acompanhadas pelo modificador *const*, cujos valores são inteiros e não podem ser alterados no decorrer da simulação de execução da rede de autômatos;
- Variáveis inteiras limitadas - variáveis cujas declarações indicam os limites inferior (-32768) e superior (32768) dentro dos quais deve residir qualquer valor atribuído à variável;
- Sincronizações binárias - canais, declarados como *chan canal*, permitem a sincronização entre arestas com etiquetas *c!* e *c?*, ou seja, caso um evento seja disparado entre dois estados cuja aresta possui a etiqueta *c!*, outra aresta que possua *c?* será ativada, levando o autômato a que pertence ao seu próximo estado;
- Sincronizações urgentes - canais de sincronização sobre os quais não podem haver atrasos, são declarados como *urgent chan canal*;
- Canais de *broadcast* - permitem a sincronização de uma única aresta etiquetada com *c!* com várias outras etiquetadas com *c?*. Todos os autômatos que puderem sincronizar em seu estado atual deveram realizar a sincronização;
- Localidades urgentes - o tempo não transcorre enquanto o sistema estiver em uma localidade urgente;

- Localidades *committed* - são mais restritivas do que localidades urgentes. Um estado *committed* não pode causar atrasos e a próxima transição deve necessariamente conter uma aresta que parta de pelo menos uma das localidades *committed*;

- Vetores - são utilizados na declaração de relógios, canais, constantes e variáveis inteiras;

- Inicializações - é possível atribuir valores iniciais a variáveis inteiras e vetores de variáveis inteiras.

O modelo UPPAAL também permite a utilização de expressões sobre relógios e variáveis inteiras, através de:

- Guardas - expressões que não possuem efeitos colaterais, são avaliadas através de valores booleanos, referenciam apenas relógios, variáveis inteiras e constantes, relógios e diferenças entre eles são comparadas apenas a expressões inteiras, guardas sobre relógios são conjunções, sendo disjunções permitidas apenas entre condições inteiras;

- Sincronizações - uma etiqueta de sincronização possui o formato *c!*, *c?* ou uma etiqueta vazia, sendo *c* o nome do canal de sincronização utilizado. A expressão deve ser livre de efeitos colaterais, ser avaliada para um canal de sincronização e referenciar apenas inteiros, constantes e canais;

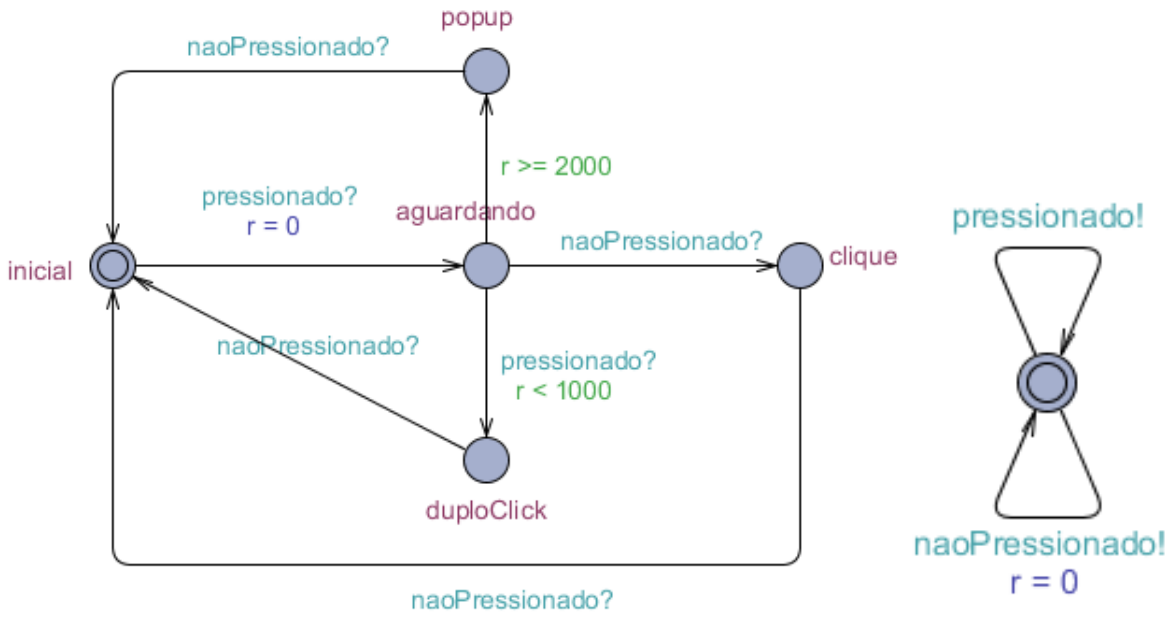
- Atribuições - uma etiqueta de atribuição possui efeitos colaterais e é formada por uma lista de expressões separadas por ponto-e-vírgula. Referenciam apenas relógios, variáveis inteiras e constantes e atribuem apenas inteiros a relógios.

- Invariantes - são expressões sem efeitos colaterais; apenas relógios, variáveis inteiras e constantes são referenciadas; são conjunções de condições no formato $c < e$ ou $c <= e$, onde c é um relógio e o valor de *e* é inteiro.

Os sistemas modelados utilizando os autômatos temporizados com extensões do UPPAAL formam uma rede de autômatos temporizados, que compartilham variáveis e relógios globais, canais de sincronização e ações, permitindo a construção de grandes modelos através de interações em paralelo dos autômatos.

Na Figura 3.6 é ilustrado um exemplo de modelagem utilizando autômatos temporizados UPPAAL da interação de um usuário com uma interface de toque, utilizada, por exemplo, em *smartphones touchscreen*. A interação do usuário com a tela pode gerar três tipos de comandos distintos que são enviados a interface gráfica:

- Um clique - caso haja um toque na tela e em seguida a tela deixe de ser pressionada;
- Duplo clique - caso hajam dois toques consecutivos na tela com um tempo menor que 1000 milisegundos entre eles;
- Menu *popup* - caso haja um toque na tela por um período de tempo maior que 2000 milisegundos, e em seguida a tela deixe de ser pressionada.



(a) Sistema (b) Usuário

Figura 3.6: Modelagem do comportamento

O autômato que modela o sistema é descrito em Figura 3.6a, que é formado por cinco localidades:

- *inicial* - estado inicial do sistema, aguardando a interação do usuário,
- *aguardando* - após o primeiro toque (sincronização *pressionado!*), o sistema irá aguardar o próximo comando do usuário para determinar a ação desejada. A aresta entre os

estados *inicial* e *aguardando* inicia o relógio do sistema (*r = 0*), no estado aguardando o relógio continuará a ser incrementado até que a execução seja encerrada ou o relógio seja reiniciado. Caso a tela deixe de ser pressionada só haverá um estado possível (*clique*). Caso o relógio atinja 2000 milisegundos e a tela continue a ser pressionada o próximo estado será *popup*, e caso haja um outro clique em um tempo menor que 1000 milisegundos o próximo estado será *duploClick*.

- *popup*, *clique* e *duploClick* - ações possíveis do sistema a partir da interação do usuário. Após cada uma dessas localidades será possível retornar ao estado *inicial*, aguardando uma nova interação.

O modelo de comportamento do usuário, descrito na Figura 3.6b, é formado por uma única localidade, sendo possível a execução de duas ações, descritas pelas arestas existentes no autômato. Caso o usuário pressione a tela (*pressionado!*), será enviado através do canal de sincronização binária *pressionado* uma mensagem para todos os autômatos (no caso o único outro autômato existente na modelagem do sistema) cujas arestas possuam uma etiqueta *pressionado?* que sincronizem, se possível. O mesmo ocorre para o canal de sincronização *naoPressionado*, e a cada liberação da tela o relógio do sistema é reiniciado (atribuição *r*=0).

A partir desta rede de autômatos descrevendo sistema (tela) e ambiente (usuário) é possível simular cenários possíveis de interação, gerando-se aleatoriamente eventos de toque ou liberação da tela obedecendo intervalos de tempo entre esses eventos. Processo semelhante, detalhado no Capítulo 5, é utilizado neste trabalho para gerar automaticamente e aleatoriamente entradas para um modelo de um programa a ser implantado em um CLP.

3.6 Rastreabilidade modelo-código

O conceito de rastreabilidade entre modelo e código pode ser definido como uma metodologia que permite rastrear informações de *design* entre modelos e código-fonte ou demais modelos gerados automaticamente a partir deles. As conexões entre modelo e código são possíveis a partir da introdução de marcações (*tags*) específicas no código gerado que armazenam informações sobre as transformações de modelos realizadas, juntamente com ligações (*links*) aos elementos de origem e destino destas transformações [11].

No cenário de geração automática de código a partir de modelos o uso de marcações entre modelo e código permite um maior controle sobre a sincronização entre ambos em caso de alterações, refletindo-as automaticamente. No método de teste proposto foi utilizado este conceito com o objetivo de permitir a associação entre modelos de teste (autômatos temporizados), vereditos de erro (eventos na rede de autômatos) e diagramas de especificação e implementação.

O arquivo XML que contem informações de rastreabilidade recolhidas pelo método é formado por tuplas descrevendo cada elemento do programa (entrada, saída ou temporizador) e sua respectiva representação em cada artefato (programa Ladder, especificação ISA 5.2 ou traço de execução de teste). Na Figura 3.7 é ilustrada a representação simplificada de uma entrada em cada um dos artefatos.

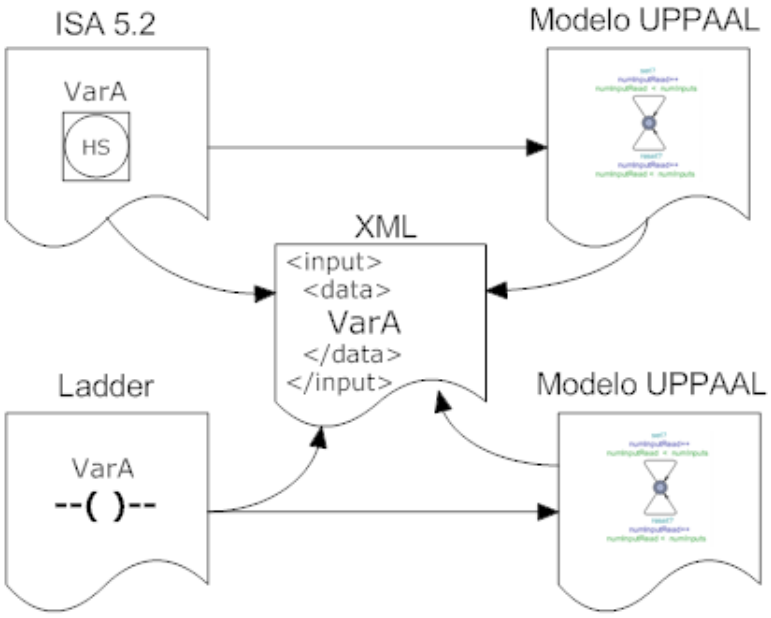


Figura 3.7: Representação simplificada de uma entrada no arquivo de rastreabilidade

Armazenando-se tais informações é possível associar instantaneamente partes do código a trechos da documentação. Em nosso método, permite que casos de teste gerados e executados automaticamente identifiquem partes do código problemáticas de acordo com a especificação.

A implementação realizada em [19] foi instrumentada de forma a prover informações de rastreabilidade entre os modelos originais (linguagens de especificação e implementação) e os modelos intermediários (redes de autômatos temporizados), de forma que os *templates*

XML no formato UPPAAL gerados contêm informações adicionais que permitem a rastreabilidade entre modelo e código na etapa de análise dos traços gerados a partir da execução dos testes. A partir de um traço de erro gerado por um veredicto *FAIL* lançado pelo testador TRON há um componente de *software* responsável por identificar que partes do código são responsáveis pela condição faltosa a partir das informações de rastreabilidade coletadas na etapa de geração dos templates. Este processo é detalhado no Capítulo 4.

Capítulo 4

O Método

Neste capítulo é detalhado o método de geração, execução e exibição de erros de forma automatizada de testes de conformidade.

O método desenvolvido faz uso da ferramenta UPPAAL-TRON, que realiza a geração e execução automática de testes de sistemas de tempo real a partir de sua modelagem em uma rede de autômatos temporizados no formato UPPAAL. A ferramenta UPPAAL-TRON possui dois modos de funcionamento:

- *Online*, gerando os eventos relativos a um caso de teste e os enviando diretamente à implementação a ser testada (IUT - *Implementation Under Test*) através do padrão OPC (*Object Linking and Embedding for Process Control*[1]);
- *Offline*, gerando a seqüencia de eventos, armazenando-a e em seguida executando-se outra instância do testador, onde será emulado o funcionamento do sistema a partir do modelo de implementação (também especificado no formato UPPAAL) para a qual serão então passados como entrada os eventos gerados a partir da especificação. Caso implementação e especificação estejam em conformidade, todos os eventos esperados pelo caso de teste serão atingidos durante a execução dessa segunda etapa.

O TRON realiza teste de caixa preta, ou seja, são gerados casos de teste a partir de um arquivo de especificação observando-se apenas as entradas e saídas possíveis para o sistema descrito, sem haver uma verificação do seu funcionamento interno.

[1] http://www.opcfoundation.org/

Durante a etapa de execução dos casos de teste são possíveis três vereditos (descrições do estado final do sistema de teste ao interromper sua própria execução) a serem emitidos pelo UPPAAL-TRON:

- *SUCESS* - caso seja atingido o tempo limite para geração e execução do caso de teste e nenhum evento inesperado tenha sido disparado, ou seja, dentro do tempo estipulado para a execução do teste o sistema se mostrou em conformidade com sua especificação;

Devido a natureza não exaustiva do método e a geração não-determinística de casos de teste não é possível determinar com certeza se implementação e especificação estão em conformidade, no entanto a cobertura de teste atingida é proporcional ao tempo de execução, sendo esta a única variável limitante para obter-se uma maior segurança sobre o veredito atingido;

- *INCONCLUSIVE* - caso uma saída emitida pelo caso de teste gerado a partir da implementação não pode ser sincronizada com uma transição do modelo de ambiente. O caso de teste gerado não prova não-conformidade, apenas que não foi possível avançar na sua execução devido a condições inesperadas pelo modelo do ambiente.

- *FAIL* - dentro do tempo de teste estipulado um evento não esperado foi detectado, portanto é certa a não-conformidade.

O método possui como entradas os diagramas ISA 5.2 e Ladder gerados durante o processo de desenvolvimento de um programa para CLP (como descrito no Capítulo 1), e exibe ao usuário eventuais erros de conformidade detectados na etapa de teste, realizada pelo TRON a partir dos modelos de autômatos temporizados gerados automaticamente.

O processo se divide em quatro fases:

- Inicialmente os diagramas de especificação (na linguagem ISA 5.2) e de implementação (na linguagem Ladder) são interpretados por um *parser* (desenvolvido em [19]) e a partir destes são gerados arquivos XML que contêm uma descrição do comportamento especificado neles em um formalismo comum, neste caso Autômatos Temporizados (AT), obedecendo ao padrão XML aceito pelo UPPAAL-TRON.

Os diagramas ISA 5.2 não são divididos explicitamente em degraus ou blocos lógicos ao serem confeccionados e não possuem uma seqüência de execução (são avaliados por inteiro

a cada iteração) [14]. Por isso a associação entre elementos e lógicas de controle desses diagramas com os diagramas Ladder não é direta, sendo necessária uma etapa de adaptação antes da geração dos modelos UPPAAL a partir desses diagramas.

Para a montagem das lógicas de controle são armazenados os elementos ISA 5.2 em uma árvore binária que é então percorrida em-ordem, a seguir a ordem de avaliação dos blocos lógicos formados é determinada a partir da seqüência das saídas do programa Ladder, sendo necessário o conhecimento prévio dessa ordem.

- A partir do comportamento descrito pelo modelo formal do diagrama ISA 5.2 são gerados os casos de teste *offline* no testador TRON. Nesta etapa o TRON é executado em modo de emulação, ou seja, é fornecido um arquivo de especificação, o testador gera os casos de teste que são passíveis de execução em um conjunto ambiente/sistema com estas especificações e armazena as entradas produzidas pelos casos de teste.

- O modelo formal do programa Ladder é interpretado em uma terceira fase, onde ocorre a simulação da execução do programa em um ambiente real, de acordo com o que foi especificado no diagrama. As entradas geradas pelos casos de teste criados na fase anterior serão utilizadas durante a execução do TRON em modo de monitoramento com o arquivo XML gerado a partir do diagrama Ladder como entrada. Nesta etapa ocorre o casamento entre as entradas permitidas pela especificação e as saídas geradas pela implementação.

Durante estas duas primeiras etapas são recolhidas informações acerca de quais elementos são representados em cada seção dos respectivos modelos XML (que representam a rede de autômatos temporizados gerada a partir destes) e é gerado um arquivo XML contendo tais informações, tornando possível a etapa de associação entre elementos dos traços de erro (eventos de AT) e o diagrama de implementação (contatos, bobinas e temporizadores Ladder).

Na Figura 4.1 é ilustrada a representação de um temporizador neste arquivo XML, identificando o elemento em cada um dos documentos de entrada e saída do método.

- Caso seja expedido um veredito do tipo *FAIL* na terceira etapa, o TRON interrompe sua execução e gera um traço de erro (*log* da simulação do programa Ladder durante

```
<timer>
    <ladder>
        <name>TIMER1</name>
        <rung>1</rung>
        <line>2</line>
        <position>14</position>
        <negated>false</negated>
    </ladder>
    <isa>
        <name>PO212</name>
        <block>1</block>
        <negated>false</negated>
    </isa>
    <trace>
        <template>TON_timer1</template>
        <type>timer</type>
        <rung>1</rung>
        <syncronization>sync1</syncronization>
    </trace>
</timer>
```

Figura 4.1: Representação de um temporizador TON no arquivo de rastreabilidade

a execução do caso de teste) que especifica que entradas, transições realizadas entre os estados possíveis do programa e saídas levaram a tal veredito. Nesta etapa há a interpretação do traço e posterior exibição em um ambiente gráfico de uma forma intuitiva do erro, facilitando a identificação de trechos de código problemáticos.

Na Figura 4.2 é ilustrado o processo automatizado. Os processos "Geração de casos de teste", "Execução de casos de teste"e os documentos "Traços de execução"no fluxograma são de responsabilidade do testador UPPAAL-TRON, os demais do método proposto neste trabalho.

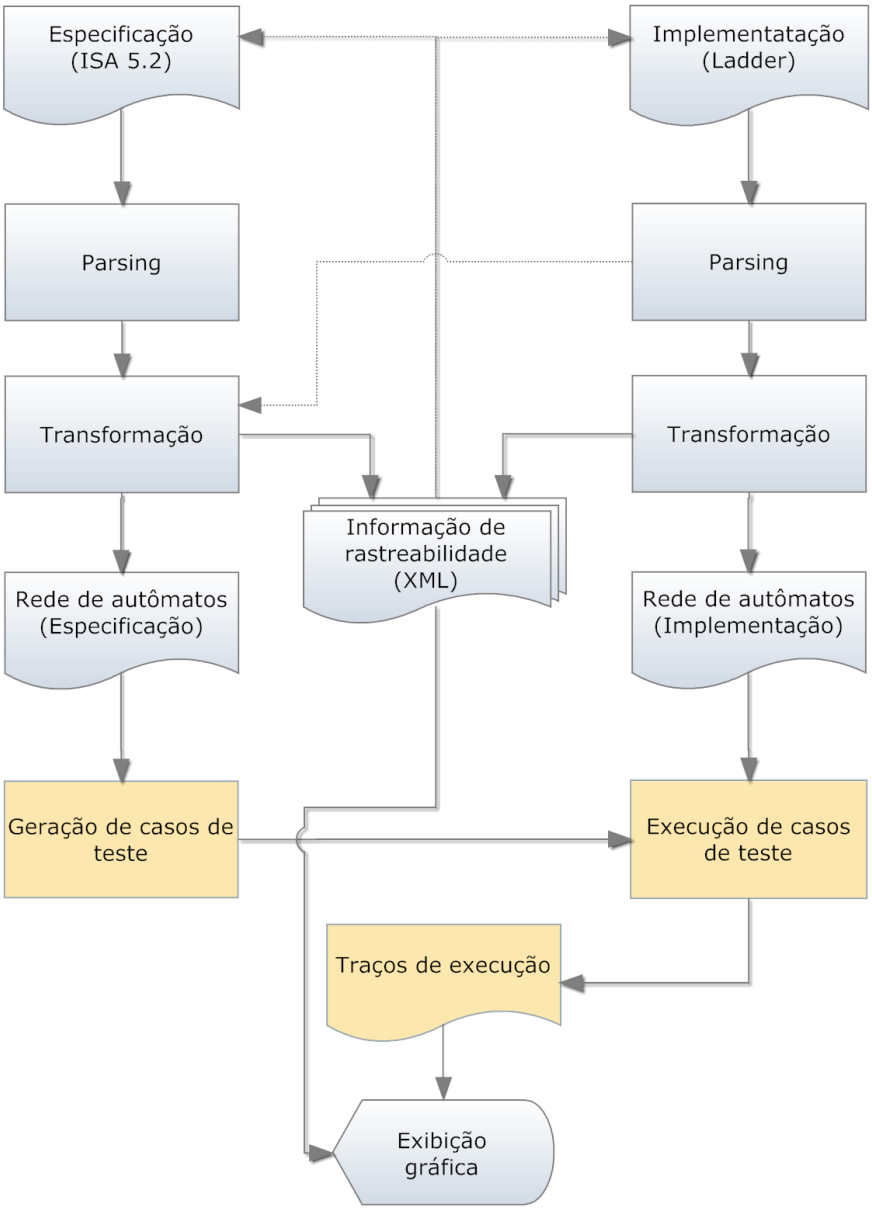


Figura 4.2: Processo de teste automatizado

Capítulo 5

Modelagem

Neste capítulo é detalhado o processo de geração dos modelos de autômatos temporizados, juntamente com o arquivo XML de rastreamento que contém as informações necessárias à etapa de interpretação dos traços de execução dos testes de aceitação. Nas seções seguintes são detalhadas as etapas de *parsing* dos arquivos de diagramas ISA 5.2 e Ladder, a criação dos modelos de autômatos temporizados no formato UPPAAL, a geração e execução dos testes de aceitação através do UPPAAL-TRON (detalhando-se os eventos presentes nos traços de teste) e por fim a criação do XML contendo informações de rastreabilidade.

5.1 *Parsing*

A etapa de *parsing* envolve a identificação de entradas, saídas e temporizadores, além das operações lógicas envolvendo estes. Diagramas Ladder, por possuírem sintaxe completamente textual, sendo compostos por caracteres ASCII, podem ser criados em editores de texto simples. Na Figura 5.1 é ilustrado um exemplo de programa Ladder aberto no editor de texto Notepad++[1].

Durante o *parsing* de diagramas Ladder são inicialmente identificados variáveis e nomes de temporizadores, é em seguida feita a associação entre esses elementos e os tipos de símbolos Ladder (contatos normalmente abertos e fechados, bobinas e tipos de temporizadores). Por fim são criadas as operações lógicas que representam cada degrau, ou seja, as lógicas de controle resultantes de cada degrau. Nesta última etapa são identificadas as operações **E** e

[1] http://http://notepad-plus-plus.org/

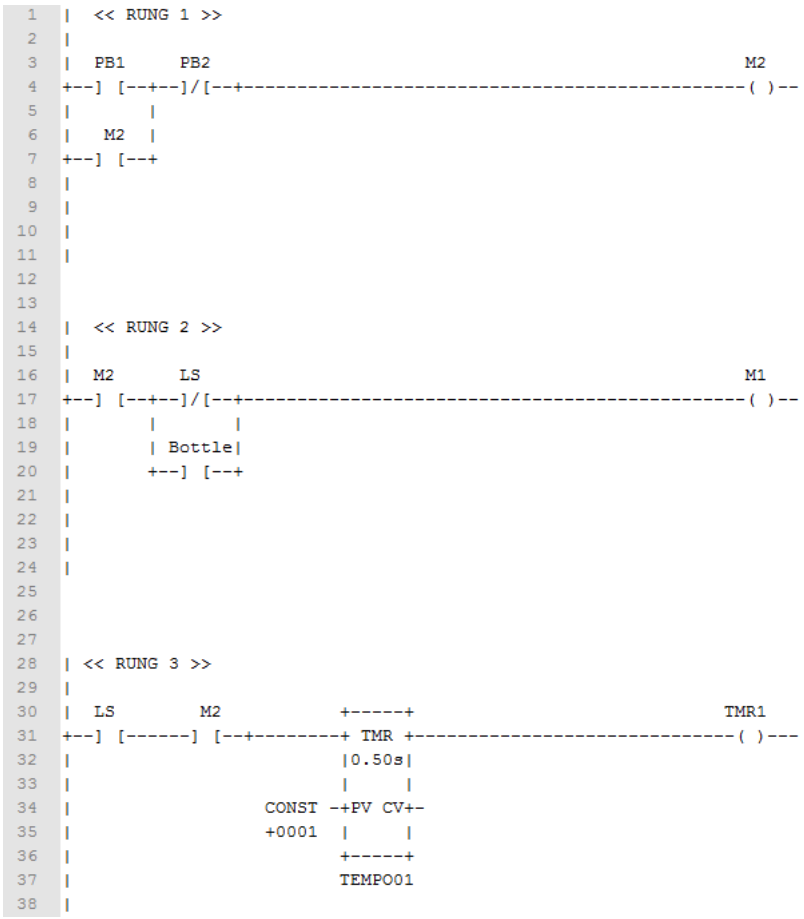


Figura 5.1: Edição de um diagrama Ladder

OU existentes no diagrama.

Diagramas ISA 5.2, por sua vez, possuem sintaxe visual, tornando necessária uma representação intermediária de forma que seja possível interpretar computacionalmente tais diagramas. Um editor visual, não implementado neste trabalho, é responsável por prover a representação intermediária (em XML) dos diagramas.

Além da representação visual, nos diagramas ISA 5.2 não existe informação explícita sobre a localização de elementos (posição dos mesmos) e divisão em degraus como nos diagramas Ladder, ou seja, o programa é representado como um único bloco de instruções interligadas. A representação em XML desses diagramas, como ilustrado na Figura 5.2, demonstra tais características.

Cada elemento do diagrama (entrada, saída, temporizador ou operação E/OU) é identificado por um nome, identificador do símbolo (*idSymbol*), um tipo (*symType*, físico ou lógico) e um subtipo (*symSubType*). As conexões aos demais elementos do diagrama são especifica-

```
<ISAFILE>
 <symbol name="PB1" idSymbol="HS_1_1" symType="fisical" symSubType="IN">
        <output feedback="false" negated="false">
            <idOutput>OUT0</idOutput>
            <outConInfo>
                <conSymName>OR1</conSymName>
                <conSymId>OR_1_1</conSymId>
                <conInpId>OR_1_1_IN_1</conInpId>
            </outConInfo>
        </output>
 </symbol>
 <symbol name="OR1" idSymbol="OR_1_1" symType="logical" symSubType="OR">
                <input feedback="false" negated="false">
                        <idInput>OR_1_1_IN_1</idInput>
                        <conSymName>PB1</conSymName>
                        <conSymId>HS_1_1</conSymId>
                 </input>
                 <input feedback="false" negated="false">
                        <idInput>OR_1_1_IN_2</idInput>
                        <conSymName>M2</conSymName>
                        <conSymId>HS_1_2</conSymId>
                 </input>
                 <output feedback="false" negated="false">
                        <idOutput>OUT6</idOutput>
                        <outConInfo>
                              <conSymName>AND1</conSymName>
                              <conSymId>AND_1_1</conSymId>
                              <conInpId>AND_1_1_IN_1</conInpId>
                        </outConInfo>
                 </output>
</symbol>
```

Figura 5.2: Trecho de um arquivo XML representando um diagrama ISA 5.2

das através das tags <*input*></*input*> e <*output*></*output*>.

Para que seja possível fazer a associação direta entre diagramas ISA 5.2 e Ladder, na etapa posterior de criação do modelo de rastreabilidade (detalhes na seção 5.4), é necessário interpretar os mesmos como se fossem diagramas Ladder, ou seja, dividindo o programa representado em blocos de instruções com uma saída cada, interpretando-se as lógicas de controle resultantes separadamente. Para isso os elementos ISA 5.2 de um determinado diagrama são lidos e representados computacionalmente através de árvores binárias, sendo as raízes as saídas de cada bloco de instruções equivalente a um degrau Ladder. Estas árvores são percorridas recursivamente em ordem infixa, gerando as expressões que representam as lógicas de controle para cada bloco de instruções.

5.2 Modelos UPPAAL

A etapa de geração dos modelos de autômatos temporizados ocorre após o *parsing* dos diagramas ISA 5.2 e Ladder do programa a ser validado, ou seja, após a representação em memória de todos os elementos de ambos os diagramas. A partir de cada diagrama é criado um modelo diferente, sendo estes modelos confrontados posteriormente através de testes de conformidade.

Cada seção do programa é modelada por um *template*, assim como o comportamento do CLP (ciclo de execução, atualização e processamento de entradas e saídas físicas). A modelagem do ambiente, ou seja, da interação do conjunto CLP e programa com o ambiente físico é simulada, permitindo a geração de dados não-determinísticos como entrada para o sistema.

Nas seções seguintes são detalhados cada um dos modelos de autômatos gerados para cada parte da rede de autômatos que representa o comportamento de um programa para CLP.

5.2.1 Entradas

Na Figura 5.3 é ilustrada uma entrada de um programa, que é modelada como um único autômato com duas transições, através das quais podem ser atribuídos os valores 0 ou 1 assim que um evento set ou reset é recebido pelos canais de sincronização *set?* ou *reset?*.

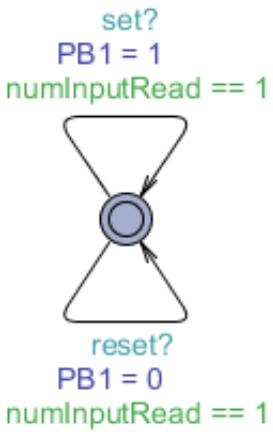


Figura 5.3: Modelo de entrada

A variável global *numInputRead* é utilizada no modelo para controlar a quantidade de entradas lidas, sendo utilizada como guarda neste autômato para possibilitar a leitura em

seqüencia das entradas. Existem outros autômatos semelhantes para as demais entradas do programa, e todos eles sincronizam ao receberem os eventos set e reset. No entanto, apenas aquele que estiver habilitado (*numInputRead* == *numeroSequenciaDaVariavel*) terá sua transição ativada e conseqüentemente irá atualizar o valor da variável (no caso *PB*=1 ou *PB*=0).

O processo de atualização das variáveis de entrada é realizado pelo autômato ilustrado na Figura 5.2.4, composto por duas localidades. A primeira destas localidades é *committed*, pois o tempo decorrido durante o processo de leitura das variáveis de entrada não faz parte do tempo de *scan* e, portanto, é ignorado pelo modelo.

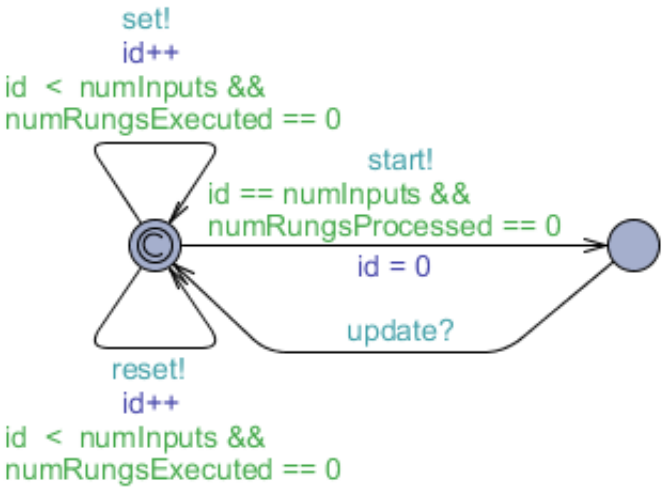


Figura 5.4: Modelo do processo de atualização das entradas

Este autômato é responsável por gerar os eventos *set!* e *reset!* que são recebidos via canais de sincronização pelo autômato da Figura 5.3 e geram os valores 0 e 1 para as entradas. Enquanto o valor de *id* for menor que a quantidade de entradas e o processo de execução da lógica (*numRungsExecuted*==0) não tiver sido iniciado, o autômato se manterá na primeira localidade, enviando eventos set e reset. Como ambas as transições são passíveis de executar no início da simulação, valores booleanos aleatórios são atribuídos às variáveis de entrada.

Após a atualização de todos os valores das variáveis de entrada será permitida a transição para o segundo estado, que iniciará a execução da lógica do programa (sincronização *start!*). Existe ainda um terceiro autômato, ilustrado na Figura 5.5, responsável por atualizar a variável global *numInputRead*, funcionando como um contador.

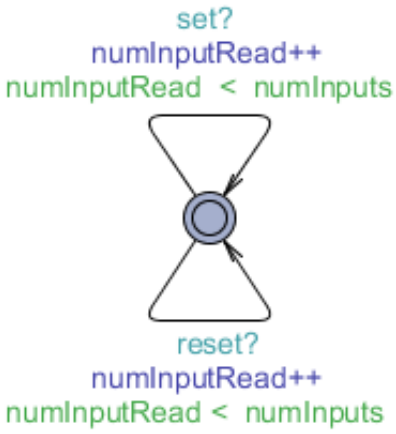


Figura 5.5: Modelo do controle da atualização dos valores de entrada

5.2.2 Saídas

Os valores das variáveis de saída não são representados individualmente por autômatos, como as entradas, e sim por um vetor de valores inteiros declarado globalmente. Assim que é iniciado o processamento dos valores de saída são enviadas sincronizações *high!* e *low!* para o autômato ilustrado na Figura 5.6, indicando o valor atribuído para cada uma delas.

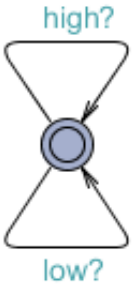


Figura 5.6: Modelo do processamento dos valores de saída

O autômato responsável pelo processo de avaliação dos valores das saídas é ilustrado na Figura 5.7, e possui funcionamento semelhante ao autômato que modela o processo de atualização dos valores das variáveis de entrada.

5.2.3 Temporizadores

Temporizadores são modelados por autômatos como o ilustrado na Figura 5.8. Os eventos de sincronização do tipo *syncNumeroTemporizador!* são enviados pelo autômato que modela a

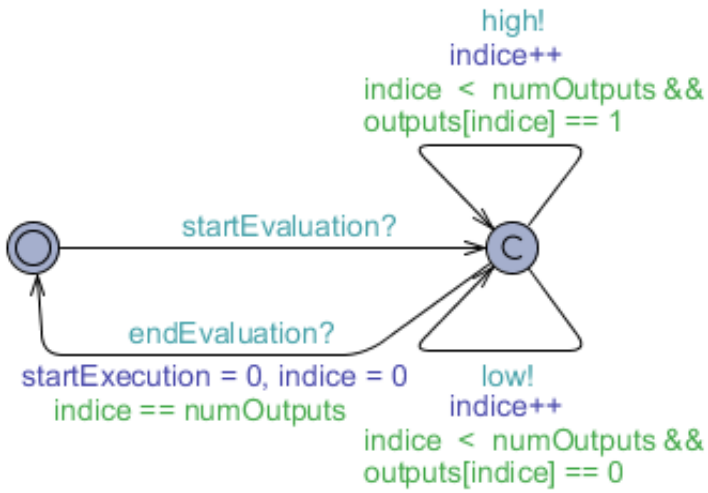


Figura 5.7: Modelo do processo de avaliação dos valores de saída

execução da lógica do programa assim que a lógica resultante na entrada do temporizador seja a esperada para sua inicialização.

A variável *tScan* representa o tempo de scan do programa, a função *inputTimer()* retorna a lógica resultante na entrada do temporizador, numCycles e control são variáveis de controle e *out_NomeDoTemporizador* representa a saída do temporizador.

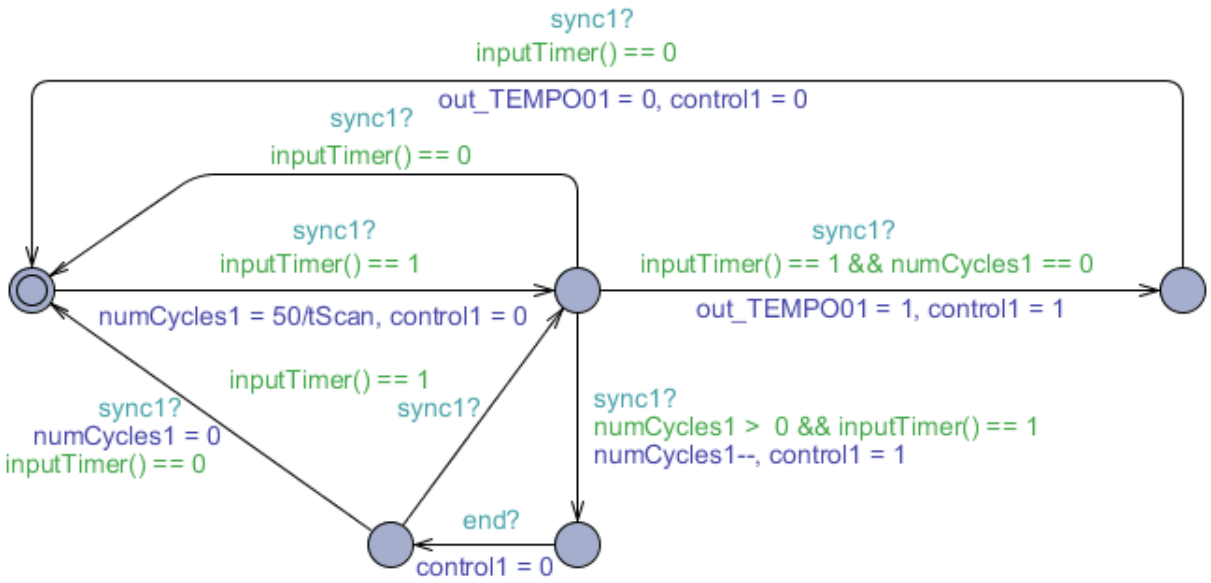


Figura 5.8: Modelo de um temporizador TON

5.2.4 Ciclo de varredura e execução da lógica do programa

Os autômatos apresentados nesta seção modelam o comportamento do CLP quando executando um programa Ladder. O ciclo de *scan* é modelado como ilustra a Figura 5.9. A sincronização *start!* é disparada assim que o processo de leitura das variáveis de entrada é terminado. Enquanto o tempo decorrido na simulação for menor que o tempo de scan (*tScan*) do programa, o autômato se manterá na segunda localidade, enviando a sincronização *execute!*, que indica que o autômato que modela a execução da lógica deve executar mais um degrau.

Assim que o tempo de *scan* for alcançado será feito o processo de atualização das variáveis de saída (*update!*) e em seguida a avaliação desses dos seus valores, através das sincronizações *startEvaluation!* e *endEvaluation!* com o autômato da Figura 5.7

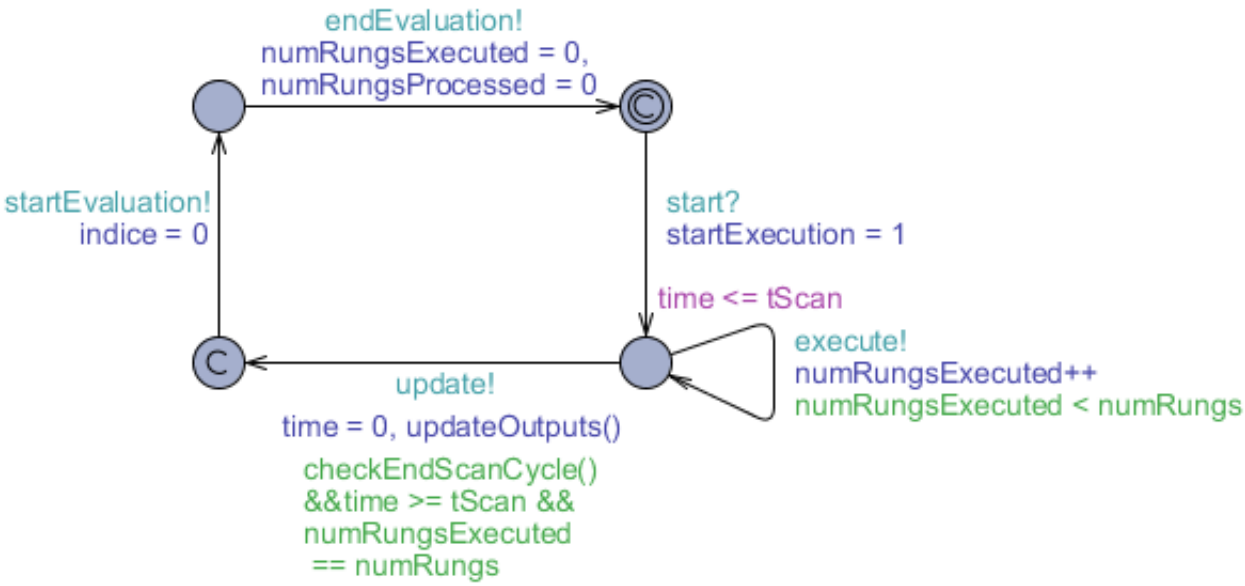


Figura 5.9: Modelo do ciclo de scan

Na Figura 5.10 é ilustrado o modelo da execução da lógica do programa, onde cada função do tipo *value_NomeDaSaida* avalia o valor de uma lógica de controle representada em um degrau do programa. O valor de cada saída de um degrau será determinada durante a execução deste autômato.

Caso exista um temporizador na lógica do degrau, o valor a ser liberado na saída dependerá da execução ou não deste autômato no momento em que ocorre a avaliação da lógica do degrau. A função *checkExecution* avalia se o temporizador deve ou não execu-

tar, e *evaluateOuput* determina o valor da saída do degrau após levar-se em consideração a saída do temporizador. Se não houverem temporizadores na lógica do programa a avaliação dos degraus segue o modelo apresentado na primeira aresta, ou seja, o valor das funções *value_NomeDaBobina()* determina a saída de cada degrau.

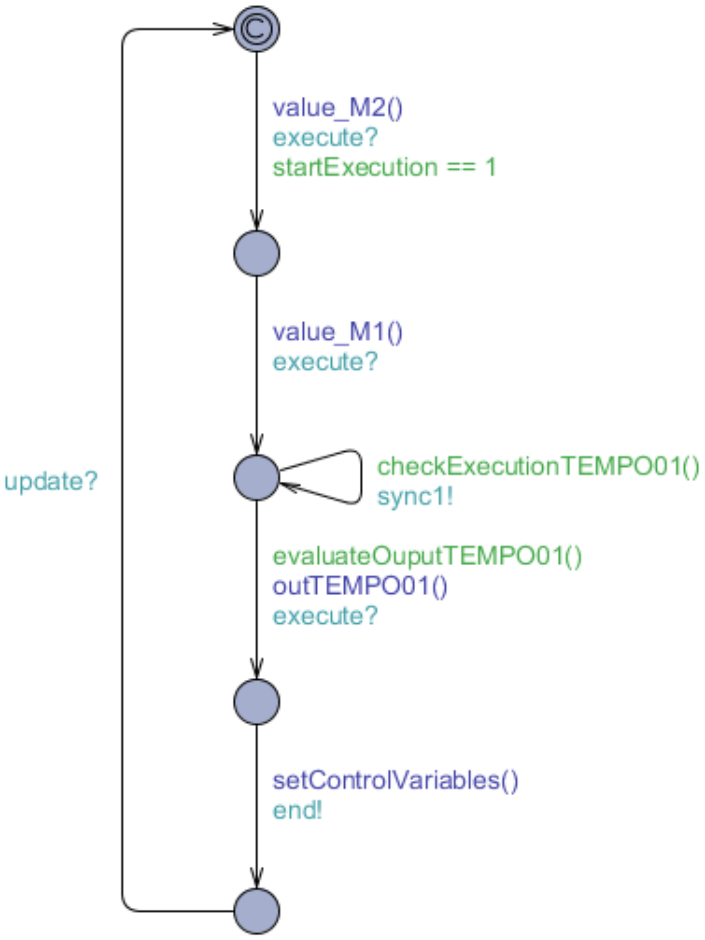


Figura 5.10: Modelo da execução da lógica do programa

5.3 UPPAAL-TRON e traços de execução

Os modelos de autômatos temporizados servem de entrada para a etapa de geração e execução automática de casos de teste. Nesta seção é explicado o funcionamento do UPPAAL-TRON assim como a saída fornecida pelo testador. O processo de geração e execução de casos de teste segue três etapas:

1. É gerada uma seqüencia de eventos de sincronização a partir do modelo de autômatos

temporizados UPPAAL criado a partir do diagrama ISA 5.2, que são especificados como entradas para o modelo. Ou seja, o próprio UPPAAL-TRON, executando em modo de monitoração das entradas, gera as seqüencias de eventos possíveis para o estado presente da rede de autômatos. Além do modelo de autômatos, é passada uma seqüencia de eventos de atraso (ou *delay*) ao testador durante a execução, simulando a passagem do tempo durante a simulação. Os eventos de atraso podem vir a disparar uma ação se o tempo decorrido for o suficiente para tal. Este processo gera um traço contendo os eventos usados como entradas para o modelo na simulação executada e os atrasos aceitos pelo modelo durante a execução.

2. A seguir o traço de execução do modo de monitoração, gerado anteriormente, é transformado em um traço de monitoração de saídas, ao invés de entradas, ou seja, todos os eventos utilizados como entrada para o modelo serão agora considerados saídas do modelo.

3. O UPPAAL-TRON recebe como entrada o modelo gerado a partir do diagrama Ladder, e agora é executado em modo de emulação, ou seja, são gerados eventos aleatórios para a rede passando-se como parâmetro os atrasos criados na primeira etapa, sendo então comparados os eventos de saída gerados com os eventos de saída registrados no traço de execução da segunda etapa. Se houver conformidade entre os diagramas não deve haver divergências nessa comparação, caso contrário o testador irá disparar uma exceção durante a execução da simulação (veredito FAIL), interromper a simulação e gerar um traço de erro descrevendo o fato.

No Apêndice A é ilustrado um traço de execução gerado pelo UPPAAL-TRON para o processo descrito neste trabalho. São especificados os parâmetros usados no teste, a seqüencia de eventos executada sobre o modelo e o detalhamento do erro, que para esta execução específica foi apontado pelo *software*.

Acompanhando-se a seqüencia de eventos é possível identificar o processo de atualização das variáveis de entrada (*set* e *reset*), o inicio da execução da lógica (*start*), execução dos degraus (*execute*) e fim da execução da lógica (*end*), o início da atualização das variáveis de saída (*update* e *startEvaluation*), a avaliação de cada saída individualmente (*high* ou *low*) e o fim da atualização das variáveis de saída (*endEvaluation*). Esta seqüencia de eventos se

repetirá até que o tempo de execução dos testes determinado (300 unidades de tempo) seja expirado ou algum erro ocorra.

De acordo com o evento responsável por um erro e seu número de seqüencia (*high*), é possível identificar qual seção do modelo de implementação apresentou um comportamento não esperado. Neste caso, uma saída cujo valor deveria ser 0 no entanto tinha valor 1.

5.4 Arquivo de rastreamento

Nesta seção é demonstrado como é feita a associação entre elementos Ladder, ISA 5.2 e eventos do traço no arquivo de rastreabilidade. Temporizadores são descritos pelas *tags* <timer> e *</timer>*. Nesta seção são descritos a que degrau/bloco lógico pertencem, se são negados ou não, linha e posição. Dentro da seção *trace* é descrito qual evento de sincronização (neste caso, *sync1*) é responsável por ativar a saída do temporizador, e em caso de erro este será o evento não esperado ou não disparado pelo modelo da implementação. Na Figura 5.11 é ilustrado um exemplo de representação de um temporizador.

```
<timer>
    <ladder>
        <name>TIMER1</name>
        <rung>1</rung>
        <line>2</line>
        <position>14</position>
        <negated>false</negated>
    </ladder>
    <isa>
        <name>PO212</name>
        <block>1</block>
        <negated>false</negated>
    </isa>
    <trace>
        <template>TON_timer1</template>
        <type>timer</type>
        <rung>1</rung>
        <syncronization>sync1</syncronization>
    </trace>
</timer>
```

Figura 5.11: Temporizador em um arquivo de rastreamento

Entradas e saídas são especificadas pelas tags *<input></input>* ou *<output></output>*, dentro das seções *<ladder>* e *<isa>* existem dados relativos à posição dos elementos (tags

<*rung*>, <*block*>, <*line*> e <*position*>). Na Figura 5.12 é ilustrado um exemplo de representação de uma entrada e uma saída.

```
<input>
    <ladder>
        <name>pb1</name>
        <rung>1</rung>
        <line>2</line>
        <position>10</position>
        <negated>false</negated>
    </ladder>
    <isa>
        <name>PB1</name>
        <rung>1</rung>
        <negated>false</negated>
    </isa>
    <trace>
        <template>PB1_Template</template>
        <type>input</type>
        <rung>1</rung>
    </trace>
</input>
(. . .)
<output>
    <ladder>
        <name>OUT2</name>
        <rung>1</rung>
        <line>2</line>
        <position>10</position>
        <negated>false</negated>
    </ladder>
    <isa>
        <name>M2</name>
        <block>1</block>
        <negated>false</negated>
    </isa>
    <trace>
        <variable>out_M2</variable>
        <type>output</type>
        <rung>1</rung>
    </trace>
</output>
```

Figura 5.12: Representação de entradas e saídas em um arquivo de rastreamento

A ocorrência de erros durante a fase de atualização das variáveis de entrada (eventos *set* e *reset* no traço de execução) é interpretada a partir das informações contidas nas tags <*input*>, identificando-se a variável de entrada responsável pela condição. O mesmo ocorre para o processo de atualização das saídas. Caso um evento *high* ou *low* não seja o esperado

pelo modelo da implementação, é possível identificar a que degrau pertence tal saída e seus demais atributos.

As lógicas de controle são armazenadas dentro das tags <*rung*>, e são utilizadas para permitir o detalhamento das causas do erro, de forma que uma entrada que não foi implementada no diagrama Ladder, por exemplo, possa ser facilmente identificada quando as lógicas de controle de especificação e implementação forem colocadas lado a lado na mensagem de erro. Na Figura 5.13 são representadas as lógicas de controle de um degrau (1, ou seja, o primeiro de um programa), onde é possível identificar a representação errônea (negada) da variável de entrada *TMR1* no diagrama Ladder.

```
<rung>
    <number>1</number>
    <ladder>
        <logic>((PB1 AND PB2) OR !TMR1)</logic>
    </ladder>
    <isa>
        <logic>((PB1 AND PB2) OR TMR1)</logic>
    </isa>
</rung>
```

Figura 5.13: Representação de duas lógicas de controle em um arquivo de rastreamento

Capítulo 6

Estudo de caso

A fim de avaliar o método proposto, a ferramenta desenvolvida foi submetida a dois estudos de casos descritos neste capítulo. Nas seções seguintes o comportamento dos sistemas propostos e os experimentos realizados são detalhados.

6.1 Sistema Instrumentado de Segurança

Para avaliar o método proposto foi usado um exemplo de Sistema de Segurança da Petrobras, que descreve um sistema responsável por manter estável a operação de uma plataforma particular de óleo após a detecção de fogo ou gás nas instalações.

O sistema é composto de dois detectores de fogo (SF1 e SF2), um tanque, três sensores de gás (SG1, SG2 e SG3), um dispositivo que libera gás CO_2 caso exista fogo na plataforma (DispCO_2), um alarme sonoro que indica a presença de fogo (AlaFDZ), um alarme sonoro que indica a presença de gás (AlaGDZ) e duas válvulas que bombeiam óleo do tanque. Uma das válvulas é chamada auxiliar (AuxiliaryValve), acionada apenas quando ocorre um vazamento de gás na válvula principal (Valve) que bombeia óleo para o tanque. A função da válvula auxiliar é manter o sistema estável e prevenir qualquer dano às instalações. O sistema opera da seguinte forma:

- Se algum dos detectores de fogo é acionado então eles detectaram a presença de fogo nas instalações.
- Se o sensor de gás 1 e o sensor de gás 2 ou 3 são acionados então foi detectada a

presença de gás nas instalações. Se o sensor 2 e o sensor 3 são acionados então foi detectada a presença de gás nas instalações.

- Se fogo for detectado nas instalações, então um alarme sonoro que indica a presença de fogo é acionado e após dois segundos o dispositivo que libera gás CO_2 é acionado e a válvula que bombeia óleo para o tanque é desligada. O gás CO_2 é liberado afim de extinguir o fogo na plataforma. O alarme sonoro é desligado quando não há mais fogo presente.

- Caso seja detectado gás na plataforma um alarme sonoro é disparado e após quatro segundos o sistema que supre óleo para o tanque é desligado e a válvula que mantém o sistema estável é aberta. Esta válvula bombeia óleo para o tanque se não existir dano às instalações da plataforma e desperdício de material. O alarme sonoro é desativado quando não há mais presença de gás.

Na Figura 6.1 é ilustrada uma instalação típica do sistema. Nas figuras 6.2 e 6.3 são apresentados, respectivamente, o diagrama ISA 5.2 e o programa Ladder do estudo de caso.

6.1.1 Erros introduzidos

Baseado nesses modelos três erros foram introduzidos no programa Ladder ilustrado na Figura 6.3 para avaliar a resposta do método de teste em relação a real causa da condição de falta:

- Erro 1: Mudar um contato normalmente aberto para um contato normalmente fechado, neste caso, no sexto degrau, a variável *GDZ* será representada por um contato normalmente fechado.

- Erro 2: Mudar a ordem de execução dos degraus, ou seja, o quarto degrau é executado primeiro e então é executado o segundo degrau.

- Erro 3: Mudar o valor de temporização (PT) do *Timer1* para 1 segundo (ou 1000 milisegundos).

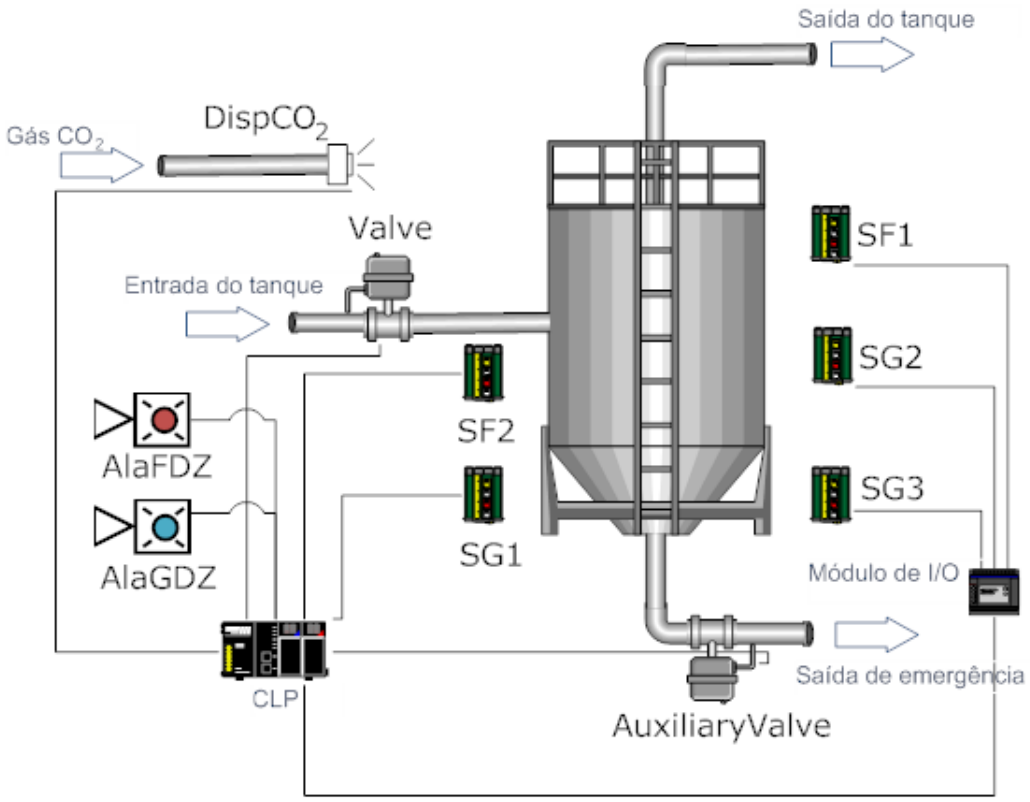


Figura 6.1: Configuração típica do Sistema de Segurança

6.1.2 Avaliação

Após a inserção do primeiro erro a ferramenta desenvolvida foi executada utilizando o diagrama ISA 5.2 como modelo de especificação e o diagrama Ladder alterado, como modelo da implementação. O traço de execução provido pelo TRON, como demonstrado a seguir, indica que uma falha ocorreu (veredito *FAIL*) porque uma saída *high()* foi esperada mas uma saída *low()* foi encontrada no tempo 20.

Em outras palavras, era esperado que o alarme de gás fosse ativado (*AlaGDZ = 1*) mas o mesmo não aconteceu. A reconstrução do traço (cujo trecho é exibido a seguir) indica que o valor de *GDZ* era *0*, então o valor de *AlaGDZ* deveria também ser 0. Uma explicação possível para a ocorrência do erro é a troca de contato normalmente aberto para fechado representando a entrada *GDZ*.

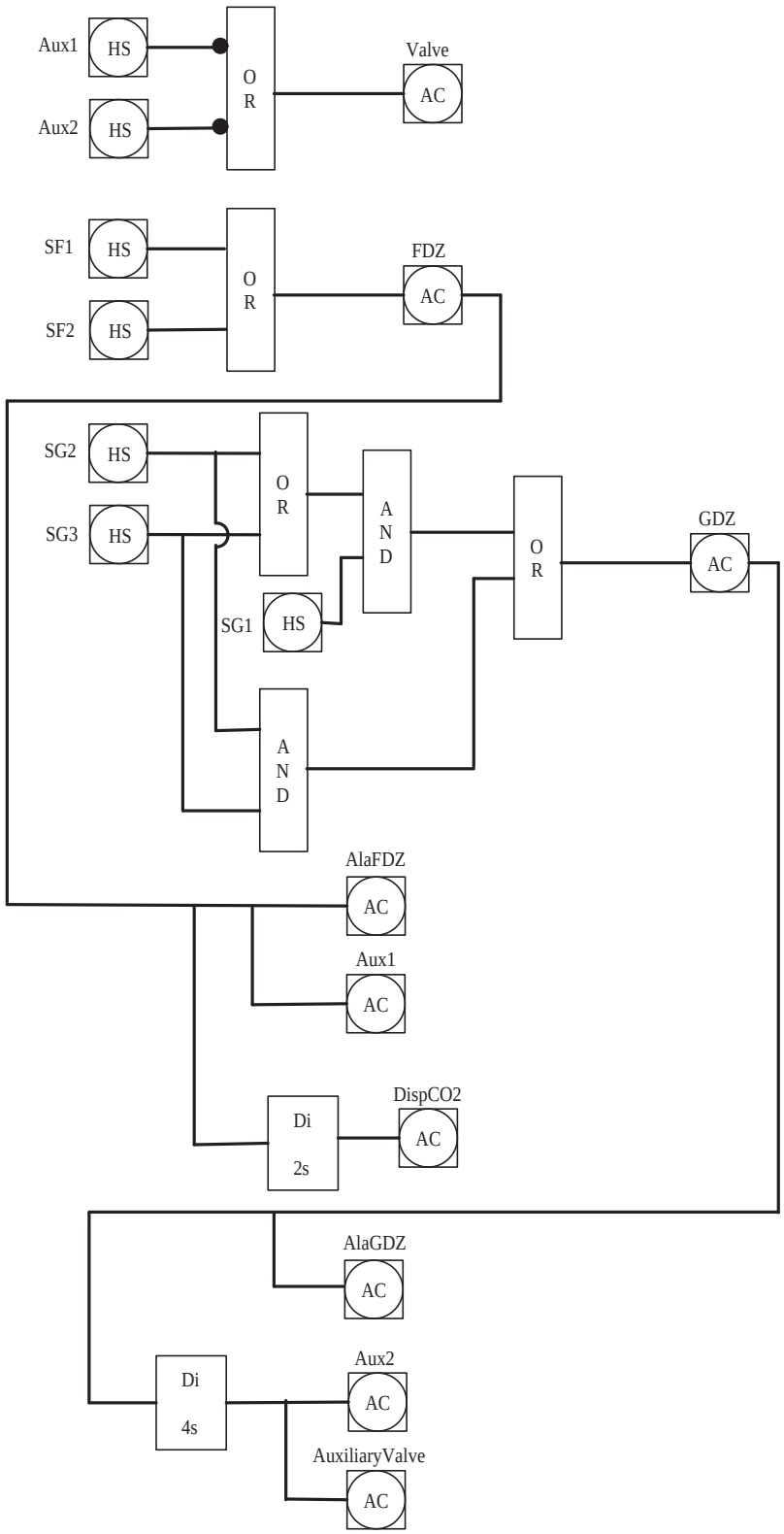


Figura 6.2: Diagrama ISA 5.2 do Sistema de Segurança

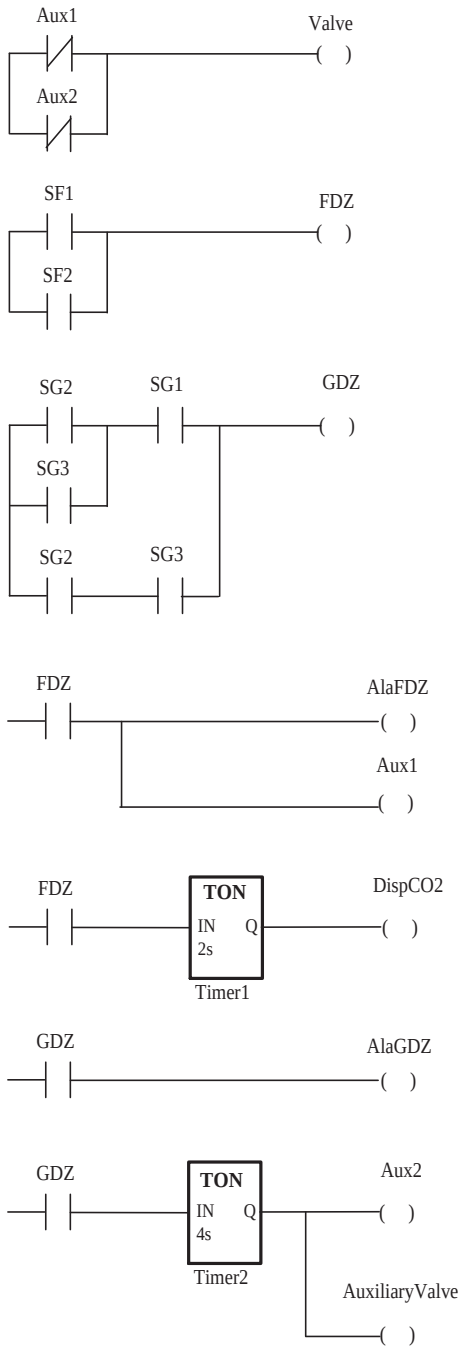


Figura 6.3: Programa Ladder do Sistema de Segurança

```
output high();
delay @20;
output low();
delay @20;
output low();

Short post-mortem analysis based on last good stateSet(1):
1)
( dump dos valores das variáveis internas do modelo e
 externas, por simplicidade, omitido )

 Options for input   : (empty)
 Options for output  : high@[40..41)
 Options for internal: (empty)
 Options for delay   : ..41)
 Last time-window    : [40..41)
Got unacceptable output: low()@20 at [40..41)
Expected outputs were: high()@0-0[40..41)

TEST FAILED: Observed unacceptable output
```

Esta informação, no entanto, é altamente dependente de como e quem lê este traço de execução. Conhecimento sobre a representação interna dos digramas como autômatos no modelo UPPAAL, treinamento em métodos formais e na operação da ferramenta UPPAAL-TRON seriam provavelmente necessários para alcançar este nível de entendimento. O método proposto adiciona uma etapa de refinamento após a execução dos testes de modo a permitir a interpretação automática da condição de erro, resultando em uma representação visual da mesma que é exibida ao usuário.

Na Figura 6.4 é ilustrada a saída do método proposto neste caso específico, que inclui o realce do elemento problemático identificado pelo caso de teste no diagrama Ladder (*GDZ*), o degrau ao qual pertence tal elemento (3) e uma mensagem textual descrevendo o problema.

Um trecho do traço gerado pelo UPPAAL-TRON na segunda execução é apresentado a seguir. O veredicto mais uma vez foi *FAIL*, uma saída *low* era esperada no entanto uma saída

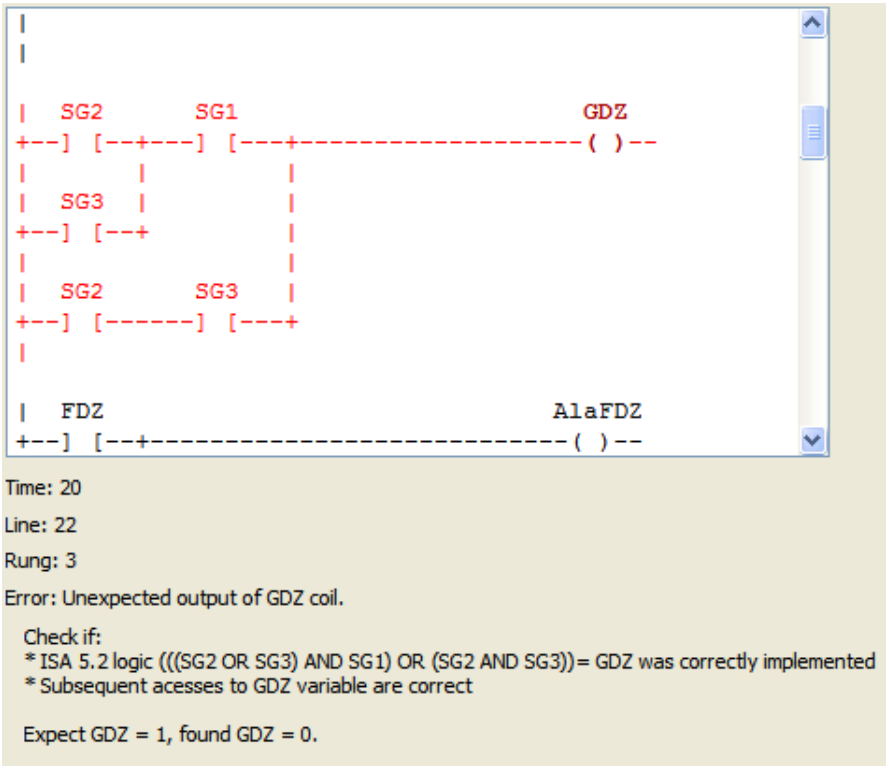


Figura 6.4: Tela de visualização do erro para a primeira execução

high foi provida no instante de tempo 20. Isto é, a saída do segundo degrau teve um valor não esperado, provavelmente devido à ordem errônea dos degraus. Mais uma vez esta informação só pode ser extraída por alguém com algum conhecimento sobre a operação do testador, mas o método proposto elimina essa necessidade. Baseado na implementação esperada da lógica de controle extraída do diagrama ISA 5.2, a mensagem de erro produzida pela ferramenta claramente indica que a lógica implementada não corresponde a esperada. Na Figura 6.5 é ilustrada a saída visual para esta execução.

```
output startEvaluation();
 delay @20;
 output low();
 delay @20;
 output high();
 Short post-mortem analysis based on last
 good stateSet(1):
 1)
 ( dump dos valores das variáveis internas do modelo e
```

```
 externas, por simplicidade, omitido )

 Options for input    : (empty)
 Options for output   : low@[40..41)
 Options for internal: (empty)
 Options for delay    : ..41)
 Last time-window     : [40..41)
Got unacceptable output: high()@20 at [40..41)
Expected outputs ere: low()@0-0[40..41)

TEST FAILED: Observed unacceptable output
delay @20;
```

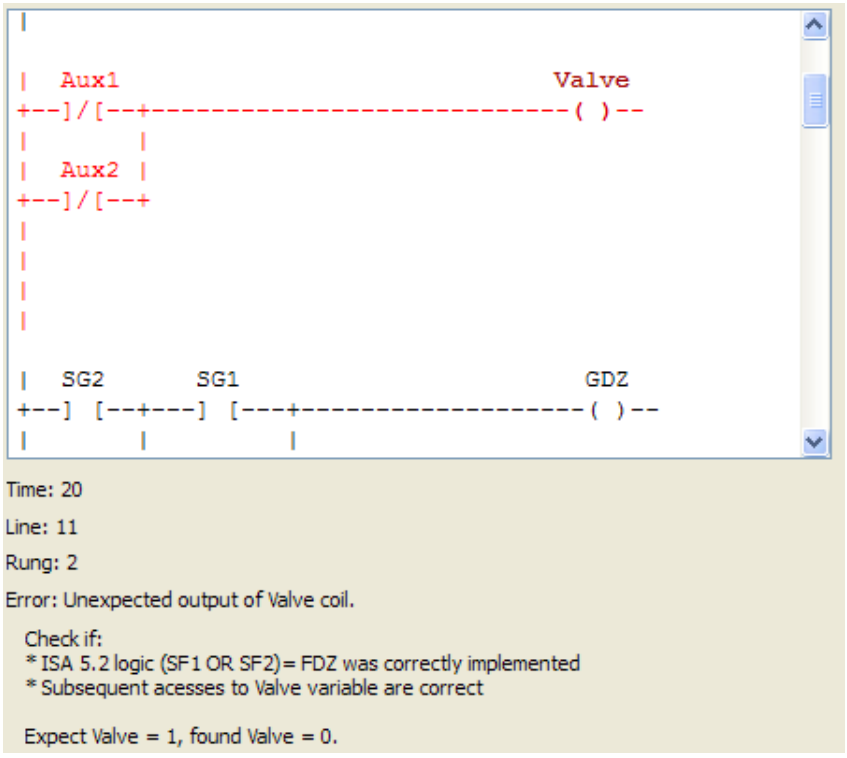


Figura 6.5: Tela de visualização do erro para a segunda execução

A terceira execução produziu um traço com um veredicto *FAIL* devido a um evento *sync1* inesperado (*Timer1* liberou sua saída antes do esperado, já que seu valor de temporização é menor que o descrito pelo arquivo ISA 5.2) no tempo 104. Na Figura 6.6 é ilustrada a saída visual provida pela ferramenta desenvolvida, e o trecho do traço que indica a condição de

erro é ilustrado a seguir.

```
output execute();
    delay @103;
    output execute();
    delay @104;
    output sync1();
    Short post-mortem analysis based on last
    good stateSet(1):
    ( dump dos valores das variáveis internas do modelo e
     externas, por simplicidade, omitido )

     Options for input   : (empty)
     Options for output  : execute@[208..241)
     Options for internal: (empty)
     Options for delay   : ..2147483646)
     Last time-window    : [208..209)
    Got unacceptable output: sync1()@104
    at [208..209)
    Expected outputs were: execute()@0-0[208..241)

    TEST FAILED: Observed unacceptable output
    delay @104;
```

6.2 Sistema de requisições concorrentes

Nesta seção é apresentado o problema ilustrado em [26] sobre o funcionamento de um sistema de controle de um jogo de perguntas e respostas implementado através de um CLP. Uma máquina de *quiz* é um equipamento utilizado em competições de perguntas e respostas, onde há um mediador responsável por iniciar e finalizar uma rodada, e vários jogadores que interagem com o sistema através de um botão e um indicador luminoso, sendo possível apenas um deles ser atendido em cada rodada. Ou seja, aquele que primeiro apertar seu botão

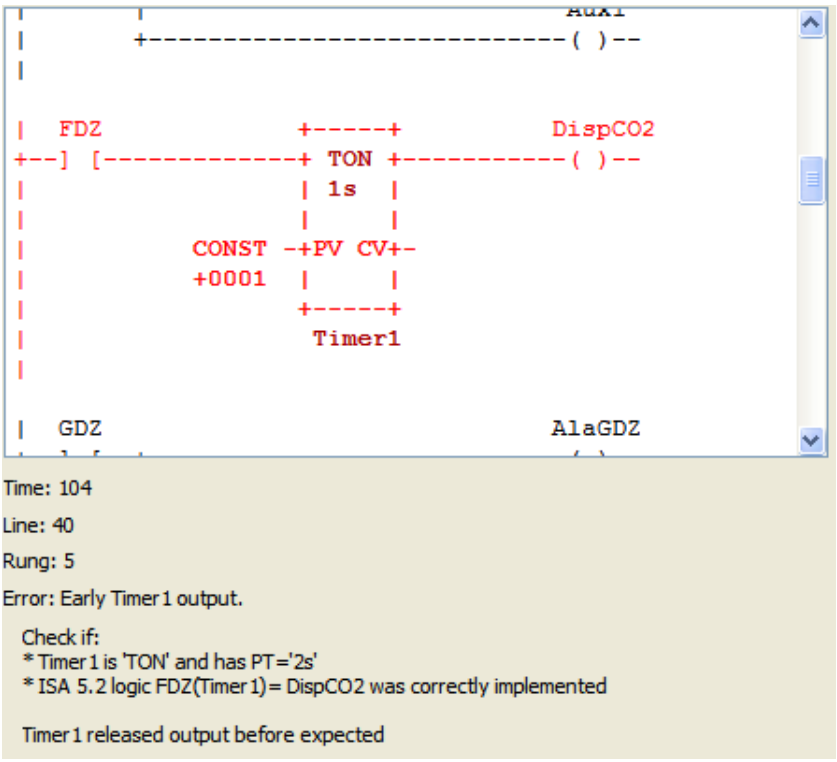


Figura 6.6: Tela de visualização do erro para a terceira execução

será o escolhido naquela rodada para responder a uma pergunta, o que é indicado pelo estado do indicador luminoso de cada participante.

O comportamento geral do sistema segue uma lógica *FIFO* (*first-in, first-out*), ou seja, o primeiro a solicitar será o primeiro a ser atendido. Lógica semelhante pode ser aplicada a outros tipos de sistema de decisão críticos, como controles de intersecções de ferrovias, onde apenas um usuário (trem) pode fazer o uso do canal (intersecção) durante um determinado tempo, cabendo a um mediador (sistema de controle) decidir qual deles deve ter acesso.

Após o início de uma rodada, o primeiro participante que pressionar seu botão dentro do tempo definido será o escolhido, sendo seu indicador luminoso energizado. Se mais de um participante pressionar seus respectivos botões ao mesmo tempo a máquina informa ao mediador o empate, requisitando um reinício de rodada. Se não houver botões pressionados pelos participantes durante o tempo definido para uma rodada o sistema informa ao mediador o ocorrido e mantém os indicadores luminosos de todos os participantes desenergizados.

O diagrama ISA 5.2 que descreve o comportamento do sistema é apresentado na Figura 6.7. Os botões de início e reinício de rodada são representados pelas entradas i0 e i1, respectivamente. As entradas i2, i3 e i4 representam cada botão dos três jogadores mode-

lados para o exemplo. As saídas o1, o2 e o3 são utilizadas para controlar as saídas l1, l2 e l3 ligadas aos respectivos indicadores luminosos de cada um dos jogadores. A saída o0 representa a ocorrência de uma rodada sem interessados ao fim do tempo estipulado.

A saída de m1 indica o início de uma nova rodada. A saída t1, ligada ao temporizador TIMER01, é utilizada para controlar o tempo decorrido após o início da rodada. As saídas m2, m3 e m4 determinam se os participantes play1, play2 ou play3 pressionaram seus respectivos botões e estão aptos a serem escolhidos. m5 indica se um dos participantes pressionou seu botão dentro do tempo pré-definido.

Três propriedades dependentes de tempo devem ser satisfeitas. Supondo o início de uma nova rodada, após o mediador ter pressionado o botão reset e logo após start, até o final do tempo estipulado:

- Se apenas um participante pressionar seu botão, seu indicador luminoso será energizado e os demais permaneceram desenergizados;
- Se pelo menos dois participantes pressionarem seus botões ao mesmo tempo, seus indicadores luminosos serão energizados e todos os demais permanecerão desenergizados antes que a rodada seja reiniciada;
- Se nenhum dos participantes pressionar seu botão o indicador de fim de tempo será energizado e os demais indicadores permanecerão desenergizados.

Na Figura 6.8 o programa Ladder que implementa o comportamento esperado do sistema é ilustrado.

6.2.1 Erros introduzidos

Como no estudo de caso anterior, foram produzidas três versões falhas do programa Ladder ilustrado na Figura 6.8 a fim de avaliar a saída obtida a partir da execução da ferramenta proposta. Em cada uma destas versões faltosas foram inseridos os seguintes erros:

- Erro 1: Representar uma operação lógica E como OU, ou seja, ao invés de dispor os elementos em série no diagrama Ladder os dispor em paralelo. O erro foi inserido no décimo degrau, onde a lógica correta seria (t1 and (not m5)), no entanto o implementado foi (t1 or (not m5)).

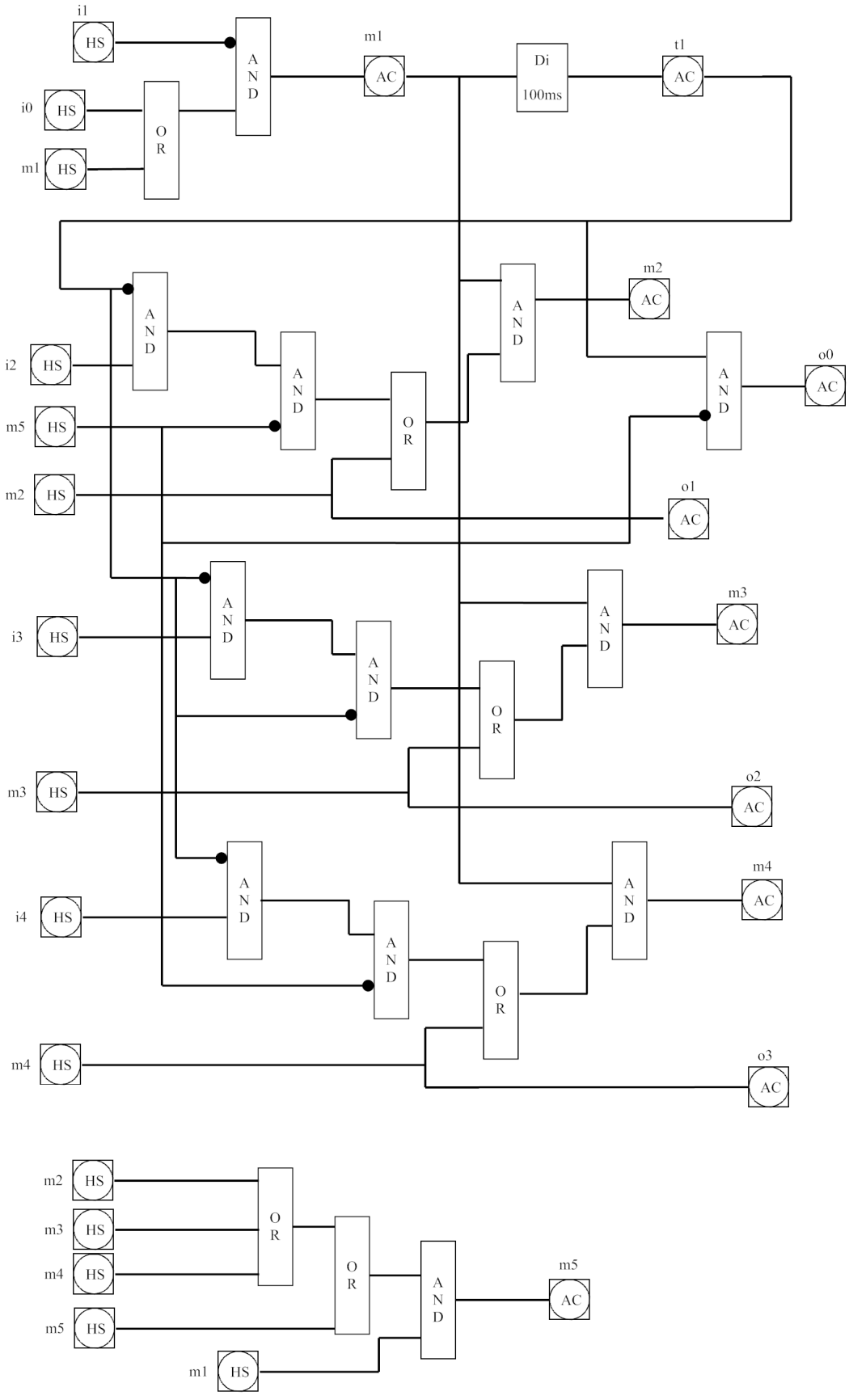


Figura 6.7: Diagrama ISA 5.2 do sistema de quiz

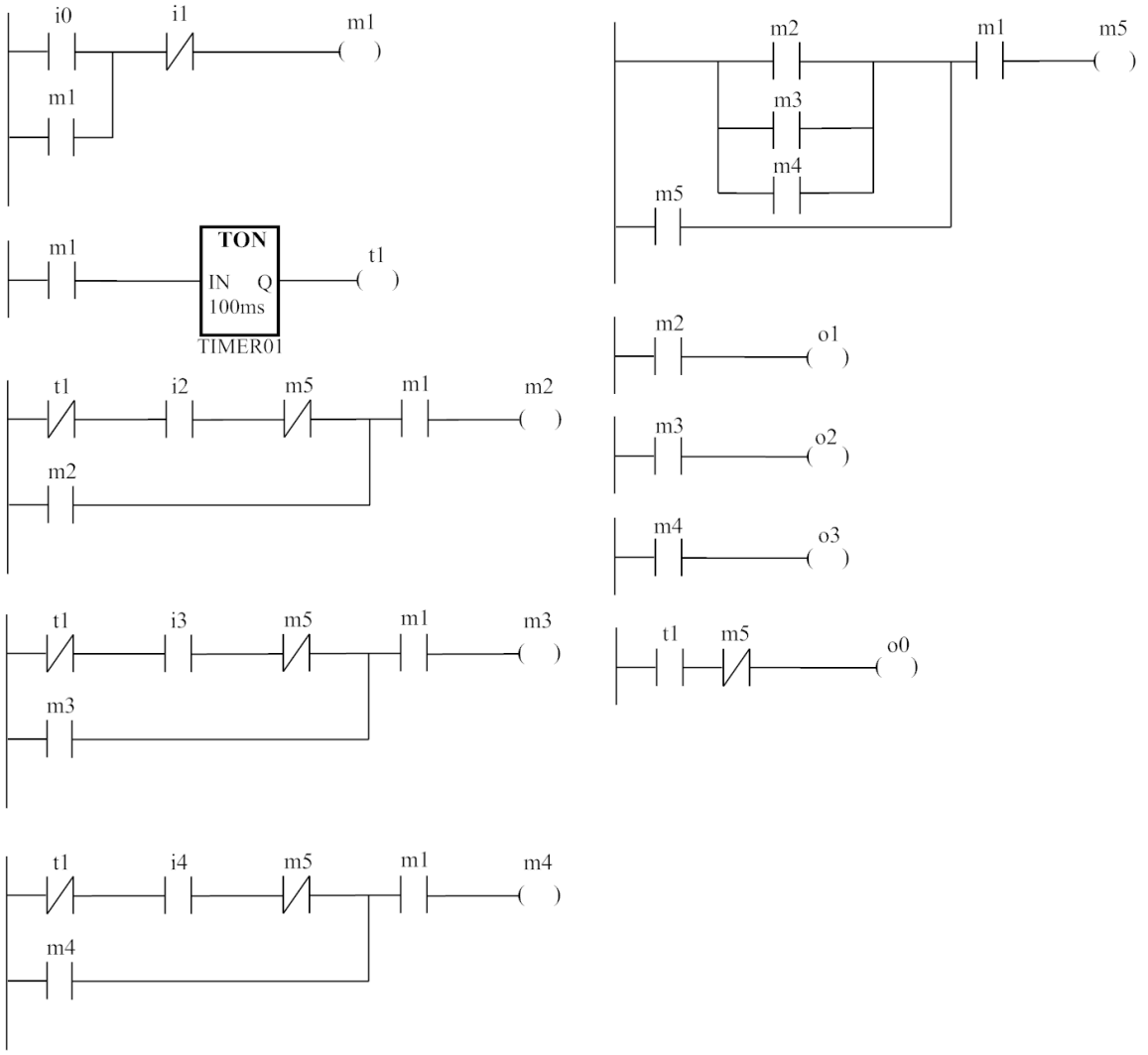


Figura 6.8: Diagrama Ladder do sistema de *quiz*

- Erro 2: Remover um degrau do programa, no caso o sexto degrau.
- Erro 3: Trocar um contato normalmente aberto (m1) por um normalmente fechado (not m1) no quarto degrau.

6.2.2 Avaliação

Após a inserção do primeiro erro a ferramenta desenvolvida foi executada utilizando o diagrama ISA 5.2 original como modelo de especificação e o diagrama Ladder faltoso como implementação. O traço de execução provido pelo UPPAAL-TRON, como demonstrado no trecho a seguir, indica que uma falha ocorreu (veredito *FAIL*) porque uma saída *high()* foi esperada mas uma saída *low()* foi encontrada no tempo 40. Era esperado que para um valor 1 de t1 e 1 de m5 a bobina o0 assumisse valor 0, no entanto com o programa faltoso o valor assumido por o0 seria 1.

```
(...)
output startEvaluation();
delay @40;
output low();
delay @40;
output low();
delay @40;
output low();
delay @40;
output low();
delay @40;
output low();
delay @40;
output low();
delay @40;
output low();
delay @40;
output low();
delay @40;
output low();
delay @40;
output low();
delay @40;
output low();
```

```
Short post-mortem analysis based on last good stateSet(1):
( dump dos valores das variáveis internas do modelo e
     externas, por simplicidade, omitido )

  Options for input    : (empty)
  Options for output   : high()@[40;40]
  Options for internal: (empty)
  Options for delay    : until 40]
  Last time-window     : [40;40]
Got unacceptable output: low()@40us at [40;40]
Expected outputs were: high()@[40;40]

TEST FAILED: Observed unacceptable output.
```

Na Figura 6.9 é ilustrada a saída provida pela ferramenta criada após a execução do ciclo de testes com o primeiro programa Ladder faltoso. Nela é possível identificar o degrau problemático identificado (décimo), assim como a mensagem de erro explicativa provida.

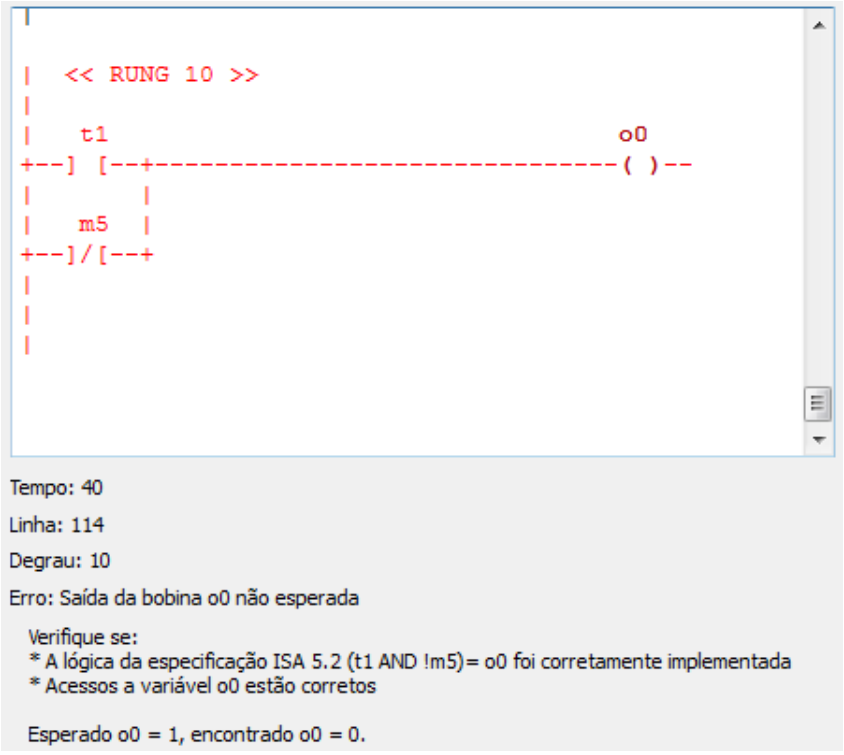


Figura 6.9: Tela de visualização do erro para a primeira execução

Um trecho do traço gerado pelo UPPAAL-TRON na segunda execução é apresentado a seguir. O veredicto mais uma vez foi *FAIL*, uma saída *set* ou *reset* era esperada no entanto uma saída *start* foi provida no instante de tempo 0, isto é, a implementação necessitava de mais um evento de atribuição para uma variável de entrada, no entanto a especificação não permite tal entrada, requisitando o início do ciclo de *scan*. Tal ocorrência se deve ao fato do degrau 6 representar a variável m5 como *feedback*, como tal degrau não existe no programa Ladder defeituoso o modelo gerado a partir da implementação interpretou a variável m5 como entrada simples, por isso requisitou um evento de atribuição a mais do que a especificação.

```
TEST in progress | 0\%
delay @0;
output reset();
delay @0;
output reset();
delay @0;
output set();
delay @0;
output reset();
delay @0;
output set();
delay @0;
output start();
Short post-mortem analysis based on last good stateSet(1):
( dump dos valores das variáveis internas do modelo e
     externas, por simplicidade, omitido )

  Options for input   : (empty)
  Options for output  : set()@[0;0], reset()@[0;0]
  Options for internal: (empty)
  Options for delay   : until 0]
  Last time-window    : [0;0]
Got unacceptable output: start()@0us at [0;0]
Expected outputs were: set()@[0;0], reset()@[0;0]
```

```
TEST FAILED: Observed unacceptable output.
```

A interpretação do erro provida pela ferramenta, como ilustrado na Figura 6.10 destaca a última ocorrência da variável m5, que foi erroneamente representada no modelo de implementação como uma entrada simples (através de um *template*), e exibe uma mensagem explicativa sobre a possível causa do erro.

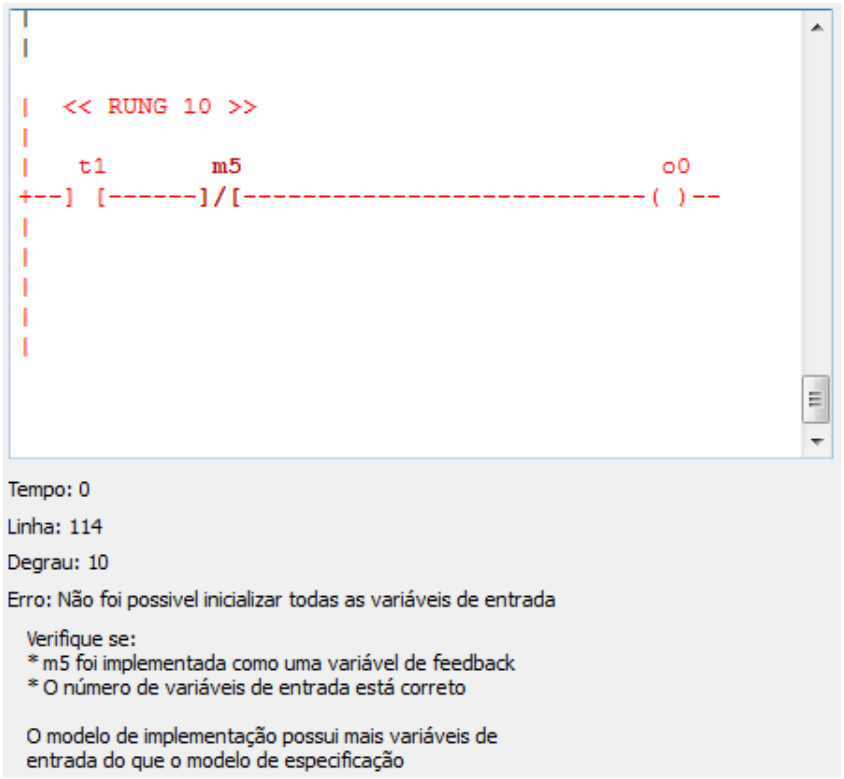


Figura 6.10: Tela de visualização do erro para a segunda execução

Por fim, a terceira execução produziu um traço com um veredicto *FAIL* devido a um evento *low* inesperado no tempo 200, sendo este evento relacionado a saída do quarto degrau. O trecho do traço que indica a condição de erro é ilustrado abaixo.

```
(...)
TEST in progress \ 5\%output execute();
delay @167;
output execute();
delay @168;
output execute();
delay @169;
output execute();
delay @170;
output execute();
delay @171;
output end();
delay @200;
output update();
delay @200;
output startEvaluation();
delay @200;
output low();
delay @200;
output low();
delay @200;
output low();
delay @200;
output low();
Short post-mortem analysis based on last good stateSet(1):
( dump dos valores das variáveis internas do modelo e
     externas, por simplicidade, omitido )

  Options for input   : (empty)
  Options for output  : high()@[200;200]
  Options for internal: (empty)
  Options for delay   : until 200]
  Last time-window    : [200;200]
Got unacceptable output: low()@200us at [200;200]
```

```
Expected outputs were: high()@[200;200]

TEST FAILED: Observed unacceptable output.
delay @200;
```

Na Figura 6.11 é ilustrada a saída visual provida pela ferramenta desenvolvida. A inversão do valor de m1 na lógica implementada no degrau 10 ocasionou o erro, por isso a mensagem solicita a checagem da lógica implementada em relação a lógica especificada no diagrama ISA 5.2. Ocasionalmente outros acessos incorretos a variável m3 podem gerar o mesmo problema, por isso é solicitada a verificação destes demais acessos.

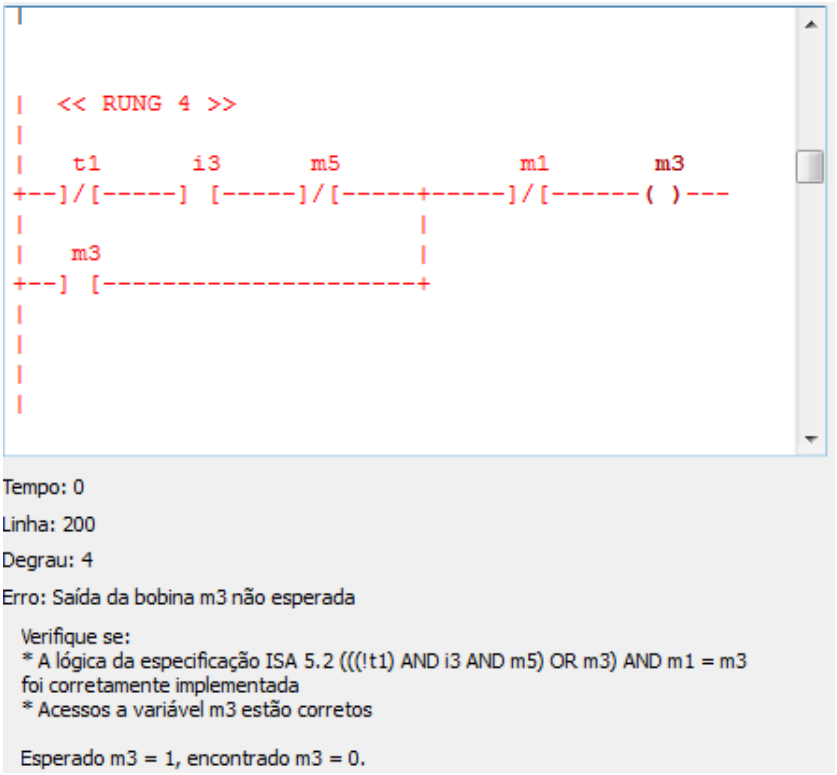


Figura 6.11: Tela de visualização do erro para a terceira execução

Ao fim da execução dos testes descritos para os dois estudos de caso desenvolvidos com a ferramenta foi possível observar a correta identificação das causas dos erros, assim como a exibição de mensagens textuais condizentes com as condições de falta.

Capítulo 7

Conclusões e trabalhos futuros

Neste trabalho é apresentado um método de validação visual de programas Ladder para Controladores Lógico Programáveis. O foco é ajudar na validação de programas para CLPs no contexto de Sistemas Instrumentados de Segurança. A principal contribuição é a definição de um método que se beneficia da teoria formal envolvida na geração automática de testes de conformidade a partir da especificação e implementação, sem demandar um conhecimento profundo destes conceitos para que seja possível operar a ferramenta desenvolvida, cuja interface é de baixa complexidade.

A utilização do método e da ferramenta introduzidos neste trabalho provêem uma forma intuitiva de se realizar testes de conformidade de software, no sentido de que escondem detalhes de implementação, permitindo que código e especificação possam ser confrontados ainda nos estágios iniciais do desenvolvimento da lógica de controle de sistemas de segurança. Erros podem ser facilmente identificados e interpretados por programadores, engenheiros e outros técnicos envolvidos. A partir dessa rápida interação, partes problemáticas do código podem ser isoladas após serem identificadas pelos testes gerados e executados automaticamente.

As principais dificuldades encontradas durante o desenvolvimento do trabalho são relativas à precisão dos vereditos extraídos a partir dos traços gerados pela ferramenta de testes UPPAAL-TRON, que algumas vezes gera falsos-positivos e não é capaz de prover informações acerca do estados das variáveis presentes nos modelos com confiança.

7.1 Trabalhos futuros

Ao final do trabalho foi possível identificar algumas possíveis extensões ao método de teste desenvolvido:

- Aumentar o suporte a outros elementos da linguagem Ladder não abordados, como funções, por exemplo, contemplando uma maior gama de componentes utilizados em aplicações industriais;
- Prover a importação de diagramas ISA 5.2 a partir formatos de arquivo proprietários de ferramentas comumente utilizadas na indústria, como editores CAD;
- Utilização de uma nova ferramenta de teste, capaz de prover maior detalhamento sobre a execução dos testes, a causa dos erros, os valores das variáveis durante a simulação, permitindo a reprodução passo a passo dos caso de teste que venham a provocar erros.
- A priorização de casos de teste através do uso de heurísticas, com o objetivo de melhorar a cobertura dos testes e conseqüentemente o grau de confiabilidade nos vereditos reportados.

Bibliografia

[1] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, PLDI '90, pages 246–256, New York, NY, USA, 1990. ACM.

[2] Rajeev Alur. Timed Automata. *Theoretical Computer Science*, 126:183–235, 1999.

[3] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[4] Darlam Fabio Bender, Benoît Combemale, Xavier Crégut1, Jean Marie Farines, Bernard Berthomieu, and François Vernadat. Ladder metamodeling and PLC program validation through time petri nets. In *ECMDA-FA '08: Proceedings of the 4th European conference on Model Driven Architecture*, pages 121–136, Berlin, Heidelberg, 2008. Springer-Verlag.

[5] Luis A. Bryan and E. A. Bryan. *Programmable Controllers: Theory and Implementation*. Industrial Text Company, 1997. Illustrator-Kory, Gina.

[6] Houshang Darabi, Rupa Sampath, and David Naylor. PLC formal control methodologies: Does academia supply what industry demands? Technical Conference Paper, International Society of Automation, 2002.

[7] Luiz Paulo de Assis Barbosa, Kyller Gorgonio, Leandro Dias da Silva, Antonio Marcus Nogueira Lima, and Angelo Perkusich. On the automatic generation of timed automata models from isa 5.2 diagrams. *Emerging Technologies and Factory Automation, 2007. ETFA. IEEE Conference on*, pages 406–412, Sept. 2007.

[8] Anton A. Frederickson and Dr Mr. Functional safety - safety instrumented systems for the process industry sector. *IEC std 61511*, 2003.

[9] G. Frey. Automatic implementation of petri net based control algorithms on plc. In *American Control Conference, 2000. Proceedings of the 2000*, volume 4, pages 2819 –2823 vol.4, 2000.

[10] V. Gourcuff, O. De Smet, and J. M. Faure. Efficient representation for formal verification of PLC programs. In *Discrete Event Systems, 2006 8th International Workshop on*, pages 182–187, 2006.

[11] Liang Guo and Abhik Roychoudhury. Debugging statecharts via model-code traceability. In Tiziana Margaria and Bernhard Steffen, editors, *ISoLA*, volume 17 of *Communications in Computer and Information Science*, pages 292–306. Springer, 2008.

[12] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8:231–274, June 1987.

[13] Anders Hessel, Kim Larsen, Marius Mikucionis, Brian Nielsen, Paul Pettersson, and Arne Skou. Testing real-time systems using uppaal. In Robert Hierons, Jonathan Bowen, and Mark Harman, editors, *Formal Methods and Testing*, volume 4949 of *Lecture Notes in Computer Science*, pages 77–117. Springer Berlin / Heidelberg, 2008.

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

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

[16] Minsuk Ko, San-Chul Park, and Gi-Nam Wang. Visual validation of PLC programs. In *22nd European Conference on Modeling and Simulations (ECMS)*, pages 410–415, Nicosia, Cyprus, June 2008.

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

[18] T. Lecomte, T. Servat, and G. Pouzancre. Formal methods in safety-critical railway systems. In *Proceedings of Brazilian Symposium on Formal Methods: SMBF 2007*, August 2007.

[19] Kézia Oliveira, Angelo Perkusich, Antônio Marcus Nogueira e Lima, Kyller Gorgonio, and Leandro Dias da Silva. Standard-based Formal Validation of Programmable Logic Controller Programs. In *IEEE-ICIT 2010: International Conference on Industrial Technology*, pages 1655 –1660, Viña Del Mar, Chile, March 2010.

[20] E. A. Parr. *Programmable Controllers An engineer´s guide*. Newnes, 3nd edition, 2003.

[21] Bernhard Peischl and Franz Wotawa. Error traces in model-based debugging of hardware description languages. In *AADEBUG'05: Proceedings of the sixth international symposium on Automated analysis-driven debugging*, pages 43–48, New York, NY, USA, 2005. ACM.

[22] PLCopen. *IEC 61131-3: a standard programming resource. In http://www.plcopen.org/.* PLCopen For Efficiency in Automation, 2004.

[23] Andre Sulflow and Rolf Drechsler. Verification of PLC programs using formal proof techniques. *Formal Methods for Automation and Safety in Railway and Automotive Systems (FORMS/FORMAT 2008)*, pages 43–50, 2008.

[24] Devinder Thapa, Chang Park, Sang Park, and Gi-Nam Wang. Auto-generation of iec standard plc code using t-mpsg. *International Journal of Control, Automation and Systems*, 7:165–174, 2009. 10.1007/s12555-009-0202-z.

[25] J. Thieme and H.-M. Hanisch. Model-based generation of modular PLC code using iec61131 function blocks. In *Industrial Electronics, 2002. ISIE 2002. Proceedings of the 2002 IEEE International Symposium on*, volume 1, pages 199 – 204 vol.1, 2002.

[26] Hai Wan, Gang Chen, Xiaoyu Song, and Ming Gu. Formalization and verification of plc timers in coq. *Computer Software and Applications Conference, Annual International*, 1:315–323, 2009.

[27] M. Bani Younis and G. Frey. Formalization of existing plc programs: A survey. In *Proceddings of CESA*, France, July 2003.

Apêndice A

Detalhamento de um traço de execução

Neste apêndice é apresentado um traço de execução gerado pela ferramenta UPPAAL-TRON em detalhes. A saída textual da ferramenta é exibida abaixo:

```
(...)

Options for the UPPAAL engine:
  Search order is breadth first
  Using no space optimisation
  State space representation uses minimal constraint systems
  Observation uncertainties: 0, 0, 0, 0 (microseconds).
  Future precomputation: closure(0 mtu).
  Input delay extended by: 0
  OS scheduler: non-real-time.
  Environment processes: stateSignalOutputs, signalsInput, input1, input2, input3,
  input4, timer1, timer2.
  Timeunit: 1us
  Timeout: 300mtu

TEST in progress | 0\%delay @0;
output set();
delay @0;
output reset();
delay @0;
output reset();
```

```
delay @0;
output reset();
delay @0;
output start();
delay @0;
output execute();
delay @1;
output execute();
delay @2;
output execute();
delay @3;
output execute();
delay @4;
output execute();
delay @5;
output execute();
delay @6;
output end();

TEST in progress / 30\%delay @100;
output update();
delay @100;
output startEvaluation();
delay @100;
output high();
delay @100;
output low();
delay @100;
output low();
delay @100;
output low();
delay @100;
output low();
delay @100;
output low();
delay @100;
output endEvaluation();
```

```
(...)

Short post-mortem analysis based on last good stateSet(1):
1)
( stateOutputs._id28 stateSignalOutputs._id29 cycleProgram._id15
 scanCycle._id23 updateInputs._id30 signalsInput._id4 input1._id0
 input2._id1 input3._id2 input4._id3 timer1._id14 timer2._id5 )
#t>=200, time<=0, time-#t<=-200, #t<=200, #t-time<=200 PB1=1 PB2=0
 LS=0 PE=1 startExecution=0 indice=1 numRungsProcessed=6 M2=1 M1=1
 TMR1=0 SOL=0 TMR2=0 Bottle=0 out_M1=1 out_SOL=0 outputs[0]=1
 outputs[1]=0 outputs[2]=0 outputs[3]=0 outputs[4]=0 outputs[5]=0
 numInputRead=1 numRungsExecuted=6 id=0 numOutputs=6 tScan=100
 numCycles1=0 numCycles2=6 control1=0 control2=0 out_Timer1=0
 out_Timer2=0

  Options for input   : (empty)
  Options for output  : low@[400..401)
  Options for internal: (empty)
  Options for delay   : ..401)
  Last time-window    : [400..401)
Got unacceptable output: high()@200 at [400..401)
Expected outputs were: low()@0-0[400..401)

TEST FAILED: Observed unacceptable output
delay @200;
```