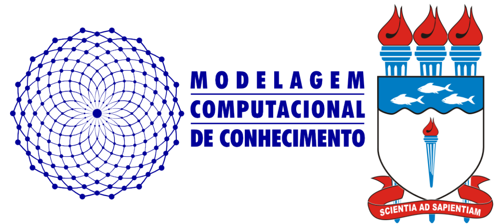


Dissertação de Mestrado

Gungnir - Uma Ferramenta para Geração e Execução Automática de Testes de Conformidade Utilizando Autômatos Temporizados

Rodrigo José Sarmento Peixoto
rodrigopex@gmail.com

Orientador:

Leandro Dias da Silva

Maceió, outubro de 2010

Rodrigo José Sarmento Peixoto

Gungnir - Uma Ferramenta para Geração e Execução Automática de Testes de Conformidade Utilizando Autômatos Temporizados

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre pelo Curso de Mestrado em Modelagem Computacional de Conhecimento do Instituto de Computação da Universidade Federal de Alagoas.

Orientador:

Leandro Dias da Silva

Maceió, outubro de 2010

Catalogação na fonte
Universidade Federal de Alagoas
Biblioteca Central
Divisão de Tratamento Técnico
Bibliotecária Responsável: Helena Cristina Pimentel do Vale

P379g Peixoto, Rodrigo José Sarmento.
Gungnir : uma ferramenta para geração e execução automática de testes de conformidade utilizando autômatos temporizados / Rodrigo José Sarmento Peixoto, 2010.
xi, 64 f. : il.

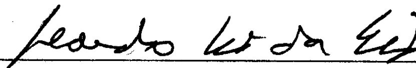
Orientador: Leandro Dias da Silva.
Dissertação (mestrado em Modelagem Computacional de Conhecimento) – Universidade Federal de Alagoas. Instituto de Computação. Maceió, 2010.

Bibliografia: f. 50-52.
Apêndices: 53-64.

1. Software – Sistema de coleta de dados. 2. Modelos computacionais – Testes. 3. autômatos temporizados. 4. Sistemas de controle. I. Título.

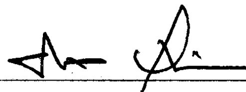
CDU: 004.028

Dissertação apresentada como requisito parcial para a obtenção do grau de Mestre em Modelagem Computacional de Conhecimento pelo Programa Multidisciplinar de Pós-Graduação em Modelagem Computacional de Conhecimento, da Universidade Federal de Alagoas, aprovada pela comissão examinadora que abaixo assina:



Prof. Dr. Leandro Dias da Silva
UFAL – Instituto de Computação
Orientador

Prof. Dr. Patrick Henrique da Silva Brito
UFAL – Instituto de Computação
Examinador



Prof. Dr. Ângelo Perkusich
UFCG – Departamento de Engenharia Elétrica
Examinador

Maceió, outubro de 2010.

Agradecimentos

Agradeço a Deus por todas as oportunidades e dificuldades, pois é a partir delas que construo meu futuro e posso crescer.

Agradeço também aos meus pais por todos ensinamentos e exemplos, pois deles formei e formo meu caráter. Devo agradecer também pelas condições que me foram dadas, na conjuntura atual, para estudar.

Agradeço aos inimigos. Não tem nenhum que eu saiba... Se alguém conhecer, avise-o que citei-o aqui, quem sabe até deixemos de ser inimigos?

Agradeço aos amigos do coração. Aqueles que sabem que são meus amigos, me ajudam e estão comigo para o que der e vier. E mais uma vez agradeço a Deus por ter encontrado alguns e dentre eles estarem meus irmãos.

Não posso deixar de fora os amigos de farra, que as vezes são tão e tão pouco freqüentes na minha vida. Dentre eles estão as amigas e "amigas".. "Quem não bebe não tem história pra contar"*Alguém...* E mais uma vez a Deus por dentre eles ter amigos do coração.

Aos amigos eventuais do dia-a-dia.

À família não precisa, os que merecem agradecimentos já foram citados nos trechos citados acima.

Acho que é isso.. e mais uma vez obrigado por tudo! Como é hora de agradecer, obrigado a você que está lendo se você achar que não está em nenhuma das "classes"citadas.

Rodrigo J. S. Peixoto

Resumo

O objetivo neste trabalho é aumentar a confiança no funcionamento de sistemas da automação através do uso de uma ferramenta de geração e execução automática de testes de conformidade. A ferramenta desenvolvida chama-se Gungnir e utiliza modelos formais, cujo padrão utilizado é o formalismo de Autômato Temporizado (AT). Os sistemas de controle são constituídos por Controladores Lógicos Programáveis (CLP) e normalmente são desenvolvidos nas linguagens Ladder e *Function Block Diagram* (FBD). A atividade da Gungnir é verificar se a implementação do sistema de controle desenvolvida na linguagem Ladder é compatível com a especificação modelada utilizando o padrão ISA 5.2. Para isso são utilizadas ferramentas de tradução de programas Ladder e diagramas ISA 5.2 para modelos de AT, definidos critérios de cobertura e criadas heurísticas as quais asseguraram menor custo computacional durante a execução dos testes.

Abstract

The aim of this work is to increase the dependability of automation systems through the use of a tool for automatic generating and executing conformance tests. The developed tool called Gungnir uses formal models to perform its actions, whose standard used is the formalism of Timed Automata (TA). The control systems consists of programmable logic controllers (PLC) and are often developed with Ladder and Function Block Diagram (FBD) languages. The Gugnir's key activity is to verify if the implementation of the control system developed in Ladder is compatible with the specification defined using the ISA 5.2 standard. To do so we used translation tools (from Ladder and ISA 5.2. to TA models), define coverages criteria and heuristics to ensure that the model was well tested.

Sumário

1 Introdução **1**
1.1 Descrição do problema . . . 2
1.2 Objetivo do trabalho . . . 3
1.3 Estrutura do Documento . . . 4

2 Fundamentação teórica **5**
2.1 Controlador lógico programável - CLP . . . 5
2.1.1 Detalhes da execução de CLPs . . . 7
2.1.2 Linguagens . . . 8
2.2 Autômatos temporizados . . . 8
2.2.1 Guardas . . . 10
2.2.2 Eventos . . . 10
2.2.3 Atribuições . . . 10
2.3 Testes baseados em Modelos . . . 11
2.3.1 Vantagens . . . 14
2.3.2 Desvantagens . . . 15
2.4 Trabalhos relacionados . . . 16
2.4.1 UPPAAL TRON . . . 16
2.4.2 Método para automatização de testes caixa preta . . . 18
2.4.3 Geração de dados de testes utilizando autômatos temporizados de Moore 21
2.4.4 Geração de testes caixa-preta baseada nos requisitos . . . 21
2.4.5 Geração de testes a partir de modelos formais utilizando o critério de cobertura de fronteira . . . 22

3 Ferramenta Gungnir **24**
3.1 Autômatos temporizados . . . 26
3.2 Testes baseados em modelos . . . 28
3.2.1 Modelos . . . 29
3.2.2 Geração de Modelos de autômatos temporizados . . . 31
3.2.3 Modelagem do ambiente . . . 32
3.2.4 Geração dos casos de testes . . . 34
3.2.5 Execução dos casos de testes . . . 37

4 Estudo de caso **39**
4.1 Utilizando o Gungnir . . . 39
4.2 Geração dos modelos . . . 41
4.3 Geração das entradas . . . 43
4.4 Execução dos testes . . . 44
4.4.1 Erro 1 . . . 45

4.4.2 Erro 2 46
4.4.3 Erro 3 46

5 Conclusões **48**
5.1 Trabalhos futuros 49

Referências **52**

Lista de Figuras

1.1 Processo de desenvolvimento da Petrobras. 2

2.1 Diagrama conceitual da aplicação de um CLP. 5
2.2 Arquitetura de um CLP. 6
2.3 Representação gráfica do *Scan Cycle*. 7
2.4 Representação interna do *Scan Cycle*. 7
2.5 Representação gráfica de uma realimentação. 8
2.6 Subconjunto de elementos do padrão ISA 5.2: (a) porta lógica *e*, (b) porta lógica *ou*, (c) temporizador DI, (d) chave de entrada e (e) acumulador de saída. 9
2.7 Subconjunto de elementos de programas Ladder:(a) entrada contato normalmente aberto, (b) entrada contato normalmente fechado, (c) saída bobina, (d) porta lógica *e*, (e) temporizador TON equivalente ao DI do ISA 5.2 e (f) porta lógica *ou*. 9
2.8 Modelo do comportamento de um possível controlador de duplo-clique de um mouse. 10
2.9 Notação do diagrama do método TBM encontrado na Figura 2.10. 12
2.10 Diagrama do método: Testes baseados em modelos. 13
2.11 Diagrama de rastreabilidade. 15
2.12 Tela do construtor de modelos do UPPAAL. 18
2.13 Tela do simulador de modelos do UPPAAL. 19
2.14 Visão geral do método desenvolvido no trabalho [9]. 20
2.15 (a) estado, (b) entrada, (c) tarefa e (d) decisão. 21
2.16 Modelo SDL para o requisito de liberação de recurso usando ficha. 22

3.1 Legenda do diagrama do método TBM adaptado. 26
3.2 Diagrama do método de testes baseados em modelos do Gungnir. 27
3.3 (a) estado, (b) estado inicial, (c) estado *committed*, não consome tempo, (d) transição, (e) atribuição e (f) guarda. 28
3.4 Processo de execução do Gungnir. 29
3.5 Exemplo de especificação ISA 5.2. 29
3.6 *ReadInputs* - um exemplo de autômato no qual é representado a leitura das variáveis de entrada do CLP. 30
3.7 Temporizador DI (TON) - autômato no qual é representado o controle dos valores de saída do temporizador timer1 da Figura 3.5. 30
3.8 *WriteOutputs* - exemplo de autômato no qual é representada a escrita das variáveis de saída do CLP. 30
3.9 (a) *ReadInputs*, (b) *timer_DI*, (c) *WriteOutputs* e (d) *TimerUpdater*. 32

3.10 (a) estado, (b) estado inicial, (c) estado final, (d) limites temporais, (e) atribuição e (f) transição. . . . 33
3.11 Modelo de ambiente que representa as ações de um usuário do equipamento x. 33
3.12 (a) porta *and*, (b) Heurística para expressões do tipo *and*, (c) porta *or*, (d) Heurística para expressões do tipo *or*, (e) combinação de portas *and* e *or* formando a expressão *(A and B) or C* e (f) Heurística para expressões *(A and B) or C*. 35
3.13 (a) Degrau com mais de uma referência para a variável A, (b) Heurística com valores impossíveis. . . . 35
3.14 Passo a passo da obtenção da heurística do degrau da Figura 3.12.e. 36
3.15 (a) Degrau de um diagrama ISA 5.2 e (b) heurística do fator determinante para o caso 36
3.16 (a) Heurística do fator determinante para degrau da Figura 3.12.e, (b) vetores satisfazem a cobertura com $n = 1$ e (c) vetores satisfazem a cobertura com $n = 2$. Os vetores sublinhados encaixam-se em mais de uma classe de valor. 37
3.17 Screenshot do aplicativo Scanion com um exemplo de arquivo **.vcd**. 38

4.1 Engarrafadora - Fonte: [6], página 485 40
4.2 Engarrafadora - (a) ISA 5.2 e (b) Ladder 40
4.3 Autômato *ReadInputs* 41
4.4 Autômato *Timer_Di01* 42
4.5 Autômato *Timer_Di02* 42
4.6 Autômato *WriteOutputs* 42
4.7 Autômato *TimeUpdater* 42
4.8 Modelo do ambiente para o sistema de controle da engarrafadora. 44
4.9 Heurística do fator determinante obtida do modelo da especificação do sistema de controle da engarrafadora. 44
4.10 Comportamento da engarrafadora para as entradas geradas sem modelo do ambiente com $n = 1$. 45
4.11 Comportamento da implementação da engarrafadora com o falha 1 injetado. . 46
4.12 Comportamento da implementação da engarrafadora com o falha 2 injetado. . 47
4.13 Comportamento da implementação da engarrafadora com o falha 3 injetado. . 47

Lista de Tabelas

4.1 Variação do valor das entradas seguindo o modelo de ambiente da Figura 4.8 para $n = 1$. 43

Capítulo 1

Introdução

A crescente demanda do mercado faz com que as indústrias invistam mais em produtividade e qualidade. Isso é alcançado, normalmente, através de sistemas de automação industrial. Grandes empresas são obrigadas a automatizar a produção, a fim de maximizar lucros e minimizar custos. Um grande exemplo é a Petrobras - uma das grandes empresas no segmento da indústria de óleo, gás e energia no mundo. Hoje, conta com uma grande quantidade de plataformas de extração em vários locais do Brasil e do mundo e com grande potencial de crescimento a partir da descoberta de petróleo e gás na região do pré-sal[1].

Empresas como a Petrobras devem aplicar várias normas internacionais em seus sistemas de automação, com a finalidade de obter segurança e confiabilidade. Um dos equipamentos mais importantes da automatização da produção é o Controlador Lógico Programável (CLP) [21], através de sensores e atuadores, os CLPs podem controlar a mais vasta gama de processos industriais. Alguns destes processos necessitam de maior cautela por tratarem de produtos perigosos e/ou caros. Sistemas são classificados como sistemas críticos quando falhas no controle produtivo podem gerar catástrofes ou perdas.

Para evitar que erros aconteçam em sistemas críticos, são utilizados os Sistemas Instrumentados de Segurança (SIS) [12] os quais são responsáveis por evitar que atitudes proibidas sejam tomadas pelo sistema de controle, desta forma diminuindo os riscos de acidente. Para aumentar a segurança do sistema como um todo, várias medidas devem ser tomadas que vão desde o cuidado na escolha das peças usadas na linha de produção ao teste exaustivo do software o qual será executado pelo do CLP que controla a produção.

O foco neste trabalho é aumentar a segurança de sistemas de controle através da melhoria no processo de testes que conta com a adaptação do método apresentado em [9] e o desenvolvimento de uma ferramenta para geração automática de testes baseados em modelos. Os sistemas, em que este trabalho se aplica, são os que utilizam a linguagem de programação Ladder [23], para implementação, e diagramas ISA 5.2 [15], para especificação do sistema.

[1]Fonte: site oficial da Petrobras - http://www.petrobras.com.br.

1.1 Descrição do problema

Atualmente, a Petrobras como grande parte de outras grandes empresas, terceirizam o desenvolvimento dos sistemas de controle. Inicialmente, os diagramas ISA 5.2 são gerados a partir de tabelas de causa e efeito e textos estruturados que caracterizam a especificação do sistema de controle. Esse diagrama é entregue à empresa responsável pelo desenvolvimento e ao fim do desenvolvimento o software é apresentado à Petrobras. Para testá-lo, executa-se o Teste de Aceitação de Fábrica (TAF). O TAF é um conjunto de testes construídos com base na tabela de causa e efeito do sistema. Os testes são executados diretamente na planta onde o sistema será usado. Caso haja algum problema, esse sistema volta para a empresa de desenvolvimento para retificação. Ao fazer isso, os custos do projeto aumentam. Na Figura 1.1, o processo está representado graficamente.

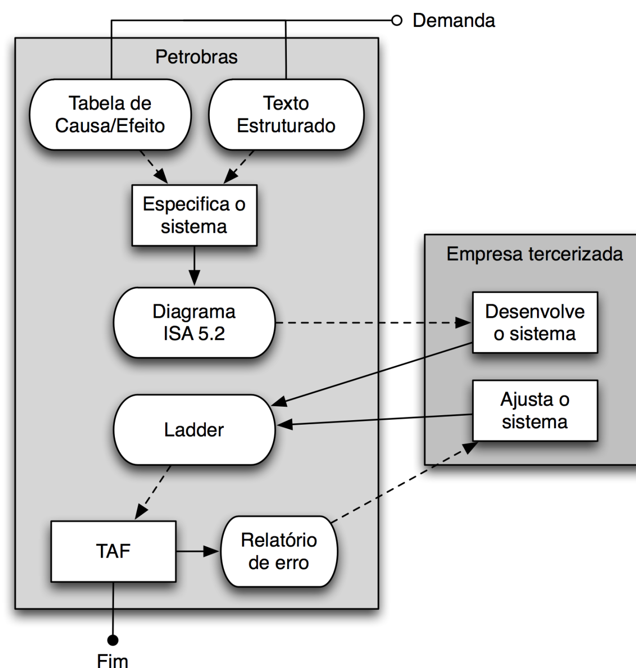


Figura 1.1: Processo de desenvolvimento da Petrobras.

Para exemplificar, vamos supor que a Petrobras vai iniciar a produção de uma nova linha. Essa nova linha precisa de um sistema para controlá-la. Esse sistema é a demanda. A partir dessa demanda, é criada uma tabela de causa e efeito que descreve o funcionamento do sistema. Para auxuliar o entendimento e aumentar o grau de detalhe com relação as ações do sistema, é feito um texto estruturado onde são especificadas observações/restrições com relação a tempo e outras características. Do texto estruturado e da tabela de causa e efeito constrói-se a especificação do sistema em ISA 5.2. Neste momento, a especificação é enviada para uma empresa tercerizada onde será desenvolvido e testado o sistema. No fim do desenvolvimento, um programa Ladder é entregue à Petrobras e, então, é feito o teste de aceitação de fábrica na própria planta de forma supervisionada. Se houver algum defeito, o

código é retornado para a empresa de desenvolvimento para correção e o teste é feito novamente. Esse processo se repete até que não sejam detectadas falhas. Não detectar falhas não implica em dizer que o sistema está livre delas.

1.2 Objetivo do trabalho

Este trabalho de pesquisa está inserido no contexto do projeto SIS em parceria com a empresa Petrobras. Os objetivos do projeto são: criação de métodos, técnicas e ferramentas para apoio no processo de desenvolvimento de sistemas de controle com o foco no aumento da confiança. O mesmo foi iniciado em 2006 e conta com a contribuição dos seguintes trabalhos:

- Módulo que gera modelos de autômatos temporizados a partir de Diagramas ISA 5.2 [8], Diagramas de Bloco de Funções (DBF) [8, 7] e programas Ladder [10];
- Módulo que gera e executa casos de teste de conformidade;

Esta dissertação apresenta novos resultados por meio de adaptações do trabalho apresentado em [9] e o desenvolvimento de uma ferramenta para geração automática de casos de testes baseados em modelos do sistema e do ambiente e a criação de heurísticas as quais definem um critério de cobertura para os modelos. Pode-se obter modelos de Autômatos Temporizados (AT) gerados a partir das especificações ISA 5.2 e do programa Ladder. Com a utilização de heurística e um critério de cobertura baseado em dados pode-se gerar os dados de testes baseados na especificação ISA 5.2. Com os dados gerados aplica-se esses dados ao modelo da especificação o qual é o oráculo do sistema e depois executam-se os mesmo testes no modelo da especificação e assim verifica-se se a implementação está de acordo com a especificação o que caracteriza testes de conformidade. É elaborado, neste trabalho, um mecanismo de geração e execução de testes para sistemas de eventos discretos.

São contribuições deste trabalho:

- A heurística do fator determinante (HFD) a qual determina um conjunto mínimo de valores de entradas que descrevem a essência do comportamento de um degrau do sistema;
- Definição do critério de cobertura baseado em dados;
- A definição de diagrama para modelagem do ambiente de execução do sistema o qual permite a criação de modelos de ambiente que tornam os testes mais efetivos;
- A ferramenta Gungnir a qual simula modelos de autômatos temporizados, gera testes de forma aleatória ou direcionada por modelos de ambiente, calcula as heurísticas do fator determinante para cada degrau do sistema e analisa a cobertura dos testes baseados nas HFDs;

1.3 Estrutura do Documento

No Capítulo 2, são apresentados os conceitos necessários para o entendimento deste trabalho e trabalhos relacionados. No Capítulo 3, é introduzida a ferramenta Gungnir e o processo de testes utilizando-a. Já no Capítulo 4, são expostos estudos de caso utilizando a ferramenta. No Capítulo 5, as conclusões e trabalhos futuros são discutidos. Por fim, são listadas as referências utilizadas neste trabalho.

Capítulo 2

Fundamentação teórica

Nesta seção, explica-se os conceitos básicos necessários para o entendimento deste trabalho. Estes são: controladores lógicos programáveis, autômatos temporizados e testes baseados em modelos. Conclui-se a fundamentação teórica com a discussão de alguns trabalhos relacionados.

2.1 Controlador lógico programável - CLP

São dispositivos de controle largamente utilizados na indústria, com a finalidade de aumentar e melhorar a produção. Usam circuitos integrados ao invés de dispositivos eletromecânicos para implementar funções de controle. São capazes de armazenar instruções de: sequenciamento, temporização, contagem, aritmética, manipulação de dados e comunicação; tudo para possibilitar um controle eficiente de máquinas industriais e processos [5]. Uma visão geral do uso de CLPs através de um diagrama conceitual é exposta na Figura-2.2.

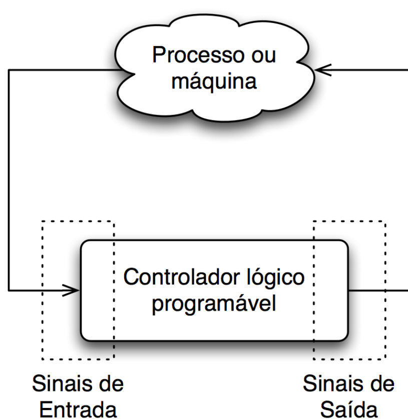


Figura 2.1: Diagrama conceitual da aplicação de um CLP.

Segundo [20], um CLP é formado por:

1. Uma CPU, unidade central de processamento, baseada em microprocessador;

2. Memória, a qual possui áreas reservadas para dados das entradas, dados da saída e processamento;

3. Pontos de entrada e saída, onde os sinais podem ser recebidos e enviados de/para o processo ou máquina, respectivamente.

Normalmente, um CLP é equipado com um sistema operacional que permite o carregamento e execução de programas e auto-checagem [20]. Na Figura 2.2, temos a arquitetura de um CLP.

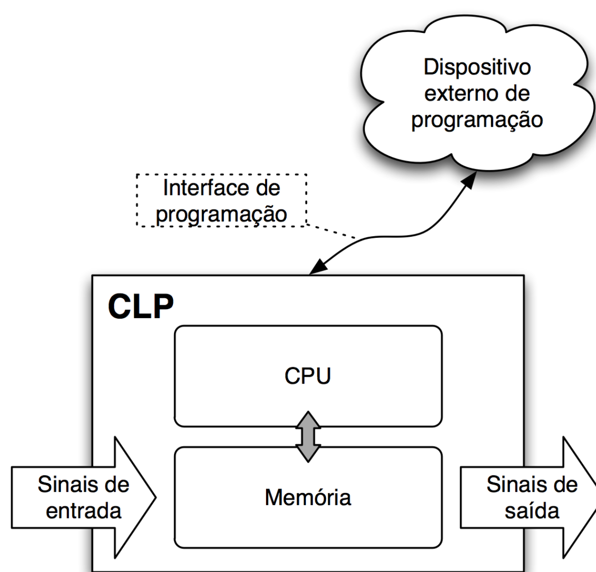


Figura 2.2: Arquitetura de um CLP.

Durante a operação de um CLP a CPU executa três tarefas básicas de forma cíclica:

1. Leitura, consiste na obtenção/aceitação das informações pela interface de entrada;

2. Processamento, executa o programa de controle armazenado no sistema de memória;

3. Escrita, atualiza as saídas do CLP via interfaces de saída.

Esse processo cíclico de leitura, processamento e escrita do CLP é chamado *Scan Cycle* [5]. Os ciclos variam de tamanho dependendo da complexidade do programa de controle. É possível observar um tempo máximo por ciclo, geralmente, na ordem de mile segundos [20]. Na Figura 2.3, está representada uma visão geral do *scan cycle*. Já na Figura 2.4, temos uma visão interna de camadas de informação e processamento do PLC, observe que o fluxo da informação vem do exterior via camada de comunicação, também chamadas de placas de entrada, a informação é copiada nas variáveis globais na etapa de leitura, processadas pelas CPUs.

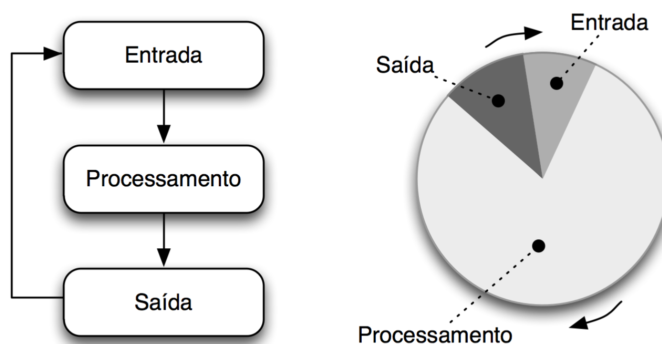


Figura 2.3: Representação gráfica do *Scan Cycle*.

Apesar de usualmente não fazer parte do CLP, os dispositivos de programação e desenvolvimento são necessários. Não há interfaces gráficas nem meios de interação com CLPs como teclados ou mouses. Portanto, deve-se desenvolver o programa em estações de trabalho para, no fim, carregá-lo no CLP. Alguns fabricantes fornecem, junto aos CLPs, dispositivos de programação; outros permitem programação direta via porta serial, USB, entre outros meios [5].

2.1.1 Detalhes da execução de CLPs

Segundo o padrão internacional IEC 61131-3 [23], um CLP lê os valores das placas de entrada e armazena nas variáveis de entrada globais, processa os dados utilizando os valores previamente lidos das placas de entrada, atualiza os valores de saída nas variáveis de saída globais. No final, as informações armazenadas nas variáveis de saída globais serão escritas nas placas de saída do CLP, caso seus valores sejam diferentes dos já armazenados nelas.

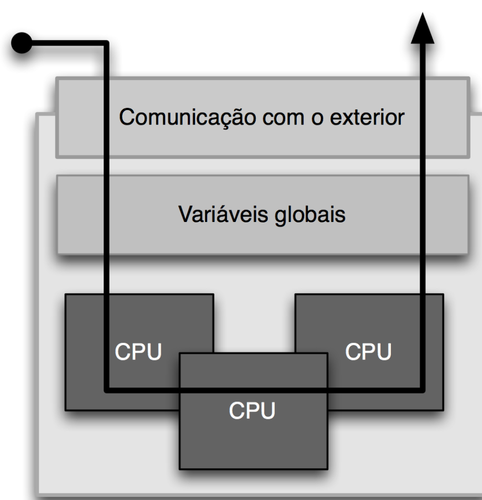


Figura 2.4: Representação interna do *Scan Cycle*.

As linhas de realimentação (*feedback*), Figura 2.5, são atualizadas no fim do *Scan Cycle*, ou seja, durante todo o *Scan Cycle* o valor da realimentação é o valor da saída no momento que as entradas são lidas.

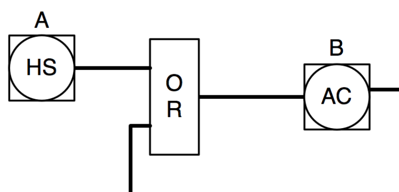


Figura 2.5: Representação gráfica de uma realimentação.

2.1.2 Linguagens

Há um padrão internacional que rege o uso de linguagens de programação para controladores lógicos programáveis, o IEC 61131-3 [23]. As linguagens tratadas no padrão são:

- Diagrama de blocos de função (do inglês, *Function Block Diagram*);
- Ladder;
- Gráfico sequencial de funções (do inglês, *Sequential Function Chart*);
- Texto estruturado;
- Lista de instruções.

Neste trabalho, duas linguagens são utilizadas, seguindo os padrões e normas da Petrobras. Para especificação, é adotada a norma ISA 5.2 e, para programação dos CLPs, a linguagem Ladder (linguagem gráfica). Ambas são largamente aplicadas em processos industriais de grandes empresas. Os elementos de ISA 5.2 e Ladder considerados neste trabalho estão representados nas Figuras 2.6 e 2.7. Dentre eles podemos encontrar portas lógicas e e ou, entradas, saídas e elementos temporizados da mesma categoria. Para os temporizadores utilizados consideramos que energizam a saída (saída = 1) quando o impulso de entrada permanece ao menos t unidades de tempo u ativado (entrada = 1). Desenergizando a saída (saída = 0) imediatamente quando a entrada for desativada (entrada = 0).

2.2 Autômatos temporizados

Autômato temporizado é uma extensão da teoria dos autômatos finitos para modelar sistemas de tempo real [11]. As transições são instantâneas e possuem relógios associados, estes

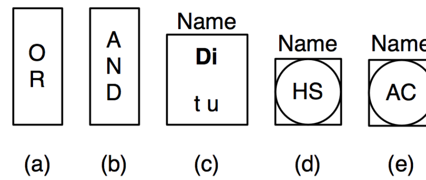


Figura 2.6: Subconjunto de elementos do padrão ISA 5.2: (a) porta lógica *e*, (b) porta lógica *ou*, (c) temporizador DI, (d) chave de entrada e (e) acumulador de saída.

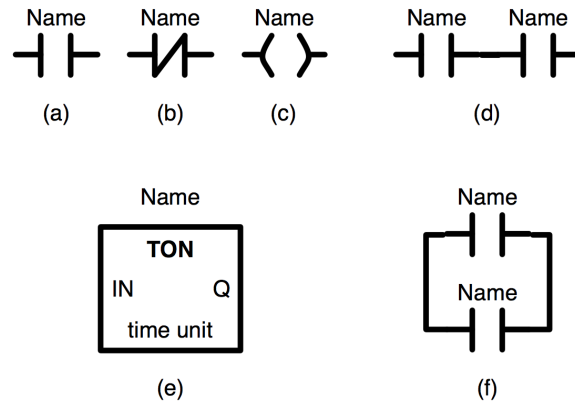


Figura 2.7: Subconjunto de elementos de programas Ladder:(a) entrada contato normalmente aberto, (b) entrada contato normalmente fechado, (c) saída bobina, (d) porta lógica *e*, (e) temporizador TON equivalente ao DI do ISA 5.2 e (f) porta lógica *ou*.

podem ser reiniciados independentemente do resto do sistema durante as transições. Todos os relógios são incrementados ao mesmo tempo e são definidos como números reais possibilitando, assim, a modelagem de sistemas contínuos [16].

Os autômatos temporizados utilizados nesta dissertação são do tipo: *Autômato temporizado de Muller* **(ATM)** - fechados sobre as operações booleanas e possuem grande poder expressivo [2]. Nesta dissertação, quando falarmos de autômatos temporizados, estaremos nos referindo a um ATM. A definição formal dos ATMs é:

Autômato temporizado (ATM) é uma 6-tupla $(\Sigma, S, S_0, C, E, \mathscr{F})$ onde:

1. Σ é um conjunto finito chamado alfabeto de entrada;
2. S é um conjunto finito chamado estados;
3. $S_0 \in S$ é o estado inicial;
4. $E \subseteq S \times S \times \Sigma \times 2^C \times \Phi(C)$ é o conjunto de arestas, análogo à função de transferência de autômatos finitos acrescido ao conjunto potência do conjunto de relógios 2^C e o conjunto de restrição de relógio $\Phi(C)$.
5. C é um conjunto finito de relógios;
6. $\mathscr{F}$ é a condição de aceitação $\mathscr{F} \subseteq 2^S$. Consiste em uma família de estados que podem ser repetidos infinitamente, 2^S é o conjunto potência de S.

Outros aspectos importantes com relação aos autômatos temporizados os quais não estão na definição formal são expostos a seguir. Para ilustrar os conceitos é utilizada a Figura 2.8. O estado com um círculo interno é o estado inicial do autômato.

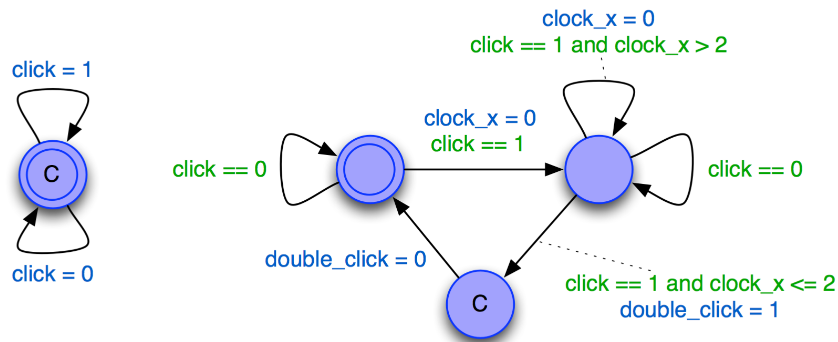


Figura 2.8: Modelo do comportamento de um possível controlador de duplo-clique de um mouse.

2.2.1 Guardas

Constituem restrições aplicadas aos relógios as quais são formadas por conjunções de expressões booleanas. Em algumas abordagens, há guardas relacionada com variáveis do sistema, como no caso do UPPAAL TRON [13] onde variáveis de entrada e/ou saída podem ser expressas numa guarda. Como exemplo de guardas podemos citar as condições $clock_x > 2$ e $clock_x <= 2$ na Figura 2.8.

2.2.2 Eventos

São quaisquer modificações internas ou externas nas variáveis do modelo. Como exemplo podemos citar os seguintes eventos: na Figura 2.8 temos algumas atribuições, $click = 1$ e as mudanças da variável $clock_x$ a qual representa um relógio por exemplo.

2.2.3 Atribuições

Por meio destas, apresentamos as mudanças que devem ser efetuadas na transição. Na Figura 2.8 podemos encontrar três exemplos de atribuições: o reinício do relógio clock_x ($clock_x = 0$), a atribuição dos valores zero ($doubleClick = 0$) e um ($doubleClick = 1$) à variável doubleClick.

2.3 Testes baseados em Modelos

Segundo [14], teste de software consiste na verificação dinâmica do comportamento de um programa sobre um conjunto finito de casos de testes apropriadamente selecionados de um domínio normalmente infinito, contra o comportamento esperado do mesmo. Ou ainda, conforme dito por [24], consiste na execução de um programa com a intenção de encontrar erros. A execução de testes adiciona valor a um sistema, pois aumenta sua qualidade e confiabilidade. Há muitas aplicações que exigem um alto grau de confiabilidade, porquanto podem causar grandes prejuízos.

Este trabalho está pautado sobre a metodologia de testes chamada **Testes Baseados em Modelos** - **TBM** (do inglês, *Model-Based Testing*) que por definição é a **automação do projeto de testes caixa-preta** [27]. Os testes caixa preta são caracterizados pela geração de casos de testes baseados na especificação, pois não se tem acesso a detalhes como comportamento ou estrutura internos do sistema em teste (do inglês, *System Under Test* - **SUT**) no momento da geração dos casos de testes. Contudo, em TBM, há uma pequena diferença com relação ao uso de testes caixa-preta, porquanto é criado um modelo do comportamento esperado do SUT o qual detêm parte ou toda especificação do mesmo ao invés de utilizar a especificação diretamente. A partir deste modelo, pode-se, automaticamente, gerar uma vasta gama de casos de testes utilizando ferramentas apropriadas.

Um modelo é uma descrição simplificada de algo. Em TBM, os modelos devem ser pequenos com relação ao tamanho do sistema, por conseguinte devem ser detalhados o suficiente para descrever características as quais devem ser testadas [24]. Os modelos descritos são utilizados como base para as ferramentas de geração de casos de testes e podem ser representados nas mais variadas formas. Abaixo temos algumas das formas de representação listadas:

- Diagramas UML enriquecidos com OCL (*Object Constraint Language*) ou máquinas de estado [27];
- Redes de Petri [22];
- Autômatos temporizados [2];
- EFSM - *Extended Finite State Machine* [27].

Existe também o conceito de Oráculo (do inglês, *Oracle* - termo bastante referenciado no âmbito de pesquisas sobre testes) o qual é responsável por gerar as saídas esperadas baseado nas entradas. Desta forma, fecha-se um ciclo na metodologia TBM, pois têm-se os modelos de descrição do SUT, os casos de testes gerados pelas ferramentas baseando-se nos modelos e os oráculos. A partir de então, deve-se executar os casos de teste e avaliar se os dados externados pelo SUT são iguais aos determinados pelo oráculo. Se, para todos os casos de

testes, essa premissa for verdadeira, o resultado dos testes é positivo; em outro caso negativo, ou seja, algum erro ocorreu. Na Figura 2.10, temos uma visão geral do uso da abordagem caixa-preta em TBM, ao fazer uso da notação mostrada na Figura 2.9.

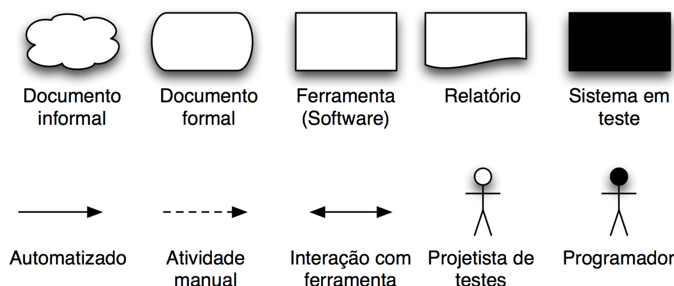


Figura 2.9: Notação do diagrama do método TBM encontrado na Figura 2.10.

A metodologia de testes baseados em modelos tem cinco passos principais:

1. **Criação do modelo**

   Primeiramente, o projetista de testes deve criar um modelo do sistema e/ou do ambiente baseado nos requisitos podendo fazer uso, também, do modelo previamente gerado pela equipe de desenvolvimento. Segundo [27], é importante não reutilizar totalmente o modelo criado pela equipe de desenvolvimento, pois em alguns momentos são inadequados com relação ao nível de detalhamento e podem fazer com que erros de especificação perdurem por mais tempo;

2. **Geração dos casos de teste abstratos**

   O próximo passo é a geração dos casos de teste abstratos que consiste na criação de casos de testes com pouca riqueza de descrição, não se adequando totalmente para a execução de testes. Sendo assim, é necessário a adaptação destes testes para um nível de descrição maior tornando-se possível a sua execução. Podemos ainda, reutilizarmos essas abstrações ao variarmos as adaptações.

   Deve-se ainda fazer uma correlação direta entre os testes abstratos e os requisitos, para que, no fim, possamos obter uma matriz de rastreabilidade. Essa matriz permite o mapeamento direto entre os casos de testes e os requisitos, possibilitando uma análise mais completa da cobertura dos testes, pois com o conjunto de caso de testes utilizado pode-se avaliar quais os requisitos estão sendo testados;

3. **Concretização dos casos de teste**

   Em seguida, temos a concretização dos casos de teste o que significa dizer que é gerado um conjunto de casos de testes com a capacidade de ser executados. A concretização é feita utilizando uma ferramenta chamada adaptador. Isso modulariza os testes, porquanto, caso haja alguma mudança, como por exemplo mudança de linguagem de

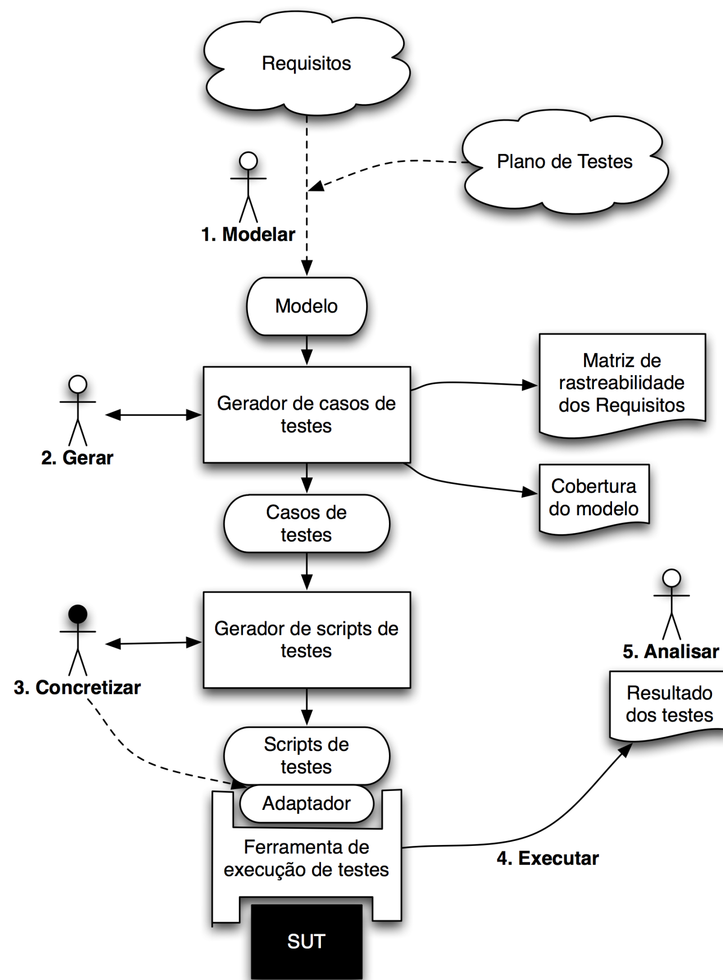


Figura 2.10: Diagrama do método: Testes baseados em modelos.

programação, os casos de teste abstratos ainda servirão, bastando apenas mudar o código do adaptador;

4. **Execução dos casos de teste**

   Agora temos a execução dos casos de teste concretos. Através de uma ferramenta de execução de testes, são obtidos os relatórios dos resultados dos testes. Há duas formas de executar testes:

   ***online*:** (*on-the-fly*) onde os testes são gerados e executados em sequência, não há registro dos casos de testes, contudo registram-se os resultados. Nesta abordagem, as fases de geração de casos de testes abstratos e concretização dos mesmos acabam sendo feitos de forma implícita à fase de execução já que não há registro de casos de teste;

   ***offline*:** onde os casos de testes são previamente armazenados e, quando demandado, devem ser concretizados, para enfim, serem executados;

5. **Análise dos resultados**

   Por último, é feita a análise dos resultados dos testes (relatórios), para assim, serem realizadas as ações corretivas necessárias. A partir do momento que surgir um problema, deve-se saber qual sua fonte. Duas novas fontes surgem com o uso de TBM: erros no código do adaptador ou no modelo (podem ser erros nos requisitos). Conforme dito em [27], grosseiramente metade dos erros encontrados serão no SUT e a outra metade será encontrada no modelo ou nos requisitos. Achar erros nos requisitos do sistema, até mesmo antes da implementação, é uma das vantagens do uso de TBM e será detalhado na Seção-2.3.1.

2.3.1 Vantagens

Esta seção está destinada a apontar os pontos positivos do uso da metodologia de testes baseados em modelos. Estes pontos dependem diretamente da qualidade dos modelos gerados, das ferramentas de geração e profissionais envolvidos com o projeto. A vantagens de se usar TBM são:

Detecção de erros no SUT: Bom para encontrar erros no sistema em desenvolvimento já que pode-se gerar uma grande quantidade de testes baseados na especificação/requisitos e escolher o critério de testes;

Redução de custo e tempo: Há uma redução considerável de tempo de execução/construção de testes com relação a outras técnicas. Bom para manutenção de testes quando o sistema evolui. Há a possibilidade de testar apenas os testes que foram afetados por uma determinada mudança;

Aumento da qualidade dos testes: Com a possibilidade de gerar um grande número de testes, escolher os critérios de testes e, ainda, analisar a cobertura dos requisitos através da matriz de rastreabilidade a qualidade dos testes tende a aumentar. Dependendo da equipe de testes a qualidade dos testes pode alcançar níveis excelentes;

Detecção de problemas nos requisitos: Por ser necessário a concepção de um modelo, erros nos requisitos informais ficam expostos, pois os projetistas acabam por se questionar sobre aspectos relevantes do projeto que podem estar incompletos ou incorretos. Quanto mais bem detalhado o comportamento do sistema, mais fiel será o modelo desse comportamento. A detecção de erros nos requisitos do sistemas é o principal aspecto positivo do uso de TBM, pois quando erros são encontrados nessa fase, é muito mais barato corrigir o sistema;

Rastreabilidade: É a habilidade de relacionar cada caso de teste com o modelo, com o critério de seleção e ainda com os requisitos informais do sistema [27]. Na Figura 2.11,

estão representas as possibilidades de rastreabilidade entre os casos de testes, o modelo e os requisitos. A rastreabilidade permite o aumento da qualidade dos testes, por exemplo: é possível saber qual a porcentagem dos requisitos que foram testados, saber se um requisito mais crítico foi mais ou menos testado que requisitos menos críticos, definir uma quantidade mínima de testes por requisitos, favorece os testes de regressão e assim por diante.

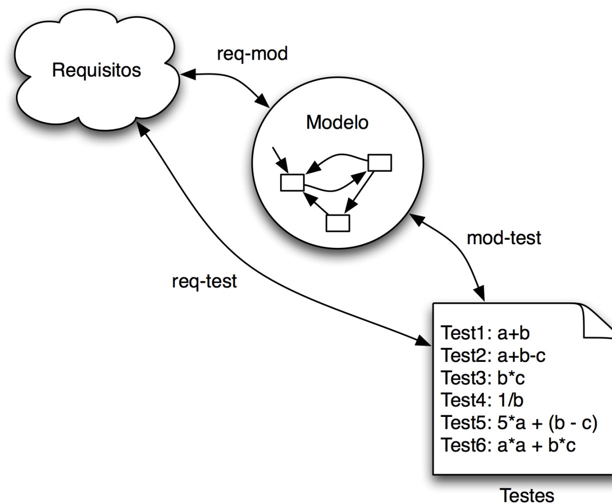


Figura 2.11: Diagrama de rastreabilidade.

2.3.2 Desvantagens

Esta seção está destinada a apontar os pontos negativos do uso da metodologia de testes baseados em modelos. O principal deles é que não há nenhuma garantia que serão encontradas todas as diferenças entre o modelo e a implementação o que é comum a qualquer tipo de teste [27]. Outras desvantagens mais específicas são listadas abaixo:

Habilidades extras: É imprescindível possuir conhecimentos além dos necessários à execução de testes manuais, pois há a necessidade de realizar outras atividades;

Apenas testes funcionais: Aspectos não-funcionais de um sistema como: usabilidade, desempenho, custo, etc, não são testados. Há trabalhos na linha de testes de desempenho utilizando TBM;

Maturidade de testes: Não deve ser aplicado em equipes cujo nível de maturidade de testes é baixa o que significa dizer que a equipe não utiliza ou não conhece técnicas e ferramentas modernas de testes.

Curva de aprendizado: Os projetistas de teste tem de aprender a usar novas ferramentas e a construir modelos apropriados às necessidades do projeto de testes baseados em modelos.

Na Seção-2.4, são descristas outras ferramentas com os mesmos intuitos do Gungnir, com a finalidade de podermos comparar e argumentar sobre pontos de melhorias e novas características que devem ser adicionadas, para que o Gungnir torne-se uma ferramenta ainda mais atraente no uso de método testes baseados em modelos.

2.4 Trabalhos relacionados

Esta seção é destinada a apresentar e discutir trabalhos relacionados que influenciaram diretamente na concepção da ferramenta Gungnir. Como base, temos os trabalhos UPPAAL TRON [13] e o gerador de modelos de ISA 5.2 e Ladder para autômatos temporizados [9]. Com fins complementares os quais são discutidos a seguir, foram analisados os trabalhos: *Timed Moore automata: test data generation and model checking* [19], *Requirement-BasedAutomated Black-Box Test Generation* [26] e o *Boundary coverage criteria for test generation from formal models* [17].

Foi constatado nas pesquisas realizadas em [9] que UPPAAL TRON é a ferramenta, dentre as pesquisadas, a qual põe em prática a maior quantidade de necessidades do projeto SIS. Contudo, o TRON não foi criado para solucionar problemas específicos de sistemas de automação e controle, por isso possui elementos desnecessários ou que necessitam de melhorias. Todos os aspectos do TRON relevantes a este trabalho serão descritos na Seção 2.4.1.

É discutido, também nesta Seção, o gerador de modelos descritos em termos de autômatos temporizados a partir de modelos descritos em ISA 5.2 e programas Ladder [9]. Esta ferramenta automatiza parte das atividades da metodologia de testes baseados em modelos e será apresentada na Seção 2.4.2.

2.4.1 UPPAAL TRON

O UPPAAL TRON é uma ferramenta para verificação de sistemas de tempo real desenvolvida pela parceria entras as universidades de Uppsala e Aalborg. Sua implementação teve início como parte da tese de mestrado e continuado como parte do projeto da tese de doutorado de Marius Mikucionis supervisionado por Kim G. Larsen e Brian Nielsen [18]. Possui uma série de estudos de caso aplicados que abrangem de protocolos de comunicação a aplicações multimídia [3].

As principais características da ferramenta baseado em [18] são:

- Aplicação de testes de conformidade. A ferramenta verifica se as execuções temporizadas do sistema em teste (SUT) estão especificadas no modelo do sistema e se nenhum comportamento ilegal é observado;
- Ênfase nos testes de propriedades funcionais e temporais. O tempo é considerado contínuo, eventos podem acontecer a qualquer momento no tempo, porém os *dea-*

dlines são definidos como inteiros. A geração de dados de teste também é possível, por conseguinte tipos de dados e seleção de valores são limitados pela linguagem de modelagem;

- A especificação é uma rede de autômatos temporizados UPPAAL [4] particionada em modelo do sistema e modelo do comportamento do ambiente. Os modelos podem ser não-determinísticos, permitindo uma liberdade apropriada para implementação de sistemas, modelando possíveis/toleráveis desvios de tempo, *soft time deadlines*;

- A primitivas de testes são geradas diretamente do modelo, executadas e as respostas do sistema são checadas ao mesmo tempo, ***online (on-the-fly)***, enquanto conectada com o SUT; isso evita *suites* de testes intermediárias gigantescas;

- Durante os testes a ferramenta segue o modelo do ambiente o qual pode ter várias propostas:

  1. um modelo completo do ambiente, o qual será suficiente para executar todos os testes de conformidade;
  2. um ambiente específico que diminui o esforço para realização de testes para um nível realista de conformidade;
  3. um ambiente como um guia de casos de uso para uma funcionalidade de particular interesse;
  4. um modelo do ambiente como execuções de testes pré-gravadas usadas para re-executar testes durante a depuração ou testes de regressão.

- Herdou o motor de exploração de modelos do projeto UPPAAL o qual permite **explorações rápidas e eficientes de modelos de autômato temporizados**;

- Se o modelo do ambiente for não-determinístico (o que é muito comum de ocorrer), então, **escolhas de valores de entrada ou tempos de espera são aleatórios;**

- Em geral, testes de conformidade para sistemas de tempo real são indecidíveis, contudo em termos numéricos, tem se mostrado suficiente em um espaço limitado de tempo.

O UPPAAL TRON também herdou do projeto UPPAAL [4] a capacidade de construir graficamente os modelos de autômatos temporizados e simulá-los. A partir dos modelos construídos, são feitos os testes. Telas do programa em uso são apresentadas nas Figuras 2.12 e 2.13.

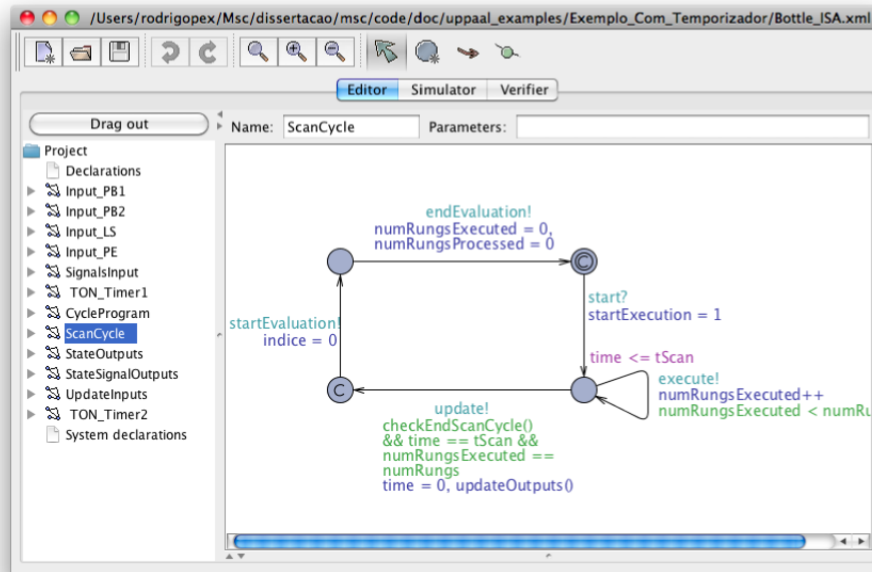


Figura 2.12: Tela do construtor de modelos do UPPAAL.

2.4.2 Método para automatização de testes caixa preta

Em [9], foi desenvolvida uma ferramenta e um método os quais contribuem para o aumento da confiança de programas para CLPs. O método consiste na utilização de testes baseados em modelos que utiliza como ferramenta de automação dos testes o UPPAAL TRON. A linguagem de implementação suportada no referido trabalho, Ladder, está em conformidade com o padrão internacional IEC 61131-3 [23].

A seguir são melhor detalhados a ferramenta e o método desenvolvido.

A ferramenta

A ferramenta é capaz de gerar modelos de autômatos temporizados utilizando o XML de uma representação da especificação em ISA 5.2 ou de um programa Ladder. O modelo de autômatos temporizados gerado reproduz o comportamento de um CLP, de modo que as gerações dos degraus de Ladder são traduzidos em um autômato chamado *Cycle Program* onde cada estado representa um degrau. Da mesma forma há um autômato que coordena as ações do modelo do CLP chamado *Scan Cycle* e as ações propriamente ditas como leitura e escrita de valores também são representados por autômatos. Todos os elementos são traduzidos de forma automática, em ATs os quais funcionam coordenados pelo AT que representa o *Scan Cycle*.

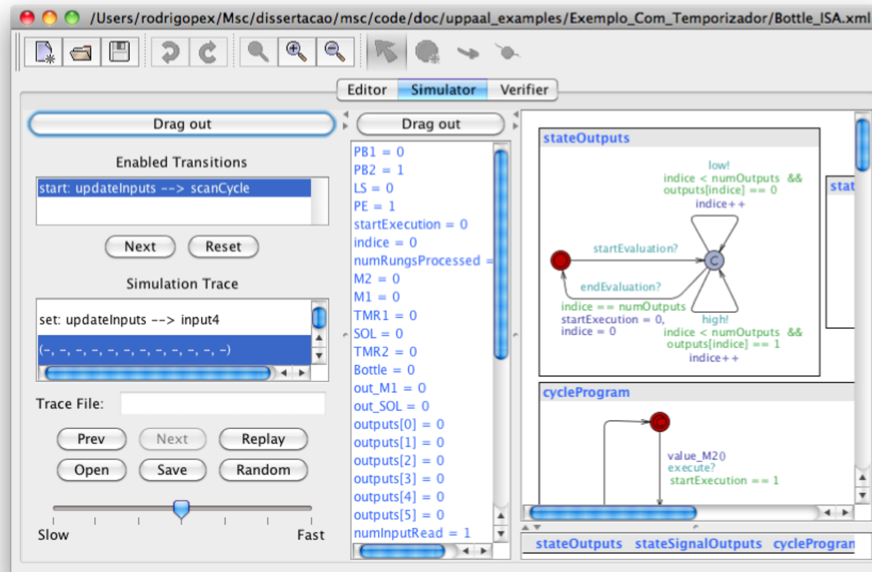


Figura 2.13: Tela do simulador de modelos do UPPAAL.

O Método

O resultado do método é um veredicto a cerca da corretude da implementação. Trata-se de um sistema correto aquele, cujo modelo da implementação está em conformidade com o modelo da especificação. Toda a geração e execução dos casos de testes são feitas de forma automática pela ferramenta UPPAAL TRON bastando apenas que os modelos AT estejam descritos no XML padrão do UPPAAL. A fim de obter o veredicto, deve-se seguir alguns passos:

1. Construir a especificação em ISA 5.2 utilizando a ferramenta SIS;

2. A partir do XML da especificação e da implementação, gerar os modelos AT no XML padrão do UPPAAL utilizando a ferramenta descrita na Seção-2.4.2;

3. Introduzir os dois modelos no UPPAAL TRON, e através de um adaptador desenvolvido para possibilitar a comunicação de duas redes de autômatos temporizados ligando, assim, os dois modelos;

4. Executar os testes *on-the-fly*, etapa responsável por gerar o *log* de veredicto;

5. Por fim, analisar o *log* manualmente e obter a informação a cerca da corretude dos testes.

Uma visão geral do método descrito é mostrada na Figura 2.14. Alguns problemas foram encontrados com o uso do UPPAAL TRON no trabalho [9], os quais serão listados abaixo:

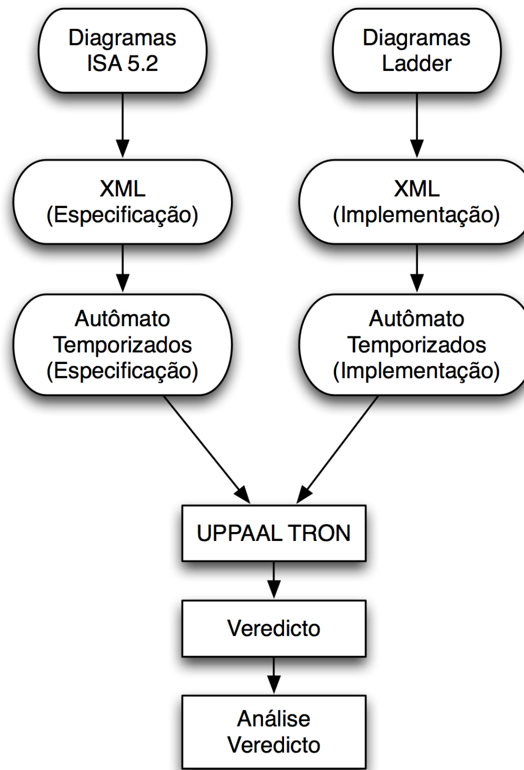


Figura 2.14: Visão geral do método desenvolvido no trabalho [9].

Projeto fechado: O código fonte do projeto não está disponível para modificação, porém a ferramenta é livre para uso.

Força bruta: A geração dos casos de testes é feita de forma aleatória, ou seja, utilizando a força bruta. Não há nenhuma análise prévia dos possíveis valores das entradas para a geração dos casos de testes. Muitos casos de testes são improváveis de acontecer o que causam desperdícios computacionais;

Dependência entre relógios: Atualmente no trabalho [9], os modelos gerados no padrão XML do UPPAAL podem possuir apenas um relógio, o que torna menos rica a descrição dos autômatos, pois todos os elementos temporizados dependem de apenas um único relógio. Isso é decorrente do problema de dependência entre os relógios. Essa limitação acarreta o problema de divisão, uma vez que, nem todos os temporizadores são múltiplos. O tempo de todo o modelo é baseado no tempo do *Scan Cycle*, podendo assim chegar a números não inteiros de ciclo;

Particionamento rígido: Os modelos utilizados no processo de teste devem estar particionados adequadamente. Variáveis e eventos de saída têm que ser diferenciados dos de entrada. Uma variável de entrada não pode ser modificada em eventos de saída e vice-versa. Assim, surgem redundâncias nos modelos, pois valores devem ser manipulados

em variáveis temporárias auxiliares;

Resultados confusos: Os dados gerados pela ferramenta UPPAAL TRON não são facilmente analisados, pois são grandes e ilegíveis a não especialistas. Há também os testes inconclusivos, os quais tornam a análise ainda mais trabalhosa.

2.4.3 Geração de dados de testes utilizando autômatos temporizados de Moore

No trabalho [19], é introduzido o conceito de autômato temporizado de Müller e é definida uma semântica operacional para componentes sequenciais com abstrações de tempo. São apresentadas, também, duas técnicas de geração de casos de testes onde uma delas é baseada em estruturas Kripke as quais podem ser definidas como o grafo de estados alcançáveis de um sistema onde os estados são os nós e as transições são as arestas. A técnica consiste em gerar os casos de testes ao passo que a estrutura kripke vai sendo explorada. Ações para que cobertura de comandos e decisões sejam alcançadas são definidas. Um problema com relação à aplicação deste trabalho no Gungnir é que a criação de estruturas Kripke torna-se muito custosa para sistemas grandes, pois são representados todos os estados possíveis do sistema.

2.4.4 Geração de testes caixa-preta baseada nos requisitos

É apresentada, em [26], uma abordagem chamada geração de testes baseados em requisitos que é uma extensão de testes baseados em modelos. Também é definida uma linguagem de especificação e descrição chamada SDL (*Specification and Description Language*). Os requisitos são descritos utilizando SDL e, a partir desta descrição modelos do sistema são gerados automaticamente em modelo EFSM. Com isso, há um modelo para cada requisito do sistema e os mesmos podem ser testados de forma independente. Na Figura 2.15 e 2.16, estão representados os elementos que fazem parte da linguagem SDL e um exemplo, respectivamente. A utilização deste trabalho no Gungnir é limitada, pois utiliza-se EFSM para modelar o comportamento do sistema e não há preocupação com o tempo em SDL.

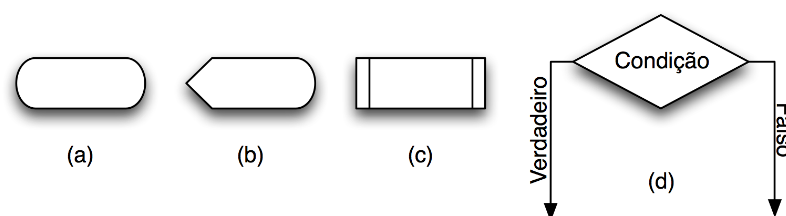


Figura 2.15: (a) estado, (b) entrada, (c) tarefa e (d) decisão.

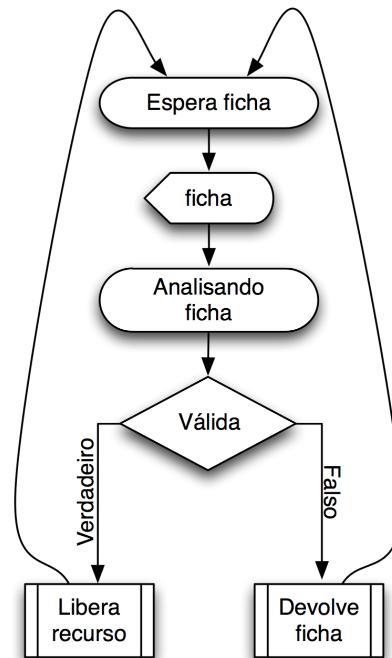


Figura 2.16: Modelo SDL para o requisito de liberação de recurso usando ficha.

2.4.5 Geração de testes a partir de modelos formais utilizando o critério de cobertura de fronteira

Em [17], é definida uma nova família de critérios de cobertura para testes baseados em modelos chamada *Boundary Coverage Criteria*. O critério de cobertura *Boundary* (ou de fronteira) é baseado em dados e pode ser usado para gerar dados ou, ainda, aplicado a um conjunto de dados já existente. Os testes são compostos de dados (os quais são atributos durante a execução do sistema) ou família de dados que interferem diretamente no comportamento do sistema, ou seja com a utilização desses dados o sistema muda de estados. São chamados de predicados efetivos dados modificadores de estado do sistema. No trabalho [17], é feita uma explanação geral sobre os critérios de cobertura existentes e também é apresentado o conceito de atributo efetivo o qual consiste em um atributo que, em determinado estado do sistema (boundary state), seu valor altera o comportamento do sistema. Segundo [17], cobrir todos os possíveis atributos efetivos de um modelo formal é muito similar a cobrir todos os caminhos independentes de um programa.

Os algoritmos para uso do critério e o próprio critério *Boundary* foram incorporados à ferramenta *BZ - Testing Tool* na podem ser manipuladas especificações B [1], Z [25] e UML/OCL [16]. O formalismo adotado neste trabalho são os autômatos temporizados o que inviabiliza o uso da ferramenta BZ-TT. Já a idéia do critério de cobertura não ser dependente do modelo e sim dos dados inseridos nos modelo é utilizado no Gungnir.

Este trabalho foi desenvolvido para resolver problemas, amenizar carências e concretizar técnicas citados acima, desta forma, torna possível o uso do método e ferramenta apre-

sentados em [9] no projeto SIS fornecendo também uma ferramenta específica para testes baseados em modelos para programas de CLPs. No Capítulo 3 são discutidos detalhes da ferramenta Gungnir e também como ela trata os problemas citados.

Capítulo 3

Ferramenta Gungnir

O Gungnir é uma ferramenta de automação de testes caixa-preta *online* a qual utiliza autômatos temporizados como formalismo para descrição dos modelos necessários aos testes. Está inserida no contexto do projeto SIS como substituta do UPPAAL TRON pelos motivos citados na Seção 2.4.2.

Para resolver/amenizar os problemas da ferramenta UPPAAL TRON, foi implementada uma séria de funcionalidades no Gungnir que vão de encontro aos pontos negativos tentando criar novos caminhos para uma solução mais específica e adequada ao problema. A seguir, temos uma lista com as principais características desta ferramenta:

- Permite a geração de autômatos temporizados que representam os modelos da especificação e da implementação baseado em arquivos XML de representações ISA 5.2 e programas Ladder, respectivamente. Essa característica foi adaptada a partir do trabalho apresentado em [9];

- Execução de redes de autômatos temporizados descritas na Seção 2.2 (modelos) com números arbitrários de estados (com ou sem consumo de tempo) e relógios;

- Número arbitrário de relógios independentes;

- Número arbitrário de transições entre os estados;

- Permite a descrição de autômatos temporizados em Python[1] tendo as atribuições limitadas pela sintaxe da linguagem;

- Geração de testes baseados em modelos do ambiente, este deve ser modelado a fim de descrever um comportamento específico do ambiente;

- Geração automática de casos de testes utilizando a força bruta;

- Execução dos testes em modo *online*;

[1] Fonte: Site oficial da linguagem Python - http://www.python.org

- Os resultados dos testes são claros e objetivos, analisáveis até mesmo por indivíduos não especialistas. Além disso, o relatório (*log*) dos testes é detalhado, demonstrando todas as modificações sofridas nas entradas e saídas do CLP durante a execução dos testes num arquivo *Value Change Dump* (**.vcd**). Pode-se, então, concluir se a implementação está ou não correta.

O método formalizado no trabalho apresentado em [9] ao utilizar o Gungnir está representado na Figura 3.2 utilizando as notações da Figura 3.1. Algumas mudanças podem ser observadas com relação ao diagrama mostrado na Figura 2.10, mudanças essas causadas por funcionalidades do Gungnir as quais estão listadas a seguir:

1. O projetista de testes é responsável pela descrição do modelo de ambiente e o seu papel é indispensável somente nos casos em que são usados testes baseados em modelos do ambiente. Caso sejam usados apenas casos de testes gerados na força bruta, o projetista faz-se dispensável, pois os testes são gerados baseados nos modelos do sistema os quais são gerados de forma automatizada;

2. O programador deve iniciar o processo junto ao projetista de testes, pois não é mais necessário criar os modelos do sistema. Basta fazer a descrição da especificação (diagramas ISA 5.2) e desenvolver o programa de implementação (programa Ladder) para a posterior geração de modelos;

3. Não há necessidade da geração de casos de testes em dois níveis (técnica utilizada em testes baseados em modelos), as vantagens da geração em dois níveis não se aplicam aqui. Os testes são executados de modo online, ou seja, a partir do modelo do ambiente e das gerações aleatórias os casos de teste são criados e executados no modelo da especificação o qual caracteriza o oráculo do sistema definindo os valores corretos de saída;

4. Os passos 2 e 3 foram embutidos no Gungnir. Não há mais o conceito de criação de modelos da implementação e da especificação por humanos. O programador não precisará mais modificar o adaptador, não há necessidade de adaptação, porque, agora, os testes podem ser aplicados ao próprio modelo da implementação;

5. A ferramenta de geração de modelo é uma extensão de parte do trabalho apresentado em [9] que foi adaptada ao Gungnir e acrescentada ao mesmo, eliminando, desta forma, o esforço na criação (normalmente manual) dos modelos do sistema[2]que são os modelos da especificação e implementação. Os modelos gerados são menos complexos, pois não são necessárias uma séria de características embutidas no modelos por conta do formalismo utilizado no UPPAAL TRON. Contudo, mantém todas as características necessárias para a geração e execução fiel do comportamento de CLPs.

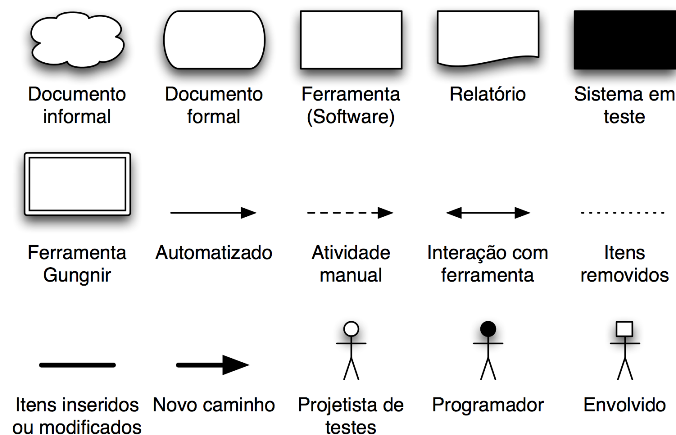


Figura 3.1: Legenda do diagrama do método TBM adaptado.

As características relativas a autômatos temporizados utilizados na ferramenta Gungnir estão descritos na próxima seção.

3.1 Autômatos temporizados

Redes de autômatos temporizados [2] são reconhecidas e simuladas no Gungnir. Os autômatos utilizados no Gungnir contém estados os quais podem ou não consumir tempo. As transições são feitas de forma aleatória, onde a transição com guarda válida entra no conjunto de possíveis sorteadas. Qualquer variável global pode entrar na verificação das guardas inclusive os relógios que também são representados nas variáveis do sistema. Na Figura 3.3 estão representadas graficamente os elementos existentes nos autômatos temporizados do Gungnir.

Os relógios são atualizados durante a simulação, o tempo de simulação é atualizado quando uma transição de um estado consumidor de tempo é escolhida. Caso não haja nenhuma transição candidata a executar em determinado momento, diz-se que aconteceu uma rejeição. A cada transição, pode-se atribuir valores às variáveis globais onde estão inclusos os relógios e variáveis de saída.

O conceito de palavra temporizada também faz parte do formalismo compreendido no Gungnir. Uma palavra temporizada é uma mudança em alguma das variáveis dos autômatos em determinados pontos no tempo. Na Expressão 3.1 temos um exemplo de palavra temporizada.

[2] A criação manual do modelo é um dos pontos fortes de TBM [27], pois os requisitos são de fato analisados, entendidos, revisados e questionados; desta forma, encontrando erros mais prematuramente baixando o custo do sistema com correção de defeitos. Através do uso da modelagem do ambiente essa característica ainda é mantida. A especificação em diagramas ISA 5.2 assume o papel do modelo na verificação de problemas nos requisitos.

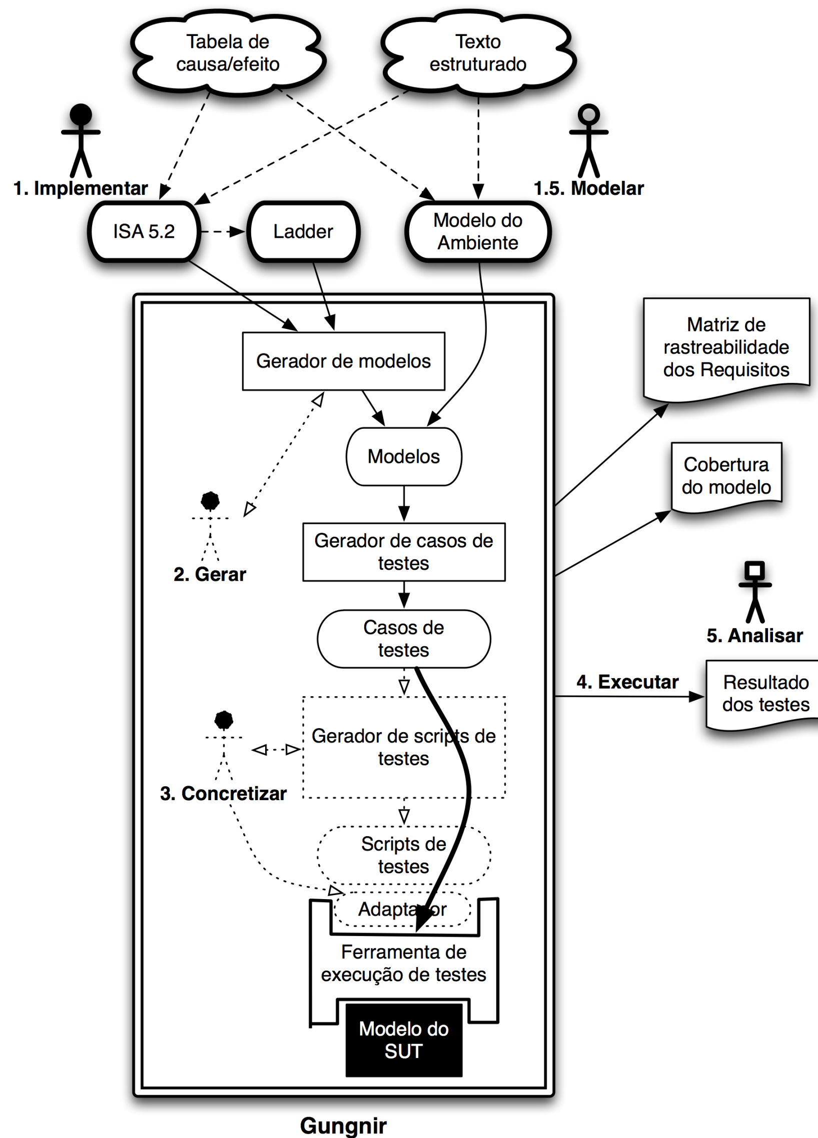


Figura 3.2: Diagrama do método de testes baseados em modelos do Gungnir.

$$TimedWWord((10,"v":1)->(12,"v":0)->(15,"v":1)->(20,"v":0)) \quad (3.1)$$

A variável v de um determinado autômato tem seu valor alterado para um, nos tempos 10 e 15, e para zero, nos tempos 12 e 20, durante a simulação de um autômato.

Podem ser simuladas no Gungnir redes de autômatos temporizados. Nas redes de autômatos todas as variáveis são globais, apenas um estado de um determinado autômato da rede é executado por vez e as atribuições são resultado de expressões lógicas.

Autômato temporizado simulados no Gungnir (ATM) é uma 5-tupla (Σ, S, S_0, C, E) onde:

1. Σ é um conjunto finito de variáveis as quais podem receber valor 0 ou 1;
2. S é um conjunto finito chamado estados o qual possui dois subconjuntos: esta-

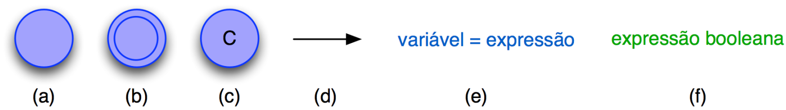


Figura 3.3: (a) estado, (b) estado inicial, (c) estado *committed*, não consome tempo, (d) transição, (e) atribuição e (f) guarda.

dos consumidores de tempo (*committed*) e não consumidores de tempo;

3. $S_0 \in S$ é o estado inicial;

4. $E \subseteq S \times S \times \Sigma \times 2^C \times \Phi(C)$ é o conjunto de arestas, análogo à função de transferência de autômatos finitos acrescido ao conjunto potência do conjunto de relógios 2^C e o conjunto de restrição de relógio $\Phi(C)$. Quando uma transição é escolhida, o estado é modificado para o estado que a transição aponta e uma atribuição é feita com o valor resultante da expressão lógica associada à atribuição contida na transição;

5. C é um conjunto finito de relógios.

Redes de autômatos temporizados simuladas no Gungnir (ATM) é uma 3-tupla (Σ, A, C) onde:

1. Σ é um conjunto finito de variáveis as quais podem receber valor 0 ou 1. É a união das variáveis dos autômatos que formam a rede;

2. A é um conjunto finito de autômatos;

3. C é a união dos conjuntos finitos de relógios do autômatos que formam a rede. Os relógios são atualizados com a mesma taxa do relógio do sistemas apenas quando estados consumidores de tempo são executados, caso contrário seus valores ficam inalterados.

A seguir, são descritas algumas características dos modelos e suas fontes de geração e a representação que o Gungnir usa para interpretá-los.

3.2 Testes baseados em modelos

O Gungnir foi construído de acordo com os conceitos da metodologia de testes baseados em modelos citada na Seção 2.3. Os modelos do sistema aqui utilizados são redes de autômatos temporizados os quais são discutidos a seguir. Uma adaptação à teoria dos autômatos temporizados foi feita para possibilitar a criação de modelos de ambiente de forma ágil, detalhes serão dados a seguir. A Figura 3.4 representa graficamente o processo execução do Gungnir. A seguir são detalhados os elementos que compõem o Gungnir apontadas na Figura 3.4.

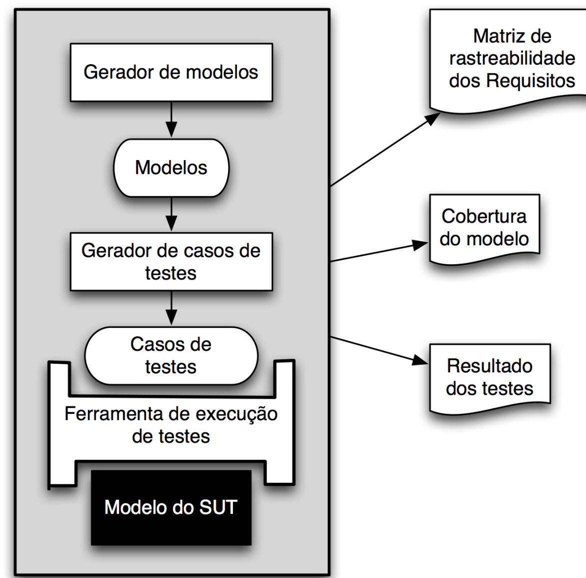


Figura 3.4: Processo de execução do Gungnir.

3.2.1 Modelos

Os modelos da especificação ISA 5.2 e do programa Ladder manipulados pelo Gungnir são descritos utilizando autômatos temporizados como formalismo. Conforme descrito na Seção 2.1.1, os CLPs possuem características especiais de execução as quais devem ser mantidas, com a finalidade de gerar modelos fieis de controladores lógicos programáveis. Para ilustrar a idéia da concepção dos modelos iremos utilizar o exemplo da Figura 3.5

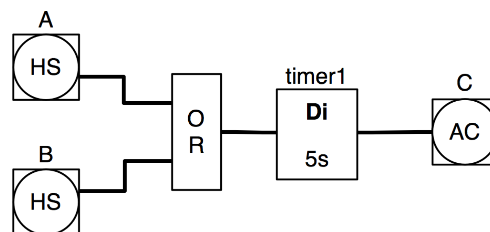


Figura 3.5: Exemplo de especificação ISA 5.2.

As etapas do *Scan Cycle* de um CLP são expressas em termos de autômatos, ou seja, para cada etapa da execução, um ou mais autômatos são gerados. Como exemplo utilizaremos o degrau mostrado na Figura 3.5. Extrai-se o autômato *ReadInputs* da primeira etapa a qual representa a leitura das placas de entrada e armazenamento dos valores nas devidas variáveis globais. Para cada variável de entrada do CLP uma transição é criada no *ReadInputs*. A Figura 3.6 mostra o resultado da geração do *ReadInputs* para o exemplo da Figura 3.5.

Na próxima etapa, a de processamento, os únicos elementos considerados no conjunto de elementos do ISA 5.2 e Ladder (citado na Seção 2.6 e Seção 2.7, respectivamente) são os

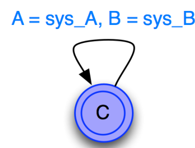


Figura 3.6: *ReadInputs* - um exemplo de autômato no qual é representado a leitura das variáveis de entrada do CLP.

temporizadores. Estes são de dois tipos: DI no ISA 5.2 e TON para o Ladder. O autômato no qual está modelado o comportamento do temporizador DI da Figura 3.5 é apresentado na Figura 3.7.

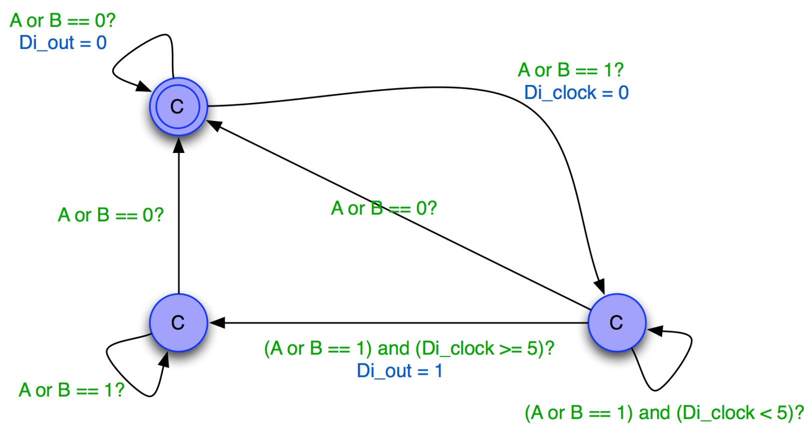


Figura 3.7: Temporizador DI (TON) - autômato no qual é representado o controle dos valores de saída do temporizador timer1 da Figura 3.5.

Por último, vem a escrita dos valores das variáveis globais de saída nas placas de saída do controlador lógico programável. A geração de autômatos para esta etapa deve seguir o mesmo raciocínio feito na etapa de leitura (*ReadInputs*), é necessário apenas um estado no autômato para gerar as saídas esperadas. É mostrado na Figura 3.8 o modelo do autômato de um único estado onde são atualizados os valores das placas de saída para o exemplo da Figura 3.5.

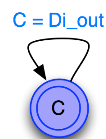


Figura 3.8: *WriteOutputs* - exemplo de autômato no qual é representada a escrita das variáveis de saída do CLP.

Observe que todos os estados utilizados até agora são *committed* (não consomem tempo), pois há a necessidade de que tudo seja executado ao mesmo tempo. Ao final do

Scan Cycle, todas as variáveis de entrada foram lidas, os temporizadores foram processados e, por fim, as variáveis de saída foram atualizadas.

Para completar o comportamento real de trabalho de um CLP, basta adicionar ao tempo do sistema o tempo de um *Scan Cycle*. Portanto, deve-se criar um último autômato chamado *TimeUpdater*, cuja função é atualizar o tempo do sistema como um todo. A seguir, são mostradas as ferramentas, estratégias e os algoritmos utilizados para gerar os modelos de autômatos temporizados discutidos até o momento.

3.2.2 Geração de Modelos de autômatos temporizados

Os modelos são obtidos a partir de arquivos XML(*eXtensible Markup Language*) os quais descrevem especificações ISA 5.2 e implementações Ladder. A lista das principais *tags* utilizadas é:

<input/> Variáveis de entrada;

<output/> Variáveis de saída;

<feedback/> Variáveis de realimentação;

<timer_DI/>, <timer_TON/> Temporizadores;

<rung/> Ligações de variáveis de entrada e *feedback* e temporizadores às variáveis de saída;

<and/>, <or/> e <not/> Operações binárias permitidas;

<ref_input/>, <ref_output/> e <ref_feedback> Referência a variáveis globais.

Exemplos de arquivos podem ser vistos no Anexo-**??**. Com a finalidade de gerar os modelos, foram desenvolvidos algoritmos que traduzem arquivos XML de especificação e implementação para as descrições Python de autômatos temporizados. O passo a passo executado para se concretizar a tradução é:

1. Cria-se, a partir de um *template*, o AT *ReadInputs* e adiciona-se a ele uma transição com uma atribuição do tipo *input_variável = sys_variável* para cada **<input/>**, deve conter também a guarda sobre o valor do *id* (identificador que indica qual transição deve ser executada numa ordem). Por exemplo, caso a transição seja a terceira a ser criada, deve conter a guarda *id == 2*. Todas as variáveis envolvidas são criadas nas variáveis globais;

2. Para cada **<feedback/>** encontrado, deve-se criar uma transição em *ReadInputs* com a guarda sobre o *id* e uma atribuição da variável de saída referente à realimentação. Por exemplo, suponha que a variável de *feedback* encontrada seja $test$, então a transição deve conter a seguinte atribuição: *feedback_test = output_test*;

3. Para cada relógio encontrado cria-se, a partir de um *template*, o AT do temporizador. A esse *template* deve-se aplicar o tempo de ativação e a expressão (obtida recursivamente sobre a entrada do temporizador) de entrada do temporizador. Deve-se cadastrar a variável de saída e o relógio interno nas variáveis globais. Em cada transição adicionar a guarda com o id do temporizador;

4. Cria-se, a partir de um *template*, o AT *WriteOutputs*. Para cada variável de saída deve-se adicionar uma transição com a guarda do *id* e uma atribuição referente à expressão de entrada (obtida recursivamente);

5. Por fim, cria-se, a partir de um *template*, o AT *TimeUpdater*.

Está ilustrado na Figura 3.9 a representação gráfica dos autômatos gerados para o exemplo da Figura 3.5. Observe que há uma sequência de *id*, na Figura 3.9, isso é o que permite a execução cíclica do CLP. Todos os estados não consomem tempo exceto o *TimeUpdater*. É nele que o tempo de *Scan* é atualizado. Como as variáveis de saída só serão vistas ao fim do ciclo, poderá se considerar que a leitura, o processamento e a escrita não gastam tempo e no fim o *Updater* espera o tempo necessário de um *Scan Cycle*.

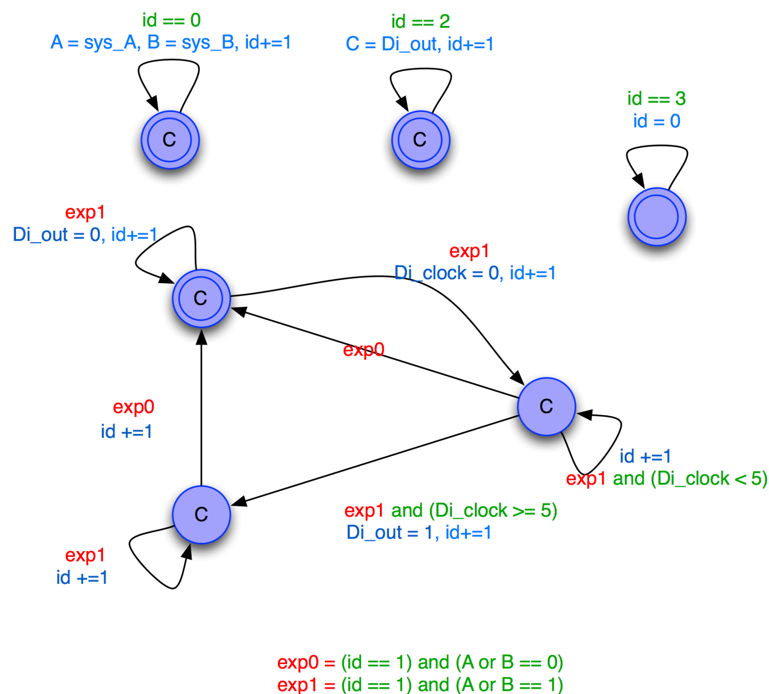


Figura 3.9: (a) *ReadInputs*, (b) *timer_DI*, (c) *WriteOutputs* e (d) *TimerUpdater*.

3.2.3 Modelagem do ambiente

O objetivo dos modelos do ambiente é criar sequências de vetores que representem as mudanças do ambiente ao longo do uso de um sistema. Nestes modelos podemos descrever

eventos relacionados ao ambiente que leva o sistema ao um estado desejado. Um estado desejado pode ser um estado raro ou crítico que deve ser especificamente testado a fim de evitar catástrofes ou grandes prejuízos. Os modelos possuem: estados, transições com ou sem restrições de tempo e atribuição de variáveis. É mostrado na Figura 3.10 os elementos pertencentes aos modelos de ambiente do Gungnir. As regras de restrição de tempo possuem dois parâmetros: a quantidade de tempo que deve ser esperado em unidades de tempo ou quando o tempo não é definido usa-se "*"e os limites de tempo que indicam o intervalo de ação do relógio. Para um tempo t, as possíveis combinações dos parâmetros para restrição de tempo são: $*$, t, $< t$, $<= t$, $> t$ e $>= t$. Assim, como exemplo de restrição de tempo podemos ter: $wait < 10$. O critério de cobertura do modelo do ambiente define que deve-se passar pelo menos uma vez por todas as transições do modelo.

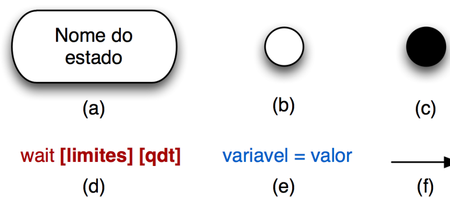


Figura 3.10: (a) estado, (b) estado inicial, (c) estado final, (d) limites temporais, (e) atribuição e (f) transição.

É apresentado na Figura 3.11 um exemplo de modelo de ambiente onde está modelado o usuário de um aparelho eletrônico x. Para que x seja ligado, deve-se apertar o botão b por 5 segundos e, para desligá-lo basta pressionar o botão que em no máximo 3 segundos. São utilizados modelos do ambiente para gerar de casos de testes direcionados no Gungnir.

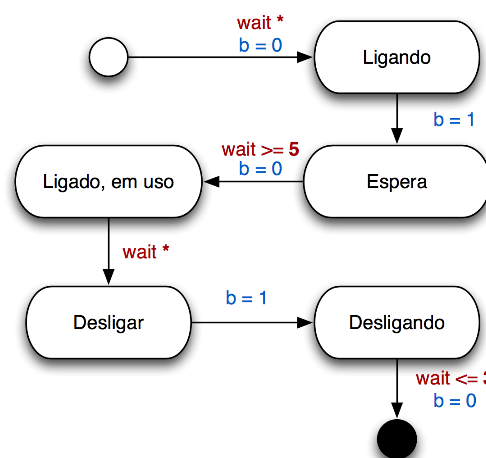


Figura 3.11: Modelo de ambiente que representa as ações de um usuário do equipamento x.

3.2.4 Geração dos casos de testes

Os casos de testes devem ser entendidos como um conjunto de dados de entrada combinado com um conjunto de dados de saída e o tempo de ocorrência desses. Para cada *Scan Cycle* temos um caso de teste. Se para um determinado conjunto de dados de entrada, o modelo da implementação apresentar na saída um conjunto de valores diferentes dos apresentados pelo modelo da especificação (oráculo), implica na ocorrência de uma falha.

Através do Gungnir são gerados aleatoriamente os valores de entrada de um CLP, fato também conhecido como força bruta. Em alguns momentos, essa forma de geração produz desperdícios computacionais por testar combinações de entrada que, baseado no modelo de falhas de um sistema ideal, nunca ocorrerão na realidade. Há também a geração de testes baseados no modelo do ambiente que direciona a simulação para a ocorrência de eventos raros, tendo em vista que em alguns momentos, principalmente, para avaliar temporizadores, a probabilidade de acontecer uma determinada combinação de valores aleatórios por ciclos seguidos é muito pequena. Resumindo, os modelos de ambiente testam cenários críticos ou raros. Por exemplo, um determinado temporizador tem sua saída em nível alto caso uma variável de entrada v mantenha seu valor em nível alto por pelo menos seis *Scan Cycles*. Obtemos, então, uma chance em torno de 1,5625% ($(1/2)^6 = 0.015625$) de acontecer, ou seja, para cada 64 sequências de 6 números, uma é a sequência de ativação do temporizador. Com isso, temos que gerar pelo menos 384 casos de testes que envolvam a variável v a fim de garantir que temporizador seja ativado. Esse quadro piora ainda mais quando há combinação de variáveis nas ativações do temporizador.

As falhas são capturadas fazendo a comparação dos valores das saídas dos modelos da especificação e da implementação para o mesmo conjunto de valores de entrada. Inicialmente, são gerados os valores de entrada, depois, esses valores são aplicados ao modelo da especificação. Desta forma, durante a simulação do modelo da especificação (o oráculo do sistema), ao ser concluído um *Scan Cycle* as saídas desse modelo são armazenadas. Então, as entradas utilizadas no modelo da especificação devem ser aplicadas ao modelo da implementação gerando, assim, valores de saída os quais devem ser comparados com o gabarito. Partindo do pressuposto que o modelo da especificação está correto, caso haja algum diferença entre os valores de saída da especificação e da implementação, uma falha na implementação foi encontrada. Logo, é exibida uma mensagem de erro indicando onde ocorreu a falha.

Baseado no subconjunto de elementos dos diagramas ISA 5.2 e dos programas Ladder utilizados neste trabalho, é observada a existência de degraus com ou sem temporizadores. Degraus não temporizados são testados utilizando força bruta. Já os degraus com temporizadores devem ser testados por meio de gerações de casos de teste baseados no modelo do ambiente.

Foi criado um critério de cobertura que faz parte da família *Boundary* que avalia o nível

de abrangência dos testes o qual utiliza a heurística do fator determinante, para definir as classes de equivalência. Essa heurística revela um conjunto de entradas que descrevem o comportamento essencial do degrau, uma espécie de radical do qual podem ser obtidos todos os valores de entrada e saída. Por exemplo, podemos extrair a heurística encontrada na Figura 3.12.b da porta *and* na Figura 3.12.a, ou seja, qualquer subconjunto de valores de entrada onde $A = 0$ ou $B = 0$ o resultado é zero. Para que o valor de saída seja um, é necessário inserir o valor um nas duas entradas. Já para Figura 3.12.c, é gerada a heurística Figura 3.12.d a qual indica que, para o valor de C ser zero é necessário que as entradas sejam zero e que para que seja um, pelo menos uma das duas entradas tem que ser um. O raciocínio segue para Figura 3.12.e e 3.12.f com a atenção para a substituição dos valores da heurística da Figura 3.12.b na primeira coluna da heurística da Figura 3.12.d conforme é demonstrado na Figura 3.14. A heurística do fator determinante tem sua eficácia garantida apenas nos casos em que, para qualquer degrau, as variáveis de entrada são referenciadas apenas uma vez; do contrário, não há cobertura completa em alguns casos como o da Figura 3.13.

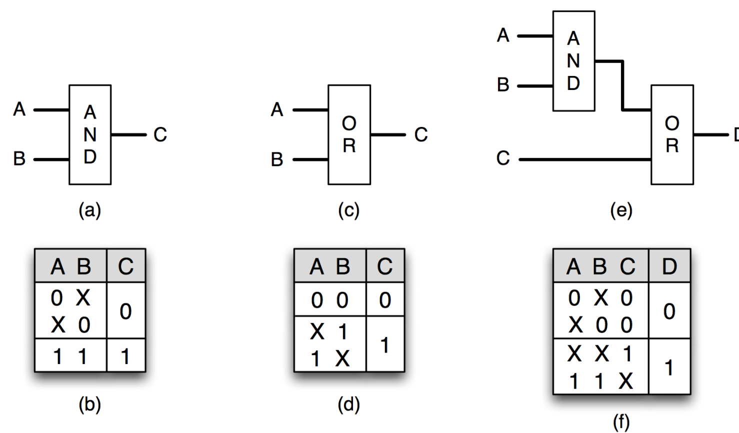


Figura 3.12: (a) porta *and*, (b) Heurística para expressões do tipo *and*, (c) porta *or*, (d) Heurística para expressões do tipo *or*, (e) combinação de portas *and* e *or* formando a expressão *(A and B) or C* e (f) Heurística para expressões *(A and B) or C*.

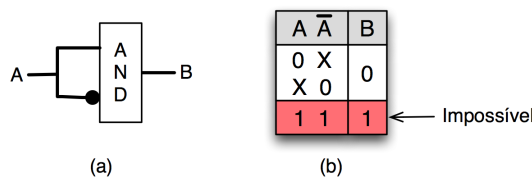


Figura 3.13: (a) Degrau com mais de uma referência para a variável A, (b) Heurística com valores impossíveis.

É mostrado na Figura 3.15 um exemplo da aplicação da heurística. Cada linha (vetor) da tabela da heurística é chamada de classe de valor. Na terceira linha da Figura 3.15.b temos a classe de valor: $A = X, B = X, C = X$ e $D = 0$ o que significa dizer que qualquer combinação

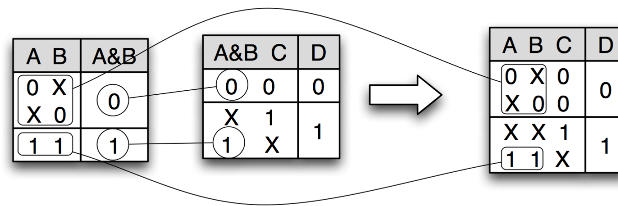


Figura 3.14: Passo a passo da obtenção da heurística do degrau da Figura 3.12.e.

de A, B e C não alteram o valor da saída E quando D é zero[3]. Um exemplo de vetor que se encaixa nessa classe é $A = 1, B = 0, C = 1$ e $D = 0$, também pode ser visto como 1010. Observe que existem oito possíveis vetores de valor que se encaixam na classe XXX0 (0000, 0010, 0100 0110, 1000,1010,1100,1110). Se testarmos os oitos, estarmos repetindo o comportamento do degrau. Há vetores que se encaixam em mais de uma classe, por exemplo no caso da Figura 3.15, o vetor 0000. A fim de mantermos a aplicação de TBM, definimos um critério de cobertura o qual diz que para todos os degraus devem ser gerados n vetores pertencentes a cada classe definida na heurística. Pode-se, então, variar a intensidade da cobertura utilizando os valores de n, o mínimo é um. A Figura 3.16 mostra um conjunto exemplo de vetores que fazem cobertura $n = 1$ para o caso da Figura 3.12.e. Os temporizadores são tratados como um degrau, portanto para cada temporizador é calculada uma heurística, e essa também faz parte do critério de cobertura.

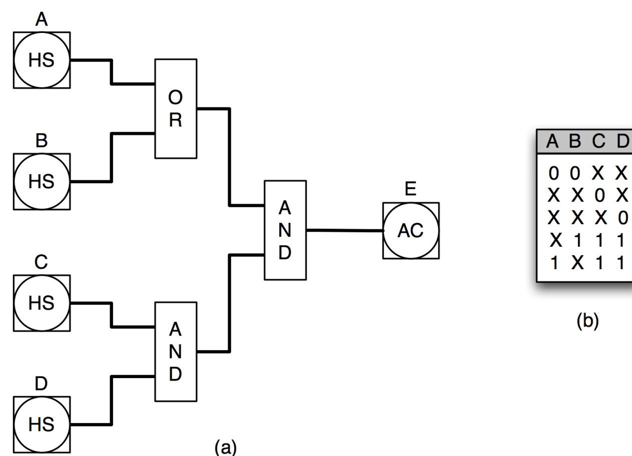


Figura 3.15: (a) Degrau de um diagrama ISA 5.2 e (b) heurística do fator determinante para o caso

Inicialmente, serão gerados os casos de testes baseados nos modelos do ambiente, a partir de então, os testes serão executados na força bruta até que o critério de cobertura seja alcançado para todos os degraus. Ao usuário resta o controle na qualidade da cobertura, basta escolher o valor de n no critério de cobertura. Quanto mais elevado o valor de n mais

[3] O X significa que não importa o valor da variável no vetor. Por exemplo, X0, pode ser 00 ou 10, pois não há alteração do valor da saída com a mudança no primeiro valor.

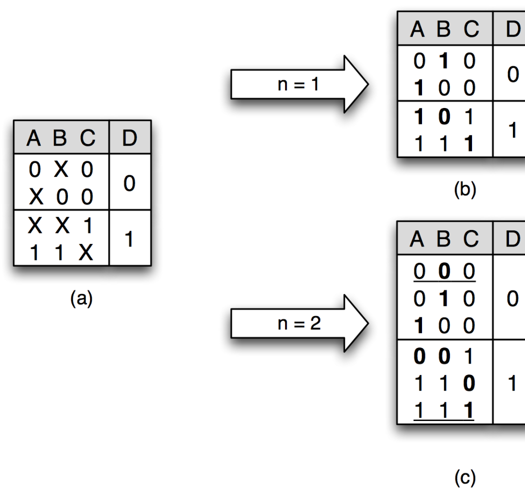


Figura 3.16: (a) Heurística do fator determinante para degrau da Figura 3.12.e, (b) vetores satisfazem a cobertura com $n = 1$ e (c) vetores satisfazem a cobertura com $n = 2$. Os vetores sublinhados encaixam-se em mais de uma classe de valor.

casos são testados. Contudo, deve-se observar que $n = 1$ já encontra qualquer problema de lógica entre a especificação e a implementação quando não temporizadas. Números grande para n, dependendo do tamanho do sistema, podem causar repetições de testes o que caracteriza desperdício. Para obter testes eficazes, deve-se desenvolver modelos de ambiente o mais próximo possível da realidade.

3.2.5 Execução dos casos de testes

A execução dos casos de testes é apenas uma chamada de comando. Deve-se passar como parâmetros o valor do nível de cobertura n e os caminhos para o xml da especificação, para o xml da implementação e para o xml do modelo do ambiente. Caso alguma saída não esteja de acordo com o esperado durante os testes, a mensagem de falha "FALHA: Implementação incorreta."é mostrada para o usuário e o degrau o qual contém o erro é indicado para possível correção. Quando os testes são executados com sucesso, a mensagem "Não foram encontradas falhas na implementação."é exibida na tela.

Como auxílio, um arquivo **.vcd** é criado com as entradas e saídas executadas no modelo da implementação para possível análise visual. É mostrado na Figura 3.17 um exemplo de visualização de um arquivo **.vcd** utilizando a ferramenta Scansion[4].

Aspectos relacionados a detalhes das implementações da ferramenta Gungnir estão disponíveis na wiki do projeto[5]. No próximo capítulo, para completar o entendimento do uso da Gungnir na geração de casos de testes utilizando modelos, é apresentado como estudo de caso o controle de engarrafadora.

[4]Fonte: site oficial da ferramenta Scansion - http://www.logicpoet.com/scansion/
[5]http://bitbucket.org/rodrigopex/msc/wiki

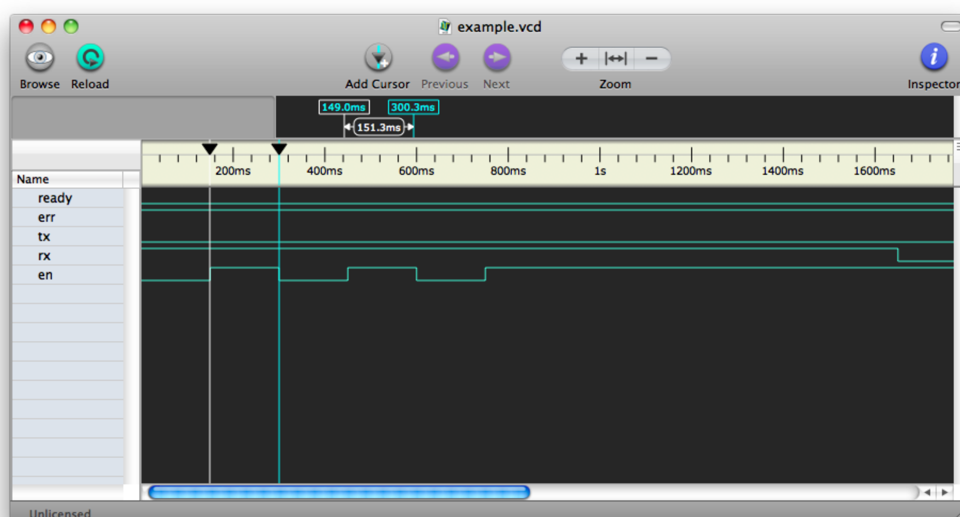


Figura 3.17: Screenshot do aplicativo Scanion com um exemplo de arquivo **.vcd**.

Capítulo 4

Estudo de caso

Este capítulo destina-se a detalhar o uso e demonstrar a eficácia do Gungnir como ferramenta para geração e execução automática de testes. Teremos como caso de teste o controle de engarrafadora o qual é apresentado a seguir.

Conforme descrito em [9], na Figura 4.1 um sistema que tem a finalidade de encher garrafas é ilustrado. Seu funcionamento ocorre da seguinte forma: uma vez que o botão de inicializar (PB1) é pressionado, o motor de auto-realimentação (M2) é ligado. Este motor permanecerá ligado até que o botão de parar (PB2) seja acionado. O motor M1 será ativado assim que o sistema for iniciado (M2 estiver ligado) e irá parar quando o sensor (LS) detectar uma garrafa na posição correta. Quando a garrafa estiver na posição correta e 0.5 segundos se passarem, o solenoide (SOL) irá abrir a válvula para liberar o refrigerante e o enchimento ocorrer até que o fotosensor (PE) detecte um nível adequado de líquido no interior da garrafa. Após ser enchida, a garrafa permanecerá nesta posição durante 0.7 segundos. Em seguida, o motor M1 é inicializado. Este irá permanecer ligado até que o sensor detecte outra garrafa.

Nas Figuras 4.2 (a) e (b) são apresentados o diagrama ISA 5.2 e o programa Ladder para a Engarrafadora.

4.1 Utilizando o Gungnir

Inicialmente, deve-se executar o Gungnir na linha de comando utilizando o seguinte comando:

```
#> gungnir [caminho para o arquivo xml do diagrama ISA 5.2]
           [caminho para o arquivo xml do programa Ladder]
           -env_model[caminho para o arquivo xml do modelo do ambiente]
           -n[critério de cobertura]
           -vcd_file_path[caminho para o arquivo vcd]
```

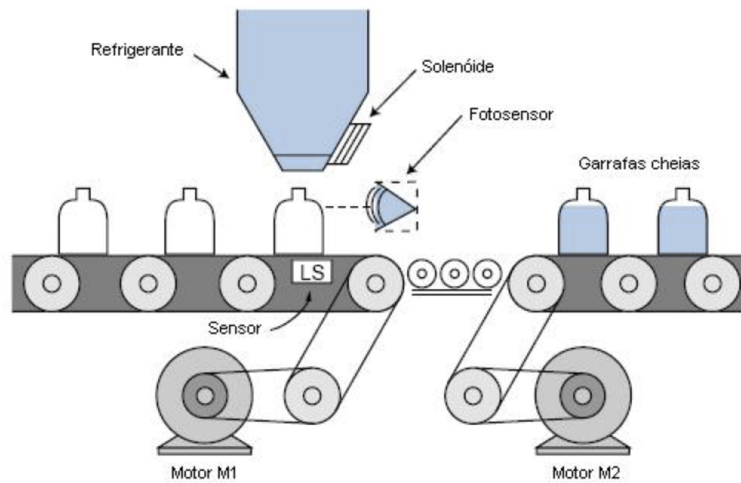


Figura 4.1: Engarrafadora - Fonte: [6], página 485

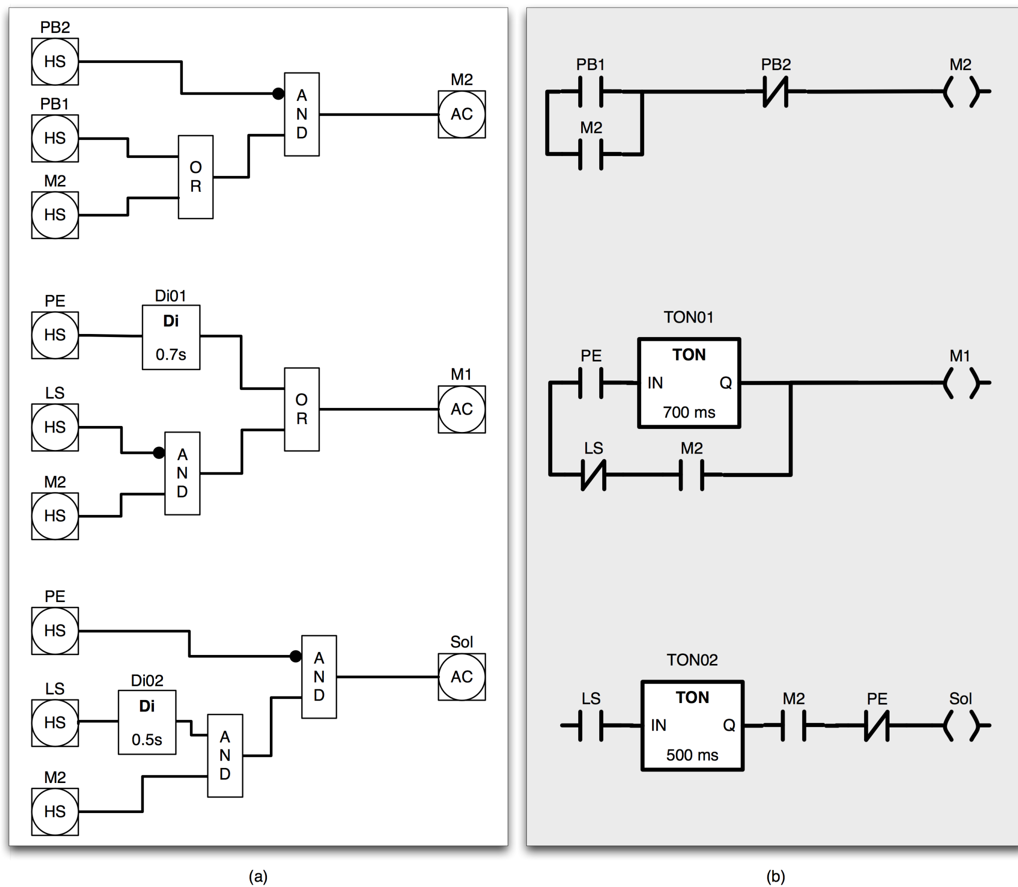


Figura 4.2: Engarrafadora - (a) ISA 5.2 e (b) Ladder

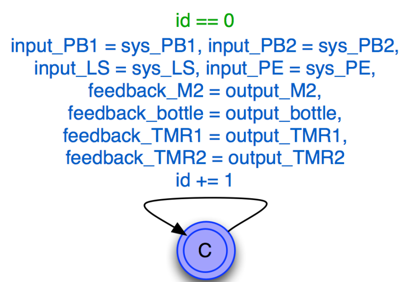


Figura 4.3: Autômato *ReadInputs*

Alguns exemplos de saída textual do comando de acionamento do Gungnir podem ser encontrados na wiki do projeto[1]. O passo a passo do processo interno da ferramenta Gungnir consiste da geração dos modelos da especificação e da implementação, geração dos vetores de entrada baseados no modelo do ambiente, aplicação dos testes ao modelo da especificação e implementação e, por fim, geração do relatório de execução.

4.2 Geração dos modelos

Conforme descrito na Seção 3.2.2, são gerados os modelos de autômatos temporizados baseados nos arquivos XML os quais descrevem o diagrama ISA 5.2 e o programa Ladder de controle da engarrafadora, Figura 4.1. Os códigos dos respectivos arquivos XML podem ser encontrados na wiki do projeto. O modelo gerado para o arquivo XML da especificação ISA 5.2 está representado nas Figuras 4.3, 4.4, 4.5, 4.6 e 4.7.

Na Figura 4.3 foi modelado o autômato o qual representa a leitura do CLP. Nesse autômato, todas as variáveis de entrada são atualizada de acordo com a palavra temporizada que o sistema gera, quando está usando a força bruta, são vetores de valor aleatório; quando utilizando o modelo do ambiente o mesmo define os valores os quais serão escritos nas variáveis de entrada. As variáveis de feedback recebem o valor atual das variáveis de saída, pois, segundo o comportamento padrão de um CLP, as variáveis de feedback recebem os valores das saídas apenas no *Scan Cycle* seguinte.

Já nas Figuras 4.4 e 4.5 estão representados os modelos de AT dos temporizadores. Esses autômatos são responsáveis pelo processamento dos valores que necessitam de tempo para serem modificados. O cálculo e atualização dos valores das variáveis de saída estão modelados no autômato temporizado da Figura 4.6. Com a finalidade de atualizar o tempo de simulação, é criado o autômato da Figura 4.7, pois este contem apenas um estado o qual consome o tempo de um *Scan Cycle* já que a granularidade de tempo de simulação é de um *Scan Cycle*. A geração é feita de forma análoga para o programa Ladder.

[1]http://bitbucket.org/rodrigopex/msc/Home

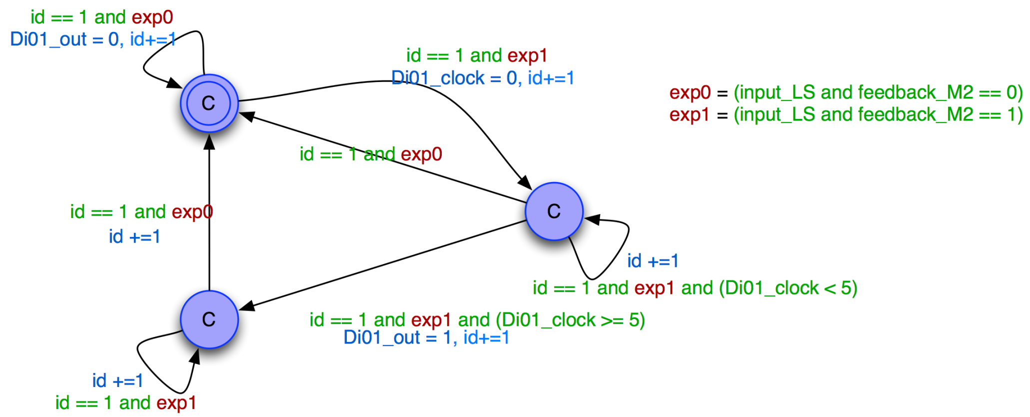


Figura 4.4: Autômato *Timer_Di01*

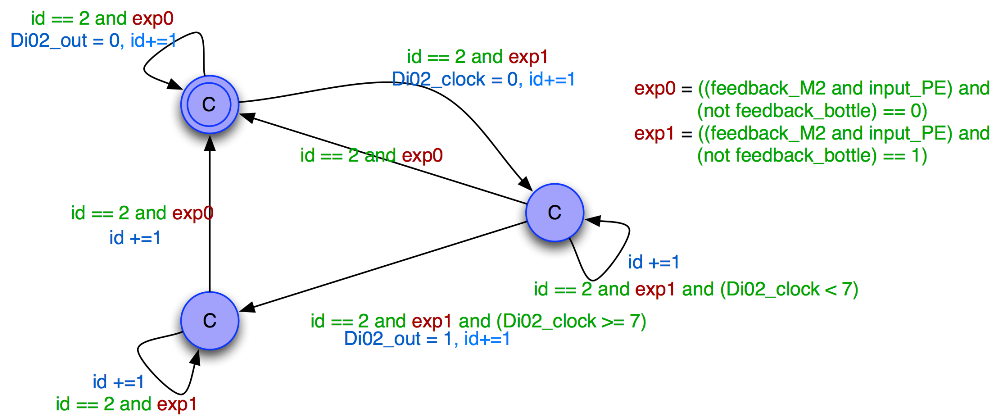


Figura 4.5: Autômato *Timer_Di02*

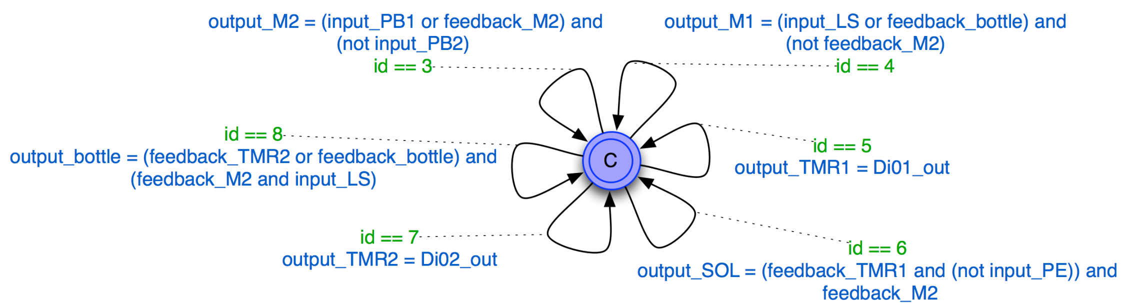


Figura 4.6: Autômato *WriteOutputs*

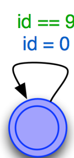


Figura 4.7: Autômato *TimeUpdater*

4.3 Geração das entradas

Os valores das entradas, inicialmente, são guiados pelo modelo do ambiente. A Figura 4.8 mostra o modelo do ambiente para a engarrafadora descrevendo as modificações, no tempo, dos elementos externos o quais interagem com o sistema. Para o modelo da Figura 4.8, um conjunto válido de vetores gerado é mostrado na Tabela 4.1.

Tabela 4.1: Variação do valor das entradas seguindo o modelo de ambiente da Figura 4.8 para $n = 1$.

Scan Cycle	**PB2**	**PB1**	**LS**	**PE**
0	0	1	0	0
1	0	1	0	0
2	0	1	0	0
3	0	1	0	0
4	0	1	1	0
5	0	1	1	0
6	0	1	1	0
7	0	1	1	0
8	0	1	1	0
9	0	1	1	0
10	0	1	1	0
11	0	1	1	0
12	0	1	1	0
13	0	1	1	1
14	0	1	1	1
15	0	1	1	1
16	0	1	1	1
17	0	1	1	1
18	0	1	1	1
19	0	1	1	1
20	0	1	1	1
21	0	1	1	0
22	0	1	0	0
23	0	1	0	0
24	0	1	0	0
25	0	1	0	0
26	1	0	0	0
27	1	0	0	0
28	1	0	0	0

A partir do momento em que for coberto o modelo de ambiente na geração de entradas, iniciam-se as gerações aleatórias a fim de completar a cobertura dos modelos da especificação. O parâmetro n, valor passado como parâmetro pelo usuário na linha de comando de execução do Gungnir, determina também quantas vezes serão geradas as entradas baseado

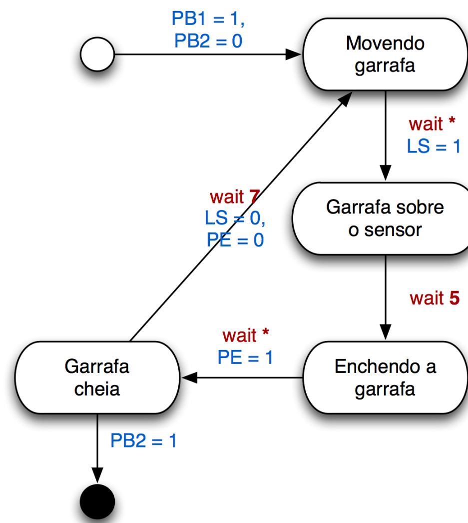


Figura 4.8: Modelo do ambiente para o sistema de controle da engarrafadora.

M2	PB1	M2*	PB2
0	0	0	x
	x	x	1
1	1	x	0
	x	1	0

M1	DI01	LS	M2
0	0	1	x
	0	x	0
1	1	x	x
	x	0	1

DI01	
T	PE
0	0
1	1

SOL	PE	DI02	M2
0	1	x	x
	x	0	x
	x	x	0
1	0	1	1

DI02	
T	LS
0	0
1	1

Figura 4.9: Heurística do fator determinante obtida do modelo da especificação do sistema de controle da engarrafadora.

no modelo do ambiente. Ao longo da execução, são marcadas as classes de valor em que se encaixam os valores gerados. Até um momento em que, para todas as heurísticas do fator determinante, todas as classes de valor foram executadas pelo menos n vezes. As heurísticas obtidas do modelo da especificação da engarrafadora estão ilustradas na Figura 4.9.

4.4 Execução dos testes

Após a geração dos casos de teste baseado no modelo do ambiente, os testes são iniciados. Os valores de entrada aleatórios são gerados *on-the-fly* durante a execução dos casos de teste. O Gungnir armazena os resultados da execução dos testes em um arquivo **.vcd**. Para o caso da engarrafadora, executamos o programa Ladder correto o qual está representado na Figura 4.2.b.

Inicialmente são gerados os testes sem usar o modelo do ambiente, como objeto de comparação. Os relatórios das execuções dos testes do sistema de controle da engarrafadora

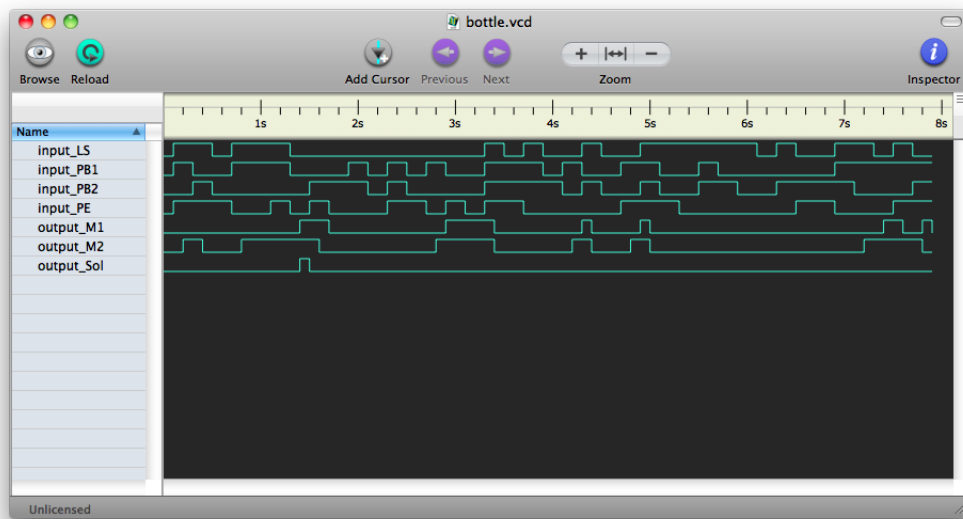


Figura 4.10: Comportamento da engarrafadora para as entradas geradas sem modelo do ambiente com $n = 1$.

com $n = 1$ e $n = 10$ podem ser encontradas na wiki do projeto[2]. Observe que no relatório para $n = 1$ foram executados 229 *Scan Cycles* para chegar ao critério de cobertura, já quando $n = 10$ temos um aumento do número de ciclos, passou a ser de 678, pois agora cada classe de valor foi gerada pelo menos 10 vezes. Em ambos os casos o resultado da execução foi positivo, ou seja, a implementação está de acordo com a especificação.

Agora são gerados os testes para o sistema de controle da engarrafadora utilizando o modelo do ambiente ilustrado na Figura 4.8. Para $n = 1$ e para $n = 10$ os relatórios de execução estão na wiki do projeto. Com a utilização do modelo do ambiente a cobertura para $n = 1$ foi alcançada com 27 *Scan Cycles*; já no caso de $n = 10$, 296 *Scan Cycles* foram necessários. Observa-se um aumento de rendimento com o uso do modelo do ambiente. Houve uma economia computacional de 88.2% quando $n = 1$ e de 56.3% com $n = 10$ com relação à geração sem modelo do ambiente.

Para validar a aplicação da ferramenta, vamos inserir falhas e analisar os arquivo modificados. Os falhas foram escolhidas de forma a abranger as mais comuns no desenvolvimento. Utilizaremos o modelo do sistema e $n = 1$.

4.4.1 Erro 1

Substituição de um contato normalmente aberto por um normalmente fechado na entrada LS.

Números de ciclos de *Scan*: 2;

[2]https://bitbucket.org/rodrigopex/msc/wiki/Home

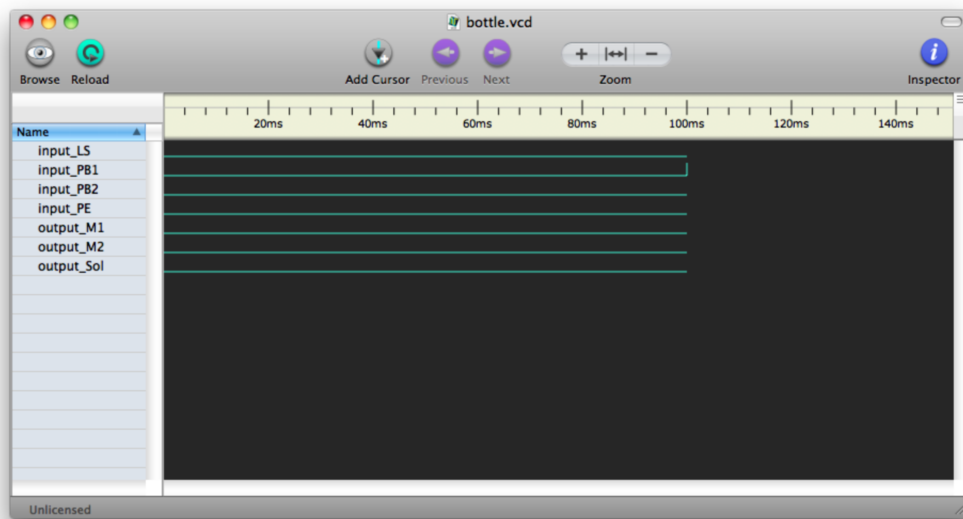


Figura 4.11: Comportamento da implementação da engarrafadora com o falha 1 injetado.

Degrau Defeituoso: M1;

Resultado do teste: Falha, implementação incorreta;

Arquivo .vcd: Figura 4.11.

4.4.2 Erro 2

Troca de uma operação lógica *and* por uma *or*.

Números de ciclos de *Scan*: 1;

Degrau Defeituoso: M2;

Resultado do teste: Falha, implementação incorreta;

Arquivo .vcd: Figura 4.12.

4.4.3 Erro 3

Aumento do tempo do temporizador timer_TON02 de 5 para 17 no degrau de Sol.

Números de ciclos de *Scan*: 12;

Degrau Defeituoso: Sol;

Resultado do teste: Falha, implementação incorreta;

Arquivo .vcd: Figura 4.13.

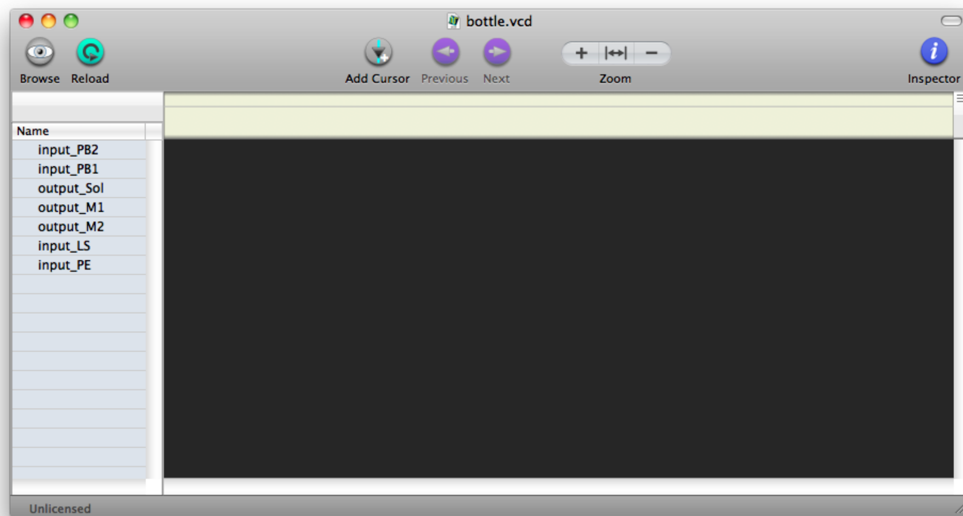


Figura 4.12: Comportamento da implementação da engarrafadora com o falha 2 injetado.

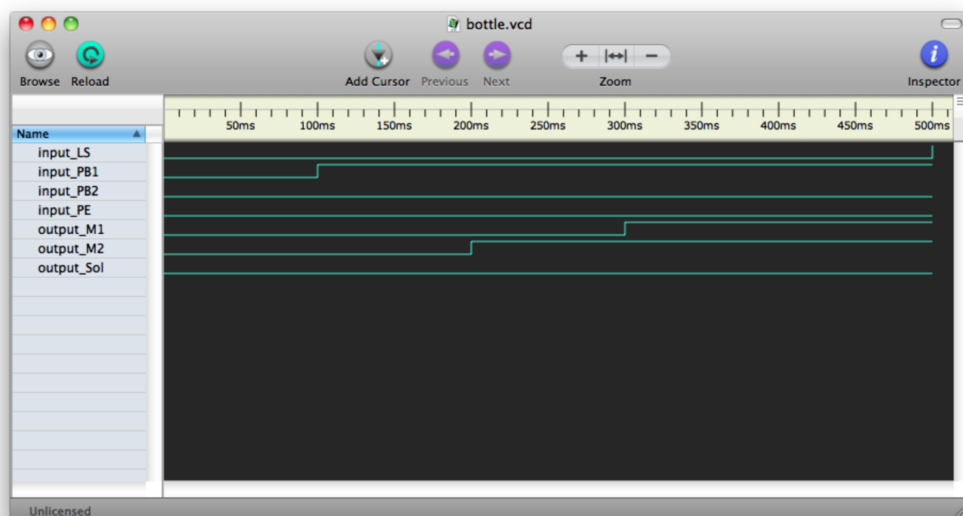


Figura 4.13: Comportamento da implementação da engarrafadora com o falha 3 injetado.

Capítulo 5

Conclusões

O objetivo principal nesse trabalho é aumentar a confiança no funcionamento de sistemas de eventos discretos fornecendo meios para que desenvolvedores possam executar testes que aumentam a confiabilidade do sistema de forma automática. Os testes são gerados de forma semi-aleatória, com a finalidade de cumprir os critérios de cobertura de elementos temporizados ou não. O trabalho foi baseado na metodologia de testes baseados em modelos e como resultado foi desenvolvida a ferramenta Gungnir. Algumas adaptações foram feitas com relação à metodologia TBM, a principal é o fato dos casos de testes serem gerados a partir de modelos da especificação e aplicados também a modelos, esses últimos de implementação. Como linguagem de programação de sistemas de controle suportada pelo Gungnir usamos o Ladder e como padrão para especificação, os diagramas ISA 5.2. Ambas fazem parte do padrão internacional IEC 61131-3.

Como contribuição deste trabalho ficam destacadas adaptações feitas nos modelos e no processo do trabalho apresentado em [9], a criação do critério de cobertura, a criação da heurística do fator determinante, disponibilização de diagramas para modelagem do ambiente, desenvolvimento do módulo para geração de testes na força bruta e baseados em modelos do ambiente e, por fim, também deve ser visto com a devida atenção o simulador de modelos de autômatos temporizados [2], parte integrante do Gungnir.

A geração automática de testes fornece ao desenvolvedor a liberdade de testar o software o quanto necessário. Quanto mais tempo for gasto na execução de testes, maior será a qualidade do sistema, ou seja, quanto menos erros maior a qualidade. O relatório de execução dos testes serve como auxilio na correção de falhas do sistema.

Este trabalho foi elaborado utilizando a idéia de *Reproducible Research*. Foi utilizado controle de versão para todos os itens: código, texto, figuras, etc. Todos os arquivos usados e gerados para essa dissertação encontram-se disponíveis para download pelo link: http://bitbucket.org/rodrigopex/msc.

5.1 Trabalhos futuros

Há muitos trabalhos a serem feitos na área de automatização de testes para sistemas de eventos discretos. Outras sugestões de temas como possíveis trabalhos futuros na área:

- Expandir o subconjunto de instruções da linguagem Ladder e do padrão ISA 5.2, pois foram utilizados os elementos lógicos *and, or* e *not* e elementos de comunicação com o exterior *handswitch* e atuadores. Há uma gama de outros elementos que devem ser inseridos no contexto do Gungnir a fim de torná-lo aplicável ao maior número de aplicações possíveis;
- Fazer uma análise estatística mais detalhada sobre a heurística e os critérios de cobertura desenvolvidos neste trabalho;
- Executar estudos de caso mais complexos;
- Criar uma interface que permita comunicar o Gungnir, em uma estação de trabalho, diretamente com o CLP via OPC (*OLE for Process Control*) possibilitando, assim, execução *online* dos testes no CLP viabilizando testes de monitoramento em ambiente real de execução.

Referências

[1] J.-R. Abrial. *The B-book: assigning programs to meanings.* Cambridge University Press, New York, NY, USA, 1996.

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

[3] G. Behrmann, David A, and Prof. K.G. Larsen. A tutorial on uppaal. In M. Bernardo and F. Corradini, editors, *LNCS, Formal Methods for the Design of Real-Time Systems (revised lectures)*, volume 3185, pages 200–237. Springer Verlag, 2004.

[4] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on UPPAAL. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS, pages 200–236. Springer–Verlag, September 2004.

[5] E.A. Bryan and L.A. Bryan. *Programmable controllers theory and implementation.* Pearson Prentice Hall, 1997.

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

[7] Leandro Dias da Silva, Luiz Paulo de Assis Barbosa, Kyller Gorgnio, Angelo Perkusich, and Antnio Marcus Nogueira Lima. On the Automatic Generation of Timed Automata Models from Function Block Diagrams for Safety Instrumented Systems. In *34th Annual Conference of the IEEE Industrial Electronics Society (IECON 2008)*, pages 291–296, Orlando, USA, November 2008. IEEE Industrial Electronics Society.

[8] 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*, pages 406–412, Sept. 2007.

[9] Kézia de Vasconcelos Oliveira. Geração automática de testes de conformidade para programas de controladores lógicos programáveis. Master's thesis, Universidade Federal de Campina Grande, Agosto 2009.

[10] Kézia de Vasconcelos Oliveira, Kyller Gorgonio, Leandro Dias da Silva, Antonio Marcus Nogueira Lima, and Angelo Perkusich. Extração automática de autômatos temporizados a partir de diagramas ladder. Sept. 2009.

[11] Jin Song Dong, Ping Hao, Shengchao Qin, Jun Sun, and Wang Yi. Timed automata patterns. *Software Engineering, IEEE Transactions on*, 34(6):844 –859, nov.-dec. 2008.

[12] William M. Goble and Harry Cheddie. *Safety Instrumented Systems Verification: Practical Probabilistic Calculations*. ISA, 2005.

[13] Anders Hessel, Kim Larsen, Marius Mikucionis, Brian Nielsen, Paul Pettersson, and Arne Skou. Testing Real-Time systems using UPPAAL. In *Formal Methods and Testing*, pages 77–117. 2008.

[14] IEEE Computer Society. *Software Engineering Body of Knowledge (SWEBOK)*. Angela Burgess, EUA, 2004.

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

[16] Edmund M. Clarke Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 1999.

[17] N. Kosmatov, B. Legeard, F. Peureux, and M. Utting. Boundary coverage criteria for test generation from formal models. pages 139 – 150, nov. 2004.

[18] Kim G. Larsen, Marius Mikucionis, and Brian Nielsen. *Uppaal Tron User Manual. In http://www.cs.aau.dk/ marius/tron/*, 2009.

[19] H. Lo andding and J. Peleska. Timed moore automata: Test data generation and model checking. pages 449 –458, apr. 2010.

[20] Angelika Mader. A classification of plc models and applications, 2000.

[21] Andrew Parr. *Programmable Controllers an Engineers Guide*. Newnes, 3nd edition, 2003.

[22] James L. Peterson. Petri nets. *ACM Comput. Surv.*, 9(3):223–252, 1977.

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

[24] Stuart Reid. The art of software testing, second edition. glenford j. myers. revised and updated by tom badgett and todd m. thomas, with corey sandler. john wiley and sons, new jersey, u.s.a., 2004. isbn: 0-471-46912-2, pp 234: Book reviews. *Softw. Test. Verif. Reliab.*, 15(2):136–137, 2005.

[25] J. M. Spivey. *The Z notation: a reference manual.* Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1992.

[26] L.H. Tahat, B. Vaysburg, B. Korel, and A.J. Bader. Requirement-based automated black-box test generation. pages 489 –495, 2001.

[27] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.